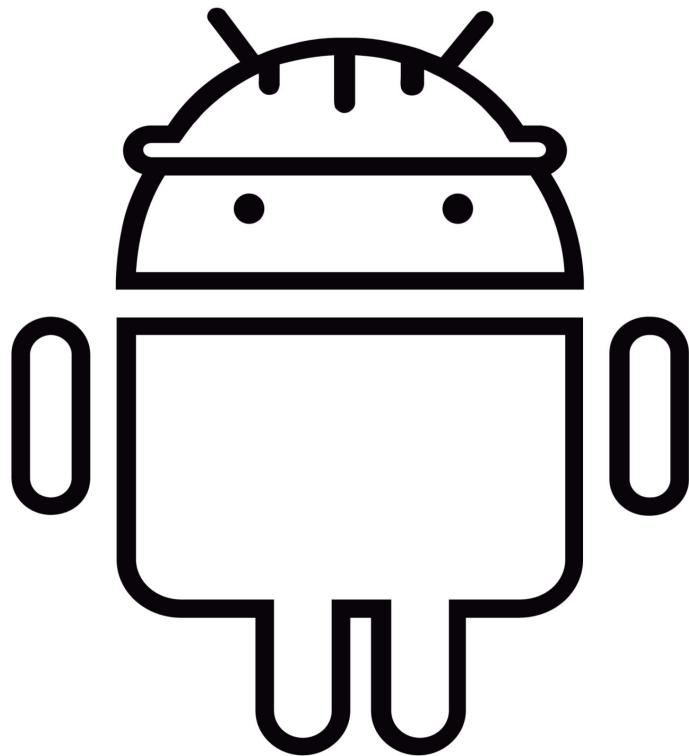


Desenvolvimento Móvel com Google Android

Curso FJ-57



 **caelum**
ensino e inovação

Apostila gerada especialmente para Willian Silva Moreira - moreiraws85@gmail.com



Conheça também:



alura

alura.com.br



Casa do Código
Livros e Tecnologia

casadocodigo.com.br

Blog da Caelum



blog.caelum.com.br

Newsletter



caelum.com.br/newsletter

Facebook



facebook.com.br/caelumbr

Twitter



twitter.com/caelum

Sumário

1 Conhecendo o mundo do Android	1
1.1 O que saberei fazer no fim desse capítulo ?	1
1.2 Aplicações Mobile	1
1.3 O Android	3
1.4 Instalação do Android	3
1.5 Dispositivos Android	4
1.6 Emuladores de Android	4
1.7 Exercício: Criando um projeto	5
2 Manipulando nossa primeira Activity	9
2.1 O que saberei fazer no fim desse capítulo ?	9
2.2 Entendendo o que foi gerado	9
2.3 Layouts no Android	10
2.4 Exercício: Modelando o layout através do editor visual	14
2.5 Mostrando botões na tela	21
2.6 Exercício: Fazendo a primeira interação com o sistema	25
3 Criando um Fluxo no Aplicativo	32
3.1 O que saberei fazer no fim desse capítulo ?	32
3.2 Criando uma Activity manualmente	32
3.3 Mostrando uma lista na tela	34
3.4 Adicionando a tela à Activity	34
3.5 Mostrando uma lista de tweets na tela	35
3.6 Porta de entrada da aplicação	38
3.7 Exercício: Exibindo uma lista no Android	39
3.8 Adicionando um botão para novo tweet	43
3.9 FloatingActionButton	44
3.10 Alerta pro Usuário	46
3.11 Exercício: Adicionando botão flutuante	47
3.12 Navegando pelas Activities	53

3.13 Exercício: Trocando de Activity	54
3.14 Utilizando Menus	55
3.15 Botão de voltar	58
3.16 Exercício: Melhorando o botão de salvar	59
4 Introdução a Arquitetura no Android	63
4.1 O que saberei fazer no fim desse capítulo ?	63
4.2 Modelando o Tweet	63
4.3 Mostrando a mensagem do tweet salvo	64
4.4 Exercício: Criando nosso primeiro modelo de maneira simples	65
4.5 Armazenando o Tweet	67
4.6 Exercício: Salvando o primeiro tweet com Room	74
4.7 Pegando a lista do banco	76
4.8 Mostrando a listagem na tela	77
4.9 Exercício: Exibindo listagem com Room	78
4.10 Mostrando a lista sempre atualizada	79
4.11 Exercício: Exibindo listagem de maneira correta	82
4.12 Estruturando nossa aplicação com as Componentes de Arquitetura	83
4.13 Exercício: Deixando nosso sistema mais maleável a mudanças	90
4.14 Atribuindo ação a um item da lista	93
4.15 Deletando tweet	94
4.16 Mostrando um alerta ao usuário	95
4.17 Exercício: Possibilitando a exclusão de um tweet	97
5 Utilizando a camera	101
5.1 O que saberei fazer no fim desse capítulo ?	101
5.2 Abrindo a câmera	101
5.3 Exercício: Abrindo o aplicativo da câmera	102
5.4 Pegando a foto tirada	103
5.5 Exercício: Esperando um resultado da câmera	109
5.6 Avisando a câmera que queremos uma resposta	113
5.7 Exercício: Prevendo quando o usuário cancelar a ação	114
5.8 Salvando a foto no banco de dados	115
5.9 Exercício: Salvando a foto	119
6 Personalizando nossa lista	122
6.1 O que saberei fazer no fim desse capítulo ?	122
6.2 Criando interface do item personalizada	122
6.3 Exercício: Exibindo a foto na lista	127

7 Desacoplando comportamentos com Fragments	132
7.1 O que saberei fazer no fim desse capítulo ?	132
7.2 Exercício: Pegando o projeto	132
7.3 Colocando um menu de navegação com BottomNavigationView	133
7.4 Exercício: Deixando nosso layout mais real	134
7.5 Reaproveitando pedaços da tela	138
7.6 Trocando fragmentos na mesma tela	139
7.7 Exercício: Criando o primeiro fragment	141
7.8 Mostrando campo de busca na tela	145
7.9 Filtrando tweets	146
7.10 Definindo OptionsMenu em Fragments	148
7.11 Exercício: Fazendo a parte de busca	149
8 Manipulando dados reais	152
8.1 O que saberei fazer no fim desse capítulo ?	152
8.2 Criando tela de login	152
8.3 Exercício: Criando nosso usuário	153
8.4 Desafio Opcional: Permitindo que o usuário tire uma foto	158
8.5 Acessando dados de outras pessoas	159
8.6 Pensando na estrutura do projeto	163
8.7 Exercício: Criando o usuário na API	167
8.8 Exercício: Logando o usuário na API	172
8.9 Exercício Opcional: Fazendo o Logout do usuário	174
8.10 Exercício: Salvando os tweets na API	175
8.11 Atualizando lista com informações do servidor	180
8.12 Exercício: Listando os tweets inseridos na API	181
8.13 Atualizando a lista com Swipe	185
8.14 Exercício: Atualizando os tweets	187
9 Trabalhando com Maps	189
9.1 O que saberei fazer no fim desse capítulo ?	189
9.2 Mostrando um mapa pro usuário	189
9.3 Exercício: Configurando o Google Maps	190
9.4 Usando o mapa da Google	191
9.5 Exercício: Exibindo o mapa	192
10 Trabalhando com GPS	194
10.1 O que saberei fazer no fim desse capítulo ?	194
10.2 Acessando a localização	194

10.3 Passando coordenadas para o tweet	197
10.4 Exercício: Fazendo nosso tweet ter posição	198
10.5 Adicionando marcadores no mapa	201
10.6 Exercício: Exibindo no mapa os tweets	203

Versão: 23.8.14

CONHECENDO O MUNDO DO ANDROID

1.1 O QUE SABEREI FAZER NO FIM DESSE CAPÍTULO ?

- Criar um projeto através do Android Studio
- Entender cada etapa da criação do projeto
- Saber qual versão escolher para cada projeto

Agora é a melhor hora de respirar mais tecnologia!



Se você está gostando dessa apostila, certamente vai aproveitar os **cursos online** que lançamos na plataforma **Alura**. Você estuda a qualquer momento com a **qualidade** Caelum. Programação, Mobile, Design, Infra, Front-End e Business! Ex-aluno da Caelum tem 15% de desconto, siga o link!

[Conheça a Alura Cursos Online.](#)

1.2 APLICAÇÕES MOBILE

Atualmente, o poder de processamento dos celulares e outros aparelhos móveis é muito alto. É muito difícil encontrar alguém nos dias de hoje que não possua pelo menos um celular; tem até várias pessoas que possuem um celular profissional e um pessoal.

Por isso, mostrar seu *portfólio* de aplicações já diretamente na sua mão é algo muito prático nos dias de hoje. Já existe uma série de possibilidades para se acessar alguma aplicação por um dispositivo *mobile*, desde criar *sites* responsivos até programar diretamente para a plataforma específica, o que chamamos de **código nativo**.

Programar um *site* responsivo é algo bem prático, pois, com um único código, temos uma aplicação para *desktop* e *mobile*, não importando a plataforma. Porém, dependendo de como foi implementado, pode ser que o usuário só consiga acessar se estiver conectado à internet, para acessar o servidor com as informações para mostrar na tela. O usuário tem que acessar o navegador para só então digitar o seu

endereço. Para melhorar esse cenário, muitas aplicações desenvolveram atalhos para o próprio endereço, permitindo ao usuário habilitar um ícone de inicialização na tela do dispositivo. Além disso, muitas aplicações armazenam temporariamente uma *cache* com as últimas buscas realizadas no servidor. Mas, ainda assim, o usuário muitas vezes precisa ter internet uma primeira vez para conseguir visualizar qualquer informação.

Para melhorar esse cenário, surgiu a ideia de usar `WebViews`, que foi iniciada com o *Apache Cordova* e permite ter uma única base de código em `HTML`, `CSS` e `JS` para programar pensando em plataformas *mobile*. Assim como ele, outras tecnologias surgiram utilizando essa mesma ideia, como *PhoneGap* e *Ionic*. Elas facilitam bastante a vida, pois quem programa para *front-end* consegue criar facilmente as telas para *mobile*, e diversos *frameworks* surgiram para facilitar esse desenvolvimento, mas, no fundo, a aplicação é basicamente um site e, por isso, a interação da tela com o usuário em muitos momentos não ocorre como ele espera. Outro ponto importante a se observar é que não temos acesso a componentes nativas como câmera, GPS, *bluetooth* e sensores de inclinação. Para podermos utilizar essas componentes, é necessário que alguém crie algum *plugin* para isso e, caso haja atualização das componentes nativas ou da forma como são acessadas, ficamos dependentes do *plugin* ser atualizado.

Outra opção muito utilizada atualmente, pensando ainda em programar um código só para todas as plataformas, ou seja, **híbridamente**, é o *React Native*. Ele é uma biblioteca `Javascript` com a qual podemos escrever praticamente todo o código em `Javascript` e uma ponte faz a conversão desse código que escrevemos para o código nativo de cada plataforma. Com isso, mantém a vantagem de quem programa pra *front-end* conseguir programar facilmente para *mobile* e permite até escrever interfaces para todas as plataformas em uma única linguagem. Além disso, o acesso às componentes nativas é mais direto, com maior manutenção a essas componentes, pois a própria linguagem dá suporte a elas. Porém, o *React Native* ainda peca em performance pois, para interações como *swipe* ou animações ele acaba acessando a ponte de conversão a cada pequena mudança, gerando conversões desnecessárias.

A última opção híbrida que está crescendo bastante é o *Flutter*, ferramenta criada pela Google que usa a linguagem `Dart`. Ela possui quase todas as vantagens do *React Native*, sua inspiração, e melhora na parte da performance, pois o código nativo é gerado em tempo de compilação; o *Flutter* não tem a ponte que o *React Native* tem. Pensando na tela, o *Flutter* desenha diretamente na tela do dispositivo. Isso deixa o código bem mais rápido, mas, no fim das contas, ele desenha do jeito dele. Por mais que siga as recomendações do `Android` e do `iOS`, algumas componentes da tela vão ficar diferentes do que a respectiva componente nativa do `Android` e do `iOS`. Há uns pontos para se atentar. No *Flutter* o código para a tela fica no mesmo arquivo que a lógica da aplicação. Falaremos mais a fundo sobre isso mais pra frente, mas a questão é que o código fica mais misturado e difícil de entender.

Essas opções todas de código único facilitam bastante em praticidade e evitam replicação da lógica,

mas acabamos perdendo nas especificidades de cada plataforma, em usar exatamente as componentes nativas, para dar uma cara como cada plataforma sugere, da forma que o usuário está mais acostumado. Além disso, o processo de funcionamento de cada plataforma é um pouco diferente e, com o código único, o processo acaba sendo o mesmo. Pensando em relação à performance, também é mais vantajoso acessar diretamente o código nativo em vez de haver qualquer adaptação ou conversão. Uma coisa que algumas empresas acabam fazendo é desenvolver uma parte mais genérica em código híbrido, mas para algumas especificidades da tela ou de componentes que só alguma plataforma tenha optam pelo código nativo. Por isso, neste curso aprenderemos código nativo para Android.

1.3 O ANDROID

O Android é um sistema operacional que roda sobre o núcleo Linux. Ele foi inicialmente desenvolvido pela Android Inc., e depois passou para as mãos da Google (que a comprou em 2005) e posteriormente pela Open Handset Alliance. A plataforma permite que os desenvolvedores escrevam software na linguagem Java ou Kotlin controlando o dispositivo via bibliotecas desenvolvidas pela Google, com o objetivo de ser uma plataforma flexível, aberta e de fácil migração para os fabricantes.

SITE PARA DESENVOLVEDORES

O link para documentação, downloads e outros é: <http://developer.android.com/>

1.4 INSTALAÇÃO DO ANDROID

Primeiramente, é necessário baixar o Android Studio para o nosso sistema operacional. O instalador está disponível na seguinte URL:

<http://developer.android.com/sdk/index.html>

O download inclui o Android Studio, o kit de desenvolvimento de software do Android (SDK), a última versão da plataforma Android e a imagem do emulador correspondente já configurado e customizado para programar no Android.

Após o download é recomendado rodar o SDK Manager e fazer o update/download das versões das APIs Android que serão utilizadas.

DURANTE O CURSO

Note que, na Caelum, já fizemos o download do Android Studio previamente com os diversos "Installed Packages" necessários no curso. Isso porque o arquivo é muito grande e nosso tempo é precioso!

É bom também baixar o Google APIs para todos os levels, para poder usar outras APIs opcionais.

Editora Casa do Código com livros de uma forma diferente



Editoras tradicionais pouco ligam para ebooks e novas tecnologias. Não dominam tecnicamente o assunto para revisar os livros a fundo. Não têm anos de experiência em didáticas com cursos.

Conheça a **Casa do Código**, uma editora diferente, com curadoria da **Caelum** e obsessão por livros de qualidade a preços justos.

[Casa do Código, ebook com preço de ebook.](#)

1.5 DISPOSITIVOS ANDROID

No mercado já existem inúmeras opções de fabricantes e modelos de dispositivos com Android como tablets, TVs e relógios de pulso.

1.6 EMULADORES DE ANDROID

O próximo passo é testar o projeto criado para nos certificarmos que o ambiente foi instalado e configurado corretamente.

Como nem sempre há um dispositivo Android adequado para testar a aplicação, é possível utilizar um dispositivo virtual Android (AVD) com as especificações desejadas utilizando o emulador disponível no SDK.

O uso do emulador é recomendado para os casos onde seja necessário testar o aplicativo em diferentes tamanhos de tela e versões do Android. Apesar de servir bem a esse propósito, é importante ressaltar que nem todos os recursos da plataforma Android são emulados corretamente.

O Android Studio já disponibiliza uma série de dispositivos virtuais pré-configurados com base em dispositivos Android reais.

1.7 EXERCÍCIO: CRIANDO UM PROJETO

Objetivo

Crie um projeto chamado TwittelumApp, que tenha suporte a Kotlin e que a versão mínima seja a API 21.

Passo a passo

1. Abra o Android Studio. Na janela `Import Android Studio Settings From...`, escolha a opção `Do not import settings`.
2. No `Android Studio Setup Wizard`, aperte `Next`, selecione o tipo `Standard` de `setup`. No próximo passo ele deveria já encontrar a pasta `Android/Sdk` dentro da sua `home`. Se não tiver encontrado e sugerir de baixar muita coisa, chame o instrutor.
3. Clique no botão de criar um novo projeto.
4. Precisamos primeiro escolher qual é o tipo de `Activity` que queremos usar. Por hora escolheremos uma `Activity` em branco.

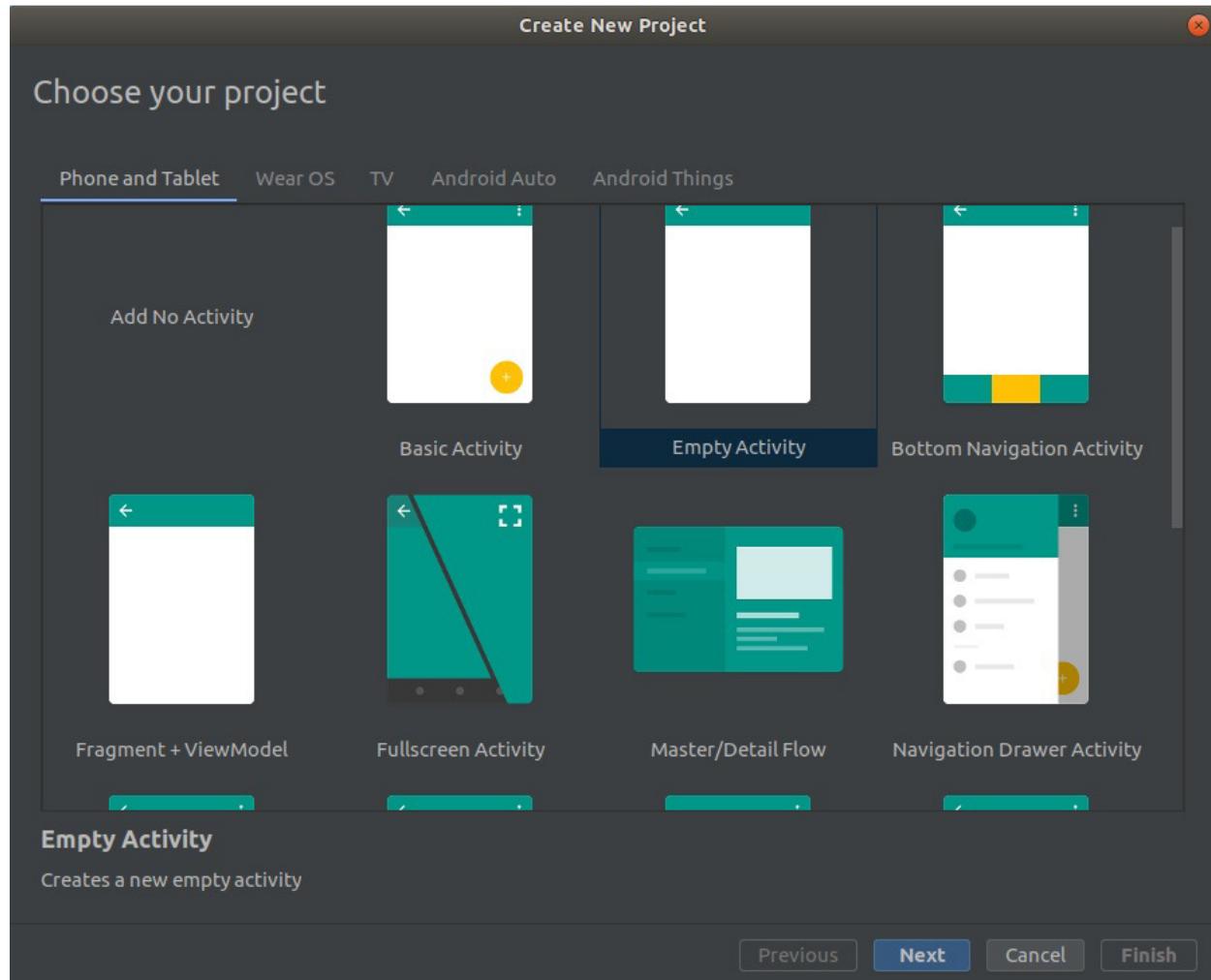


Figura 1.1: Escolhendo o tipo de Activity

5. Deixe o nome do projeto como `TwittelumApp`, troque o nome do pacote para `br.com.caelum.twittelumapp`, mantenha a linguagem de `Kotlin` e selecione a versão mínima da API para 21 (Android 5.0).

6 1.7 EXERCÍCIO: CRIANDO UM PROJETO

Apostila gerada especialmente para Willian Silva Moreira - moreiraws85@gmail.com

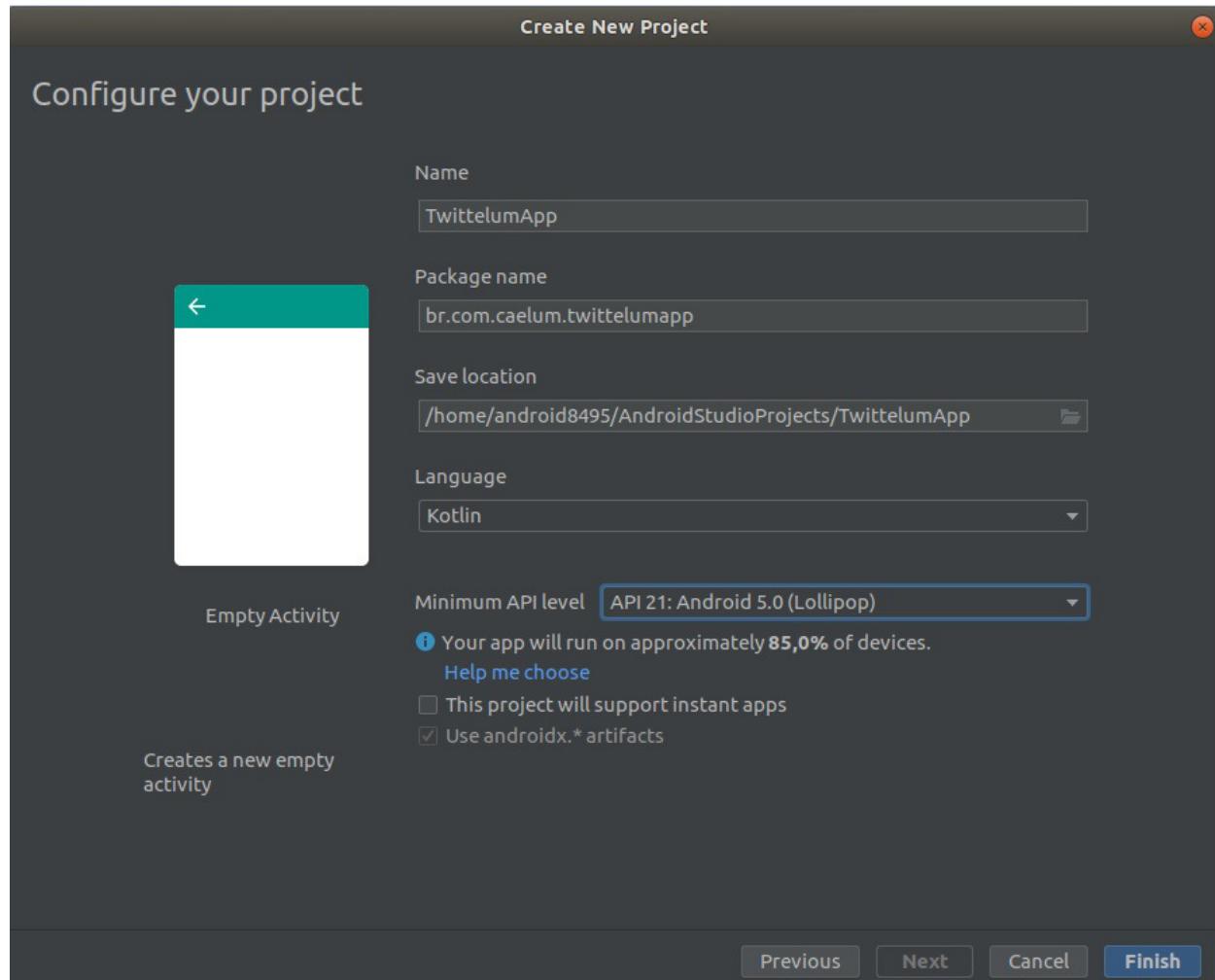


Figura 1.2: Configurando o nome do projeto

6. Rode o novo projeto em um emulador e veja que vai abrir um aplicativo no emulador apenas com o texto `HelloWorld!`.

Já conhece os cursos online Alura?



A Alura oferece centenas de **cursos online** em sua plataforma exclusiva de ensino que favorece o aprendizado com a **qualidade** reconhecida da Caelum. Você pode escolher um curso nas áreas de Programação, Front-end, Mobile, Design & UX, Infra e Business, com um plano que dá acesso a todos os cursos. Ex aluno da Caelum tem 15% de desconto neste link!

[Conheça os cursos online Alura.](#)

MANIPULANDO NOSSA PRIMEIRA ACTIVITY

2.1 O QUE SABEREI FAZER NO FIM DESSE CAPÍTULO ?

- Manipular layouts
- Entender o que são constraints e como usá-las
- Criar eventos personalizados
- Recuperar informações da tela
- Exibir Toasts na tela

Já conhece os cursos online Alura?



A Alura oferece centenas de **cursos online** em sua plataforma exclusiva de ensino que favorece o aprendizado com a **qualidade** reconhecida da Caelum. Você pode escolher um curso nas áreas de Programação, Front-end, Mobile, Design & UX, Infra e Business, com um plano que dá acesso a todos os cursos. Ex aluno da Caelum tem 15% de desconto neste link!

[Conheça os cursos online Alura.](#)

2.2 ENTENDENDO O QUE FOI GERADO

Ao longo do curso iremos entender com mais detalhes alguns arquivos gerados quando pedimos para o Android Studio criar um novo projeto para a gente. Se deixarmos nosso projeto com a visualização de *Android*, veremos as seguintes pastas dentro de *app*:

- **manifests:** vai armazenar todos os arquivos de manifesto do aplicativo, em geral o *AndroidManifest.xml*, que serve para passar algumas configurações do aplicativo ao dispositivo Android em que a aplicação irá ser executada.
- **java:** local em que ficará todo nosso código fonte do projeto, que poderá ser escrito tanto na

linguagem *Java* quanto na linguagem *Kotlin*.

- **generatedJava**: diretório em que ficará o código que é gerado no momento de compilação.
- **res**: pasta em que ficam armazenados os recursos estáticos utilizados na aplicação.

Há uma outra estrutura que aparece na visualização de *Android*, que é **Gradle Scripts**. Nela ficam, entre outras coisas, as configurações sobre as dependências e versões relacionadas ao projeto, além de informações como localização de algumas ferramentas.

Podemos pedir ao *Android Studio* para criarmos uma nova **Activity** tanto durante a criação do projeto quanto depois de termos um projeto criado e, quando pedimos para gerar uma activity em branco, o *Android Studio* já cria para a gente dois arquivos: *TweetActivity.kt* e *activity_tweet.xml*. Vamos entender um pouco a função de cada um deles.

Para o *Android*, cada tela é representada por uma **Activity**. Precisamos informar para cada Activity o que queremos que o usuário veja, ou seja, o *layout* da tela. Mas, além de visualizar, um usuário pode interagir com qualquer tela de um aplicativo. Precisamos então disponibilizar também as interações que o usuário poderá ter com nossa Activity, ou seja, a parte da *lógica*. Pensando nas telas de qualquer aplicação *mobile*, uma única tela possui inúmeras interações com o usuário. Se juntassemos todas as componentes que aparecem na tela (*layout*) com as inúmeras interações (*lógica*) que o usuário pode ter em um único arquivo, ficaria quase impossível para dar manutenção a este arquivo por causa do tamanho dele e da quantidade de coisas que ele estaria guardando. Pensando nisso, os desenvolvedores do *Android* resolveram separar essas duas funções em dois arquivos diferentes: a parte visual, do *layout*, ficará dentro da pasta *res/layout* e será definida em um arquivo escrito em XML; a parte da interação com o usuário, da *lógica*, ficará dentro de *java* e de algum pacote, e será definida por arquivos escritos em *Java* ou *Kotlin*.

2.3 LAYOUTS NO ANDROID

Para definirmos o que vai aparecer na tela, usamos a linguagem XML e, parecido com quando vamos estruturar o que vai aparecer no HTML, precisamos definir cada uma das componentes de tela através de *tags*. Precisamos informar ao *Android* cada uma das informações visuais que queremos mostrar ao usuário.

Componentes de tela

Ao longo do curso veremos algumas componentes necessárias para a construção de uma tela, mas existem inúmeras outras que podem ser encontradas nas documentações do *Android*. Cada componente é representada por uma *tag* com a sintaxe `<NomeDaComponente>` e é chamada de **View**. As primeiras que veremos são:

- **TextView**: usada para mostrar apenas um **texto** ao usuário.

- **EditText**: usada para disponibilizar um **campo de texto** ao usuário.

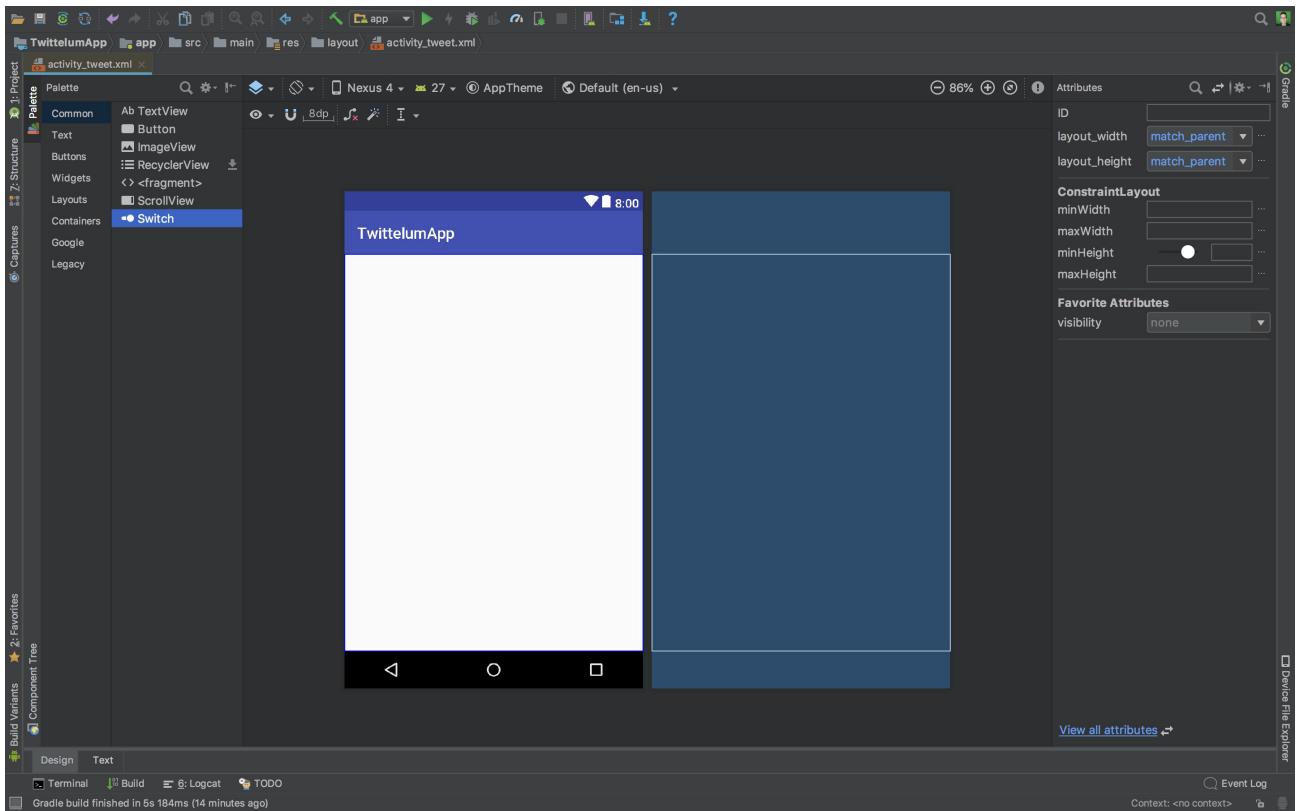
Para cada componente que utilizamos podemos definir diversas características, ou **atributos**. Por exemplo, sempre que colocamos algum elemento na tela, precisamos definir qual largura e qual altura queremos que esse elemento tenha. Fazemos isso utilizando os atributos `layout_width` e `layout_height`, respectivamente. Para esses atributos, é possível utilizar uma medida específica, como em pixels, mas não é recomendado. As opções relativas são mais frequentes, como o `wrap_content`, determinando que a componente terá o tamanho ocupado por seu conteúdo, e o `match_parent`, determinando que a componente terá o mesmo tamanho que a componente que a engloba, a componente "mãe". Como é impossível algum elemento aparecer na tela sem largura e altura definidos, estes dois atributos são os únicos **obrigatórios** para qualquer `View`. Outro atributo muito comum de definirmos para um elemento é o `id`, para identificarmos um elemento e posteriormente atrelar alguma lógica a ele. Há também características específicas para algumas componentes; por exemplo, a view `EditText` possui o atributo `hint`, para dar uma dica ao usuário de o que ele deve digitar no campo.

Uma `Activity` é capaz de receber apenas uma única `View` como conteúdo. Desse modo, se for necessário incluir mais de uma componente em uma tela, será necessário utilizar uma `View` que funcione como um *container*, onde seja possível adicionar quantas `Views` forem necessárias. No Android, tal *container* é denominado **ViewGroup**. Usamos componentes do tipo `ViewGroup` tanto para definir o *layout* principal da tela quanto para estruturar grupos de elementos da tela. Um exemplo do primeiro uso, para definir o *layout* principal da tela, seria o **ConstraintLayout**. Um `ViewGroup` também é uma `View` e portanto pode ser incluído em outros `ViewGroups`, conforme a imagem a seguir:

Uma das maneiras de se editar um *layout* é utilizando o **Editor visual** do Android Studio, e é bem indicada se estivermos utilizando o `ConstraintLayout`.

Usando o Editor Visual

Ao abrir um arquivo de *layout* na aba de *Design*, veremos a janela abaixo:



O centro da janela mostra duas imagens de pré-visualização: a da direita é apenas a estrutura dos elementos que estão configurados para essa tela; na da esquerda está incluído o tema gráfico do aplicativo como um todo, com cores, estilos, cabeçalhos e rodapés padrões para todo o projeto.

Usando a aba à esquerda, de `Palette` (ou paleta, em português), podemos escolher que tipo de elemento queremos adicionar à nossa tela e, assim que escolhermos um, ele já vai aparecer nas duas pré-visualizações da tela que ficam no centro da janela. Essa aba também possui uma lupa para buscar pelo nome da componente que queremos inserir.

Na lateral direita podemos definir os atributos da componente que está selecionada no momento.

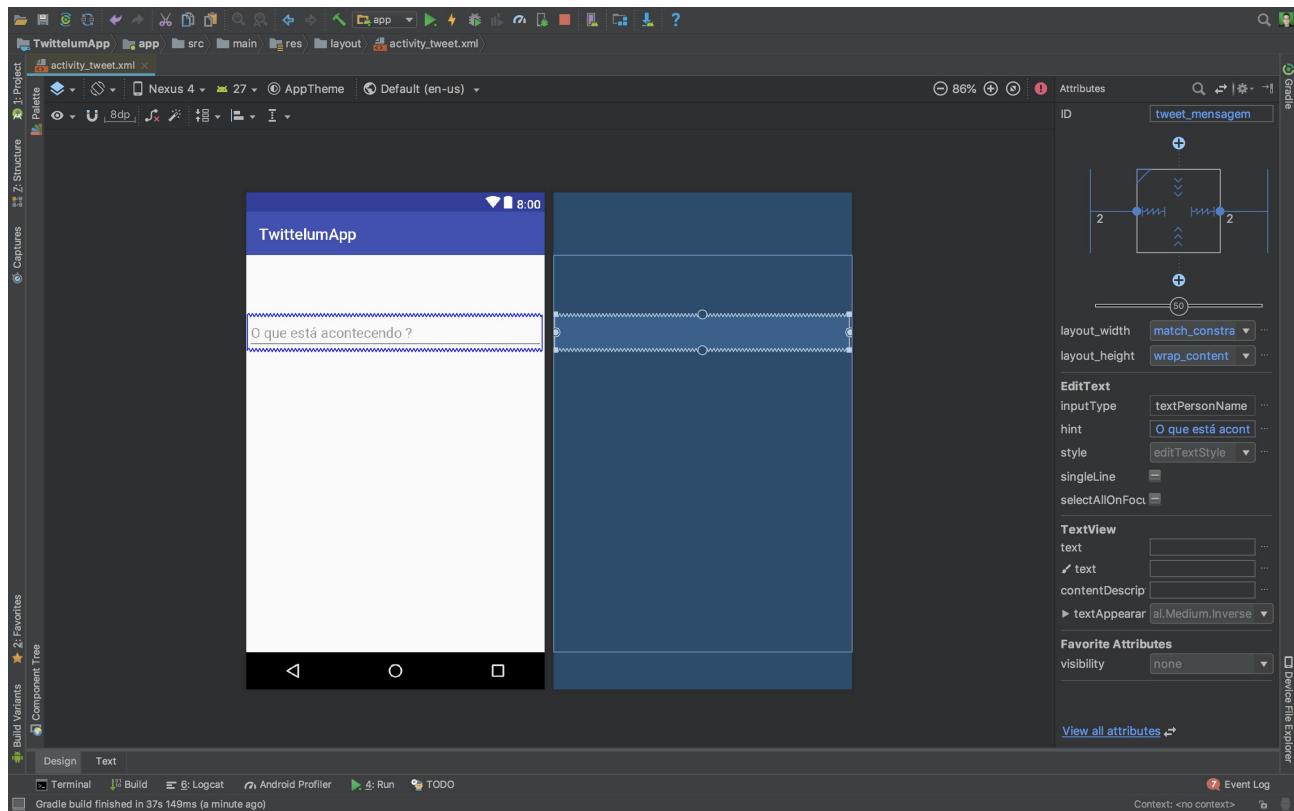
Trabalhando com o `ConstraintLayout`

Vimos que um dos tipos de *layout* que podem ser usados como *container* de uma tela é o `ConstraintLayout`. Ele estrutura as componentes internas **baseado em restrições**. Portanto, as opções de tamanho para largura e altura mais utilizados nesse *layout* são `wrap_content`, que já vimos, e `match_constraint`, que afirma que o tamanho da componente será definido apenas pelas restrições que ela possui. Para o `ConstraintLayout`, o *Android* também identifica como `match_constraint` se colocarmos o valor `0dp`, onde `dp` significa *density pixels*.

Se formos alterar pelo Editor Visual uma tela com o `ConstraintLayout`, veremos na aba dos atributos um quadrado com vários `+`. Através deles podemos definir as restrições que o elemento

selecionado terá. Enquanto estiver o  , significa que o elemento não tem restrição nenhuma para aquela margem, ou seja, não está preso a nenhuma outra componente nesse lado específico. Podemos definir uma restrição pelo quadrado à direita ou pelas pré-visualizações no centro da janela.

- Usando o quadrado, basta clicarmos no mais que ele vai virar uma caixinha para colocarmos um número. O número que colocarmos será processado com a medida de `dp` .
- Usando as pré-visualizações, primeiro clicamos no componente. Aparecerão 4 bolinhas ao redor do componente, 1 em cada lateral. Em seguida, basta clicarmos em alguma bolinha e arrastar até alguma borda de outro componente.



Além de utilizarmos as restrições para definir o tamanho da componente, também a utilizamos para determinar a posição em que o elemento ficará na tela. Se não definirmos nada, o elemento ficará colado com o início (pela esquerda e por cima) do elemento "pai". Se definirmos a restrição, o elemento terá a distância que definirmos para o elemento "pai", contada a partir da margem que especificamos.

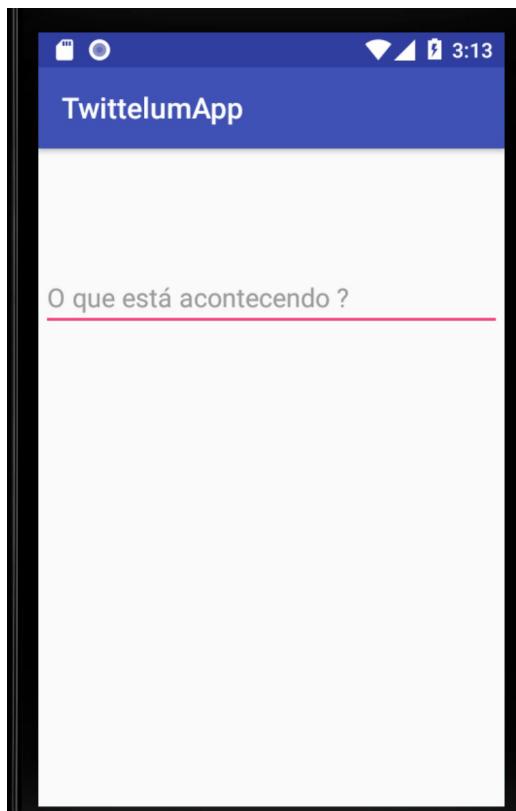
O grande diferencial do *Constraint Layout*, portanto, é definir a posição e tamanho das componentes de acordo com as restrições estabelecidas entre elas. Por exemplo, podemos dizer com as restrições que um botão deve ficar abaixo de um campo de texto e um label deve ficar acima desse campo de texto. Com isso, o `ConstraintLayout` consegue calcular as posições e tamanho desses elementos apenas respeitando as restrições estabelecidas.

2.4 EXERCÍCIO: MODELANDO O LAYOUT ATRÁVES DO EDITOR VISUAL

Objetivo

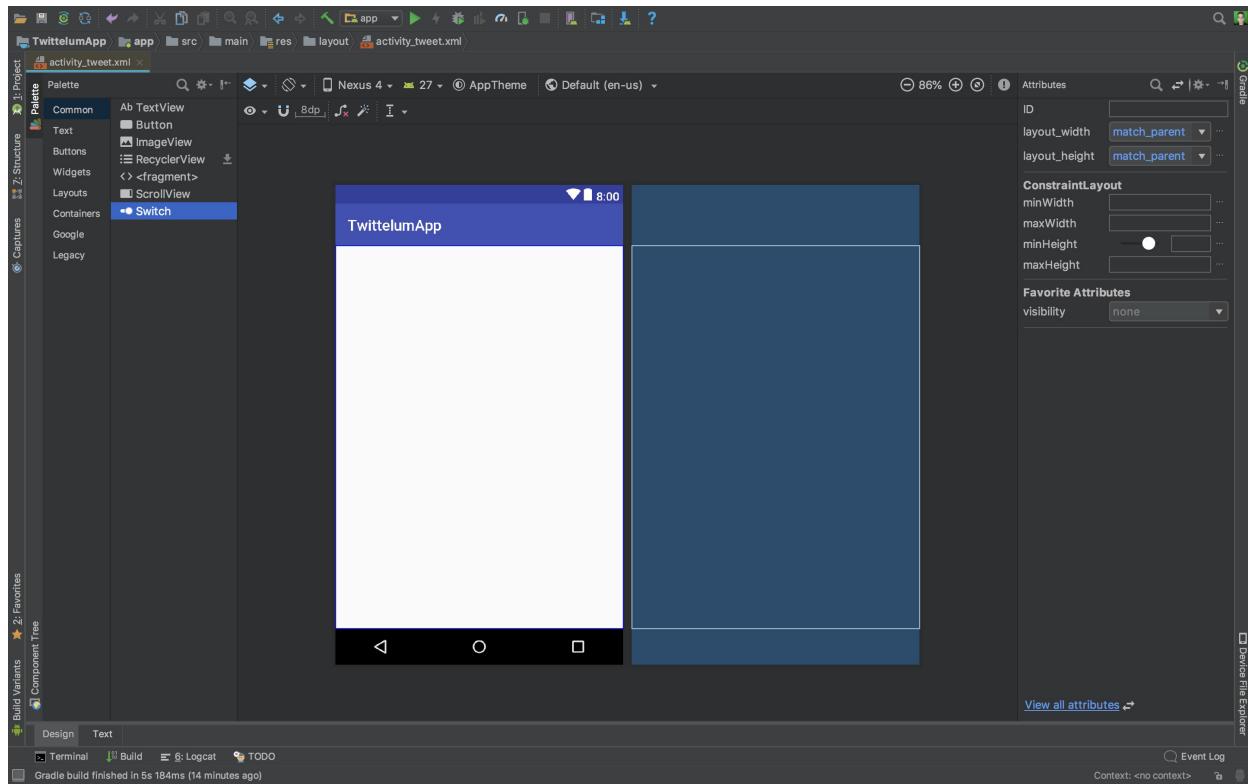
Exibir nossa Activity com um `EditText` contendo a dica: "O que está acontecendo ?".

A tela deve ficar similar a essa:

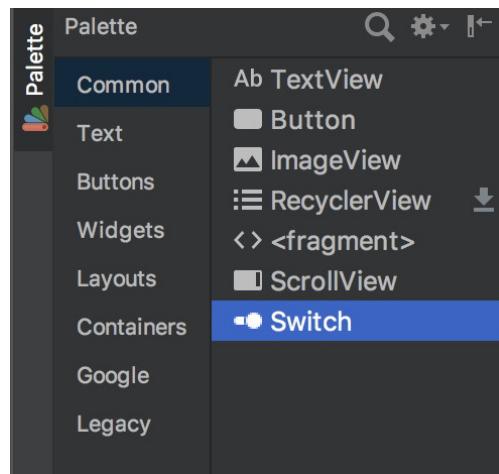


Passo a passo

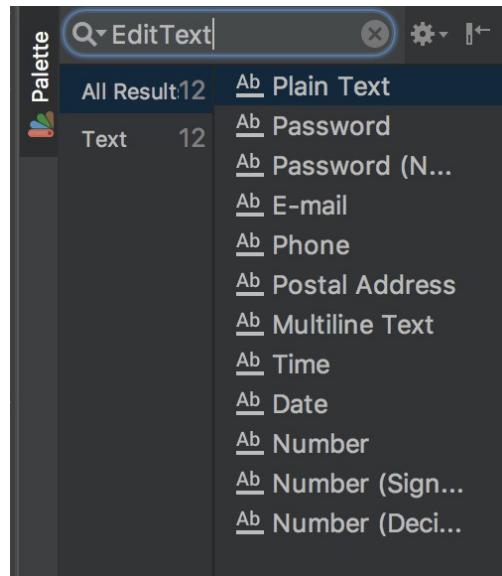
1. Ache o arquivo `activity_main.xml`, que está dentro de `res/layout/`, e o renomeie para `activity_tweet.xml` com o atalho `shift+F6`. Em seguida, abra o arquivo.
2. Clique com o botão direito no texto `hello world` que está sendo exibido e escolha a opção `Delete`. Outra opção é utilizar a tecla `delete` do teclado para excluir o texto. Seu layout deve estar próximo disso:



3. Encontre a paleta de componentes que fica na parte superior esquerda:

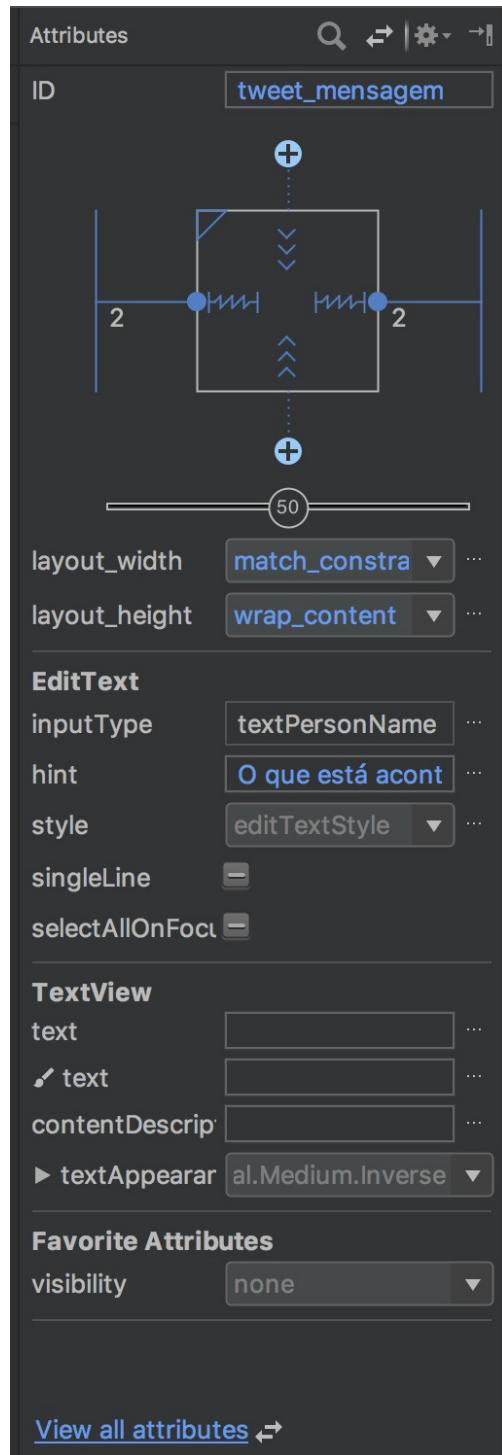


4. Clique no ícone de pesquisa (lupa) e procure pelo campo de entrada de texto `EditText` :

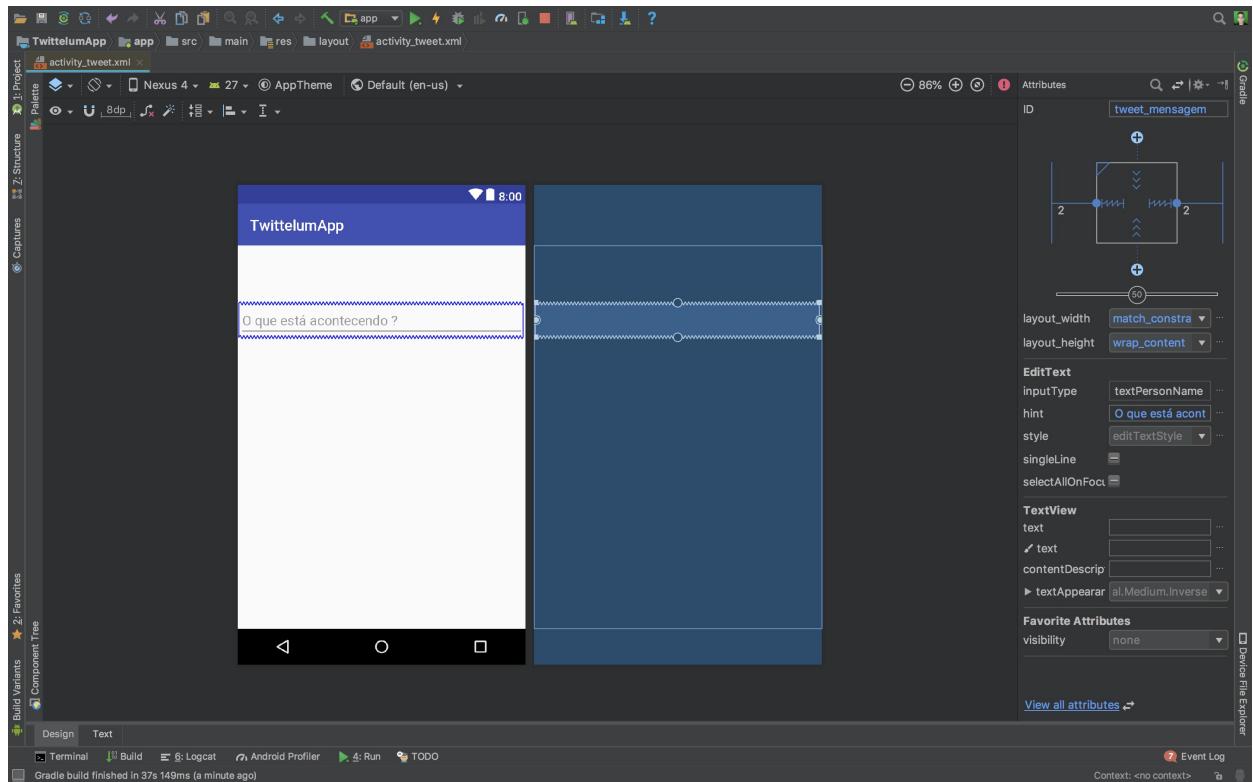


5. A primeira opção é `Plain Text`. Clique nela e arraste-a para a tela do seu aplicativo. Configure as propriedades dessa `View` na paleta que fica na direta no Android Studio conforme abaixo:

- mude o id para `tweet_mensagem`
- mude as margens da esquerda e da direita para 2
- altere a largura do componente para ser `match_constraint (0dp)`
- coloque no campo `hint` a mensagem da dica: "*O que está acontecendo ?*"
- apague o texto no campo `text`

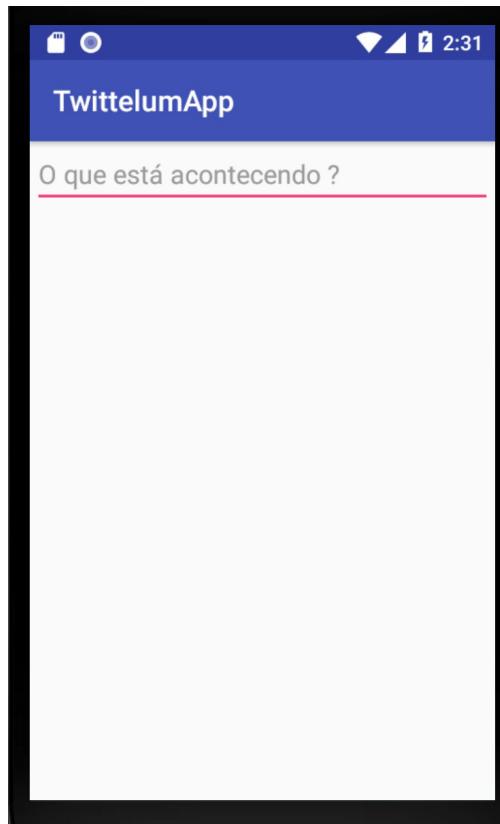


6. Arraste o `EditText` para próximo do meio da tela, deixando similar à imagem:

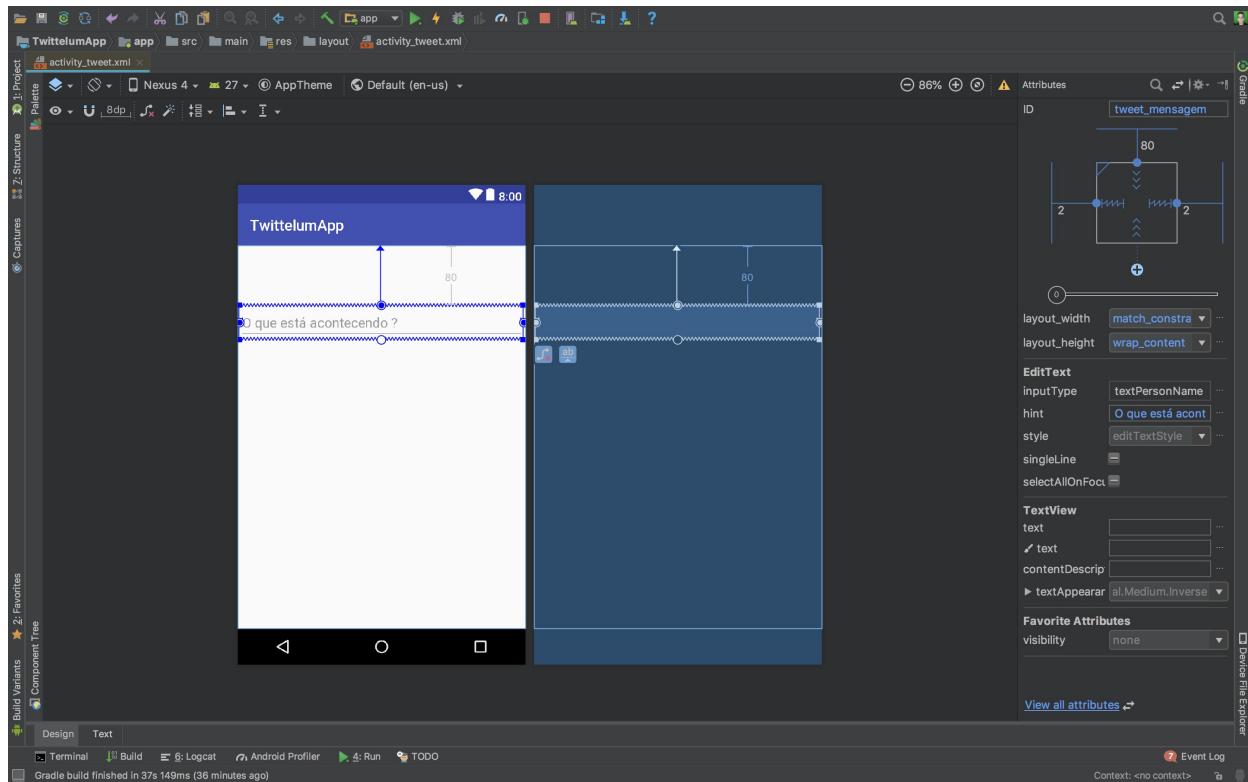


7. Rode o aplicativo no emulador.

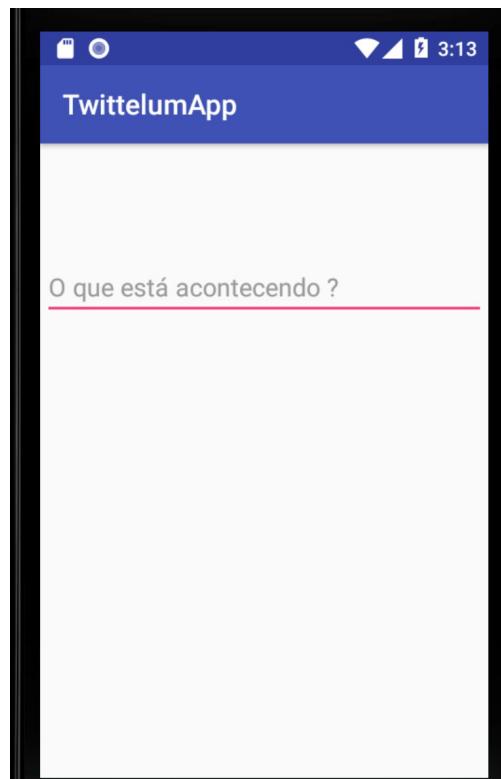
Provavelmente o resultado vai ser parecido com a imagem abaixo:



8. Ficou faltando um pequeno detalhe para chegarmos no que queríamos. Precisamos definir uma *constraint* do nosso componente para prendê-lo ao topo da tela. Para adicionar restrições em um componente, primeiro clicamos nele. Aparecerão 4 bolinhas ao redor do componente, 1 em cada lateral. Precisamos clicar na bolinha do topo do componente e arrastar até o topo da tela. Se com esse último passo a componente ficou colada no topo, basta arrastá-la de volta para a posição anterior, mas agora mantendo a restrição vertical.



9. Rode o aplicativo novamente e veremos um resultado parecido com esse:



Saber inglês é muito importante em TI



cotidianas. Além disso, todas as aulas possuem explicações gramaticais, para você entender completamente o que está aprendendo. Aprender inglês é fundamental para o profissional de tecnologia de sucesso!

[Pratique seu inglês na Alura Língua.](#)

Na **Alura Língua** você reforça e aprimora seu inglês! Usando a técnica *Spaced Repetitions* o aprendizado naturalmente **se adapta ao seu conhecimento**. Exercícios e vídeos interativos fazem com que você pratique em situações

2.5 MOSTRANDO BOTÕES NA TELA

Uma das formas de um usuário interagir com nosso aplicativo é através do clique. Assim como em um site, em aplicativos *mobile* é possível clicar em qualquer componente, mas o mais visual e intuitivo para o usuário é clicar em um **botão**. Para mostrarmos um botão no *layout*, há uma componente chamada **Button**. No entanto, geralmente há diversos botões em uma mesma tela, e o usuário precisa saber o que vai acontecer quando clicar em um botão específico. Ele costuma decidir o botão pelo que está escrito dentro dessa componente. Podemos mostrar texto dentro de uma componente através do atributo **text**. Usando o Editor Visual, basta procurar por esse atributo na lateral direita e colocar o texto que desejar.

Pensando ainda no visual, outra coisa que podemos fazer é centralizar o botão (ou qualquer outra componente), tanto na vertical, quanto na horizontal. O Editor Visual facilita nosso trabalho nesse momento, sendo possível apenas clicar com o botão direito e escolher como queremos centralizar a componente e outras restrições que quisermos fazer.

A partir do momento em que o usuário clica no botão, ele espera que uma ação aconteça. Precisamos atrelar ao botão uma lógica. Se vamos trabalhar com lógica, o mais recomendado é deixar de editar o arquivo de *layout* para editar o arquivo de *lógica*, o `TweetActivity.kt`. Este arquivo já tem algumas coisas escritas na linguagem Kotlin, mas vamos entendê-las mais à frente. O que precisamos saber nesse momento é que, se quisermos trabalhar com uma componente em uma `Activity`, podemos acessar a componente dentro do método `onCreate`, depois da chamada do método `setContentView`, código em que é atrelada a `Activity` com o *layout*. Para recuperar neste arquivo as características e

comportamentos de uma componente que definimos no arquivo de *layout*, *activity_tweet.xml* , usamos o método `findViewById` . No entanto, cada tipo específico de `view` possui características e comportamentos diferentes. Com Orientação a Objetos, cada tipo específico é uma classe diferente. Assim, podemos especificar pra esse método qual o tipo da `view` que estamos identificando pelo *generics*, como do Java. Por exemplo, se queremos pegar um botão da tela, fazemos `findViewById<Button>` .

Agora precisamos identificar com qual `view` queremos trabalhar. Uma das maneiras de identificar nomes é usando o formato de texto, mas isso aumenta as chances de erro, pois só percebemos em tempo de execução. Por isso, o Android gera em tempo de compilação uma classe chamada `R` , onde os atributos correspondem aos diferentes tipos de recursos (*layout*, *id*, *menu*, etc). Para cada recurso, são gerados atributos correspondentes ao nome do arquivo do recurso ou, para recursos do tipo *id*, são gerados atributos correspondentes aos ids das componentes. Por exemplo, se definimos no *activity_tweet.xml* o botão com id `criar_tweet` , para acessar o elemento com esse id, chamamos `R.id.criar_tweet` . A vantagem é que percebemos o erro em tempo de compilação e as IDEs nos fornecem auto-complete. O código até o momento ficaria assim:

```
class TweetActivity : AppCompatActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_tweet)

        findViewById<Button>(R.id.criar_tweet)
    }
}
```

Esse método nos fornece um objeto do tipo `View` com base nas características e componentes que a componente possui. Então, como queremos manipular esse botão, vamos armazená-lo em uma variável. Para criar variáveis cujo valor (referência para o objeto) não mudará usando Kotlin, usamos a seguinte sintaxe:

```
val nomeDaVariavel = valorImutavel
```

Mas nossa aplicação não sabe quando que o usuário vai decidir clicar. Pensando nisso, os desenvolvedores do Android usaram a mesma ideia das aplicações Web de ficar *ouvindo* as interações do usuário, por meio dos chamados *listeners*. Várias componentes possuem essa capacidade de ter listeners, e podemos definí-los por meio de métodos. No nosso caso queremos ouvir a ação do clique, então vamos usar o método `setOnClickListener` . Esse método espera receber uma instância de uma classe que implemente a interface `View.OnClickListener` . Podemos criar essa classe:

```
class BotaoListener : View.OnClickListener {
    override fun onClick(v: View?) {
        // lógica que queremos realizar no clique do botão
    }
}
```

No código acima vemos algumas novidades na sintaxe do Kotlin:

- o modificador de acesso padrão é o `public` ;
- para implementar uma interface, usamos " : ";
- na assinatura de qualquer método sobrescrito, é necessário usar a palavra `override` ;
- sempre que vamos criar um método, precisamos usar o termo `fun` ;
- para definirmos o tipo de um parâmetro, usamos a sintaxe `nomeDaVariavel: TipoDaVariavel` ;
- se quisermos informar que uma variável pode possuir o valor nulo, colocamos o caractere `?` colado com o tipo da variável, logo em seguida: `nomeDaVariavel: TipoDaVariavel? ;`

Veremos algumas dessas sintaxes com mais calma e profundidade mais à frente.

Com a classe `BotaoListener`, podemos adicionar uma instância dela ao botão com o seguinte código:

```
val botao = findViewById<Button>(R.id.criar_tweet)
botao.setOnClickListener( BotaoListener() )
```

Podemos ver pelo código acima que, para instanciar um objeto de uma classe com o Kotlin não é usado a palavra `new`. Apenas é preciso chamar o construtor da classe e pronto!

Porém, dessa forma, não temos acesso aos outros campos que estão na `Activity`. Há algumas maneiras de resolver isso. Uma delas é passar uma referência ao objeto da `Activity` como parâmetro no construtor da nossa classe, mas teríamos que criar um construtor desse tipo para todas as classes de *listeners*, ou poderíamos pedir no construtor apenas as informações que precisarmos, mas esse construtor pode acabar ficando bem grande. Uma outra abordagem é pensarmos que geralmente um *listener* é criado para ser utilizado em apenas um único lugar, então podemos criar a classe `BotaoListener` direto dentro da classe onde vamos utilizá-la:

```
class TweetActivity : AppCompatActivity() {
    class BotaoListener : View.OnClickListener {
        override fun onClick(v: View?) {
            // lógica que queremos realizar no clique do botão
        }
    }
}
```

Dessa forma conseguimos ter acesso aos campos da `TweetActivity` dentro da classe `BotaoListener`. Mas acabamos perdendo um tempo pensando no nome da classe `BotaoListener`, deixamos ela solta dentro de outra classe e só a usamos em um único lugar no projeto todo. Os desenvolvedores tiveram a ideia então de criar a classe anônima, em que, quando quisermos usar a instância dessa classe, só precisamos dizer quem estamos implementando que internamente o compilador da linguagem faz esse trabalho de criar uma classe que implementa a interface e instanciá-la para a gente. O código de classe anônima em Kotlin fica assim:

```
botao.setOnClickListener(View.OnClickListener {
    // lógica que queremos realizar no clique do botão
})
```

```
})
```

O código fica mais curto, mas pode atrapalhar a legibilidade e tornar a classe muito grande e com responsabilidades demais. Caso conheça pouco de classes anônimas, recomendamos o post no blog da Caelum a respeito delas: <http://blog.caelum.com.br/classes-aninhadas-o-que-sao-e-quando-usar/> .

Logando informações

Em vários momentos durante o desenvolvimento queremos ver se a execução da nossa aplicação passou por determinado ponto ou queremos apenas verificar alguns valores para conferir se está tudo funcionando conforme o esperado. Uma das formas é checar os valores na hora executando a aplicação em modo de *debug*, verificando passo a passo. A outra maneira é apenas registrar as informações que queremos conferir. Para o segundo, o Android fornece a classe `Log`. Ela vai escrever a mensagem que pedirmos no *console* do *Android Studio*, o **LogCat**. A vantagem é que podemos especificar o tipo do registro, chamando o método específico pra cada tipo: `i` para `info` , `d` para `debug` , `e` para `error` , etc.

```
Log.i("TWEET", "sucesso ao criar um tweet")
```

Então, dependendo do tipo de execução ou do modo de visualização que vemos no console, veremos apenas as mensagens para aquele tipo. Outra coisa que definimos quando chamamos um método da classe `Log` é uma *tag*, um nome para facilitar nossa busca por aquela mensagem no console.

Pegando informações de uma componente

Em muitas situações em que o usuário clica em um botão, queremos pegar alguma informação que ele digitou em algum campo de texto. A primeira coisa que precisamos fazer, como já falamos, é pegar um objeto com as informações da componente na tela:

```
val campoDeTexto = findViewById<EditText>(R.id.campo_mensagem)
```

Em seguida, podemos pedir a esse objeto para pegar alguns valores dele, como o texto que o usuário digitou, através de métodos *getters*. No entanto, no Kotlin, pegar valores de objetos funciona um pouco diferente. Ao invés de chamarmos os métodos *getters* diretamente, acessamos as **propriedades** do objeto e o compilador transforma esse acesso em uma chamada ao método do tipo *get*. Por exemplo, se chamarmos o código:

```
campoDoTexto.text
```

Internamente está sendo processado o código:

```
campoDoTexto.getText()
```

Mas o próprio *Android Studio* e o *Kotlin* sugerem fortemente usar as *propriedades*.

Propriedades

Não tenha medo, mais à frente no curso, quando precisarmos criá-las, veremos como elas funcionam e como devemos usá-las.

Mostrando um aviso temporário ao usuário

Em diversos momentos o usuário gostaria de sentir que a interação dele com a aplicação está ocorrendo como o esperado. Mas, às vezes, a ação que está acontecendo não está visualmente de fácil acesso ao usuário. Podemos pensar em alguns exemplos: quando algum processamento como buscar dados de um servidor é demorado, quando alguma aplicação que está em segundo plano recebe alguma informação (wi-fi), ou quando dá algum erro em qualquer processamento e o comportamento é continuar exatamente na mesma tela que já está visível ao usuário. Nesses momentos, seria legal ao menos mostrar uma mensagem ao usuário para ele saber o que está acontecendo, ter algum *feedback* sobre o resultado da ação que aconteceu.

Podemos fazer isso usando a classe `Toast`, que permite justamente que mostremos um aviso temporário na tela do dispositivo, independente de qual aplicação está em primeiro plano naquele momento. Essa classe possui um método estático chamado `makeText`, que serve para processar a criação de um aviso e recebe três valores:

- a informação de quem está pedindo para esse aviso ser mostrado, geralmente a própria activity em que o código está escrito é que está pedindo, então usamos o `this`;
- o texto que queremos mostrar ao usuário;
- e a duração em que o `toast` irá aparecer na tela, que pode ser curta ou longa. Com o `toast` criado, precisamos mostrá-lo ao usuário, chamando o método `show`:

```
Toast.makeText(this, mensagemDoTweet, Toast.LENGTH_LONG).show()
```

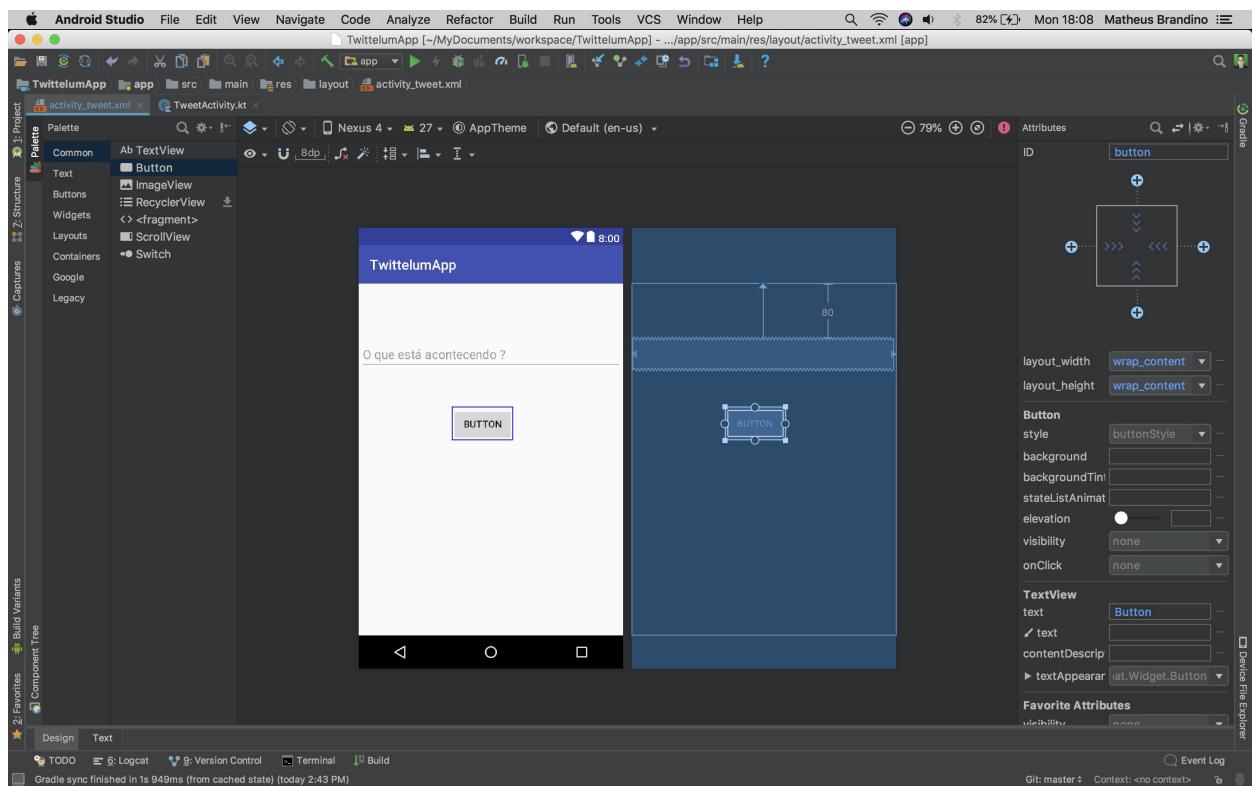
2.6 EXERCÍCIO: FAZENDO A PRIMEIRA INTERAÇÃO COM O SISTEMA

Objetivo

Fazer o aplicativo ter um botão com o nome `PUBLICAR TWEET` que, ao ser clicado, pegue o conteúdo do `EditText` que está acima do botão e exiba o conteúdo em um `Toast`.

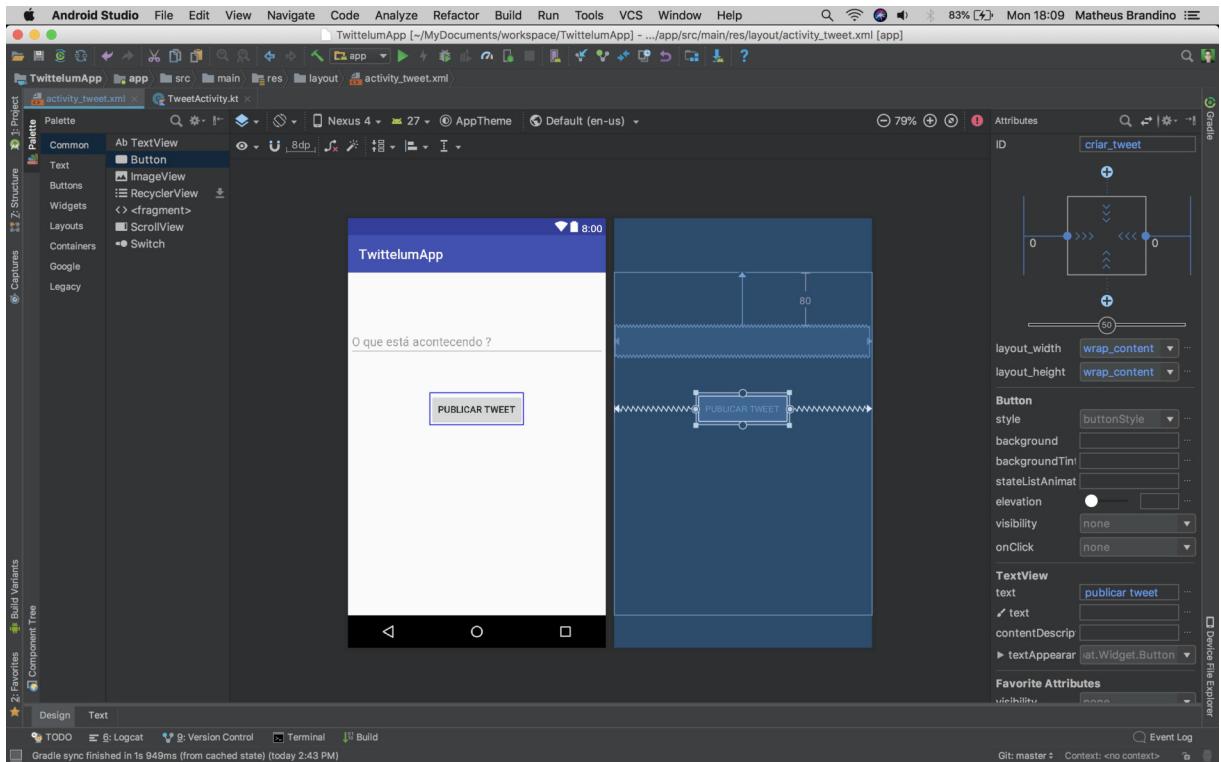
Passo a passo

1. Novamente pelo modo visual vamos adicionar um novo componente na tela. Procure por `Button` e o coloque logo abaixo do nosso `EditText` :

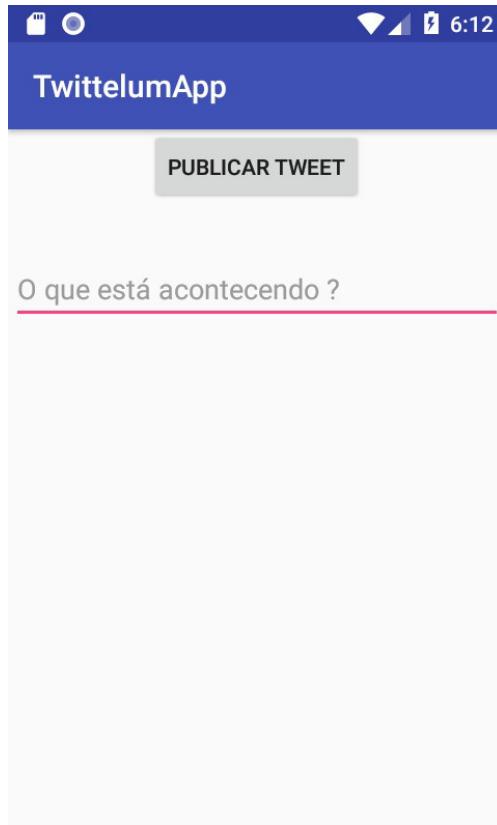


2. Agora vamos deixar nosso botão um pouco melhor para nosso usuário, conforme imagem abaixo:

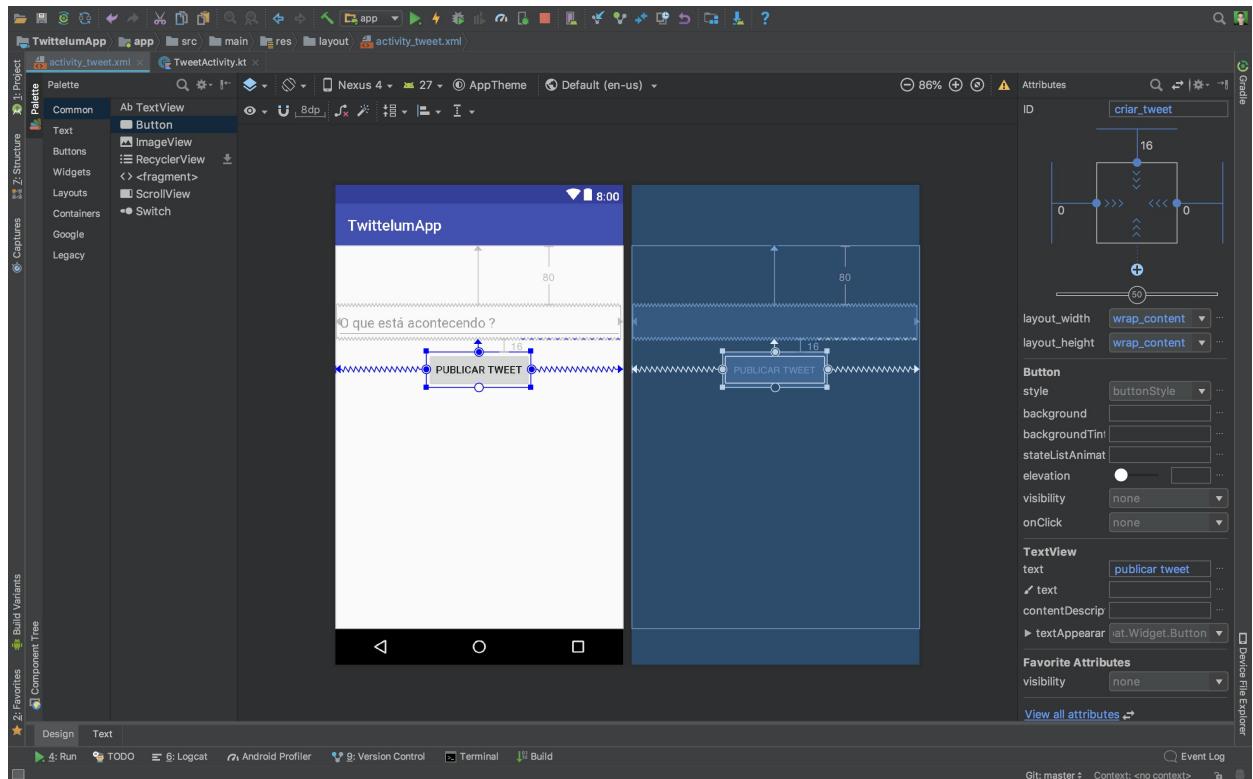
- mude o texto para *Publicar tweet*
- mude o campo de id para `criar_tweet`
- deixe o botão no centro, clicando com o botão direito e escolhendo a opção *Center* e, depois, *Horizontally*



3. Rode o aplicativo agora para vermos como nossa tela está. Veja que o resultado deve ser algo muito próximo disso:



4. Parece que esquecemos de atribuir a *constraint* que nosso botão precisa ter para ficar na posição desejada. Adicione a *constraint* do **topo** do **Button** com o **fundo** do **EditText**:



5. Rode agora novamente o aplicativo e veja se o resultado está satisfatório:



6. Agora precisamos colocar algum comportamento nesse nosso botão. Para isso, vamos abrir a classe `MainActivity`. Renomeie-a para `TweetActivity` com o atalho `shift+F6`. A primeira coisa que precisamos fazer nessa classe é recuperar o botão utilizando o método `findViewById`:

```
override fun onCreate(savedInstanceState: Bundle?) {  
    super.onCreate(savedInstanceState)  
    setContentView(R.layout.activity_tweet)  
  
    val botao = findViewById<Button>(R.id.criar_tweet)  
}
```

Sempre que precisarmos importar uma classe, basta usarmos o atalho `alt+Enter`.

7. Como já recuperamos nosso botão, basta definirmos o comportamento que ele terá. Faremos isso usando o método `setOnItemClickListener` que **toda** View possui:

```
// restante do código  
  
botao.setOnClickListener(View.OnClickListener { publicaTweet() })
```

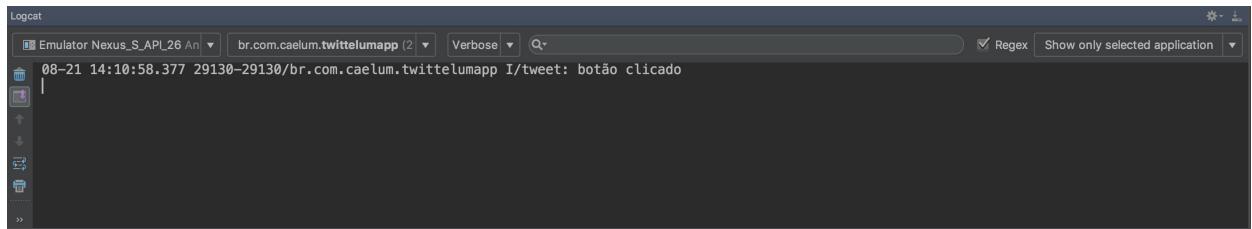
Nesse momento o código não vai compilar.

8. Vamos criar o método `publicaTweet`:

```
private fun publicaTweet() {  
    Log.i("tweet", "botão clicado")
```

```
}
```

9. Rode o aplicativo e clique no botão. Veja no **Logcat**(console) se nossa mensagem foi exibida.



Para facilitar a busca pela mensagem, selecione a opção `Show only selected application` e procure no campo de busca pela *tag* que passamos pro método `i` do `Log`.

10. Agora queremos exibir o conteúdo que está dentro do `EditText` assim que o `Button` for clicado:

- Primeiro precisaremos recuperar nosso `EditText`:

```
private fun publicaTweet() {  
    val campoDeMensagemDoTweet = findViewById<EditText>(R.id(tweet_mensagem)  
}
```

- Agora precisamos recuperar o conteúdo desse campo. Faremos isso por meio da **propriedade** `text`.

```
private fun publicaTweet() {  
    val campoDeMensagemDoTweet = findViewById<EditText>(R.id(tweet_mensagem)  
  
    val mensagemDoTweet : String = campoDeMensagemDoTweet.text.toString()  
}
```

- Por fim, precisamos criar e mostrar o `Toast` na tela:

```
private fun publicaTweet() {  
    val campoDeMensagemDoTweet = findViewById<EditText>(R.id(tweet_mensagem)  
  
    val mensagemDoTweet : String = campoDeMensagemDoTweet.text.toString()  
  
    Toast.makeText(this, mensagemDoTweet, Toast.LENGTH_LONG).show()  
}
```

11. Rode o aplicativo e veja se o `Toast` foi exibido:



Aprender Android com Kotlin é
muito legal

PUBLICAR TWEET

CRIANDO UM FLUXO NO APLICATIVO

3.1 O QUE SABEREI FAZER NO FIM DESSE CAPÍTULO ?

- Manipular layouts através do xml
- Entender o AndroidManifest.xml
- Trabalhar com listas
- Adicionar uma biblioteca externa ao projeto
- Trocar de Activity
- Manipular menus

Editora Casa do Código com livros de uma forma diferente



Editoras tradicionais pouco ligam para ebooks e novas tecnologias. Não dominam tecnicamente o assunto para revisar os livros a fundo. Não têm anos de experiência em didáticas com cursos.

Conheça a **Casa do Código**, uma editora diferente, com curadoria da **Caelum** e obsessão por livros de qualidade a preços justos.

[Casa do Código, ebook com preço de ebook.](#)

3.2 CRIANDO UMA ACTIVITY MANUALMENTE

Agora que temos a tela de adicionar um *tweet*, vamos permitir que o usuário veja quais *tweets* já foram escritos com uma tela de listagem de *tweets*. Para isso, precisamos criar uma nova `Activity`. A que temos até o momento foi obtida junto com a criação do projeto, mas o projeto já está criado, não podemos seguir a mesma abordagem. Se observarmos nossa `Activity`, vemos que ela é uma classe Kotlin. Vamos criar então uma nova classe Kotlin:

```
class ListaActivity {  
}
```

Mesmo sem escrevermos o modificador de acesso `public`, essa classe já é pública, pois este é o modificador de acesso padrão no `Kotlin` para classes.

Só que, para essa classe possuir todos os comportamentos que uma `Activity` deve ter, a `ListaActivity` precisa **herdar** de `Activity`. Para herdar de uma classe no Java, a sintaxe é:

```
public class ListaActivity extends Activity {  
}
```

Com o `Kotlin`, não precisamos usar o termo `extends`. A sintaxe ficará assim:

```
class ListaActivity : Activity() {  
}
```

A classe `Activity` possui todos os comportamentos específicos para o funcionamento de uma tela na versão definida pelo projeto. Porém, a maioria dos aplicativos que usamos funciona para as versões anteriores também, mesmo que às vezes mostre uma diferença no *layout*. Essa diferença acontece porque, mesmo utilizando componentes mais novas, em versões anteriores a essas componentes, o aplicativo adapta seu *layout* para as componentes existentes na versão mais antiga. Para isso ser possível, o Android disponibiliza uma outra classe, `AppCompatActivity`, que se preocupa em adaptar os *layouts* para usarem apenas as componentes que existem naquela versão. Vamos deixar nosso projeto funcional para as versões anteriores:

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

Agora precisamos dizer que componentes essa `Activity` terá. Mas já vimos que para definir as componentes de uma tela usamos um arquivo específico de *layout*, com o nome `activity_lista.xml`. Esse arquivo deve ficar dentro da pasta de recursos da aplicação, `res`, e dentro da pasta específica para *layouts*, `layout`. Já vimos como editar o arquivo de *layout*, pelo Editor Visual; agora veremos como editar diretamente pelo código. Quando pedimos para criar um arquivo de *layout* com o `Android Studio`, o arquivo fica aproximadamente assim:

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

Vamos entender o que foi gerado. A primeira linha mostra apenas que o conteúdo dentro deste arquivo está usando a linguagem XML e está baseado na tabela de caracteres UTF-8. A linha seguinte já está definindo a componente que será o *container* desse arquivo. Por padrão, a IDE usa a `ConstraintLayout`, mas podemos alterar durante a criação do arquivo, nas janelas do passo a passo,

ou depois do arquivo criado, por qualquer uma das maneiras de edição de arquivo. Vale ressaltar que essa componente veio da biblioteca que fornece suporte às versões anteriores do Android, por isso seu nome na *tag* vem com o nome do pacote, para a importação correta. Dentro da abertura da *tag container* de um arquivo XML , temos que dizer de onde virão os atributos para a IDE nos ajudar com a compilação e os auto-completes. Fazemos isso com o atributo xmlns:android . Mas não se preocupe, a própria IDE já cria esse atributo para a gente. Por fim, como já vimos que toda componente tem que ter largura e altura, a IDE já adiciona à *tag container* os dois atributos que definem essas informações: android:layout_width e android:layout_height , respectivamente. Para a *tag container*, esses dois valores já recebem o valor match_parent para dizer que a tela que estamos criando terá o mesmo tamanho que a tela do dispositivo.

3.3 MOSTRANDO UMA LISTA NA TELA

Para mostrarmos uma lista na tela, há uma componente específica para isso, a ListView . Além dos atributos que já vimos que são obrigatórios para toda componente, android:layout_width e android:layout_height , a ListView precisará também de um id para ser recuperada na classe. Para definirmos um novo id pelo XML , usamos a sintaxe @+id/ , seguido do nome que queremos usar para identificar a componente:

```
<ListView android:id="@+id/lista_tweets">
</ListView>
```

Também precisamos definir as restrições que ela deverá ter dentro do ConstraintLayout . Já vimos como definir as restrições pelo Editor Visual; agora veremos como é possível definí-las direto pelo XML . Para cada borda da componente, há alguns atributos que podemos usar para definir sua restrição. Por exemplo, para limitar a borda de baixo, podemos usar app:layout_constraintBottom_toBottomOf ou app:layout_constraintBottom_toTopOf , para definir o limite como sendo a borda de baixo de alguma componente ou a borda de cima de alguma componente, respectivamente. Passamos a esses atributos qual a componente que vai limitar a ListView . No nosso caso, queremos que a componente "mãe" limite a ListView , então podemos usar o termo parent : app:layout_constraintBottom_toBottomOf="parent" . Perceba que pra estes atributos o prefixo, chamado de namespace, é app em vez de android . Isso acontece para atributos que são customizados, que não são atributos do Android SDK. Para que esse namespace seja reconhecido neste arquivo, é necessário importá-lo: xmlns:app="http://schemas.android.com/apk/res-auto" , sempre na *tag container* do arquivo. Mas não se preocupe em guardar esse nome; a própria IDE já adiciona essa linha automaticamente quando escrevemos qualquer atributo com esse prefixo.

3.4 ADICIONANDO A TELA À ACTIVITY

Agora que temos nosso *layout* com a lista, precisamos pedir para a *Activity* que estruturamos mostrar esta tela ao ser criada. Toda *Activity* tem o método `onCreate`, que define que lógicas serão executadas no momento de sua criação. É justamente nesse método que definimos qual tela será mostrada enquanto esta *Activity* estiver em primeiro plano, por meio do método `setContentView`. Portanto, sobrescreveremos o método `onCreate`:

```
class ListaActivity : AppCompatActivity() {  
    override fun onCreate(savedInstanceState: Bundle?) {  
    }  
}
```

Vale lembrar que sempre que sobrescrevermos um método em Kotlin, precisamos da palavra `override`.

Da mesma forma que fizemos para recuperar uma *View* pelo `id`, usamos a classe `R`, que possui o atributo `layout`, que por sua vez possui um atributo para cada `XML` de *layout* que criamos, com o nome do arquivo `XML`. Assim, para termos acesso ao arquivo `activity_lista.xml`, fazemos `R.layout.activity_lista`. No entanto, antes de definir que componentes aparecerão em uma *Activity*, precisamos pedir para todos os comportamentos padrões durante a criação de qualquer *Activity*, como processar o tema e o *template* da tela, se mantenham. Para isso, precisamos dentro do nosso `onCreate`, chamar a implementação do `onCreate` da nossa "mãe":

```
class ListaActivity : AppCompatActivity() {  
    override fun onCreate(savedInstanceState: Bundle?) {  
        super.onCreate(savedInstanceState)  
        setContentView(R.layout.activity_lista)  
    }  
}
```

Já conhece os cursos online Alura?



A Alura oferece centenas de **cursos online** em sua plataforma exclusiva de ensino que favorece o aprendizado com a **qualidade** reconhecida da Caelum. Você pode escolher um curso nas áreas de Programação, Front-end, Mobile, Design & UX, Infra e Business, com um plano que dá acesso a todos os cursos. Ex aluno da Caelum tem 15% de desconto neste link!

[Conheça os cursos online Alura.](#)

3.5 MOSTRANDO UMA LISTA DE TWEETS NA TELA

Se executarmos nossa aplicação neste momento, considerando que a tela de lista é a única na aplicação, veremos apenas uma tela vazia! Isso porque, apesar de termos uma componente de lista na tela, ela não possui nenhum item. Precisamos recuperar a componente do *layout* e preenchê-la com tweets.

Importação das Views com Kotlin

Já vimos que podemos usar o método `findViewById` para recuperar uma `View`. Por exemplo, se nossa `ListView` possui o `id lista_tweets`, podemos recuperar essa componente da seguinte forma:

```
val listView = findViewById<ListView>(R.id.lista_tweets)
```

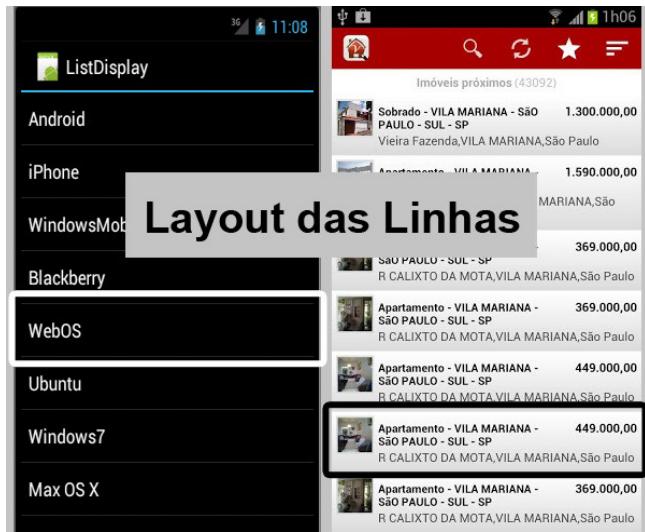
Mas chamar esse método para cada componente que queremos recuperar em um formulário extenso, por exemplo, começa a ficar bem cansativo e a deixar nossa classe com muito código estrutural. Pensando nisso, o Kotlin nos permite importar diretamente uma `view` definida dentro de um arquivo XML de *layout* usando seu `id` como variável:

```
val listView = lista_tweets
```

Passando os tweets para a ListView

Agora que temos acesso à `ListView`, precisamos ter uma lista de tweets para popular a componente. Podemos pegar essa lista de uma tela de formulário anterior, de algum local interno em que ela está armazenada, de um servidor, ou podemos definir pelo próprio código. Não vamos nos ater a como pegar a lista agora pois vamos primeiro focar em como se preenche uma `ListView`.

Uma vez que temos uma lista de tweets, precisamos popular a `ListView`. No entanto, a tela do Android só aceita objetos do tipo `View`, e, ao trabalhar com tweets no código Kotlin, teremos uma lista de objetos de um tipo que represente um tweet, ou pelo menos do tipo `String`. Nesse primeiro momento, iremos trabalhar com uma lista mais simples, de elementos `String`. Para conseguir mostrar essa lista na `ListView`, vamos precisar descrever o processo de como converter uma `String` em uma `View`. Além disso, temos que descrever também como será a aparência de cada item da lista. Abaixo podemos observar dois exemplos de uso de `ListView`, um deles mais simples, outro mais complexo:



Para atingir nosso objetivo e criar uma tela com um `ListView` populado vamos utilizar um `Adapter`, uma classe do `Android` especializada em *adaptar* para a tela um objeto qualquer com a criação de um objeto do tipo `View`. Um dos adapters já disponíveis para uso é o `ArrayAdapter`, que é capaz de converter listas ou arrays em listas de views que contém um `TextView`. Em seu construtor o `ArrayAdapter` nos pede 3 argumentos:

- um ***Context***, que é uma interface do `Android` para conseguir informações de qualquer ambiente da aplicação, por exemplo, uma `Activity` implementa `Context`; usaremos o `this` pois o `Adapter` precisará obter algumas informações da `Activity` que está criando esse adapter;
- o ***id*** do layout para as linhas, que pode ser um *layout* customizado nosso ou usar um *layout* de item já criado pelo `Android`, acessado por `android.R.layout.simple_list_item_1`, que possui apenas um `TextView`;
- a lista ou array de objetos que queremos mostrar no `ListView`; aqui vamos usar uma variável de nome `tweets`, do tipo `List<String>`, que guardará uma lista de *tweets*;

Definimos também qual o tipo do objeto que está na lista ou no array. A instanciação do `ArrayAdapter` fica algo parecido com:

```
val adapter = ArrayAdapter<String>(this, android.R.layout.simple_list_item_1, tweets)
```

Para que a `ListView` possa buscar seus elementos usando esse adapter que acabamos de criar, precisamos disponibilizar esse adapter à `ListView` atribuindo-o à propriedade `adapter` da `ListView`:

```
lista_tweets.adapter = adapter
```

Internamente o `Kotlin` vai chamar o método `setter` para popular essa propriedade.

Com isso, a `ListaActivity` está pronta para mostrar a lista de tweets que ela tiver disponível. Mas apenas criamos uma `Activity` nova na aplicação. Agora que temos duas `Activity`s, qual será a primeira a ser executada quando o aplicativo for inicializado?

3.6 PORTA DE ENTRADA DA APLICAÇÃO

Nossa aplicação vai continuar mostrando a primeira `Activity` que criamos, pois ela está configurada para ser a porta de entrada da nossa aplicação e não mudamos esta configuração em nenhum momento.

Porém, na maioria dos aplicativos, a primeira tela que aparece, sem contar a tela de login, se tiver, é sempre uma lista. Podemos pensar em vários exemplos: Facebook, com a lista de publicações, WhatsApp, com a lista de mensagens, Instagram, com a lista de publicações, agenda de contatos, com a lista de contatos. Faremos o mesmo com nossa aplicação: deixaremos a `ListaActivity` como sendo a primeira a ser executada. Precisamos passar essa configuração ao `Android` para ele saber por qual `Activity` iniciar nossa aplicação. Sempre que quisermos passar uma configuração ao `Android`, usamos o arquivo `AndroidManifest.xml`.

O arquivo `AndroidManifest.xml`

A primeira configuração que precisamos lembrar sempre é que toda `Activity` **precisa** estar registrada nesse arquivo para ser reconhecida pelo `Android`. Então, o primeiro passo a fazermos é registrar a `ListaActivity`. Como a `Activity` faz parte da nossa aplicação, ela fica dentro da tag `application`:

```
<manifest ...>
    <application ...>
        <activity android:name=".ListaActivity"/>
    </application>
</manifest>
```

Para saber qual é a `Activity` que deve ser iniciada, o `Android` procura no arquivo `AndroidManifest.xml` pela tag `activity` que possua a tag `intent-filter` com ação `MAIN` (principal) e categoria `LAUNCHER` (inicializadora). Se olharmos nosso `AndroidManifest.xml` nesse momento, a `TweetActivity` possui em seu registro a tag `intent-filter` com essas informações:

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

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

<activity android:name=".ListaActivity" />
```

...

Como esses atributos indicam que `Activity` será a porta de entrada da nossa aplicação, precisamos deixar essas informações apenas na `tag activity` da `ListaActivity`, pois queremos que apenas a `ListaActivity` seja a inicializadora e principal da nossa aplicação. Não esqueça de tirar esses atributos da `tag activity` da `TweetActivity`:

```
<activity android:name=".TweetActivity" />

<activity android:name=".ListaActivity">
 <intent-filter>
 <action android:name="android.intent.action.MAIN" />

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

## 3.7 EXERCÍCIO: EXIBINDO UMA LISTA NO ANDROID

### Objetivo

Criar uma **nova** `Activity` que exiba uma lista de textos na tela, fazendo o aplicativo começar por essa tela.

*A parte de layout deve ser feita por meio do xml e não pelo editor visual.*

### Passo a passo

1. Vamos criar uma nova classe chamada `ListaActivity.kt` no novo pacote `activity`, onde ficarão todas as nossas `activities`. Não esqueça de mover também a `TweetActivity.kt` para esse pacote.

```
class ListaActivity {
}
```

2. Agora fazemos a classe se tornar uma `Activity`:

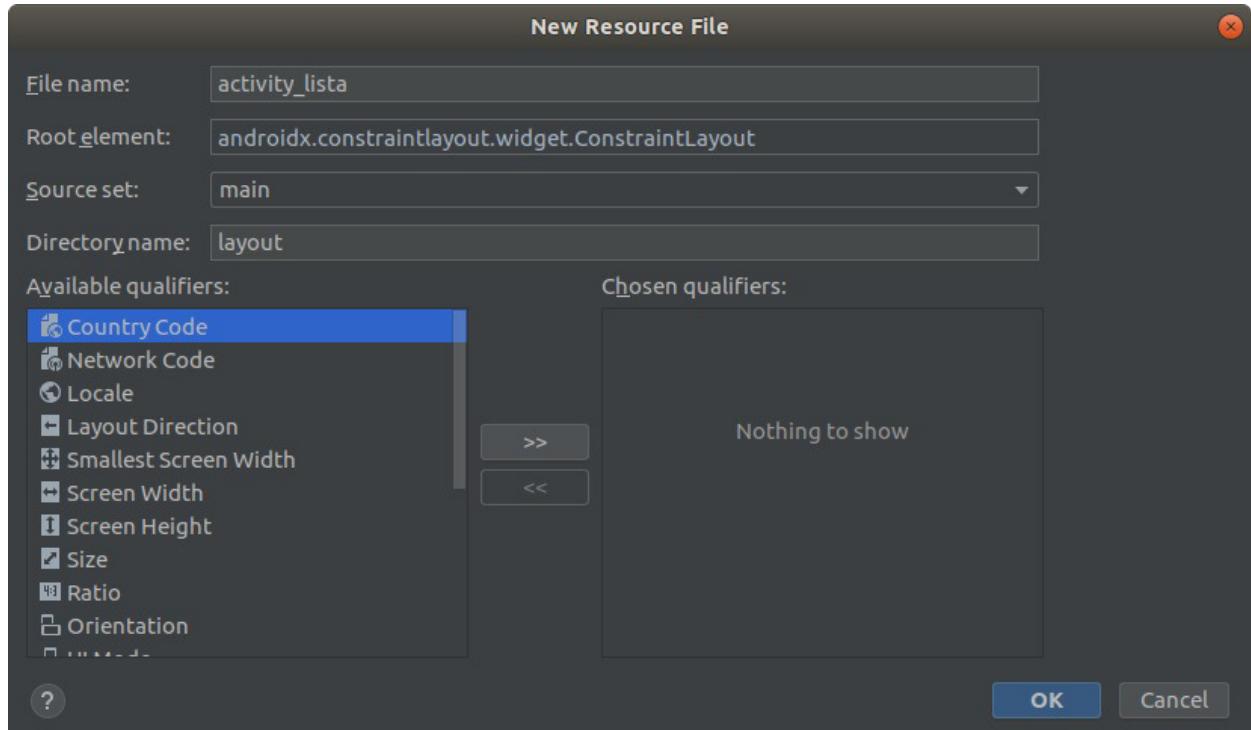
```
class ListaActivity : AppCompatActivity() {
}
```

3. O próximo passo é sobrescrever o método responsável pela criação da `Activity`:

```
class ListaActivity : AppCompatActivity() {
 override fun onCreate(savedInstanceState: Bundle?) {
 super.onCreate(savedInstanceState)
 }
}
```

4. Agora precisamos definir o `layout` da nossa `Activity`. Para isso, vamos criar um arquivo `xml` que

represente nosso *layout*. Vá até a pasta `res` e, na subpasta `layout`, clique com o botão direito e escolha a opção **new -> Layout resource file**. Deixe o nome do arquivo a ser gerado como `activity_lista` e certifique-se de que o campo `Root element` está com o `ConstraintLayout`. Se o elemento raiz for outro, basta procurar pelo `ConstraintLayout`:



5. Com o arquivo criado, troque do editor visual para o editor de texto usando a aba que fica na parte de baixo do editor. Em seguida, adicione um `ListView`:

```
<androidx.constraintlayout.widget.ConstraintLayout
 xmlns:android="http://schemas.android.com/apk/res/android"
 xmlns:app="http://schemas.android.com/apk/res-auto"
 android:layout_width="match_parent"
 android:layout_height="match_parent">

 <ListView />
</androidx.constraintlayout.widget.ConstraintLayout>
```

- Toda `View` **precisa obrigatoriamente** ter a sua altura e largura definidas no *layout*. Vamos definir o tamanho da nossa lista para que ela utilize o tamanho inteiro da tela. Como estamos usando um `ConstraintLayout`, vamos usar o valor `0dp` (equivalente a usar `match_constraint`):

```
<androidx.constraintlayout.widget.ConstraintLayout
 xmlns:android="http://schemas.android.com/apk/res/android"
 xmlns:app="http://schemas.android.com/apk/res-auto"
 android:layout_width="match_parent"
 android:layout_height="match_parent">

 <ListView
 android:layout_width="0dp"
```

```
 android:layout_height="0dp"/>
</androidx.constraintlayout.widget.ConstraintLayout>
```

- Agora será necessário definir programaticamente todas as *constraints* que nosso `ListView` precisa ter:

```
<ListView
 android:layout_width="0dp"
 android:layout_height="0dp"
 app:layout_constraintBottom_toBottomOf="parent"
 app:layout_constraintEnd_toEndOf="parent"
 app:layout_constraintStart_toStartOf="parent"
 app:layout_constraintTop_toTopOf="parent" />
```

- Como vamos querer acessar essa `View` na `Activity`, precisamos definir um `id` para ela:

```
<ListView
 android:id="@+id/lista_tweet"
 android:layout_width="0dp"
 android:layout_height="0dp"
 app:layout_constraintBottom_toBottomOf="parent"
 app:layout_constraintEnd_toEndOf="parent"
 app:layout_constraintStart_toStartOf="parent"
 app:layout_constraintTop_toTopOf="parent" />
```

6. Agora precisamos vincular o *layout* que acabamos de criar com a `ListaActivity`. Para isso, faremos uso do método `setContentView`, passando a referência do arquivo que é gerada na classe R :

```
class ListaActivity : AppCompatActivity() {
 override fun onCreate(savedInstanceState: Bundle?) {
 super.onCreate(savedInstanceState)
 setContentView(R.layout.activity_lista)
 }
}
```

7. Vamos criar uma lista de `String`s para representar alguns tweets para teste usando o método `listOf` do `Kotlin`:

```
class ListaActivity : AppCompatActivity() {
 override fun onCreate(savedInstanceState: Bundle?) {
 super.onCreate(savedInstanceState)
 setContentView(R.layout.activity_lista)

 val tweets = listOf("Meu primeiro tweet", "Meu segundo tweet", "Meu terceiro tweet", "Meu quarto tweet", "Meu quinto tweet")
 }
}
```

8. Nesse instante precisamos passar para nosso `ListView` a lista de `String`s para exibir os *tweets*. Usaremos um objeto do tipo `ArrayAdapter` para **adapta**r nossa lista de *tweets* e o `ListView` conseguir exibi-la:

```
class ListaActivity : AppCompatActivity() {
 override fun onCreate(savedInstanceState: Bundle?) {
 super.onCreate(savedInstanceState)
 setContentView(R.layout.activity_lista)
```

```

 val tweets: List<String> = listOf("Meu primeiro tweet", "Meu segundo tweet", "Meu terceiro
tweet", "Meu quarto tweet", "Meu quinto tweet")

 val adapter = ArrayAdapter<String>(this, android.R.layout.simple_list_item_1, tweets)
 }
}

```

9. Agora vamos usar o *import* automático que o `Kotlin` possui para buscar os componentes da tela e disponibilizar suas referências na `Activity`. Para isso, basta escrevermos parte do *id* da `View` que estamos buscando e ele cuidará de fazer o restante. Faremos isso para buscar nosso `ListView` e colocarmos o `Adapter` dentro dele:

```

package br.com.caelum.twittelumapp.activity

import android.os.Bundle
import android.widget.ArrayAdapter
import androidx.appcompat.app.AppCompatActivity
import br.com.caelum.twittelumapp.R

// esse import será gerado automaticamente
import kotlinx.android.synthetic.main.activity_lista.*

class ListaActivity : AppCompatActivity() {
 override fun onCreate(savedInstanceState: Bundle?) {
 super.onCreate(savedInstanceState)
 setContentView(R.layout.activity_lista)

 val tweets: List<String> = listOf("Meu primeiro tweet", "Meu segundo tweet", "Meu terceiro
tweet", "Meu quarto tweet", "Meu quinto tweet")

 val adapter = ArrayAdapter<String>(this, android.R.layout.simple_list_item_1, tweets)

 lista_tweet.adapter = adapter
 }
}

```

10. Rode o aplicativo.

11. Ainda estamos exibindo a nossa tela de criação de `tweets` mas queremos iniciar o aplicativo pela nossa lista de `tweets`. Precisamos alterar o arquivo `AndroidManifest.xml` para indicar que a nossa `ListaActivity` é a tela inicial do aplicativo. Dentro da tag `<application>` deverá ficar o seguinte código:

```

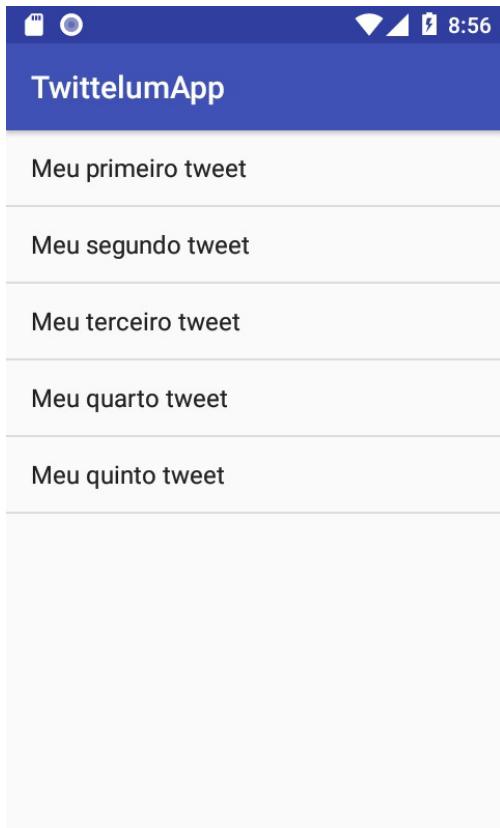
<activity android:name=".activity.ListaActivity">
 <intent-filter>
 <action android:name="android.intent.action.MAIN" />

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

<activity android:name=".activity.TweetActivity" />

```

12. Rode novamente o aplicativo. Agora você deve estar vendo algo parecido com isso:



Saber inglês é muito importante em TI

**alura** língua

Na **Alura Língua** você reforça e aprimora seu inglês! Usando a técnica *Spaced Repetitions* o aprendizado naturalmente **se adapta ao seu conhecimento**. Exercícios e vídeos interativos fazem com que você pratique em situações cotidianas. Além disso, todas as aulas possuem explicações gramaticais, para você entender completamente o que está aprendendo. Aprender inglês é fundamental para o profissional de tecnologia de sucesso!

[Pratique seu inglês na Alura Língua.](#)

### 3.8 ADICIONANDO UM BOTÃO PARA NOVO TWEET

O usuário, quando abrir nosso aplicativo, vai ver a lista de *tweets*. Olhando os *tweets* já criados, ele

pode querer em algum momento criar um novo *tweet*. Seria interessante então ter um botão para o usuário poder criar um novo *tweet*. Esse comportamento na realidade é muito comum em diversos aplicativos usados no dia a dia, como o WhatsApp e o Gmail. Já vimos como colocar um botão na tela, então podemos fazer o mesmo na nossa `ListaActivity`.

Acontece que um botão com o texto "Novo Tweet" no centro e na parte de baixo da tela pode ser confundido com os itens da lista, ficando difícil de saber no que clicar e o que é item e o que não é. Se olharmos nos botões desses aplicativos citados acima, eles tem uma aparência de que estão flutuando, pra dar a ideia de que não são a mesma componente que a lista. Além disso, eles são arredondados e tem um tamanho padrão.

Esse padrões, junto com vários outros, foram resultado de muitos estudos em usabilidade e design que acabaram gerando o **Material Design** (<https://material.io/>), um projeto que envolve diretrizes, componentes e ferramentas que deem suporte às boas práticas de *design* de interface do usuário. Ele foi criado em uma linguagem que seja bem fácil de ser compreendida, visual, com a intenção de fornecer uma experiência unificada ao usuário, independente do dispositivo que ele usar, mas permitindo uma grande flexibilidade aos desenvolvedores para criarem as aplicações com suas características. O Android começou a dar suporte a essas diretrizes a partir do Android 5.0 (Lollipop), e fornece informações específicas do *Material Design* para Android : <https://developer.android.com/design/> .

Segundo o *Material Design*, esses botões que dão a ideia de flutuação levam à principal funcionalidade disponível ao usuário naquela tela. Como já comentamos, a principal ação que o usuário vai querer fazer da tela de listagem é criar um novo *tweet*. Vamos então adicionar um **botão flutuante** na tela de listagem! Como esse tipo de botão se tornou muito utilizado, os desenvolvedores do Android já criaram uma componente que representa esse tipo de botão: a `FloatingActionButton`.

## 3.9 FLOATINGACTIONBUTTON

### Importando a componente

Como essa componente foi criada baseada no *Material Design*, ela não está na biblioteca de componentes padrão do Android , então precisamos importar uma biblioteca externa, a `com.google.android.material:material`. O Android usa o gerenciador de dependências Gradle para organizar as versões e dependências necessárias no nosso projeto. Então, uma das maneiras de importar uma dependência é alterando o arquivo `build.gradle` (`Module: app`) . Nesse arquivo estão várias informações relacionadas às versões, pacotes, bibliotecas e plugins utilizados na nossa aplicação. Dentro dele há a seção `dependencies` , dentro da qual podemos colocar todas as bibliotecas que precisarmos utilizar. Depois do nome da dependência, colocamos " : " seguido da versão daquela biblioteca que utilizaremos:

```
dependencies {
```

```
 implementation 'com.google.android.material:material:1.0.0'
 }
```

## Customizando o botão

Esses botões flutuantes geralmente mostram algum ícone sinalizando a ação que será feita, pois não possuem espaço suficiente para textos. Para isso, a componente possui o atributo `android:src`, que espera receber a imagem que ficará dentro do botão.

### Definindo um ícone

O *Material Design* já recomenda uma grande variedade de ícones para serem utilizados em diversas situações. O Android disponibiliza vários deles já prontos para utilizarmos, no formato vetorial, para manter a qualidade da imagem dependendo da resolução da tela. Veremos no exercício como escolhermos esses ícones vetorizados pelo Android Studio. Os ícones que são usados nas nossas telas ficam dentro da pasta `res` e dentro ainda de uma pasta mais específica, `drawable`.

Uma vez que tivermos o ícone, precisamos recuperá-lo no botão. Já vimos que, para acessar recursos da aplicação no código podemos usar a classe `R`, mas nesse momento estamos em um arquivo `XML`. Em arquivos desse tipo, a sintaxe utilizada para acessar um outro recurso é `@tipoDoRecurso`. Por exemplo, se queremos passar para o `FloatingActionButton` um ícone criado em `res/drawable` com o nome `ic_novo`, usamos o seguinte código:

```
<com.google.android.material.floatingactionbutton.FloatingActionButton
 android:src="@drawable/ic_novo" />
```

### Seguindo o Material Design

Pelas diretrizes do *Material Design*, o botão não pode estar colado com a borda da tela. Por isso, atribuiremos uma **margem** ao botão com o atributo `android:layout_margin="16dp"`, atributo existente em quase todas as componentes.

#### Boas práticas de Layout

Você pode ver as melhores práticas do *material design* na documentação:

<https://material.io/design/components/buttons-floating-action-button.html>

### Alterando a cor do botão

A componente `FloatingActionButton` já utiliza por padrão as cores padrões definidas no tema da nossa aplicação. Se quisermos mudar a cor, precisamos utilizar os atributos com *namespace* `app`: `app:backgroundTint` muda a cor de fundo do botão, enquanto que `app:tint` muda a cor do ícone

em si e `app:rippleColor` define a cor que aparece no clique do botão. Para definir uma cor para qualquer um desses atributos, podemos passar direto a cor no formato hexadecimal, por exemplo, para o branco podemos fazer:

```
app:backgroundTint="#ffffffff"
```

Podemos também procurar por uma cor que o Android já tenha dado um nome para o valor hexadecimal, acessando os recursos do Android com a sintaxe `@android:tipoDoRecurso`. Por exemplo, para a cor branca podemos definir:

```
app:backgroundTint="@android:color/white"
```

## 3.10 ALERTA PRO USUÁRIO

Queremos que o usuário vá para a tela de novo *tweet* ao clicar no botão flutuante, mas, como é uma nova componente que estamos utilizando, vamos em um primeiro momento apenas alertar o usuário que foi feito o clique.

Já vimos uma maneira de mostrar um aviso temporário ao usuário, usando o `Toast`. Mas, dessa forma, não fica claro a qual aplicação ou tela está relacionado o aviso. Se a mensagem no aviso for muito genérica, como "a ação está sendo processada" ou "foram liberados 3GB no celular", não será possível o usuário saber que aplicativo enviou essa mensagem. Há duas maneiras de solucionar: enviar uma mensagem mais completa, mas aí o texto pode começar a ficar muito grande, ou deixar o aviso "grudado" à tela que ele está relacionado. Por exemplo, o Google Fotos faz um aviso desse tipo ao avisar que terminou a limpeza de arquivos do aplicativo, usando a classe `Snackbar`. Usaremos essa classe, que vem da mesma biblioteca de suporte que o `FloatingActionButton`, para alertar o usuário que o botão flutuante foi clicado. O `Snackbar` funciona bem parecido com o `Toast`: primeiro é feita a criação de um objeto do tipo `Snackbar` com o método estático `make` para depois ser feito o pedido de mostrá-lo na tela com o método `show`. O método `make` recebe a `View` a qual ela estará presa, a mensagem para o usuário e a duração da exibição da mensagem. Sendo `view` uma variável que guarda alguma `View`, podemos criar e mostrar o `Snackbar` com o seguinte código:

```
Snackbar.make(view, "FAB clicado", Snackbar.LENGTH_SHORT).show()
```

## Arrumando o Layout

Quando a componente `Snackbar` fica visível, ela acaba ocupando o espaço do botão flutuante e, consequentemente, esconde parte do botão. Esse visual fica bem estranho para o usuário. O ideal seria continuarmos mostrando a `FloatingActionButton` mesmo com outras componentes como a `Snackbar` na tela. Para isso, existe uma componente, chamada `CoordinatorLayout`, que coordena as componentes que estão na tela. É possível especificar como cada componente vai reagir com mudanças na tela, definindo a interação entre as `View`s da tela. Algumas componentes já possuem um

comportamento padrão quando são filhas de `CoordinatorLayout`, como por exemplo a `FloatingActionButton`: ela vai automaticamente se deslocar para não ser encobrida pela `Snackbar`. Portanto, basta adicionar ao nosso `layout` a `GroupView CoordinatorLayout` como `container`, ou seja, como a `View` principal nessa tela.

**Aprenda se divertindo na Alura Start!**



Você conhece alguém que tem potencial para tecnologia e programação, mas que nunca escreveu uma linha de código? Pode ser um filho, sobrinho, amigo ou parente distante. Na **Alura Start** ela vai poder criar games, apps, sites e muito mais! É o **começo da jornada com programação e a porta de entrada para uma possível carreira de sucesso**. Ela vai estudar em seu próprio ritmo e com a melhor didática. A qualidade da conceituada Alura, agora para Starters.

[Conheça os cursos online da Alura Start!](#)

### 3.11 EXERCÍCIO: ADICIONANDO BOTÃO FLUTUANTE

#### Objetivo

Colocar um `FloatingActionButton` na tela de listagem que, ao ser clicado, mostre um aviso ao usuário usando um `Snackbar`.

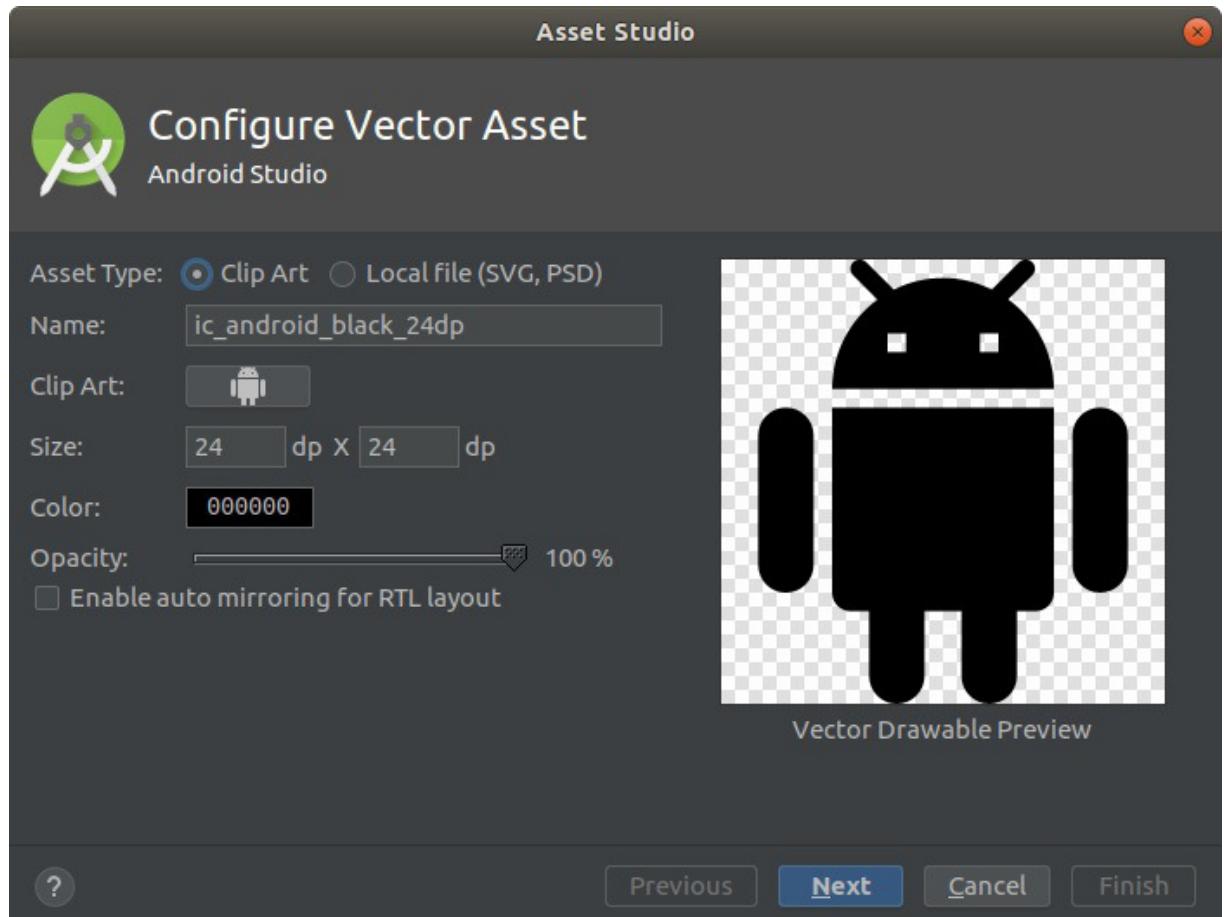
#### Passo a passo

1. Vamos adicionar uma biblioteca de componentes do **Material Design** ao nosso projeto. Primeiro procure e abra o arquivo `build.gradle` (`module: app`) na aba lateral de projeto do Android Studio. Em seguida procure no arquivo a parte de dependências e adicione a seguinte linha:

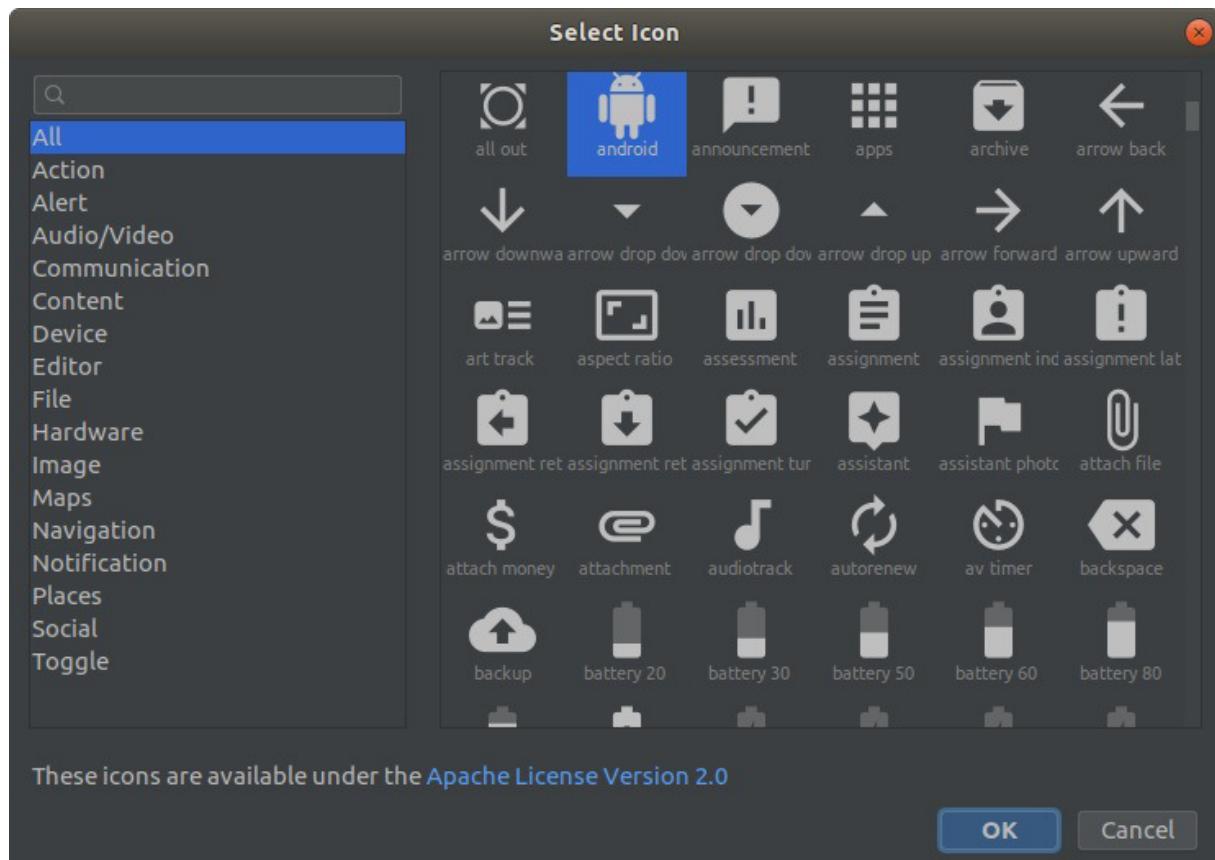
```
dependencies {
 // demais dependências
 implementation 'com.google.android.material:material:1.0.0'
}
```

2. Vamos gerar um ícone vetorizado para usar no botão.

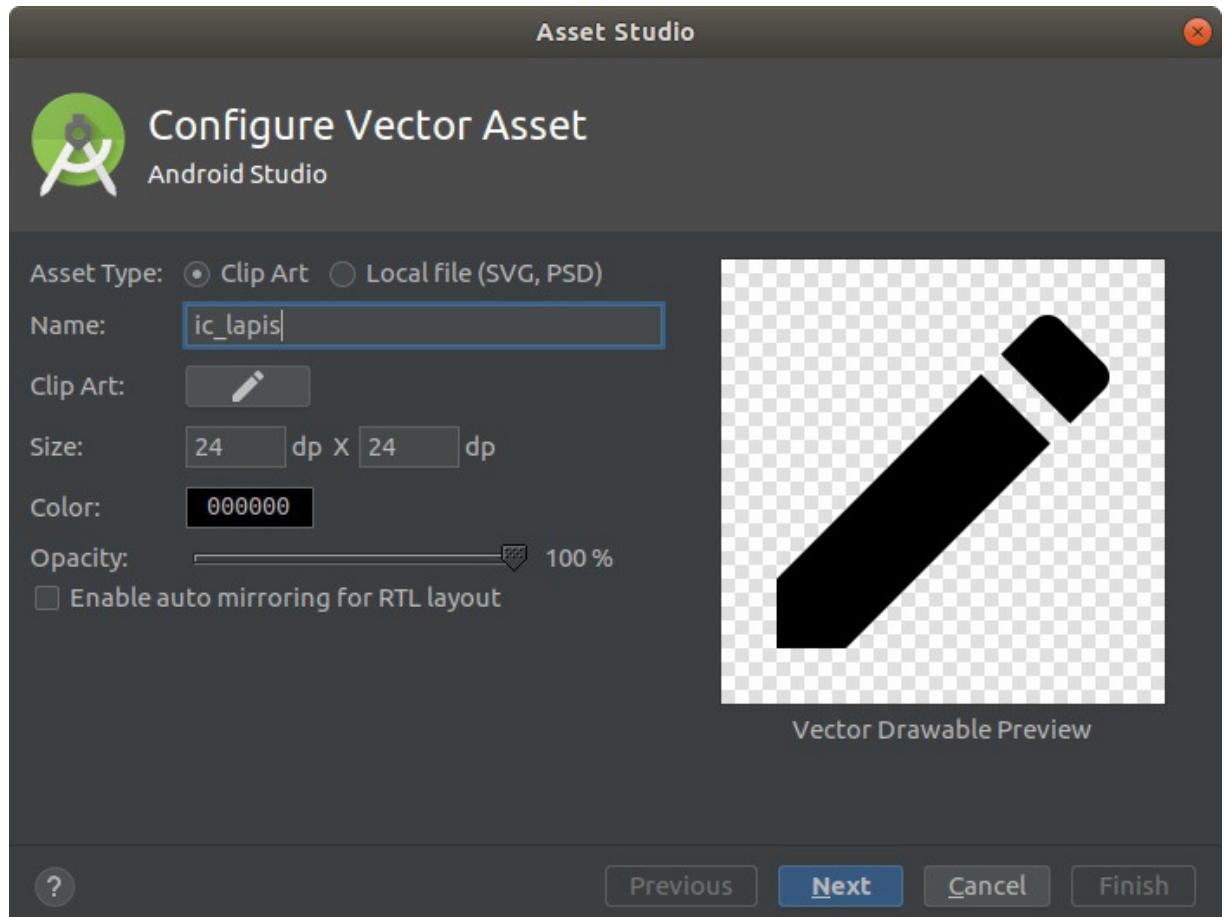
- Clique com o botão direito na pasta `res/drawable`, depois em `New -> Vector Asset`. Aparecerá uma nova janela para configuração do ícone.



- Para escolher um ícone, vamos clicar no desenho do Android ao lado de `Clip Art` e veremos uma lista de vários ícones sugeridos pelo **Material Design**.



- Para esse botão de novo tweet, podemos procurar pelo ícone `create`, de um lápis escrevendo, selecioná-lo e dar o nome de `ic_lapis`. Finalmente, vamos adicioná-lo ao nosso projeto clicando em `Next` e depois em `Finish`.



3. Agora vamos adicionar o `FloatingActionButton` ao layout da nossa listagem de tweets editando o arquivo `activity_lista.xml`. O xml de layout ficará assim:

```

<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.ConstraintLayout xmlns:android="http://schemas.android.com/apk/res/android"
 xmlns:app="http://schemas.android.com/apk/res-auto"
 android:layout_width="match_parent"
 android:layout_height="match_parent">

 <ListView
 android:id="@+id/lista_tweet"
 android:layout_width="0dp"
 android:layout_height="0dp"
 app:layout_constraintBottom_toBottomOf="parent"
 app:layout_constraintEnd_toEndOf="parent"
 app:layout_constraintStart_toStartOf="parent"
 app:layout_constraintTop_toTopOf="parent" />

 <com.google.android.material.floatingactionbutton.FloatingActionButton
 android:id="@+id/fab_add"
 android:layout_width="wrap_content"
 android:layout_height="wrap_content"
 app:layout_constraintBottom_toBottomOf="parent"
 app:layout_constraintEnd_toEndOf="parent"
 android:layout_margin="16dp"/>

```

```

 android:src="@drawable/ic_lapis"
 app:backgroundTint="@android:color/holo_blue_dark"
 app:rippleColor="@android:color/holo_blue_light" />

 </androidx.constraintlayout.widget.ConstraintLayout>

```

4. Vamos implementar o comportamento do clique em nosso `FloatingActionButton` fazendo com que ele exiba uma `SnackBar` indicando que o botão foi clicado. Para isso, vamos alterar a classe `ListaActivity`:

```

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

 val tweets: List<String> = listOf("Meu primeiro tweet", "Meu segundo tweet", "Meu terceiro tweet", "Meu quarto tweet", "Meu quinto tweet")

 lista_tweet.adapter = ArrayAdapter<String>(this, android.R.layout.simple_list_item_1, tweets)

 fab_add.setOnClickListener { Snackbar.make(it, "FAB clicado", Snackbar.LENGTH_SHORT).show() }
}

```

5. Rode o aplicativo e clique no botão flutuante. Provavelmente você verá algo próximo disso:



6. O resultado não foi bem o que queríamos já que a `SnackBar` foi exibida por cima do

`FloatingActionButton` . Para corrigir esse problema, vamos utilizar o `CoordinatorLayout` . Sendo assim, nosso *layout* ficará dessa maneira:

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.coordinatorlayout.widget.CoordinatorLayout xmlns:android="http://schemas.android.com/apk/res/android"
 xmlns:app="http://schemas.android.com/apk/res-auto"
 android:layout_width="match_parent"
 android:layout_height="match_parent">

 <androidx.constraintlayout.widget.ConstraintLayout
 android:layout_width="match_parent"
 android:layout_height="match_parent">

 <ListView
 android:id="@+id/lista_tweet"
 android:layout_width="0dp"
 android:layout_height="0dp"
 app:layout_constraintBottom_toBottomOf="parent"
 app:layout_constraintEnd_toEndOf="parent"
 app:layout_constraintStart_toStartOf="parent"
 app:layout_constraintTop_toTopOf="parent" />

 </androidx.constraintlayout.widget.ConstraintLayout>

 <com.google.android.material.floatingactionbutton.FloatingActionButton
 android:id="@+id/fab_add"
 android:layout_width="wrap_content"
 android:layout_height="wrap_content"
 android:layout_gravity="end|bottom"
 android:layout_margin="16dp"
 android:src="@drawable/ic_lapis"
 app:backgroundTint="@android:color/holo_blue_dark"
 app:rippleColor="@android:color/holo_blue_light"
 />

</androidx.coordinatorlayout.widget.CoordinatorLayout>
```

7. Rode o aplicativo novamente e clique no botão para ver se acontece algo diferente.



## 3.12 NAVEGANDO PELAS ACTIVITIES

### Iniciando outra Activity

Vimos que o botão flutuante está funcionando. Mas, na verdade, o que queremos mesmo fazer ao clicar no botão é ir para a tela de criação de tweet que já temos criada, ou seja, queremos que a `TweetActivity` seja iniciada. Quem define quando uma `Activity` será iniciada é o próprio `Android`. A gente apenas pede para que uma determinada `Activity`, se possível, seja iniciada por meio do método `startActivity`. Faremos essa lógica no *listener* do botão flutuante.

### Intents Explícitas

O método `startActivity` recebe uma `Intent`, que representa a nossa **intenção** de iniciar uma `Activity`. Nesse caso, queremos ir para uma `Activity` específica, ou seja, queremos avisar **explicitamente** que a `TweetActivity` será iniciada. Para criarmos uma `Intent explícita`, precisamos informar no construtor quem está pedindo para a nova `Activity` ser criada, ou seja, o contexto do momento, e a definição da classe que controla a nova `Activity`:

```
Intent(this, TweetActivity::class.java)
```

Para o contexto usamos a própria `Activity` do momento. Para a definição da classe, é pedida a representação em `Java`. Como a `TweetActivity` é escrita em `Kotlin`, usamos a sintaxe do `Kotlin` para pegar a definição dela em `Java`: `TweetActivity::class.java` (se fosse uma classe `Java`, seria `TweetActivity.class`, e se fosse a referência em `Kotlin`, seria `TweetActivity::class`).

## Encerrando uma Activity

Depois do usuário mudar para a tela de novo `tweet`, ele vai digitar o `tweet` e clicar no botão `Publicar Tweet`. Em seguida, ele vai esperar que o `tweet` seja salvo e que ele vá para a tela de listagem. Mas, se ficarmos iniciando telas o tempo inteiro, vai começar a ter muito objeto `Activity` armazenado no nosso aplicativo consumindo assim muitos recursos de memória.

Se olharmos mais a fundo, o que realmente o usuário quer quando ele clica no botão `Publicar Tweet` é voltar para a tela anterior, bastando **encerrar** a `Activity` atual.

O `Android` gerencia as `Activity`s ativas por meio de uma **pilha**: cada `Activity` nova que é criada é colocada em cima das outras que já estavam na pilha, e a `Activity` que está em primeiro plano é a que está no topo dessa pilha. Ao encerrarmos a `Activity` do topo, a segunda `Activity` da pilha passa a ser a primeira e a ficar visível ao usuário.

Portanto, para voltar à `Activity` anterior, basta pedirmos para o `Android` finalizar a `Activity` atual por meio do método `finish()`. Faremos isso no exercício abaixo no momento do clique do botão `Publicar Tweet` da `TweetActivity`.

## 3.13 EXERCÍCIO: TROCANDO DE ACTIVITY

### Objetivo

- Fazer com que o clique no `FloatingActionButton` leve o usuário para a tela de inclusão de `tweet`
- Fazer com que o usuário volte para a tela anterior ao clicar no botão `Publicar Tweet`

### Passo a passo

1. Vamos fazer o aplicativo trocar de tela quando o botão for clicado. Primeiro, vamos declarar para o `Android` que temos a intenção de trocar de `Activity`. Para isso, no método `onCreate` da `ListaActivity`, faremos uso de uma `Intent` que será passada para o `Android` com o método `startActivity`:

```
fab_add.setOnClickListener {
 // remova o snack bar
```

```
 val intencao = Intent(this, TweetActivity::class.java)
 startActivity(intencao)
}
```

2. Se rodarmos o aplicativo e clicarmos no botão, seremos levados para a tela de novo *tweet* e poderemos então escrever um texto para publicarmos.
3. Porém, quando clicarmos no botão *Publicar tweet*, continuaremos vendo a tela de cadastro. Queremos, em vez disso, voltar para a tela de lista de *tweets*. Para resolver isso, vamos mudar a lógica do clique no botão da classe `TweetActivity`, alterando o *listener* dele e pedindo para essa *Activity* ser finalizada:

```
botao.setOnClickListener (View.OnClickListener {
 publicaTweet()
 finish()
})
```

4. Vamos rodar o aplicativo novamente e veremos que agora, depois de irmos pra tela de inclusão de *tweet* e escrevermos um *tweet*, o botão *Publicar tweet* nos levará de volta para a tela de listagem.

#### Seus livros de tecnologia parecem do século passado?



Conheça a **Casa do Código**, uma **nova** editora, com autores de destaque no mercado, foco em **ebooks** (PDF, epub, mobi), preços **imbatíveis** e assuntos **atuais**.

Com a curadoria da **Caelum** e excelentes autores, é uma abordagem **diferente** para livros de tecnologia no Brasil.

[Casa do Código, Livros de Tecnologia.](#)

## 3.14 UTILIZANDO MENUS

O nosso botão `Publicar Tweet` está fazendo bem a lógica que determinamos para ele até o momento, de mostrar um `Toast` e finalizar a `TweetActivity`, mas, se as informações que estão acima do botão ocuparem muito espaço, o botão passa a ficar fora da tela visível e o usuário precisará arrastar a tela para poder encontrar o botão. Pensando na boa experiência do usuário, o melhor seria o botão estar visível a todo momento. De fato, se pensarmos em diversos aplicativos muito utilizados, há um **menu** no canto superior direito da tela mostrando as principais ações disponíveis para aquela tela. Por exemplo, no *Google Calendar* é possível ir para o dia atual e atualizar o calendário; no *Whatsapp* é possível realizar buscas nas conversas, iniciar um novo grupo ou transmissão, olhar as configurações,

entre outros. No nosso caso, mostraremos apenas a ação de salvar o *tweet* que a pessoa digitou.

A partir do Android 3.0 , surgiram os menus aparecendo no canto superior direito. Esse tipo de menu é chamado de *Menu de Opções* e fica dentro da *Barra de Ações (ActionBar)*. Como toda Activity pode ter um *Menu de Opções*, todas possuem um método chamado `onCreateOptionsMenu` , que recebe um objeto do tipo `Menu` . O retorno desse método indica se esse menu deve aparecer na tela: retorna `true` (o retorno padrão da Activity ) para informar que o *Menu de Opções* estará visível.

```
override fun onCreateOptionsMenu(menu: Menu?): Boolean {
 return true
}
```

Podemos ver mais uma novidade na sintaxe do `Kotlin` : informamos o tipo do retorno de um método depois do nome dele, assim como é feito para as variáveis.

Um menu só faz sentido existir se possuir itens; portanto, se não inserirmos nenhum item no objeto `Menu` , ele também não será mostrado na tela.

## Itens do Menu

Uma das formas de adicionarmos itens em um menu é utilizar o próprio objeto `Menu` e chamar o método `add` para cada item que quisermos inserir. Por exemplo, para adicionar o item de `Publicar tweet` ao menu, podemos escrever o código:

```
menu.add("Publicar Tweet")
```

A partir do momento em que o menu possui pelo menos um item, ele já ficará visível ao usuário na tela. Como comportamento padrão, o sistema colocará como nome do item o parâmetro passado no método `add` .

Apesar da possibilidade da criação programática do menu, ele é um elemento de tela, no qual podemos definir ícones e visibilidade. Por isso, damos a preferência de criar seus itens por meio de um `xml`, assim como fazemos com nossos arquivos de *layout*.

```
<menu xmlns:android="http://schemas.android.com/apk/res/android"
 xmlns:app="http://schemas.android.com/apk/res-auto">

 <item android:title="Publicar Tweet" />

</menu>
```

Como esse `xml` é específico de menu, ele ficará em uma pasta reservada pra esse tipo de *layout*: `res/menu` . O único atributo obrigatório para um item de menu é o título, que é a mesma informação que passaríamos ao método `add` do objeto `Menu` . Além disso, podemos ter vários itens e ações específicas para cada um deles, portanto precisamos identificá-los com o atributo `android:id` . Por

fim, há várias telas de aplicativos em que, ao lado do menu, aparecem alguns ícones. Esses ícones nada mais são que itens que estão aparecendo diretamente na *Action Bar*, graças a uma característica que receberam: o atributo `app:showAsAction`, que determina quando aquele item será mostrado na *Barra de Ações*, se *sempre*, com `always`, se *nunca*, com `never`, se *apenas se tiver espaço*, com `ifRoom`, e outras possibilidades. Se vamos mostrar um ícone, precisamos definir um com o atributo `android:icon`. O botão `Publicar Tweet` poderia virar um menu com as seguintes configurações:

```
<menu xmlns:android="http://schemas.android.com/apk/res/android"
 xmlns:app="http://schemas.android.com/apk/res-auto">

 <item android:title="Publicar Tweet"
 android:id="@+id/tweet_menu_publicar"
 android:icon="@android:drawable/ic_menu_save"
 app:showAsAction="always" />

</menu>
```

Agora precisamos fazer o Android criar elementos na tela, ou seja, *Views*, para cada item do menu descrito no *xml*. Para isso, o Android possui um especialista: o `MenuInflater`.

## Inflando o Menu com `MenuInflater`

O Android precisa ter a capacidade de ler os *xmls* que criamos e criar objetos do tipo `View` a partir das tags. Essa é a especialidade dos ***Inflators***. No caso específico de um *xml de menu*, podemos utilizar um `Inflater` especializado em interpretar esse tipo de arquivo. Trata-se do ***MenuInflater***.

Dentro do método `onCreateOptionsMenu`, podemos fazer uma chamada a `getMenuInflater`, um método da classe `Activity` que nos dá acesso a um `MenuInflater`.

Repare que o método `onCreateOptionsMenu` já nos fornece um objeto do tipo ***Menu***. Basta utilizarmos o inflater para ler nosso *xml* e criar a partir dele os itens que serão colocados no menu. Podemos então utilizar o método mais importante de um `Inflater`: o `inflate`, que precisa da informação do *xml* que será inflado e o objeto `Menu` que receberá os itens que serão inflados.

## Adicionando lógica ao Menu

Uma vez que o usuário clicar no ícone de `Publicar Tweet`, ele vai esperar que o *tweet* que ele escreveu seja salvo. Precisamos então atrelar uma lógica ao clique neste item. Uma maneira de fazer isso é adicionando um *listener*, mas teríamos que adicionar um *listener* para cada item do menu. Para evitar esse monte de *listeners* que criariam, o Android disponibiliza um método chamado `onOptionsItemSelected`, que recebe um objeto `MenuItem` referente ao item que foi clicado e nele definimos qual a lógica que queremos executar no clique de um item. O retorno desse método informa se só essa lógica será executada, com o retorno `true`, ou se outros *listeners* e demais processamentos serão executados depois da lógica definida no método, com o retorno `false`. Sobrescrevendo o

método, nosso código ficaria assim:

```
override fun onOptionsItemSelected(menuItem: MenuItem?): Boolean {
 // lógica de publicar um tweet
 return false
}
```

Se só tivermos um item no menu, o código acima funciona muito bem, mas, na maioria das vezes, temos vários itens em um menu. Além disso, esse método da `AppCompatActivity` já trata vários itens por padrão. Portanto, é uma boa prática só executar a lógica relacionada a um item se este for o item clicado, verificando seu `id`. Precisamos então comparar a *propriedade* `itemId` de `menuItem` com o `id` do item, que no nosso caso é `tweet_menu_publicar`.

Porém, se repararmos no tipo de `menuItem`, vemos que ela pode ter o valor `null` por causa da "`? ?`". Se chamarmos uma propriedade ou método de uma variável com valor nulo, será lançada aquela exceção `NullPointerException`. Podemos evitar essa exceção fazendo um `if`, mas precisamos lembrar de escrevê-lo e sempre comparamos uma variável com o valor `null` nesses casos. O sistema de tipagem no `Kotlin` visa ajudar o programador a evitar a maior parte desses `NullPointerException` dando erro de compilação no código sempre que tentamos acessar um método ou atributo de uma variável que pode ter valor nulo. Uma das formas de fazer o código compilar é com o `if` comentado acima, verificando se a variável é nula. Como alternativa, o `Kotlin` disponibiliza o operador `?.`, chamado de *safe call* (chamada segura), que faz a mesma coisa: só acessa a propriedade ou executa o método se a variável não for nula. Portanto, se quisermos acessar o `itemId` do `menuItem` usando o *safe call*, escrevemos o código:

```
menuItem?.itemId
```

Para comparar o `id` do item clicado com os itens do menu, podemos usar qualquer operador de verificação, como o `if` ou o `when`. O `when` segue a mesma lógica do `switch/case`. Se só queremos executar a lógica de publicar o `tweet` quando o `itemId` do `menuItem` for igual a `R.id(tweet_menu_publicar)`, o código ficaria assim:

```
when (menuItem?.itemId) {
 R.id(tweet_menu_publicar) -> {
 // lógica de publicar o tweet
 }

 return false
}
```

## 3.15 BOTÃO DE VOLTAR

Quando o usuário estiver na tela de criar um novo `tweet`, ele pode desistir da ideia e querer voltar para a lista. Podemos fazer isso adicionando um outro item do *Menu de Opções* com o ícone de "x",

para adicionar a funcionalidade de *cancelar* o que seria feito na tela do momento. Mas uma outra abordagem comum em muitos aplicativos é mostrar uma setinha apontada para a esquerda na *Action Bar*, dando a ideia de que voltaremos para a tela anterior. Para deixar essa setinha visível, precisamos ter acesso à *Action Bar* e depois pedir para ativar o ícone do canto esquerdo mostrando a setinha apontada para a esquerda.

```
supportActionBar?.setDisplayHomeAsUpEnabled(true)
```

Só escrevendo o código acima, já podemos visualizar a setinha na *Action Bar*.

Porém, o Android não sabe para onde ir quando o usuário clicar na setinha. Como esse ícone aparece na *Action Bar*, ele é considerado também como um item do *Menu de Opções*, e já possui o `id home`. Portanto, podemos definir a lógica para ele usando também o método `onOptionsItemSelected`. Se queremos atribuir a essa setinha a lógica de voltar para a tela anterior, podemos apenas pedir para a *Activity* que está ativa no momento ser finalizada no clique desse item:

```
override fun onOptionsItemSelected(item: MenuItem?): Boolean {
 when (item?.itemId) {
 android.R.id.home -> finish()
 }
 return false
}
```

## 3.16 EXERCÍCIO: MELHORANDO O BOTÃO DE SALVAR

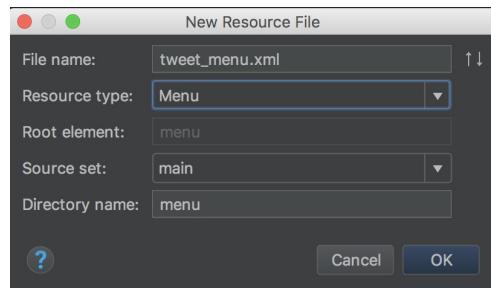
### Objetivo

- Remover o `Button` que tínhamos para publicar o *tweet* e colocar a mesma funcionalidade em um botão de **menu**
- Incluir o botão de voltar na tela de criação de *tweet*

### Passo a passo

1. Para melhorar um pouco o layout da tela de cadastrar um *tweet*, vamos trocar o botão de publicar o *tweet* por um botão de **menu**. A primeira coisa a fazer é criar um arquivo de menu. Para isso, vamos clicar com o botão direito na pasta `res` do projeto, selecionar a opção `new` e por fim escolher `android resource file`.
  - Deixe o nome do arquivo como `tweet_menu.xml`
  - Mude o tipo do arquivo para `menu`

- Clique no botão `ok` para gerar o arquivo



2. Agora precisamos definir como será o nosso menu. O único item que queremos é o botão cadastrar, então vamos adicionar um item ao menu. Como ele vai aparecer na `Action Bar`, vamos gerar um novo ícone com o Android Studio da mesma forma que fizemos para o ícone do botão na lista e vamos dar a ele o nome `ic_save`. O xml do menu ficará conforme abaixo:

```
<menu xmlns:android="http://schemas.android.com/apk/res/android"
 xmlns:app="http://schemas.android.com/apk/res-auto">

 <item
 android:id="@+id/tweet_menu_cadastrar"
 android:title="Cadastrar"
 android:icon="@drawable/ic_save"
 app:showAsAction="always" />

</menu>
```

*Sugestão: para esse ícone, procure pela palavra "check".*

3. Para exibir o nosso menu na tela, precisamos sobrescrever o método `onCreateOptionsMenu` na classe `TweetActivity`:

```
override fun onCreateOptionsMenu(menu: Menu?): Boolean {
}
```

4. Agora precisamos *inflar* nosso xml de menu no momento da criação da nossa `Activity`:

```
override fun onCreateOptionsMenu(menu: Menu?): Boolean {
 menuInflater.inflate(R.menu.tweet_menu, menu)

 return true // deve mostrar ou não ?
}
```

5. Por fim, precisamos definir qual será o comportamento que o item do menu terá ao ser clicado. Fazemos isso sobrescrevendo outro método, nesse caso o `onOptionsItemSelected`:

```
override fun onOptionsItemSelected(item: MenuItem?): Boolean {
}
```

6. Dentro dele vamos verificar o item que foi clicado usando o `when`, que é bem similar ao `switch`

do Java:

```
override fun onOptionsItemSelected(item: MenuItem?): Boolean {
 when (item?.itemId) {
 R.id(tweet_menu_cadastrar -> {
 publicaTweet()
 finish()
 }
 }
 return false
}
```

7. Como acabamos de adicionar a opção de publicar no menu, podemos remover o botão da tela que está embaixo do campo de texto. Vamos então alterar o arquivo `activity_tweet.xml` removendo o botão.
8. Além disso, precisaremos remover a busca que fizemos pela `view` do botão na nossa `TweetActivity`:

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

 val botao = findViewById<Button>(R.id.criar_tweet)

 botao.setOnClickListener(View.OnClickListener{
 publicaTweet()
 finish()
 })
}
```

9. Rode o aplicativo e clique no botão para ir pra tela de inclusão. Você verá uma tela com um ícone no canto direito superior. Ao clicar no ícone de salvar, vamos ver um `Toast` aparecer com o texto digitado no campo de texto, que é a nossa lógica atual de publicar um *tweet*, e retornaremos para a tela de listagem.
10. Vamos aproveitar e adicionar também um botão de voltar à barra de menu do aplicativo para que o usuário possa voltar à tela de lista sem ter que publicar um *tweet*. Fazemos isso invocando o método `setDisplayHomeAsUpEnabled` na referência para a barra de menu:

```
class TweetActivity : AppCompatActivity() {

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

 supportActionBar?.setDisplayHomeAsUpEnabled(true)
 }
```

```
 }

 //demais métodos
}
```

11. Precisamos agora definir o comportamento do botão de voltar. Como ele também está na `ActionBar`, iremos tratá-lo como se ele fosse um item de menu:

```
override fun onOptionsItemSelected(item: MenuItem?): Boolean {

 when (item?.itemId) {

 R.id(tweet_menu_cadastrar -> {

 publicaTweet()

 finish()

 }

 android.R.id.home -> finish()

 }

 return false
}
```

12. Rode o aplicativo novamente, vá pra tela de novo *tweet* e teste o que acontece ao clicar no botão de voltar e também o que acontece ao publicar um *tweet*.

**Agora é a melhor hora de respirar mais tecnologia!**



Se você está gostando dessa apostila, certamente vai aproveitar os **cursos online** que lançamos na plataforma **Alura**. Você estuda a qualquer momento com a **qualidade** Caelum. Programação, Mobile, Design, Infra, Front-End e Business! Ex-aluno da Caelum tem 15% de desconto, siga o link!

[Conheça a Alura Cursos Online.](#)

# INTRODUÇÃO A ARQUITETURA NO ANDROID

## 4.1 O QUE SABEREI FAZER NO FIM DESSE CAPÍTULO ?

- Entender o que é um `data class`
- Conhecer os Architecture Components
- Trabalhar com Room
- Trabalhar com ViewModel
- Trabalhar com LiveData
- Conhecer o conceito de `companion object`
- Entender o conceito de `Repository`

### Editora Casa do Código com livros de uma forma diferente



Editoras tradicionais pouco ligam para ebooks e novas tecnologias. Não dominam tecnicamente o assunto para revisar os livros a fundo. Não têm anos de experiência em didáticas com cursos.

Conheça a **Casa do Código**, uma editora diferente, com curadoria da **Caelum** e obsessão por livros de qualidade a preços justos.

[Casa do Código, ebook com preço de ebook.](#)

## 4.2 MODELANDO O TWEET

Deixamos nossas telas comunicando entre si e os cliques funcionando, mas nem todos estão fazendo a lógica esperada. Ao clicar no ícone de salvar um *tweet*, o usuário espera que o *tweet* seja realmente armazenado em algum local para depois ser listado na tela seguinte. Para manipularmos um *tweet* na linguagem `Kotlin`, podemos guardar todas as características e comportamentos de um *tweet* em uma classe, modelando o que significa um *tweet* na nossa aplicação. Vamos criar a classe `Tweet` em

Kotlin :

```
class Tweet
```

## Propriedades em Kotlin

Queremos que essa classe tenha um atributo chamado `mensagem`. Geralmente criamos um atributo, também chamado de campo, com visibilidade privada e, para acessarmos esse campo, usamos o método `getter` e, possivelmente, o `setter`. A junção do campo com esses métodos é chamada em algumas linguagens, como o Java, de **propriedade**. O Kotlin, buscando encurtar o código que digitamos, substituiu a ideia dos *campos com métodos de acesso* pelas *propriedades*. Se a propriedade for **apenas de leitura**, a declaramos com o `val`, e o Kotlin vai gerar para a gente um *campo* e o método `getter`; se a propriedade for **para escrita**, a declaramos como `var`, e o Kotlin vai gerar para a gente um *campo* e os métodos `getter` e `setter`. A propriedade `mensagem`, que será apenas de leitura, ficará assim:

```
class Tweet {
 val mensagem: String
}
```

Mas precisamos garantir que todo objeto `Tweet` que for criado terá sempre uma `mensagem`. Fazemos isso com um construtor. No Kotlin, se quisermos definir apenas um construtor para nossa classe com a propriedade `mensagem`, usamos a seguinte sintaxe:

```
class Tweet(val mensagem: String)
```

## 4.3 MOSTRANDO A MENSAGEM DO TWEET SALVO

Agora usaremos esse modelo na nossa aplicação sempre que formos mexer com um `tweet`. Por exemplo, na hora de avisar ao usuário com o `Toast` que o `tweet` com a mensagem "Boa tarde!" foi publicado, podemos escrever o texto: "Tweet salvo: Boa tarde!".

Para enviar esse texto ao `Toast`, precisamos juntar um texto estático com a mensagem do `tweet`:

```
"Tweet salvo: " + tweet.mensagem
```

## Interpolação

Se continuarmos juntando mais textos estáticos com esse ou outros valores dinâmicos, precisamos abrir e fechar aspas e concatenar toda hora. Fica um pouco trabalhoso. Várias linguagens desenvolveram um recurso chamado **interpolação**, em que dentro do conteúdo estático é possível colocar valores dinâmicos, sem precisar encerrar a parte do texto. Em Kotlin, outra forma de escrever o código para o texto acima é:

```
"Tweet salvo: ${tweet.mensagem}"
```

## Convertendo para String

---

Toda vez que quisermos imprimir um *tweet*, precisamos pegar o valor da propriedade `mensagem` do objeto `Tweet`. Podemos acabar esquecendo de chamar a propriedade `e`, assim, a impressão  `${tweet}` nos traria a referência da memória. Isso porque a JVM vai chamar o método `toString` do objeto `tweet`. Como não há uma implementação desse método na classe `Tweet`, vai ser executada a implementação da classe `Any`. O ideal pra gente seria não precisar ficar chamando as propriedades necessárias para representar um *tweet*, mas isolar essa lógica no método `toString`, que tem justamente a função de transformar o objeto no tipo `String`.

## Classes de dados

No nosso caso, queremos que sejam usadas as propriedades de um *tweet* na hora de imprimí-lo. Como essa implementação para o `toString` é muito usada, assim implementações padrões para os métodos `hashcode` e `equals`, o Kotlin criou um modificador para representar as classes de dados, o `data`, que implementa em tempo de compilação justamente estes três métodos para a classe.

```
data class Tweet(val mensagem: String)
```

Com nossa classe `Tweet` declarada como acima, a impressão  `${tweet}` para um *tweet* com a mensagem "Boa tarde!" nos trará o texto: `Tweet(mensagem=Boa tarde!)`.

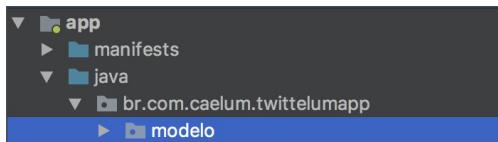
## 4.4 EXERCÍCIO: CRIANDO NOSSO PRIMEIRO MODELO DE MANEIRA SIMPLES

### Objetivo

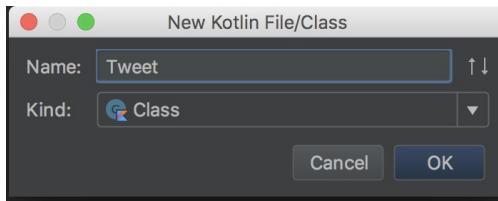
- Criar a classe `Tweet` para representar um *tweet* na nossa aplicação, contendo apenas uma `String` para a mensagem do *tweet*.
- Usar uma instância da classe `Tweet` para pegar o texto digitado pelo usuário no `EditText` e passá-lo para o `Toast` que já estamos mostrando na `TweetActivity`.

### Passo a passo

1. Crie o pacote `modelo` dentro do projeto



2. Dentro desse pacote crie a **classe** `Tweet`



3. Adicione uma propriedade na classe `Tweet` para representar a mensagem do tweet:

```
class Tweet(val mensagem: String)
```

4. Crie uma instância do `Tweet` no método `publicaTweet` :

```
private fun publicaTweet() {
 val campoDeMensagemDoTweet = findViewById<EditText>(R.id(tweet_mensagem)

 val mensagemDoTweet: String = campoDeMensagemDoTweet.text.toString()

 val tweet = Tweet(mensagemDoTweet)

 Toast.makeText(this, tweet.toString(), Toast.LENGTH_LONG).show()
}
```

5. Rode o aplicativo e tente cadastrar um tweet.

6. A mensagem que foi exibida no `Toast` foi a referência de memória. Para contornarmos isso, vamos usar um recurso do kotlin que facilitará bastante nossa vida. Aplicaremos o conceito de `data class`, com o qual ganharemos os métodos `toString`, `hashCode` e `equals`:

```
data class Tweet(val mensagem: String)
```

7. Rode novamente o aplicativo e veja o que apareceu no `Toast`.

8. Podemos melhorar mais um pouco nosso código usando a interpolação de `String`s, mais um recurso que o Kotlin nos oferece:

```
private fun publicaTweet() {

 // restante do código

 Toast.makeText(this, "$tweet", Toast.LENGTH_LONG).show()
}
```

9. Rode novamente o aplicativo e veja que continua aparecendo a mesma mensagem no `Toast`.

**Já conhece os cursos online Alura?**



A Alura oferece centenas de **cursos online** em sua plataforma exclusiva de ensino que favorece o aprendizado com a **qualidade** reconhecida da Caelum. Você pode escolher um curso nas áreas de Programação, Front-end, Mobile, Design & UX, Infra e Business, com um plano que dá acesso a todos os cursos. Ex aluno da Caelum tem 15% de desconto neste link!

[Conheça os cursos online Alura.](#)

## 4.5 ARMAZENANDO O TWEET

O usuário ainda está se frustrando na nossa aplicação pois ele clica pra salvar um *tweet* mas não vê esse *tweet* na tela de listagem. Isso porque o que estamos fazendo até agora é apenas mostrar o *tweet* para o usuário em um *Toast*. Daremos início a resolver esse problema agora. Para podermos mostrar na tela de listagem em qualquer momento o *tweet* criado, precisamos primeiro de fato armazená-lo em algum lugar. Geralmente, quando queremos salvar informações para recuperá-las depois, usamos **bancos de dados**. O Android já possui um banco de dados interno, o *SQLite*. Em muitas situações lidamos com uma imensa quantidade de dados, por isso o próprio Android recomenda esse banco apenas para testes ou para armazenar informações temporárias, por exemplo enquanto o dispositivo estiver *off-line*.

### Usando o SQLite

Nesse primeiro momento do curso, usaremos apenas o banco de dados interno, para depois explorarmos outras possibilidades. Podemos nos comunicar diretamente com o servidor de bancos de dados do dispositivo, mas, para isso, precisamos antes de tudo criar uma conexão com ele e gerenciá-la. Além disso, precisamos criar as tabelas para os dados que queremos armazenar. No momento em que formos implementar a lógica de salvar um *tweet* no banco de dados, também teremos que pegar cada uma das propriedades do objeto *tweet* e mapeá-lo para alguma coluna de uma tabela no banco de dados. Para listar, teremos o processo contrário para fazer: pegar cada coluna do banco de dados e mapeá-la para uma propriedade do *tweet*. Isso acontece porque estamos programando em um mundo voltado à Orientação a Objetos e o mundo do banco de dados que estamos utilizando é relacional. Porém, além da informação dos nomes e tipos das propriedades, todo o resto é o mesmo código de mapear as informações de um mundo para o outro independente de que modelo está sendo salvo.

O ideal seria então automatizar esse código que é igual sempre (*boilerplate*) e apenas configurar os

tipos e nomes. Surgiram várias ferramentas em várias linguagens de programação que fazem esse papel de mapear os objetos do mundo de orientação a objetos com o mundo relacional. Essas ferramentas receberam o nome de **ORM** (*Object Relational Mapping*). O mapeamento dessas ferramentas funciona da seguinte forma:

- cada classe corresponde a uma tabela no banco de dados
- cada atributo ou propriedade geralmente corresponde a uma coluna no banco de dados
- cada objeto corresponde a um registro no banco de dados

Para o `Android`, o **ORM** mais conhecido é o **Room**.

## Trabalhando com o Room

O **Room** é uma biblioteca externa que faz parte das **Componentes de Arquitetura do Android** (*Architecture Components*), conjunto de bibliotecas que auxiliam bastante o desenvolvedor a criar aplicações mais robustas, testáveis e de fácil manutenção. Ela vai justamente implementar essas lógicas que são sempre parecidas mas necessárias para a gente não precisar fazer a conversão entre os dois mundos na mão.

### Importando a biblioteca

A primeira coisa que precisamos fazer é importar a biblioteca do `Room`. Quando formos utilizar o `Room`, vamos apenas passar algumas configurações por anotações para ele, mas todo aquele código repetitivo, porém necessário, terá que ser criado em algum momento. Felizmente, quem fará isso não é a gente, mas o `Room`. Como ele vai criar código para a gente, essa geração de código terá que ser feita em tempo de compilação. Portanto, além da dependência com as classes relacionadas ao `Room`, precisamos importar uma segunda dependência, que terá a responsabilidade de processar as anotações relacionadas ao `Room`:

```
implementation 'androidx.room:room-runtime:2.2.1'
kapt 'androidx.room:room-compiler:2.2.1' // processador de anotações
```

Mas, para o código acima compilar, é necessário adicionarmos o plugin `kapt` ao mesmo arquivo `build.gradle` (Module: app) :

```
apply plugin: 'kotlin-kapt'
```

### Mapeando o Tweet

Vamos querer que o `Room` mapeie apenas algumas classes da nossa aplicação, por isso precisamos configurar para ele quais classes devem ser mapeadas, ou seja, quais classes serão consideradas **entidades**. Para essa configuração usamos a anotação `@Entity`:

```
@Entity
data class Tweet(val mensagem: String)
```

Quando o código for compilado, o `Room` já vai criar automaticamente o código para criar no banco de dados uma tabela de nome `Tweet` e coluna `mensagem` com o valor `TEXT`. Uma informação bem importante que nossa tabela precisa ter é como será feito para identificar cada registro unicamente. Geralmente criamos uma coluna com o nome `id` no banco de dados e avisamos que esta será a chave primária para a tabela, mas quem vai gerenciar o banco de dados é o `Room`. Precisamos então criar na nossa classe `Tweet` uma propriedade do tipo inteiro (`Int`) para identificar os objetos. Como essa propriedade deve ser imutável para um mesmo objeto, usaremos o `val` para declará-la. Em seguida, vamos avisar ao `Room` que ela será a chave primária com `@PrimaryKey`. Avisaremos também que quem vai gerenciar os valores desta chave primária será o banco de dados, usando o atributo `autoGenerate` do `@PrimaryKey`:

```
@PrimaryKey(autoGenerate = true) val id: Int
```

Quando buscarmos um *tweet* no banco de dados, vamos querer criar um objeto `Tweet` com o valor do `id` já preenchido, então seria legal já ter um construtor que consiga receber tanto a `mensagem` como o `id`. Para os momentos de inserção do *tweet*, já é preferível usar o construtor que já temos criado, apenas com a `mensagem`. Podemos sair criando um monte de construtores, usando a ideia de sobrecarga, mas a classe começa a ficar muito grande sem estarmos pensando necessariamente na lógica de negócio. Com `Kotlin`, essas sobrecargas podem ser evitadas especificando valores `default` para as propriedades que quisermos deixar "opcionais", ou seja, que quisermos ter a opção de não passar um valor a ela durante a criação de um objeto. A classe `Tweet` pode então ter no seu único construtor declarado as duas propriedades e deixar o `id` opcional de ser preenchido:

```
@Entity
data class Tweet(val mensagem: String,
 @PrimaryKey(autoGenerate = true) val id: Int = 0)
```

## Criando o DAO

Para cada modelo da nossa aplicação teremos várias lógicas para gerenciar os dados desse modelo. Para cada lógica, é necessário um método, e os métodos geralmente ficam em classes. Se esses métodos ficarem cada um em uma classe diferente, para alterarmos todas as lógicas referentes ao acesso aos dados de *tweets*, por exemplo, teremos que procurar esses métodos em várias classes diferentes, ou podemos acabar recriando alguns métodos por não lembrarmos que já existem, levando à repetição de código. Uma boa maneira de evitar essa confusão no código é deixar em apenas uma classe todas as formas de gerenciar o objeto *tweet*. Essa ideia de termos **objetos** específicos para **acessar os dados** da nossa aplicação viraram um padrão de projeto (*Design Pattern*) chamado **DAO** (*Data Access Object*). Na convenção sugerida pelo padrão, colocamos o nome do modelo seguido da sigla `Dao` ou `DAO` no nome da classe, para ficar fácil de identificar qual a responsabilidade principal dessa classe.

Como queremos gerenciar os dados relacionados ao `Tweet`, criaremos a classe `TweetDao`:

```
class TweetDao
```

Nessa classe teremos vários métodos para gerenciar o `Tweet` no banco de dados. Mas, se a classe é nossa, quem terá que implementar os métodos todos é a gente! Se lembremos do que decidimos agora há pouco, não queremos mais implementar esses métodos na mão. Ao invés disso, vamos pedir para o `Room` implementar para a gente esses métodos. A biblioteca precisa apenas saber quais lógicas deverão ser implementadas. Para isso, vamos definir quais métodos devem ser implementados por meio de uma interface e a biblioteca do `Room` que cuidará de criar uma classe que implementa essa interface que criamos:

```
interface TweetDao {
}
```

Nela podemos então apenas definir a assinatura dos métodos que queremos que existam:

```
interface TweetDao {
 fun armazena(tweet: Tweet)
}
```

Queremos que, dentro de várias interfaces que podemos ter no nosso projeto, o `Room` implemente apenas algumas. Configurando essas interfaces com a anotação `@Dao`, o `Room` saberá quais implementar. Feito isso, nossa classe vai ter vários métodos, mas, como a gente que tá criando a interface, não tem nada definindo o nome dos métodos. Não tem como o `Room` adivinhar qual processamento com o banco de dados queremos que ele implemente para cada método da interface. A forma que eles viram de resolver isso é, de novo, configurando por meio de anotações. A anotação `@Insert` em cima de um método do `Dao` indica ao `Room` que a lógica que será implementada é a lógica de inserção no banco de dados do objeto recebido como parâmetro:

```
@Dao
interface TweetDao {
 @Insert
 fun armazena(tweet: Tweet)
}
```

Agora precisamos forçar que o `Room` crie uma classe que implemente esta interface. Faremos isso pedindo um objeto do tipo `TweetDao`. Como provavelmente usaremos esse objeto em muitas classes da nossa aplicação e teremos outros `Dao`s para gerenciar outros modelos, vamos pedir para serem criados os objetos que seguem o *Design Pattern* do `Dao` em uma mesma classe, que vai possuir todos os objetos de acesso ao banco que iremos utilizar. Vamos chamá-la de `MeuBanco`.

Uma das formas de pedirmos a uma biblioteca algum objeto de qualquer tipo nosso é usando `reflection`, mas essa abordagem acaba deixando o celular lento. Outra abordagem é criarmos um método que retorne o nosso tipo. Por exemplo, a criação do objeto `TweetDao` na classe `MeuBanco` fica assim:

```
class MeuBanco {
 fun getTweetDao(): TweetDao {
 // lógica para criar um objeto tweetDao
 }
}
```

```
}
```

Mas, com esse código, nós temos que nos preocupar com criar uma classe que implemente TweetDao e instanciá-la. Só que é justamente esse processo que queremos que o Room faça pela gente! Para esse método não ter implementações, vamos deixá-lo **abstrato** para não nos preocuparmos mais com sua implementação:

```
class MeuBanco {
 abstract fun getTweetDao(): TweetDao
}
```

Quando uma classe tem um método abstrato, ela também precisa ser abstrata, portanto:

```
abstract class MeuBanco {
 abstract fun getTweetDao(): TweetDao
}
```

Agora precisamos criar uma classe concreta que herde dessa classe abstrata. Se continuarmos nesse caminho de sempre criar classes abstratas que nos deem objetos de outros tipos nossos, não vamos terminar nunca. Vamos olhar melhor para o que significa essa classe MeuBanco que criamos. Ela tem a responsabilidade de criar todos os nossos Dao's e, para isso, precisará saber lidar com a conexão com o banco de dados.

## Configurando o banco de dados

Os códigos necessários para o gerenciamento da conexão também são códigos repetitivos e felizmente o Room já os implementa para a gente com a classe RoomDatabase. Precisamos então que a classe que vai lidar diretamente com conexões herde a capacidade que a RoomDatabase tem de gerenciar as conexões. Vamos aproveitar a MeuBanco criada agora há pouco:

```
@Database
abstract class MeuBanco : RoomDatabase() {
 abstract fun getTweetDao(): TweetDao
}
```

Para o Room saber da existência dessa classe, vamos configurá-la com a anotação `@Database`. Podemos definir vários bancos de dados diferentes para a mesma aplicação, portanto precisamos avisar ao Room quais entidades podem ser acessadas quando for feita uma conexão com essa classe. Definimos isso dentro da anotação `@Database`, passando para o parâmetro `entities` um `array` com as referências das classes referentes às entidades. Para esse momento, essas referências devem ser em Kotlin, por exemplo, `Tweet::class`. Ao longo do tempo podem ocorrer mudanças na modelagem do banco de dados, então, caso algum dispositivo esteja em uma versão anterior do banco de dados e o aplicativo for atualizado, o Room precisa saber em que versão nosso banco está no momento. Fazemos isso com outro parâmetro na anotação `@Database`: o `version`.

```
@Database(entities = [Tweet::class], version = 1)
abstract class MeuBanco : RoomDatabase() {
 abstract fun getTweetDao(): TweetDao
```

```
}
```

Já definimos um monte de informações por meio da `MeuBanco`, mas ela ainda é uma classe abstrata, e ainda precisamos conseguir ter um objeto que possa ser referenciado por `MeuBanco`. Felizmente, a biblioteca que estamos usando nos ajuda com isso por meio do método `databaseBuilder` da classe `Room`. Esse método recebe:

- o contexto que será usado para o banco de dados; o banco vai trabalhar com informações da nossa aplicação, então precisamos falar quem somos
- a referência em java da classe abstrata que é filha de `RoomDatabase`; o `Room` vai usar essa informação para criar a classe concreta a partir da `MeuBanco` para a gente e nos dar uma instância dela
- o nome do banco que queremos que ele gerencie; para ele conseguir encontrar as tabelas, precisamos informá-lo qual é o banco

Esse método retorna um `RoomDabatase.Builder` e, para conseguirmos a nossa instância, precisamos chamar o método `build`:

```
Room.databaseBuilder(context, MeuBanco::class.java, "TwittelumBD").build()
```

Sempre que quisermos deixar nosso código mais claro e coeso, podemos criar um método para separar uma lógica que pode ser confusa:

```
fun criaBanco(contexto: Context): MeuBanco {
 return Room.databaseBuilder(contexto, MeuBanco::class.java, "TwittelumBD").build()
}
```

Agora precisamos colocar esse método em algum local. Podemos criar uma nova classe apenas para ele, mas esse método tem como única responsabilidade instanciar um objeto que possa ser representado por `MeuBanco`. No fundo, esse método é uma fábrica do tipo `MeuBanco`. Criar métodos com essa responsabilidade para encapsular códigos complexos ou trabalhosos é uma solução documentada por outro padrão de projeto, a **Factory**.

### Criando uma Factory

Faria mais sentido deixá-lo dentro da própria classe `MeuBanco`. Porém, se criarmos um método de objeto (o padrão) na classe, teremos que instanciá-la primeiro para chamar o método, mas o método é justamente para realizar a instanciação! Temos que criar um método que possa ser chamado direto da classe `MeuBanco`. Para fazer isso com `Kotlin`, criamos um **objeto companheiro** (*companion object*):

```
abstract class MeuBanco : RoomDatabase() {
 companion object {
 fun criaBanco(contexto: Context): MeuBanco {
 return Room.databaseBuilder(contexto, MeuBanco::class.java, "TwittelumBD").build()
 }
 }
}
```

Com o código acima, podemos chamar `MeuBanco.criaBanco` em qualquer lugar da nossa aplicação e teremos um objeto que possa ser representado por `MeuBanco`, ou seja, a própria classe está dizendo como ela deve ser criada.

#### Usando o Design Pattern Singleton

Parando pra pensar mais um pouco sobre esse objeto que gerencia o banco de dados, como o banco é o mesmo durante a execução do aplicativo, é recomendado que a instância também seja a mesma em todo o processo. Vamos então garantir que só será feita uma única instância do objeto, ou seja, que o método `criaBanco` só é chamado uma única vez. Para isso, vamos criar um novo método `getInstance`, convenção de nome para métodos responsáveis por nos dar uma instância, sem sabermos se é nova ou se é reutilizada. Esse método vai verificar se já há uma instância criada; se já tiver, vai retornar essa instância, senão vai criar uma chamando o método `criaBanco`. Como sua responsabilidade é na mesma linha que o `criaBanco`, deixaremos ele também dentro do **companion object**:

```
companion object {
 fun getInstance(context: Context): MeuBanco {
 // lógica de retornar a instância única
 }

 fun criaBanco(contexto: Context): MeuBanco {
 return Room.databaseBuilder(contexto, MeuBanco::class.java, "TwittelumBD").build()
 }
}
```

Para a lógica do `getInstance`, basicamente vamos criar uma variável e verificar se ela é nula. Se ela não for nula, usaremos ela; se ela for nula, chamaremos o método `criaBanco`. Com o `Kotlin`, podemos fazer isso sem `if`s e de maneira mais enxuta por meio do **Elvis operator** (`"?:"`):

```
companion object {
 private var banco: MeuBanco? = null

 fun getInstance(context: Context): MeuBanco {
 return banco ?: criaBanco(context)
 }
}
```

Pensando no caso em que `banco` está com valor `null`, o retorno será um objeto do tipo `MeuBanco`, mas o valor de `banco` ainda será `null`. Precisamos então, nesse caso, **além de** chamar o `criaBanco` e usar seu retorno, **também** atribuir o retorno deste método para a variável `banco`.

Fazemos isso com as **funções de escopo**, outro recurso do `Kotlin` que permite executar blocos de lógica dentro do contexto de um objeto. Chamamos essas funções a partir de um objeto e dentro delas temos acesso ao objeto em si, para processarmos lógicas com ele usando *expressão lambda*. Uma dessas funções de escopo é o `also`, que permite executar o método que quisermos "*e também fazer um determinado bloco de código*" com o retorno do método anterior. Dentro do `also` temos acesso ao

objeto pelo qual ele foi chamado por meio da palavra reservada `it` :

```
companion object {
 private var banco: MeuBanco? = null

 fun getInstance(context: Context): MeuBanco {
 return banco ?: criaBanco(context).also { banco = it }
 }
}
```

Agora nosso método `getInstance` está fazendo a lógica que queremos! Apenas chamando `MeuBanco.getInstance`, sempre conseguiremos a mesma instância de um objeto representado por `MeuBanco`. Essa solução de guardar em um método a garantia de que exista uma única instância de um tipo é chamada de ***Singleton***.

## Entendendo Threads no Android

O acesso ao banco de dados pode ser bem demorado e acabar travando a tela. Para evitar isso, devemos trabalhar com *threads*. A *thread* principal é sempre a que vai gerenciar a tela, por isso é chamada de ***UI Thread*** ou de ***Main Thread***. Sempre que formos fazer algum processo demorado, o recomendado é não usar a *UI Thread*. O `Room` já se previne para isso, por isso, quando usarmos o método `getTweetDao` na `TweetActivity` para salvar o `tweet` no banco de dados, o `Room` nem vai deixar inicializar a aplicação e receberemos a exceção `IllegalStateException` se não gerenciamos as *threads* de alguma forma.

Como estamos querendo apenas focar no funcionamento do `Room` nesse momento, vamos pedir pra ele autorizar acessos a esse banco de dados pela *thread* principal com o método `allowMainThreadQueries` durante a criação do banco:

```
fun criaBanco(contexto: Context): MeuBanco {
 return Room.databaseBuilder(contexto, MeuBanco::class.java, "TwittelumBD")
 .allowMainThreadQueries()
 .build()
}
```

## 4.6 EXERCÍCIO: SALVANDO O PRIMEIRO TWEET COM ROOM

### Objetivo

- Colocar a dependencia do Room na aplicação.
- Transformar a classe `Tweet` em uma entidade.
- Criar o `Dao` para salvar o `Tweet` no banco.
- Criar o `Database` para implementar o `Dao`.
- Criar a fábrica de database.

### Passo a passo

---

- Precisamos adicionar o plugin `kapt` no nosso projeto para conseguirmos usar bibliotecas que processam anotações em tempo de compilação. Vamos para o `build.gradle` (module: app) :

```
// outros plugins
apply plugin: 'kotlin-kapt'
```

- Agora que o plugin foi incluído na aplicação, vamos adicionar a dependência do Room no mesmo `build.gradle` :

```
dependencies {
 // demais dependencias

 implementation 'androidx.room:room-runtime:2.2.1' // Room
 kapt 'androidx.room:room-compiler:2.2.1' // leitor de anotações
}
```

- Vamos transformar nossa classe numa entidade. Para isso, usaremos a anotação `@Entity` :

```
import androidx.room.Entity

@Entity
data class Tweet(val mensagem: String)
```

- Toda entidade deve ter uma chave primária, então precisamos cadastrar uma em nossa entidade:

```
@Entity
data class Tweet(val mensagem: String,
 @PrimaryKey(autoGenerate = true) val id: Int = 0)
```

- Agora precisamos dar um jeito de persistir o `tweet` no banco. Para isso, vamos criar uma *interface* que vai possuir todos os métodos de acesso ao banco. Essa ideia é conhecida como padrão DAO (*Data Access Object*). Abaixo segue o código da nossa interface de DAO, que criaremos no novo pacote `bancodedados` :

```
@Dao
interface TweetDao {
```

- Em nosso DAO precisamos criar um método que será responsável por salvar um `Tweet` :

```
@Dao
interface TweetDao {

 @Insert
 fun salva(tweet: Tweet)

}
```

- Agora precisamos criar o banco de dados e deixar claro quais são nossas entidades e nossos daos. Para isso, vamos criar uma classe abstrata no mesmo pacote `bancodedados` :

```
@Database(entities = [Tweet::class], version = 1)
abstract class TwittelumDatabase : RoomDatabase() {

 abstract fun tweetDao(): TweetDao
```

```
}
```

8. Agora precisamos dar um jeito de termos uma classe concreta para essa classe abstrata. Faremos isso criando uma *Factory*. O Kotlin já nos dá suporte nativo para isso através do conceito de *Companion Object*:

```
@Database(entities = [Tweet::class], version = 1)
abstract class TwittelumDatabase : RoomDatabase() {

 abstract fun tweetDao(): TweetDao

 companion object {

 private var database: TwittelumDatabase? = null

 private val DATABASE = "TwittelumDB"

 fun getInstance(context: Context): TwittelumDatabase {
 return database ?: criaBanco(context).also { database = it }

 }

 private fun criaBanco(context: Context): TwittelumDatabase {
 return Room.databaseBuilder(context, TwittelumDatabase::class.java, DATABASE)
 .allowMainThreadQueries()
 .build()
 }
 }
}
```

9. Agora usaremos nosso banco para salvar nosso aluno alterando o método `publicaTweet` que já existe na classe `TweetActivity`:

```
private fun publicaTweet() {

 val campoDeMensagemDoTweet = findViewById<EditText>(R.id(tweet_mensagem)

 val mensagemDoTweet: String = campoDeMensagemDoTweet.text.toString()

 val tweet = Tweet(mensagemDoTweet)

 val tweetDao = TwittelumDatabase.getInstance(this).tweetDao()
 tweetDao.salva(tweet)

 Toast.makeText(this, "$tweet foi salvo com sucesso :D", Toast.LENGTH_LONG).show()
}
```

10. Rode o aplicativo e tente salvar algum `Tweet`.

## 4.7 PEGANDO A LISTA DO BANCO

Agora que estamos armazenando os *tweets* no banco, precisamos conseguir mostrar eles na tela de listagem para atendermos às expectativas do usuário. Da mesma forma que fizemos para o momento da inserção, faremos agora para a listagem: usaremos a classe especialista em acessar os dados de *tweets*, a

`TweetDao`. Como queremos ter a lógica de pegar os `tweets` do banco, precisamos ter nessa classe um método que nos devolva a lista de `tweets`:

```
@Dao
class TweetDao {
 fun lista(): List<Tweet>
}
```

Mas não queremos implementar na mão a lógica de pegar todos os registros do banco e, para cada registro, pegar os valores das colunas para colocar nas propriedades correspondentes. Em vez disso, estamos usando a biblioteca do `Room`. Para dizer que íamos fazer a inserção, usamos a anotação `@Insert`, mas infelizmente não há uma anotação pronta para a listagem. Precisaremos escrever pelo menos a consulta que queremos fazer para buscar todos os `tweets` do banco: `select * from Tweet`. Para conseguir associar essa consulta com o método `lista` que criamos, há uma anotação chamada `@Query`:

```
@Dao
interface TweetDao {
 @Query("select * from Tweet")
 fun lista(): List<Tweet>
}
```

### Saber inglês é muito importante em TI



cotidianas. Além disso, todas as aulas possuem explicações gramaticais, para você entender completamente o que está aprendendo. Aprender inglês é fundamental para o profissional de tecnologia de sucesso!

Na **Alura Língua** você reforça e aprimora seu inglês! Usando a técnica *Spaced Repetitions* o aprendizado naturalmente **se adapta ao seu conhecimento**. Exercícios e vídeos interativos fazem com que você pratique em situações

[Pratique seu inglês na Alura Língua.](#)

## 4.8 MOSTRANDO A LISTAGEM NA TELA

Precisamos lembrar que até o momento a lista que aparecia na tela era composta apenas por objetos `String`. Agora nossa lista é composta por objetos do tipo `Tweet`. Mas, no final, precisamos mesmo é de uma lista de `View`. Olhando a `ListaActivity`, a forma que fizemos para transformar a lista de

`String` em lista de `View` foi com o `ArrayAdapter`, informando o tipo dos objetos da nossa lista. Portanto, para fazer com que o `ArrayAdapter` saiba converter de uma lista de `Tweet` para uma lista de `View` precisamos especificar o tipo no *generics* do `ArrayAdapter`:

```
ArrayAdapter<Tweet>(this, android.R.layout.simple_list_item_1, tweets)
```

Como esse *layout* que estamos usando tem apenas um campo de texto por item, ele vai pegar cada elemento da lista e transformá-lo em texto, chamando o método `toString`.

## 4.9 EXERCÍCIO: EXIBINDO LISTAGEM COM ROOM

### Objetivo

- Fazer o `Dao` devolver uma lista de `Tweets`
- Colocar a lista do banco no `ListView`

### Passo a passo

1. Vamos criar no `Dao` um método que liste todos os tweets:

```
@Dao
interface TweetDao {

 @Insert
 fun salva(tweet: Tweet)

 @Query("select * from Tweet")
 fun lista(): List<Tweet>

}
```

2. Agora será necessário usar essa nova função em nossa `ListaActivity` ao invés de usarmos a lista estática:

```
class ListaActivity : AppCompatActivity() {

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

 // delete esse código

 val tweets: List<String> = listOf("Meu primeiro tweet", "Meu segundo tweet", "Meu terceiro
 tweet", "Meu quarto tweet", "Meu quinto tweet")

 // adicione esse código

 val tweetDao = TwittelumDatabase.getInstance(this).tweetDao()

 val tweets: List<Tweet> = tweetDao.lista()

 // mude o tipo do adapter
```

```
 lista_tweet.adapter = ArrayAdapter<Tweet>(this, android.R.layout.simple_list_item_1, tweets
)

 // restante do código
}
}
```

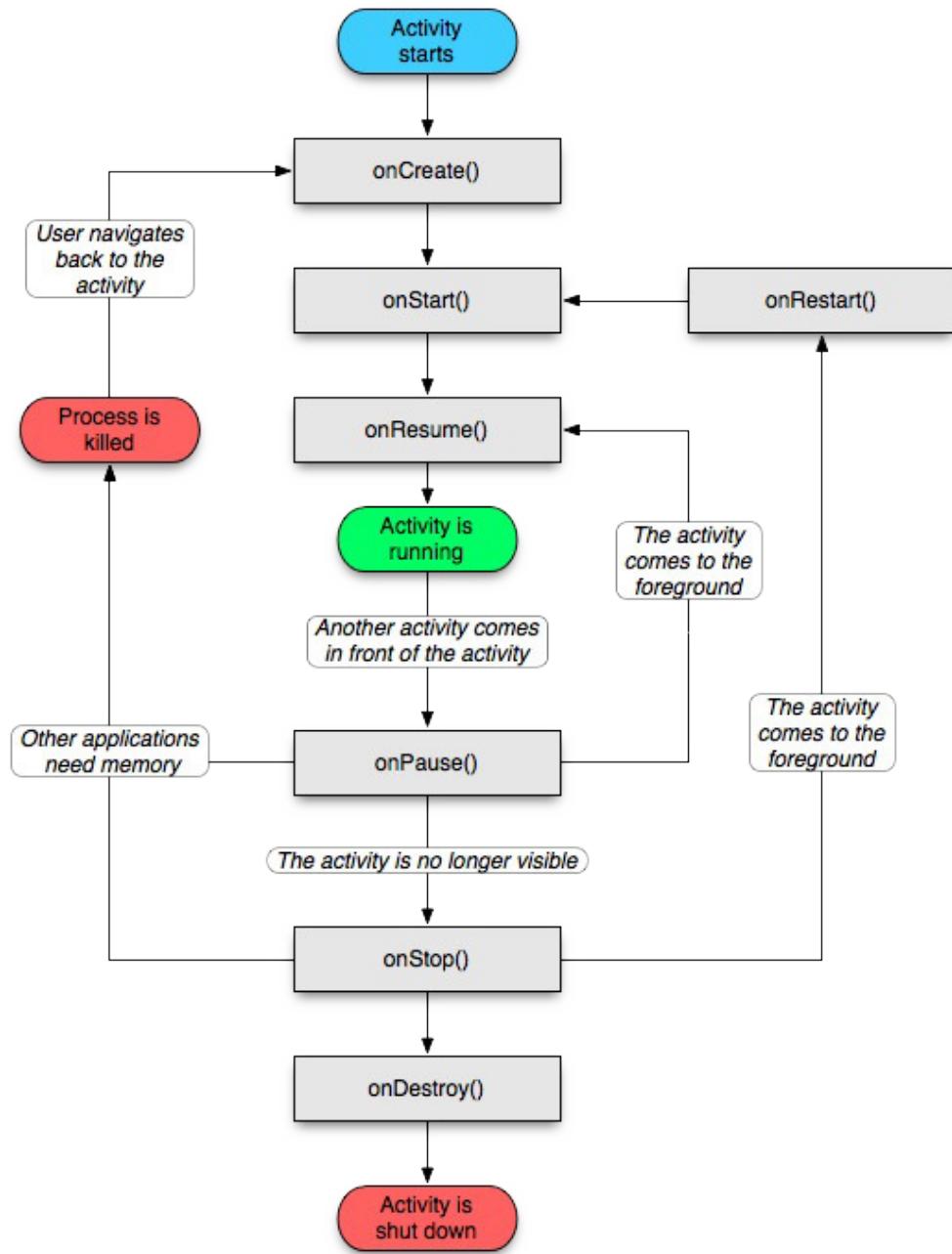
3. Rode a aplicação. Veja se a lista aparece com os tweets que já foram cadastrados e tente cadastrar um novo tweet.

## 4.10 MOSTRANDO A LISTA SEMPRE ATUALIZADA

Agora toda vez que o usuário abrir nossa aplicação, ele vai ver os *tweets* existentes no banco de dados naquele momento. Aí o usuário vai clicar para criar um novo *tweet*, vai digitar alguma mensagem e clicar na confirmação. Quando ele fizer isso, ele vai voltar para a tela de listagem, mas não vai ver ainda o *tweet* recém-criado. Se ele encerrar o aplicativo e reabri-lo, aí ele vai conseguir ver esse novo *tweet*. Isso acontece por causa do **ciclo de vida** que coordena todas as `Activity`s no Android .

### Ciclo de Vida de uma Activity

Toda `Activity` tem o ciclo de vida representado na figura a seguir. Sempre que quisermos adicionar algum comportamento a algum momento do ciclo, podemos sobrescrever estes métodos quando criarmos nossas próprias Activities.



**onCreate()** Chamado quando a Activity é criada. É onde se criam as Views, faz chamadas de banco de dados. Quando executado, o sistema recebe um Bundle com o estado da última execução da atividade.

**onStart()** Chamado antes da Activity ficar visível na tela. Se der tudo certo, vai para `onResume()`, senão, para `onStop()`.

**onResume()** Chamado após o `onStart()` se a sua Activity for para primeiro plano. Neste ponto, você está interagindo com o usuário. É aqui que sua aplicação pode iniciar ou retornar as ações

necessárias para atualizar as interfaces com o usuário.

**onPause()** Acontece quando o Android chama uma atividade diferente. Você perde os direitos da tela, então é o momento de parar animações e tudo que consuma recursos do sistema e bateria.

**onStop()** Chamado quando outra atividade obteve o primeiro plano, ou quando sua atividade está sendo eliminada.

**onDestroy()** Última oportunidade da sua aplicação fazer alguma coisa antes de ser eliminada. Lembre-se que este método pode ser chamado quando o Android precisa de recursos, ou porque o usuário optou por finalizar a aplicação.

Quem coordena o ciclo de vida de toda `Activity` é sempre o `Android`.

## Entendendo a `ListaActivity`

Voltando para a nossa aplicação, a lista está sendo carregada e a `ListView` está sendo preenchida no método `onCreate`, ou seja, no processo de criação da `ListaActivity`. Quando o usuário clica no botão flutuante, para ele, nossa aplicação apenas muda de tela. Internamente, o `Android` executou os métodos `onPause` e `onStop` da `ListaActivity` e, em seguida, os métodos `onCreate`, `onStart` e `onResume` da `TweetActivity`.

Em um segundo momento, quando ele clica para salvar o `tweet`, para o usuário, ele pode estar voltando para a tela anterior. Internamente, o `Android` executou os métodos `onPause`, `onStop` e `onDestroy` da `TweetActivity` e, em seguida, os métodos `onStart` e `onResume` da `ListaActivity`. Então, o método `onCreate`, no qual está o código de carregar a lista, não foi chamado.

O melhor a fazer é colocar toda a lógica de carregar a lista do banco e preencher a `ListView` no `onStart` ou no `onResume`. Para garantir que a tela estará atualizada o máximo possível, colocaremos no método `onResume`, pois é o momento em que a tela recebe o foco das ações do usuário.

**Aprenda se divertindo na Alura Start!**



Você conhece alguém que tem potencial para tecnologia e programação, mas que nunca escreveu uma linha de código? Pode ser um filho, sobrinho, amigo ou parente distante. Na **Alura Start** ela vai poder criar games, apps, sites e muito mais! É o **começo da jornada com programação e a porta de entrada para uma possível carreira de sucesso**. Ela vai estudar em seu próprio ritmo e com a melhor didática. A qualidade da conceituada Alura, agora para Starters.

[Conheça os cursos online da Alura Start!](#)

## 4.11 EXERCÍCIO: EXIBINDO LISTAGEM DE MANEIRA CORRETA

### Objetivo

Faça a busca ser processada no método `onResume`

### Passo a passo

- Precisamos fazer a busca ser executada em outro método do ciclo de vida:

```
class ListaActivity : AppCompatActivity() {

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

 // tire esse código daqui

 val tweetDao = TwittelumDatabase.getInstance(this).tweetDao()

 val tweets: List<Tweet> = tweetDao.lista()

 lista_tweet.adapter= ArrayAdapter<Tweet>(this, android.R.layout.simple_list_item_1, tweets)

 // restante do código
 }

 override fun onResume() {
 super.onResume()

 val tweetDao = TwittelumDatabase.getInstance(this).tweetDao()

 val tweets: List<Tweet> = tweetDao.lista()
```

```

 lista_tweet.adapter = ArrayAdapter<Tweet>(this, android.R.layout.simple_list_item_1, tweets
)
}

}

```

## 4.12 ESTRUTURANDO NOSSA APLICAÇÃO COM AS COMPONENTES DE ARQUITETURA

Atualmente nossa aplicação está armazenando e buscando os *tweets* no banco de dados local, mas, como já comentamos, o próprio *Android* não recomenda essa abordagem. Em casos de aplicativos como o *Facebook*, a quantidade de dados para mostrar ao usuário é enorme, e armazenar todas essas informações no dispositivo do usuário fica inviável. Além disso, muitas vezes a informação precisa ser compartilhada por muita gente. Por isso, geralmente as aplicações *mobile* utilizam algum servidor ou API para armazenar as informações necessárias. Em outras situações, por exemplo para manter algumas funcionalidades enquanto o dispositivo está *off-line*, há o suporte para armazenar e buscar dados tanto em uma API quanto no banco de dados local.

Pensando na nossa aplicação, se quisermos mudar para usar uma API em vez do banco de dados local, teremos que alterar em **todos** os lugares em que estamos chamando os *daos*. Teremos que chamar novas classes com outros métodos e precisaremos excluir todas as linhas de código relacionadas a banco de dados nas nossas *Activity*s. Além disso, nossas *Activity*s estão tendo que saber em que lugar estão sendo salvos os *tweets*, sendo que a responsabilidade principal de uma *Activity* é definir uma tela e suas interações com o usuário.

### A camada Repository

Para resolver os problemas que surgiram na manutenção do nosso código e o excesso de responsabilidade para as *Activity*, vamos isolar a parte do código que define como serão buscados e armazenados os dados em outra classe. Essa classe terá como única função gerenciar como serão feitos os acessos e alterações nos dados da nossa aplicação. No nosso caso, em que temos apenas uma forma de gerenciar *tweets*, essa classe vai ser uma camada a mais na nossa aplicação para encapsular a forma como estão sendo gerenciados os dados dela. Essa camada é chamada de **Repository**. Vamos criar uma classe chamada *TweetRepository*. Como ela vai chamar as lógicas do *TweetDao*, vamos pedir um objeto desse tipo no construtor. Dentro dela vamos criar métodos para encapsular os métodos do *TweetDao*:

```

class TweetRepository(private val fonteDeDados: TweetDao) {
 fun lista() = fonteDeDados.lista()
 fun salva(tweet: Tweet) = fonteDeDados.salva(tweet)
}

```

Agora nossa *ListaActivity* vai acessar a camada do repositório em vez de acessar direto os métodos do *TweetDao*:

```

class ListaActivity : AppCompatActivity() {
 override fun onResume() {
 super.onResume()

 val tweetDao = TwittelumDatabase.getInstance(this).tweetDao()
 val tweetRepository = TweetRepository(tweetDao)

 val tweets: List<Tweet> = tweetRepository.lista()

 lista_tweet.adapter = ArrayAdapter<Tweet>(this, android.R.layout.simple_list_item_1, tweets)
 }
}

```

## O contexto da aplicação

Já que estamos pensando nas melhorias para a nossa aplicação, se observarmos nossa `TwittelumDatabase`, ela tem a responsabilidade de gerenciar o banco de dados da nossa aplicação, então faz mais sentido enviar o contexto da aplicação, em vez de o contexto específico da `Activity`. Se olharmos no arquivo `AndroidManifest.xml`, veremos que há uma tag chamada `application`, então o Android já gerencia para a gente esse contexto criando um objeto do tipo `Application`. Como toda `Activity` só é criada se está em uma aplicação, podemos acessar o objeto `Application` a partir de uma `Activity`, chamando a propriedade `application` dentro da `Activity`:

```

val tweetDao = TwittelumDatabase.getInstance(this.application).getTweetDao()
val tweetRepository = TweetRepository(tweetDao)

```

## A camada ViewModel

Deixamos nossa aplicação mais estruturada e semântica, mas vamos trabalhar um pouco mais nesses pontos. Já comentamos que a responsabilidade principal de uma `Activity` é gerenciar a tela, o *layout* e as interações com o usuário. Mas, olhando pra nossas `Activity`s, estamos acessando o banco de dados com o código:

```
TwittelumDatabase.getInstance(this.application)
```

O ideal seria nossa `Activity` não processar nenhuma lógica, mas **delegar para outras camadas** todas as lógicas que tiverem que serem realizadas.

Quando diz respeito a **dados**, como nesse caso da busca e inserção de `tweets` que estamos analisando agora, há uma outra coisa que temos que pensar sempre. Entenderemos melhor com a seguinte situação: se o usuário estiver vendo a lista de `tweets` e girar a tela, pode ser que os dados demorem para aparecer. Isso porque, quando há uma rotação da tela, o *layout* para o modo **paisagem** pode ser completamente diferente do *layout* para o modo **retrato**, então o Android destroi totalmente a `Activity` e cria novamente ela, passando por todos os métodos para destruir e para criar uma `Activity` conforme vimos sobre o **ciclo de vida**. Quando é executado novamente o método `onResume`, está sendo realizada uma nova consulta no banco de dados para pegar a lista de `tweets`. Mas, se apenas a tela do aplicativo foi

rodada e os *tweets* estão sendo gerenciados apenas localmente, é bem difícil que um novo *tweet* tenha sido criado.

O ideal seria a gente armazenar em algum lugar a lista de *tweets* antes da `Activity` ser destruída e pegar essa lista de volta quando a `Activity` for novamente criada. Mas, para isso, teremos que sobrescrever vários métodos do ciclo de vida e gerenciar essa lista de *tweets* pensando no ciclo de vida da `ListaActivity`.

Para entendermos melhor esse problema, vamos pensar em aplicativos de vídeo como o YouTube: enquanto a tela está em primeiro plano e ativa, o vídeo está executando. Porém, quando saímos da tela, o vídeo para e, quando voltamos para a tela, o vídeo volta no exato momento em que estava quando saímos da tela. Para que esta funcionalidade ocorra como planejado, precisaremos armazenar, por exemplo, no `onPause`, o instante em que o vídeo estava, e recuperar essa informação por exemplo no `onResume`, para sabermos a partir de que instante do vídeo executar.

Felizmente, o `Android` já disponibiliza uma classe capaz de armazenar e gerenciar dados relacionados à tela pensando no ciclo de vida das `Activity`s: a `ViewModel`. Essa classe vem de uma biblioteca externa, mas do `Android`, que possui justamente uma série de classes para nos dar suporte ao ciclo de vida das `Activity`s, chamada:

```
'androidx.lifecycle:lifecycle-extensions:2.1.0'
```

Para a nossa nova camada, criaremos uma classe que terá os comportamentos e características que a `ViewModel` possui e que irá chamar todas as lógicas do `TweetRepository`:

```
class TweetViewModel(private val repository: TweetRepository) : ViewModel() {

 fun lista() = tweetRepository.lista()

 fun salva(tweet: Tweet) = tweetRepository.salva(tweet)

}
```

Agora nossa `ListaActivity`, ao invés de chamar a camada `Repository`, vai chamar a camada `ViewModel`:

```
class ListaActivity : AppCompatActivity() {
 override fun onResume() {
 super.onResume()

 val tweetDao = TwittelumDatabase.getInstance(this.application).getTweetDao()
 val tweetRepository = TweetRepository(tweetDao)
 val tweetViewModel: TweetViewModel = TweetViewModel(tweetRepository)

 val tweets: List<Tweet> = tweetViewModel.lista()

 lista_tweet.adapter = ArrayAdapter<Tweet>(this, android.R.layout.simple_list_item_1, tweets)
 }
}
```

No entanto, se a gente instanciar um objeto `TweetViewModel` dentro de algum método do ciclo de vida de uma `Activity`, a gente que tá tendo o controle de sua existência. Ao invés disso, vamos deixar a propriedade `viewModel` da `ListaActivity` fora de qualquer método do ciclo de vida. Além disso, vamos pedir a alguém para nos prover um objeto `TweetViewModel` e, assim, gerenciar este objeto para a gente. Quem pode fazer isso é a classe `ViewModelProviders`. Ela vai nos ajudar a conseguir um objeto `TweetViewModel`, descobrindo se é preciso criar um novo objeto ou nos devolver algum objeto já criado.

Mas, para esse provedor conseguir criar um objeto `TweetViewModel` quando necessário, precisamos ensiná-lo como se cria um objeto desse tipo definindo uma fábrica (**Factory**) de `TweetViewModel`. Como quem vai chamar o método da nossa fábrica é a `ViewModelProviders`, ela precisa ter certeza que o objeto `TweetViewModel` implementa o método que ela vai chamar. Isso é possível por meio de *interfaces*, no nosso caso, a **interface** `ViewModelProvider.Factory`:

```
class ViewModelFactory : ViewModelProvider.Factory {

 override fun <T : ViewModel?> create(modelClass: Class<T>): T = TweetViewModel(tweetRepository) as T

}
```

O construtor de `TweetViewModel` que temos precisa de um `tweetRepository`, que criaremos usando a mesma lógica que estava na `Activity` antes:

```
class ViewModelFactory : ViewModelProvider.Factory {

 override fun <T : ViewModel?> create(modelClass: Class<T>): T {

 val tweetDao = TwittelumDatabase.getInstance(this.application).getTweetDao()
 val tweetRepository = TweetRepository(tweetDao)

 return TweetViewModel(tweetRepository) as T
 }
}
```

Porém, pela `ViewModelFactory`, não temos acesso ao objeto `Application` usando o `this`. Precisaremos ver outra maneira de ter acesso à `Application`. Queremos acessar a instância de `Application` de qualquer classe da nossa aplicação, ou seja, queremos ter uma lógica que nos dê essa instância sempre que quisermos. A melhor classe para gerenciar a instância de `Application` é uma `Application`. Assim como pra definir uma `Activity` com lógicas específicas criamos uma classe que estenda `Activity`, podemos fazer o mesmo para uma `Application`:

```
class TwittelumApplication : Application()
```

Nessa classe criaremos a lógica de nos dar a instância de `TwittelumApplication`, mas, para fazermos isso sem criar uma nova instância, esse método terá de ser estático. Criaremos novamente um *objeto companheiro* que terá o método `getInstance`:

```

class TwittelumApplication : Application() {

 companion object {
 fun getInstance() : TwittelumApplication
 }
}

```

Esse método deverá devolver a instância de `TwittelumApplication`, mas dentro do *objeto companheiro* não temos acesso direto a uma instância de `TwittelumApplication`. Por isso, criaremos um objeto do tipo dessa classe dentro do *objeto companheiro*:

```

class TwittelumApplication : Application() {

 companion object {
 private val instance: TwittelumApplication

 fun getInstance() = instance
 }
}

```

Mas essa variável não pode ser inicializada já com a instância, portanto ela deverá ter o valor `null` nesse momento e só dentro de algum método da `Application` podemos inicializá-la. A `Application` possui o método `onCreate`, que é logo após sua inicialização. Usaremos esse método para capturarmos a instância da nossa `TwittelumApplication`:

```

class TwittelumApplication : Application() {

 override fun onCreate() {
 super.onCreate()
 instance = this
 }

 companion object {
 private var instance: TwittelumApplication? = null

 fun getInstance() = instance
 }
}

```

Para não deixar a propriedade `instance` ser mutável e aceitar valores `null`, não podemos usar *lazy initialization* porque teríamos que conseguir acessar diretamente a instância de `TwittelumApplication` no *objeto companheiro*, o que já vimos não ser possível. Felizmente, há outra técnica de inicialização que o `Kotlin` nos fornece: a ***inicialização atrasada (lateinit)***:

---

```

class TwittelumApplication : Application() {

 override fun onCreate() {
 super.onCreate()
 instance = this
 }

 companion object {
 private lateinit var instance: TwittelumApplication

 fun getInstance() = instance
 }
}

```

---

```
}
```

Por fim, como criamos uma classe específica para nossa aplicação, precisamos pedir ao SO do dispositivo usar a `TwittelumApplication` para criar um objeto para nossa aplicação, ao invés de se basear na classe `Application`. Fazemos isso registrando nossa classe no `AndroidManifest.xml`, passando para a tag `<application>` o nome da nossa classe:

```
<application android:name=".application.TwittelumApplication">
```

Como a única função dessa classe é ser uma fábrica de objetos de outro tipo, não é muito útil criarmos uma nova instância de `ViewModelFactory` sempre que quisermos um novo `TweetViewModel`. Podemos ter apenas uma instância da nossa fábrica que já será suficiente, ou seja, podemos ter um `Singleton` da `ViewModelFactory`. Com o `Kotlin`, podemos fazer com que nossa classe siga o padrão `Singleton` usando o termo `object`:

```
object ViewModelFactory : ViewModelProvider.Factory {
 override fun <T : ViewModel?> create(modelClass: Class<T>): T = TweetViewModel() as T
}
```

Agora, na `ListaActivity`, podemos apenas pedir ao `ViewModelProviders` um objeto do tipo `TweetViewModel`. Usaremos o `TweetViewModel` em vários métodos, mas o `ViewModelProviders` precisa receber a instância da activity para criar o `ViewModel`, o que só será possível em um método:

```
class ListaActivity : AppCompatActivity() {

 private var viewModel: TweetViewModel

 override fun onCreate() {
 viewModel = ViewModelProviders.of(this, ViewModelFactory).get(TweetViewModel::class.java)
 }

 // restante do código
}
```

Isso vai nos forçar a deixar o `viewModel` ser mutável e precisaremos inicializá-la com um valor, que será `null`, ou seja, essa *propriedade* terá que ser mutável e aceitar valores nulos, justamente o contrário do que queremos pra essa *propriedade*. Para deixar nosso código melhor, há uma técnica de inicialização diferente que a linguagem de Kotlin nos permite fazer: a ***inicialização preguiçosa (lazy initialization)***. Com essa técnica, definimos qual o código que será executado para inicialização da variável no momento da sua declaração, mas esse código só será executado na primeira vez que a propriedade for utilizada, ou seja, a propriedade só será inicializada no momento de seu primeiro uso. Nas próximas vezes que for usada, o valor será o mesmo:

```
class ListaActivity : AppCompatActivity() {

 private var viewModel: TweetViewModel by lazy {
 ViewModelProviders.of(this, ViewModelFactory).get(TweetViewModel::class.java)
 }
```

```
// restante do código
}
```

Com isso, garantimos que o objeto do tipo `TweetViewModel` só será criado se realmente for necessário, e conseguimos armazená-lo em uma propriedade imutável.

## A camada LiveData

Agora nossa tela de listagem vai pegar os *tweets* do banco e vai manter a lista mesmo se a tela for girada. O mesmo valerá para a tela de criação de *tweet*, na qual também usaremos o `TweetViewModel` para gerenciar o *tweet* que será salvo.

Mas, se o usuário estiver na tela de lista com três *tweets*, for para a tela de criação *tweet* e criar um novo *tweet*, ao voltar para a tela de lista, vai continuar aparecendo apenas três *tweets*, como anteriormente. O *tweet* recém-criado não vai aparecer na tela de lista. Isso porque a tela de listagem é a mesma de antes, portanto a lista de *tweets* que o `TweetViewModel` armazena vai continuar igual.

Para o usuário ver a lista atualizada, ele precisa sair da tela de listagem e voltar para ela, o que não é nada prático. O ideal é que quem está armazenando a lista, no caso, o `TweetViewModel`, perceba que um *tweet* foi adicionado e atualize sua lista. Para isso, o `TweetViewModel` precisa **observar** quando um *tweet* for salvo no banco para aí então adicionar esse *tweet* à sua lista. Há um **Design Pattern** chamado **Observer** que segue justamente essa ideia. Como é muito comum um `ViewModel` observar alterações nos dados da aplicação, o `Android` já criou uma classe que pode ser *observada*: `LiveData`.

O `Room` já se integra bem com essa classe, podendo retornar o tipo `LiveData<List<Tweet>>` ao invés de apenas `List<Tweet>` para o método `lista` na classe `TweetDao`:

```
@Dao
interface TweetDao {

 @Query("select * from Tweet")
 fun lista(): LiveData<List<Tweet>>

}
```

Com isso, o método `lista` do `TweetViewModel` também vai retornar um `LiveData<List<Tweet>>`, permitindo que essa lista seja observável:

```
class TweetViewModel(private val repository: TweetRepository) : ViewModel() {

 fun lista(): LiveData<List<Tweet>> = repository.lista()
}
```

[melhorar] Assim, o `Room` vai avisar através desse "dado vivo" que foi adicionado um *tweet* e, portanto, a lista de *tweets* dentro do `LiveData` no `TweetViewModel` será atualizada. Como o `LiveData` pode ser observado, vamos chamar o método que permite observá-lo:

```
viewModel.lista().observe()
```

Esse método pede um objeto que possua um ciclo de vida no `Android` e uma implementação da interface `Observer`. Essa interface define o método `onChanged()`, que será chamado caso ocorra alguma mudança no objeto que o `LiveData` guarda e recebe o objeto já atualizado.

Como a lógica que queremos fazer ao alterar a lista de `tweets` no banco é atualizar a `ListView` com os `tweets`, precisamos criar uma implementação de `Observer`:

```
Observer { tweets ->
 lista_tweet.adapter = ArrayAdapter(this, android.R.layout.simple_list_item_1, tweets)
}
```

Lembrando que, como essa interface possui um único método, o `onChanged()`, não é necessário escrevê-lo no código; a linguagem já infere que a lógica dentro das chaves é desse método. Esse método recebe a lista de `tweets` atualizada, portanto podemos utilizá-la para passar ao construtor do `ArrayAdapter`. Passando uma instância da implementação da interface `Observer` ao método `observe`, nosso código fica assim:

```
viewModel.lista().observe(Observer { tweets ->
 lista_tweet.adapter = ArrayAdapter(this, android.R.layout.simple_list_item_1, tweets)
})
```

Como queremos observar o `LiveData` com a lista de `tweets` desde o momento em que a `ListaActivity` for criada, podemos chamar o método `observe` dentro do `onCreate`.

## 4.13 EXERCÍCIO: DEIXANDO NOSSO SISTEMA MAIS MALEÁVEL A MUDANÇAS

### Objetivo

- Isolar todo o acesso ao banco no domínio do `Repository`
- Criar a classe `TwittelumApplication` para prover um contexto ao `Repository`
- Utilizar `LiveData` na busca do banco
- Isolar o acesso do `Repository` no `ViewModel`
- Usar o `ViewModel` nas Activities

### Passo a passo

1. Adicione no `build.gradle` a dependência para o `ViewModel` e o `LiveData`:

```
dependencies {
 // ViewModel and LiveData
 implementation 'androidx.lifecycle:lifecycle-extensions:2.1.0'
}
```

2. Crie a classe `TweetRepository` no pacote `bancodedados` :

```
class TweetRepository(private val fonteDeDados: TweetDao) {

 fun lista() = fonteDeDados.lista()

 fun salva(tweet: Tweet) = fonteDeDados.salva(tweet)

}
```

3. Altere no `TweetDao` o retorno do método `lista` em nosso `Dao` :

```
@Dao
interface TweetDao {

 @Insert
 fun salva(tweet: Tweet)

 @Query("select * from Tweet")
 fun lista(): LiveData<List<Tweet>>

}
```

4. Crie a classe `TweetViewModel` em um novo pacote `viewmodel` :

```
class TweetViewModel(private val repository: TweetRepository) : ViewModel() {

 fun lista() = repository.lista()

 fun salva(tweet: Tweet) = repository.salva(tweet)

}
```

5. Agora será necessário criar uma fábrica do nosso `ViewModel`, também no pacote `viewmodel` :

```
object ViewModelFactory : ViewModelProvider.Factory {

 private val database = TwittelumDatabase.getInstance(TwittelumApplication.getInstance())

 private val tweetRepository = TweetRepository(database.tweetDao())

 override fun <T : ViewModel?> create(modelClass: Class<T>): T = TweetViewModel(tweetRepository)
 as T

}
```

Esse código ainda não vai compilar.

6. Para o `ViewModelFactory` compilar, será necessário criarmos a classe `TwittelumApplication` no pacote `application` :

```
class TwittelumApplication : Application() {

 override fun onCreate() {
 super.onCreate()
 instance = this
 }

 companion object {
```

```

 private lateinit var instance: TwittelumApplication

 fun getInstance() = instance
}
}

```

7. Agora que temos uma classe que representa a `Application`, precisamos deixar isso claro ao Android, registrando-a no `AndroidManifest`:

```

<application
 android:name=".application.TwittelumApplication">
 // restante

```

8. Agora precisamos usar nosso `ViewModel` em nossas `Activities`, vamos começar pela listagem:

- crie uma propriedade que será inicializada em breve:

```

class ListaActivity : AppCompatActivity() {

 private val viewModel: TweetViewModel by lazy {
 ViewModelProviders.of(this, ViewModelFactory).get(TweetViewModel::class.java)
 }

 // restante do código
}

```

- Remova o método `onResume` dessa activity
- Vamos ficar observando agora nossa listagem:

```

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

 viewModel.lista().observe(this, observer())

 // restante do código
}

```

O código acima ainda não vai compilar.

- Crie o método `observer`:

```

private fun observer(): Observer<List<Tweet>> {
 return Observer {
 lista_tweet.adapter = ArrayAdapter(this, android.R.layout.simple_list_item_1, it)
 }
}

```

9. Agora precisamos fazer pequenas alterações em nossa `TweetActivity`:

- crie uma propriedade que será inicializada em breve:

```

class TweetActivity : AppCompatActivity() {

 private val viewModel: TweetViewModel by lazy {
 ViewModelProviders.of(this, ViewModelFactory).get(TweetViewModel::class.java)
}

```

```
 }

 // restante do código
}
```

- Vamos alterar a forma de publicar o tweet:

```
private fun publicaTweet() {

 val campoDeMensagemDoTweet = findViewById<EditText>(R.id.tweet_mensagem)

 val mensagemDoTweet: String = campoDeMensagemDoTweet.text.toString()

 val tweet = Tweet(mensagemDoTweet)

 viewModel.salva(tweet)

 Toast.makeText(this, "$tweet foi salvo com sucesso :D", Toast.LENGTH_LONG).show()
}
```

10. Rode novamente o aplicativo e veja que tudo continua funcionando como antes.

#### Seus livros de tecnologia parecem do século passado?



Conheça a **Casa do Código**, uma **nova** editora, com autores de destaque no mercado, foco em **ebooks** (PDF, epub, mobi), preços **imbatíveis** e assuntos **atuais**.

Com a curadoria da **Caelum** e excelentes autores, é uma abordagem **diferente** para livros de tecnologia no Brasil.

[Casa do Código, Livros de Tecnologia.](#)

## 4.14 ATRIBUINDO AÇÃO A UM ITEM DA LISTA

Ao ver a lista de *tweets*, o usuário pode se arrepender de algum texto que escreveu e querer alterá-lo ou deletar direto o *tweet*. Para realizar qualquer ação em um item de uma lista, o usuário vai clicar neste item. O clique em um item de uma lista pode ser associado a uma ação através de *listeners*, como já fizemos antes com o `onClickListener`. Porém, na `ListaActivity` só temos acesso direto à `ListView`. Adicionar o *listener* `onClickListener` à `ListView` vai atrelar uma ação ao clique na lista como um todo, sem a informação de qual item foi clicado. Mas precisamos saber que item foi clicado para poder alterar suas informações ou deletar o item, caso o usuário escolha essa opção. Para adicionarmos uma ação específica a um item de uma lista, há um outro *listener*: `onItemClickListener`. Ele recebe justamente o `item` que foi clicado, assim como outras

informações bem úteis quando estamos lidando com uma lista, como a posição e o id do item que foi clicado. Além disso, esse método recebe também um objeto do tipo `AdapterView`, ou seja, o adapter que foi utilizado para transformar os objetos em `View`s:

```
AdapterView.OnItemClickListener {
 adapter, item, posicao, id ->
}
}
```

Para conseguir as informações do `tweet`, precisamos conseguir um objeto do tipo `Tweet`. Para termos o objeto `Tweet` a partir da `view` que foi clicada, podemos usar o `adapter`, que possui um método chamado `getItemAtPosition`, bastando apenas passar a posição ao método:

```
AdapterView.OnItemClickListener {
 adapter, item, posicao, id ->
 adapter.getItemAtPosition(posicao) as Tweet
}
```

Também é possível conseguir o objeto referente à `view` clicada diretamente pela `ListView` com o método `getItemAtPosition`:

```
AdapterView.OnItemClickListener {
 adapter, item, posicao, id ->
 lista_tweets.getItemAtPosition(posicao) as Tweet
}
```

Lembrando que ambos os métodos podem retornar qualquer tipo de objeto, ou seja, retornam `Any`. Como sabemos que nesse caso o objeto é o `Tweet`, fizemos *casting*.

## 4.15 DELETANDO TWEET

Para o usuário poder deletar o `tweet` ao clicar nele, precisamos ter na nossa aplicação a lógica de excluir do banco de dados um `tweet`. Estamos usando o `Room` para implementar para a gente essas lógicas relacionadas ao banco de dados e escrevermos o mínimo possível, mas precisamos avisá-lo que lógica ele deve implementar. Ele cria a implementação com base na nossa interface anotada com `@Dao`. Precisamos definir nela um método que será responsável pela exclusão:

```
@Dao
interface TweetDao {
 fun exclui(tweet: Tweet)
}
```

Mas, só com o nome do método o `Room` não consegue saber qual lógica deve ser feita. Precisamos configurar esse método com a anotação que avisará o `Room` qual ação fazer no banco de dados:

```
@Dao
interface TweetDao {
 @Delete
 fun exclui(tweet: Tweet)
}
```

## Passando pelas camadas

Agora, nossa `ListaActivity` vai pedir para deletar um `tweet` quando um usuário clicar em um item da lista, mas sempre que ela vai acessar qualquer dado, ela usa a camada `ViewModel`. Portanto, ela deverá pedir ao `TweetViewModel` para deletar um `tweet`:

```
AdapterView.OnItemClickListener {
 adapter, item, posicao, id ->
 val tweet = lista_tweets.getItemAtPosition(posicao) as Tweet
 viewModel.deleta(tweet)
}
```

A camada `ViewModel`, por sua vez, não sabe de onde vem os dados para deletá-los, então vai pedir a quem sabe dessa informação: a camada `Repository`:

```
class TweetViewModel(private val repository: TweetRepository) : ViewModel() {
 fun deleta(tweet: Tweet) = repository.deleta(tweet)
}
```

Por fim, o `TweetRepository` irá pedir ao `TweetDao` para remover o `tweet`, chamando o método que acabamos de criar:

```
class TweetRepository(private val dao: TweetDao) {
 fun deleta(tweet: Tweet) = dao.exclui(tweet)
}
```

## 4.16 MOSTRANDO UM ALERTA AO USUÁRIO

Nossa aplicação no momento está com a lógica de, assim que o usuário clicar em um item, já deletá-lo. Mas o usuário pode ter clicado no item por diversos outros motivos, como ver mais detalhes do item ou ver as opções de ação para esse item, sem querer necessariamente deletá-lo e perder tudo que tinha desse item. Pensando nisso, seria legal mostrar um alerta ao usuário perguntando se ele realmente quer excluir esse item. Para tarefas que exigem uma decisão ou ação do usuário, o pessoal que desenvolveu as diretrizes do *Material Design* definiu as caixas de diálogo, caixas modais que ocupam parte da tela, tomam o foco do usuário e só somem com alguma ação do usuário.

No caso do `Android`, a classe `AlertDialog` é bastante utilizada para caixas de diálogo que servem de alerta para o usuário tomar alguma decisão. Essa classe já cria um alerta com algumas configurações padrões e precisamos apenas definir os conteúdos da caixa de diálogo, como o título do alerta e a mensagem ou pergunta para o usuário.

## Construindo o AlertDialog

Como é preciso passar essas e outras informações para criar uma caixa de diálogo, a `AlertDialog` tem uma classe especializada em construir um objeto do tipo `AlertDialog`: a `AlertDialog.Builder`. Na criação desse *builder*, precisamos passar a informação de a qual contexto esse alerta está

relacionado, para a decisão do usuário realizar alguma ação com a componente que ativou o alerta:

```
val dialogBuilder = AlertDialog.Builder(this)
```

Para esse *builder* passamos então as informações que queremos mostrar ao usuário:

```
dialogBuilder.setTitle("Atenção")
dialogBuilder.setMessage("Deseja mesmo apagar esse tweet?")
```

Com a pergunta que fazemos ao usuário, ele pode tanto responder positivamente, concordando com a ação pretendida, negativamente, discordando com a ação, ou de forma neutra, mostrando que não se importa se a ação será feita ou não. Para cada uma dessas três possibilidades, podemos mostrar um botão ao usuário utilizando métodos do `AlertDialog.Builder` : `setPositiveButton` , `setNegativeButton` e `setNeutralButton` , respectivamente. Cada um desses métodos recebe o texto que irá aparecer no botão e a lógica que está relacionada à escolha do usuário. No entanto, essa lógica será chamada pelo `AlertDialog.Builder` , então ele precisa saber o nome do método onde nossa lógica estará guardada, e faz isso pedindo para a gente implementar uma interface, a `DialogInterface.OnClickListener` .

No nosso caso, se o usuário discordar da pergunta, ou seja, não quiser deletar o *tweet*, não faremos ação nenhuma, então não precisamos passar nenhuma implementação da interface:

```
dialogBuilder.setNegativeButton("Não", null)
```

Se o usuário concordar com a pergunta, queremos efetivamente deletar o *tweet*, então precisamos criar uma implementação da interface e passar sua instância para o método `setPositiveButton` . A interface define o método `onClick` , que recebe a caixa de diálogo que recebeu o clique e o botão que foi selecionado ou, se for uma caixa de diálogo com itens, a posição do item que foi selecionado. Nossa implementação, juntamente com sua instanciação, pode ficar assim:

```
DialogInterface.OnClickListener(){ dialog: DialogInterface, which: Int ->
 viewModel.deleta(tweet)
}
```

Olhando para o código acima, vemos que nenhum dos dois parâmetros que recebemos no método `onClick` são utilizados pela nossa lógica. Pela sintaxe do `lambda` , se um método recebe dois ou mais parâmetros, eles devem ser listados antes do operador " `->` ", mas não os utilizamos, então não deveríamos precisar pensar em um nome de variável para eles. Para esses casos, o `Kotlin` permite declararmos um parâmetro com o caractere reservado " `_` ":

```
DialogInterface.OnClickListener(){ _: DialogInterface, _: Int ->
 viewModel.deleta(tweet)
}
```

Passando essa lógica para o método `setPositiveButton` , não precisamos escrever o nome da interface, pois o método já espera um objeto de uma implementação da interface:

```
dialogBuilder.setPositiveButton("Sim", { _: DialogInterface, _: Int ->
```

```
 viewModel.deleta(tweet)
 })
```

Como o método `setPositiveButton` pede por último o parâmetro que pode ser passado como um `lambda`, um outro jeito de passar essa lógica a ele é pela seguinte sintaxe:

```
dialogBuilder.setPositiveButton("Sim") { _: DialogInterface, _: Int -> viewModel.deleta(tweet) }
```

Perceba que, ao invés de incluir o `lambda` dentro do parênteses do método `setPositiveButton`, passamos primeiro todos os outros parâmetros (no nosso caso, apenas um) dentro do parênteses, fechamos o parênteses do método `setPositiveButton` e só depois escrevemos o `lambda` dentro das chaves. São apenas duas maneiras diferentes de passar uma lógica com `lambda` para um método.

## Mostrando o AlertDialog

Até agora, só configuramos o que queremos que apareça na caixa de diálogo, mas ainda estamos trabalhando com o objeto `AlertDialog.Builder`. Precisamos em algum momento avisar ao *builder* que terminamos de customizar o alerta e queremos mostrá-lo na tela. Fazemos isso com o método `show` do *builder*:

```
dialogBuilder.show()
```

**Agora é a melhor hora de respirar mais tecnologia!**



Se você está gostando dessa apostila, certamente vai aproveitar os **cursos online** que lançamos na plataforma **Alura**. Você estuda a qualquer momento com a **qualidade** Caelum. Programação, Mobile, Design, Infra, Front-End e Business! Ex-aluno da Caelum tem 15% de desconto, siga o link!

[Conheça a Alura Cursos Online.](#)

## 4.17 EXERCÍCIO: POSSIBILITANDO A EXCLUSÃO DE UM TWEET

### Objetivo

Ao clicar no `ListView`, exibir uma mensagem perguntando se o usuário quer ou não deletar o `tweet`. Se ele clicar no sim, delete o `tweet` selecionado.

### Passo a passo

1. Adicione um *listener* na listagem de tweets:

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

 // código já existente

 lista_tweet.onItemClickListener = listener
}
```

O código acima ainda não vai compilar.

2. Agora será necessário definir esse *listener* que estamos passando:

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

 // código já existente

 val listener = AdapterView.OnItemClickListener {}

 lista_tweet.setOnItemClickListener = listener
}
```

3. Nossa código não deve estar funcionando nesse momento, reclamando com a falta de alguns parâmetros, faremos isso agora:

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

 // código já existente

 val listener = AdapterView.OnItemClickListener {
 adapter, item, posicao, id ->

 }

 lista_tweet.setOnItemClickListener = listener
}
```

4. Agora precisamos recuperar o `Tweet` que foi clicado e ativou esse *listener*:

```
val listener = AdapterView.OnItemClickListener {
 adapter, item, posicao, id ->

 val tweet = lista_tweet.getItemAtPosition(posicao) as Tweet

 // podemos pegar dessa maneira também:
 // adapter.getItemAtPosition(posicao) as Tweet
}
```

5. Com o `Tweet` em mãos podemos ver se o usuário quer deletá-lo, faremos isso num método separado:

```
val listener = AdapterView.OnItemClickListener {
 adapter, item, posicao, id ->
```

```

 val tweet = lista_tweet.getItemAtPosition(posicao) as Tweet
 perguntaSePrecisaDeletarEsse(tweet)
 }

```

O código acima ainda não vai compilar.

6. O método `perguntaSePrecisaDeletarEsse(tweet)` será responsável por exibir a mensagem na tela:

```

class ListaActivity : AppCompatActivity() {

 override fun onCreate(savedInstanceState: Bundle?) {
 // código
 }

 // restante do código

 private fun perguntaSePrecisaDeletarEsse(tweet: Tweet) {
 val dialog = AlertDialog.Builder(this)

 dialog.setMessage("Deseja mesmo apagar esse tweet?")
 dialog.setTitle("Ação")
 dialog.setPositiveButton("Sim") { _: DialogInterface, _: Int -> viewModel.deleta(tweet) }
 dialog.setNegativeButton("Não", null)

 dialog.show()
 }
}

```

O código acima ainda não vai compilar.

7. Contudo, nosso `ViewModel` ainda não possui o método de deleção. Para o código acima funcionar, vamos implementá-lo:

```

class TweetViewModel(private val repository: TweetRepository) : ViewModel() {

 fun lista() = repository.lista()

 fun salva(tweet: Tweet) = repository.salva(tweet)

 fun deleta(tweet: Tweet) = repository.deleta(tweet)

}

```

O código acima ainda não vai compilar.

8. Agora precisamos propagar isso para nosso `Repository` :

```

class TweetRepository(private val fonteDeDados: TweetDao) {

 fun lista() = fonteDeDados.lista()

 fun salva(tweet: Tweet) = fonteDeDados.salva(tweet)

```

```
 fun deleta(tweet: Tweet) = fonteDeDados.deleta(tweet)
}
```

O código acima ainda não vai compilar.

9. Por fim, precisamos adicionar ao nosso `TweetDao` o método `deleta`:

```
@Dao
interface TweetDao {

 @Insert
 fun salva(tweet: Tweet)

 @Query("select * from Tweet")
 fun lista(): LiveData<List<Tweet>>

 @Delete
 fun deleta(tweet: Tweet)
}
```

10. Rode o aplicativo e tente deletar algum *tweet*.

# UTILIZANDO A CAMERA

## 5.1 O QUE SABEREI FAZER NO FIM DESSE CAPÍTULO ?

- Manipular a camera
- Salvar a foto localmente no dispositivo
- Utilizar Base64 para salvar no banco de dados
- Chamar uma Activity esperando um retorno
- Extentions Functions

### Editora Casa do Código com livros de uma forma diferente



Editoras tradicionais pouco ligam para ebooks e novas tecnologias. Não dominam tecnicamente o assunto para revisar os livros a fundo. Não têm anos de experiência em didáticas com cursos.

Conheça a **Casa do Código**, uma editora diferente, com curadoria da **Caelum** e obsessão por livros de qualidade a preços justos.

[Casa do Código, ebook com preço de ebook.](#)

## 5.2 ABRINDO A CÂMERA

Quando pensamos em *tweets*, é bem comum vermos uma imagem dentro de um *tweet* e, para conseguí-la, o usuário pode sempre pegar da galeria ou tirar uma foto enquanto está escrevendo o *tweet*. Focaremos nesse momento na possibilidade de tirar a foto durante a criação do *tweet*.

Uma maneira de fazer isso é nossa aplicação gerenciar diretamente o *hardware* da câmera. [checar a próxima info] Mas, para isso, teríamos que conseguir conversar com os componentes físicos da câmera, como a lente, a abertura, o tempo de exposição. Ao invés disso, podemos utilizar algum aplicativo que já tenha a capacidade de gerenciar os componentes físicos da câmera. De qualquer forma, sempre estamos indo para uma nova tela, onde o usuário vai poder tirar a foto.

Já vimos que, para mudar de tela no `Android`, avisamos ao Sistema Operacional apenas que temos a **intenção** de mudar de tela com a `Intent`, pois não temos o controle de que `Activity` será iniciada ou destruída. A forma de criar a `Intent` que vimos é pedindo a `Activity` de destino e o contexto que está querendo mudar de tela. Se nós mesmos gerenciarmos diretamente o *hardware* da câmera, saberemos qual é a `Activity` que deverá ser chamada. Porém, se utilizarmos um outro aplicativo que já saiba lidar com a câmera, precisaríamos saber o nome exato da `Activity` desse aplicativo que permite tirar a foto. Além disso, há inúmeros aplicativos que tiram foto, e precisaríamos definir um específico utilizando essa forma de criar uma `Intent`. Isso porque a `Intent` que construimos da forma que conhecemos é chamada de **Intent explícita**, já que definimos *explicitamente* qual `Activity` será chamada.

No nosso caso de agora, não queremos definir uma `Activity` específica a ser aberta, queremos *apenas* definir uma **ação** a ser feita pela `Activity` que será aberta, ou seja, estamos criando uma **Intent implícita**, sem falar qual `Activity` será iniciada. Por exemplo, para criar uma `Intent` com a ação de tirar foto, usamos o seguinte construtor:

```
val intent = Intent(MediaStore.ACTION_IMAGE_CAPTURE)
```

Com essa forma de criação da `Intent`, quando chamarmos a lógica de iniciar uma `Activity`, `startActivity`, o Sistema Operacional vai procurar pelos aplicativos do dispositivo que saibam lidar com essa ação. Se houver apenas um aplicativo, este já será iniciado automaticamente quando for a hora; se houver mais de um aplicativo, será aberto um aviso ao usuário perguntando qual dos aplicativos deve ser iniciado; se não houver nenhum aplicativo que saiba tratar essa ação, ocorrerá um erro.

## 5.3 EXERCÍCIO: ABRINDO O APLICATIVO DA CÂMERA

### Objetivo

Criar um botão no menu da tela de cadastrar um novo tweet que seja responsável por abrir o aplicativo de camera.

### Passo a passo

- Precisamos adicionar um novo item no menu da `TweetActivity`, para isso altere o arquivo `tweet_menu.xml`:

```
<menu xmlns:android="http://schemas.android.com/apk/res/android"
 xmlns:app="http://schemas.android.com/apk/res-auto">

 <item
 android:id="@+id/tweet_menu_foto"
 android:icon="@android:drawable/ic_menu_camera"
 android:title="Foto"
 app:showAsAction="always" />
```

```

<item
 android:id="@+id/tweet_menu_cadastrar"
 android:title="Cadastrar"
 android:icon="@drawable/ic_save"
 app:showAsAction="always" />

</menu>

```

2. Agora precisamos adicionar mais uma condição de tratamento dos itens de menu na `TweetActivity`:

```

override fun onOptionsItemSelected(item: MenuItem?): Boolean {
 when (item?.itemId) {
 //demais casos
 R.id(tweet_menu_foto) -> {
 tiraFoto()
 }
 return true
 }
}

```

O código acima ainda não vai compilar.

3. Ainda está faltando definirmos o método `tiraFoto()`, que será responsável por chamar a câmera:

```

private fun tiraFoto() {
 val vaiPraCamera = Intent(MediaStore.ACTION_IMAGE_CAPTURE)
 startActivityForResult(vaiPraCamera)
}

```

4. Rode o aplicativo e veja se o aplicativo de câmera foi aberto.

## 5.4 PEGANDO A FOTO TIRADA

Com o aplicativo de câmera aberto, o usuário vai tirar a foto que quiser, mas essa foto ainda está no aplicativo de câmera. O aplicativo de câmera pode então enviar a foto de volta para nosso aplicativo. Isso realmente acontece, mas a foto devolvida é muito pequena, quase do tamanho de um ícone. Se quisermos usar essa imagem pequena em um espaço maior, ela vai ficar toda quadriculada, quebrada. O melhor então é conseguir uma versão maior da foto. Como enviar uma foto em tamanho grande de um aplicativo pra outro ficaria muito pesado, os aplicativos de câmera usam a abordagem de salvar a foto em um arquivo e mexer com a localização desse arquivo.

Geralmente, quando pedimos para abrir um aplicativo de câmera, queremos ter acesso ao caminho da foto, que foi tirada justamente pensando na nossa aplicação. Por esse motivo, a gente mesmo define em que lugar será armazenada a foto e passamos a localização para o aplicativo de câmera. Queremos

então passar uma informação a outro aplicativo. Já vimos que, para pedir para o Android inicializar outra Activity , seja da nossa aplicação ou de outra, sempre usamos uma Intent . Ela é portanto nossa forma de comunicação com a Activity que queremos inicializar. Usaremos então a Intent que definiu a ação de tirar foto para enviar a localização em que a foto deverá ser salva. Como essa informação é opcional para a Activity de tirar foto funcionar, o método usado é o putExtra . Mas podemos passar inúmeras informações para uma Activity usando o putExtra e a Activity precisa saber o que significa cada um desses dados que enviarmos, ou seja, precisamos identificar cada uma das informações que enviarmos. Essa informação do caminho para salvar a foto será usado pelo aplicativo de câmera, então deve ser um nome já pré-estabelecido: MediaStore.EXTRA\_OUTPUT :

```
val intent = Intent(MediaStore.ACTION_IMAGE_CAPTURE)

intent.putExtra(MediaStore.EXTRA_OUTPUT, caminhoDaFoto)
```

## Definindo o caminho do arquivo

Para definir o caminho da foto, primeiro precisamos pensar em que diretório queremos armazenar essa imagem. Como é uma foto apenas para a nossa aplicação, deixaremos dentro de uma pasta específica da nossa aplicação, mas separado dos arquivos internos da aplicação, permitindo ser colocada em mídias removíveis. Para isso, usaremos o método getExternalFilesDir . Esse método pede como parâmetro o tipo do diretório que queremos utilizar. Por exemplo, para usar o diretório de fotos para armazenar a imagem, podemos passar como tipo a constante Environment.DIRECTORY\_PICTURES e ele vai nos retornar o caminho contendo o diretório de fotos que está disponível para o usuário. Se não passarmos nenhum texto para o tipo do diretório, ou seja, deixarmos com null , o método nos devolve por padrão o diretório raiz para os arquivos da nossa aplicação:

```
/storage/sdcard/Android/data/pacote.da.app
```

Agora que definimos o diretório em que nossa foto será salva, precisamos definir o nome do arquivo que conterá a foto. Esse nome, porém, deve ser único, pois todas as fotos ficarão dentro da mesma pasta. Podemos utilizar o instante atual para definir o nome do arquivo, usando o método System.currentTimeMillis , não esquecendo de colocar uma extensão de imagem, por exemplo, .jpg . O arquivo vai estar com o seguinte caminho, fazendo uso da interpolação para adicionar coisas dinâmicas ao texto:

```
val localDaFoto =
"${getExternalFilesDir(Environment.DIRECTORY_PICTURES)}/ ${System.currentTimeMillis()}.jpg"
```

Agora temos o caminho para o arquivo no qual queremos que o aplicativo de câmera salve a foto, mas o aplicativo de tirar foto está esperando a informação que vai chegar com a identificação MediaStore.EXTRA\_OUTPUT em um formato específico: Uri (vem de Uniform Resource Identifier, ou seja, Identificador de Recurso Uniforme). Com esse formato, a localização da foto deve ser única e seguir o protocolo e formatação de uma URI . Precisamos então transformar esse texto em um objeto

`Uri`. Primeiro, vamos criar um objeto representando um arquivo a partir da localização:

```
val arquivo = File(localDaFoto)
```

Em seguida, vamos transformar para `Uri`:

```
Uri.fromFile(arquivo)
```

Essa transformação direta de arquivo para URI monta a URI baseada no esquema `file:///`. Porém, só conseguimos controlar o acesso às permissões para um arquivo com uma URI desse tipo se modificarmos diretamente as permissões pelo sistema de arquivos. Essas permissões valem para *qualquer* aplicativo que tente acessar esse arquivo e permanecem válidas até que mudemos a permissão de novo. Esse tipo de controle de acesso é muito inseguro, então não é mais recomendado utilizar o esquema `file:///` para URI de arquivos no Android. Na verdade, a partir do Android 6, API 23, o Sistema Operacional lança a exceção `FileUriExposedException` caso tentemos utilizarmos esse esquema.

Há um outro esquema que pode ser utilizado, o `content://`. Ele permite darmos acesso de leitura e escrita temporariamente e apenas para a `Activity` ou serviço que vai receber o arquivo, bem mais seguro que com o outro esquema. Para usar esse esquema, precisamos utilizar um provedor de arquivos, portanto, ao invés de criar uma URI diretamente de um arquivo, pediremos a um provedor de arquivos criar pra gente com o método `FileProvider.getUriFromFile`. Esse método pede, além do arquivo, o `Context` e a autoridade (identificação) do provedor de arquivo que será utilizado nesse momento. Precisamos então ainda definir um provedor de arquivo. O próprio Android já nos fornece um provedor de arquivos: `FileProvider`, mas, para avisar o Android que este `FileProvider` será utilizado e informar quais configurações serão feitas, precisamos registrar o `FileProvider` no `AndroidManifest.xml` usando a tag `provider`. Para essa tag precisamos incluir pelo menos os seguintes atributos: `android:name` com o nome completo da classe, `androidx.core.content.FileProvider`; `android:authorities` com a identificação desse *provider*, que pode ser qualquer nome mas, para evitar conflitos e ser único, é recomendado que seja o domínio da aplicação junto com o nome `fileprovider`; `android:grantUriPermissions` setado para verdadeiro, para permitir acesso temporário para o arquivo. O código no `AndroidManifest.xml` ficará assim:

```
<manifest ...>
 ...
 <application ...>
 ...
 <provider
 android:name="androidx.core.content.FileProvider"
 android:authorities="br.com.twittelumapp.fileprovider"
 android:grantUriPermissions="true">
 </provider>
 </application>
 </manifest>
```

Um `FileProvider` só consegue gerar uma URI com o esquema `content://` para arquivos dentro de diretórios pré-especificados. Quando registramos o `FileProvider` no `AndroidManifest.xml`, precisamos já informar com quais diretórios ele vai trabalhar, por meio da tag `meta-data` com atributo `android:name android.support.FILE_PROVIDER_PATHS`. Como podemos informar vários diretórios, por organização, passamos para o atributo `android:resource` apenas a referência a outro arquivo `xml` que irá conter os diretórios com que o `FileProvider` vai trabalhar:

```
<manifest ...>
 ...
 <application ...>
 ...
 <provider ...>
 <meta-data
 android:name="android.support.FILE_PROVIDER_PATHS"
 android:resource="@xml/provider_paths" />
 </provider>
 </application>
</manifest>
```

Precisamos então criar esse `xml`. Para ser localizado com a sintaxe acima, ele deve estar dentro de `resources/xml` e ter o nome `provider_paths.xml`. Perceba que a sintaxe é parecida com a usada para pegar uma imagem de dentro de `resources/drawable`. Dentro do arquivo `provider_paths.xml`, utilizaremos a tag `raiz paths` para envolver todos os caminhos com que o `FileProvider` poderá trabalhar. Usaremos como tag filha da `paths` uma tag diferente para cada tipo do diretório a que o caminho fizer parte. Por exemplo, se o caminho está definido pelo método `getExternalFilesDir`, ou seja, é um caminho cujo arquivo pode ser colocado em mídias removíveis, usaremos a tag `external-path` para definí-lo. Se queremos que o `FileProvider` consiga gerenciar o acesso a qualquer subdiretório dentro do diretório definido por `getExternalFilesDir`, podemos passar para o atributo `path` da tag `external-path` o caractere de diretório atual, " ". , portanto, nosso arquivo `provider_paths.xml` ficará assim:

```
<?xml version="1.0" encoding="utf-8"?>
<paths>
 <external-path
 name="external_files"
 path="/" />
</paths>
```

Com o `FileProvider` devidamente configurado, podemos chamá-lo pelo método estático `getUriFromFile` da classe `FileProvider`:

```
FileProvider.getUriFromFile(this, "br.com.twittelumapp.fileprovider", arquivo)
```

Agora que temos a URI com o esquema `content://`, podemos enviá-la para o aplicativo de tirar foto usando a `Intent`, como já fizemos acima. Agora, quando o usuário clicar no ícone de câmera para tirar uma foto, o aplicativo de câmera será aberto e a foto que for tirada será salva na localização definida pela URI. Queremos agora pegar a foto e mostrar na tela do formulário.

Precisamos então de uma componente que mostre na tela do formulário uma imagem a partir de um arquivo. Usaremos a view `ImageView` do Android para isso, não esquecendo de sempre definir uma altura e largura para qualquer componente e de colocar as restrições para ela ficar abaixo do `EditText`. Como vamos querer acessar essa componente no código em `Kotlin` para inserir uma imagem nela, precisamos também informar um `id` para ela:

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.ConstraintLayout
 xmlns:android="http://schemas.android.com/apk/res/android"
 xmlns:app="http://schemas.android.com/apk/res-auto"
 android:layout_width="match_parent"
 android:layout_height="match_parent">

 <EditText ... />

 <ImageView
 android:id="@+id/tweet_foto"
 android:layout_width="300dp"
 android:layout_height="300dp"
 android:layout_margin="5dp"
 app:layout_constraintEnd_toEndOf="parent"
 app:layout_constraintStart_toStartOf="parent"
 app:layout_constraintTop_toBottomOf="@+id/tweet_mensagem"/>

</androidx.constraintlayout.widget.ConstraintLayout>
```

Agora, voltando à classe correspondente à tela do formulário, podemos pegar a referência pra componente `ImageView` da tela e passar para ela a imagem que está salva na localização que definimos. Porém, só temos até agora informação sobre a localização da imagem. Precisamos passar já a imagem em si para a `ImageView`. O Android possui uma classe que justamente representa uma imagem, guardando a descrição de cada pixel. Essa classe é a `Bitmap`. Como um *bitmap* pode ser criado a partir de vários tipos de recursos, o Android também nos disponibilizou uma *factory* de *bitmaps*, a `BitmapFactory`. Usaremos ela para decodificar um *bitmap* a partir de um arquivo, passando o texto contendo a localização do arquivo:

```
val bitmap: Bitmap = BitmapFactory.decodeFile(localDaFoto)
```

Agora podemos passar esse *bitmap* para a `ImageView`, acessando-a com o importe estático das `views` chamando o `id` da componente:

```
tweet_foto.setImageBitmap(bitmap)
```

Com isso, a foto que o usuário tirar irá aparecer na tela do formulário! Mas, dependendo do tipo de foto que o usuário tirar, podendo ser modo retrato ou paisagem, a foto por padrão pode não ocupar toda a área ocupada pela `ImageView`, pensando nas dimensões. Por exemplo, no nosso caso, a `ImageView` está com dimensão igual para a largura e para a altura, ou seja, está no formato de um quadrado. A foto tirada pelo aplicativo de câmera, por padrão, é retangular. Para definir como queremos que a imagem fique em relação às dimensões da `ImageView`, a componente possui uma propriedade chamada `scaleType` que tem justamente essa função. Por exemplo, se quisermos que a imagem se distorça para ocupar toda a

extensão da `ImageView`, escrevemos o código:

```
tweet_foto.scaleType = ImageView.ScaleType.FIT_XY
```

Uma outra coisa a se pensar é que a foto tirada geralmente é muito grande, e vamos mostrá-la em um espaço bem menor que o tamanho dela. Para não trabalharmos com arquivos muito pesados, podemos criar um *bitmap* reduzido a partir do *bitmap* que foi criado com o tamanho real usando o método `Bitmap.createScaledBitmap`, que recebe o *bitmap* original, a largura e a altura que queremos para o novo *bitmap*, e um booleano para se deve ou não ser aplicado um filtro para a transformação de imagem (se o filtro não for aplicado, a qualidade da imagem reduz bastante, enquanto que a diferença na performance quase não é notada):

```
Bitmap.createScaledBitmap(bitmap, 300, 300, true)
```

## Melhorando o visual

Agora que nosso formulário já é capaz de mostrar a foto que o usuário tirou, vamos deixar ela um pouco mais atraente ao usuário, por exemplo, deixando as bordas dela mais arredondadas e deixando mais claro que a imagem é algo independente do resto da tela, dando uma noção de diferença de profundidade ao deixar uma sombra em volta dela. Podemos fazer tudo isso usando atributos da `ImageView`, mas, como essas duas mudanças no estilo começaram a ser muito utilizadas e fazem parte das diretrizes do *Material Design*, os desenvolvedores do *Android* criaram uma componente que define exatamente esses estilos, chamada de `CardView`. Como essa componente foi criada depois do *Material Design*, precisamos incluir a biblioteca externa `androidx.cardview:cardview:1.0.0` no arquivo `build.gradle`.

Em seguida, usamos essa componente no *layout*, envolvendo a `ImageView`. Para definir o quanto arredondado ficam as bordas, ela tem o atributo `app:cardCornerRadius`:

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.ConstraintLayout
 xmlns:android="http://schemas.android.com/apk/res/android"
 xmlns:app="http://schemas.android.com/apk/res-auto"
 android:layout_width="match_parent"
 android:layout_height="match_parent">

 <EditText ... />

 <androidx.cardview.widget.CardView
 android:layout_width="0dp"
 android:layout_height="300dp"
 android:layout_margin="5dp"
 app:cardCornerRadius="5dp"
 app:layout_constraintEnd_toEndOf="parent"
 app:layout_constraintStart_toStartOf="parent"
 app:layout_constraintTop_toBottomOf="@+id/tweet_mensagem">

 <ImageView
 android:id="@+id/tweet_foto"
 android:layout_width="match_parent"
```

```
 android:layout_height="match_parent" />

 </androidx.cardview.widget.CardView>

</androidx.constraintlayout.widget.ConstraintLayout>
```

Agora nossa imagem fica arredondada e com a sensação de que está elevada! Mas, se o usuário tiver um dispositivo pequeno, pode ser que ele não enxergue a foto inteira. Quando não conseguirmos ver tudo que uma tela possui, geralmente "scrollamos" a tela. Por padrão, as *views* de *layout* não possuem essa capacidade, então precisamos usar uma componente específica que adicione essa capacidade à tela: a `ScrollView`. Colocaremos ela envolvendo toda a tela, deixando ela como *container* principal do *layout*:

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

 <androidx.constraintlayout.widget.ConstraintLayout
 android:layout_width="match_parent"
 android:layout_height="match_parent">

 ...
 </androidx.constraintlayout.widget.ConstraintLayout>
</ScrollView>
```

Já conhece os cursos online Alura?



A Alura oferece centenas de **cursos online** em sua plataforma exclusiva de ensino que favorece o aprendizado com a **qualidade** reconhecida da Caelum. Você pode escolher um curso nas áreas de Programação, Front-end, Mobile, Design & UX, Infra e Business, com um plano que dá acesso a todos os cursos. Ex aluno da Caelum tem 15% de desconto neste link!

[Conheça os cursos online Alura.](#)

## 5.5 EXERCÍCIO: ESPERANDO UM RESULTADO DA CÂMERA

### Objetivo

Ao chamar a câmera, passe o local onde ela deve armazenar a foto e trate a resposta na `TweetActivity` colocando a foto em um `ImageView`

## Passo a passo

- Precisamos passar valor para a `Activity` de câmera. Fazemos isso através do método `putExtra`.

Vamos pra `TweetActivity` alterar o método `tiraFoto`:

```
private fun tiraFoto() {
 val vaiPraCamera = Intent(MediaStore.ACTION_IMAGE_CAPTURE)

 vaiPraCamera.putExtra(MediaStore.EXTRA_OUTPUT, caminhoFoto)

 startActivity(vaiPraCamera)
}
```

O código acima não vai compilar.

- Precisamos criar esse caminho:

```
class TweetActivity : AppCompatActivity() {

 private var localFoto: String? = null

 //demais métodos

 private fun tiraFoto() {

 val vaiPraCamera = Intent(MediaStore.ACTION_IMAGE_CAPTURE)

 val caminhoFoto = defineLocalDaFoto()

 vaiPraCamera.putExtra(MediaStore.EXTRA_OUTPUT, caminhoFoto)

 startActivity(vaiPraCamera)
 }

 fun defineLocalDaFoto(): Uri? {

 localFoto = "${getExternalFilesDir(Environment.DIRECTORY_PICTURES)}/${System.currentTimeMillis()}.jpg"

 val arquivo = File(localFoto)

 return FileProvider.getUriForFile(this, "br.com.twittelumapp.fileprovider", arquivo)
 }
}
```

- Precisamos definir agora nosso provider e a permissão de salvar arquivos em nosso `AndroidManifest.xml`

```
<manifest>

<application ...>

 <!-- demais códigos -->

 <provider
 android:name="androidx.core.content.FileProvider"
 android:authorities="br.com.twittelumapp.fileprovider"
```

```

 android:grantUriPermissions="true">
 <meta-data
 android:name="android.support.FILE_PROVIDER_PATHS"
 android:resource="@xml/provider_paths" />
</provider>

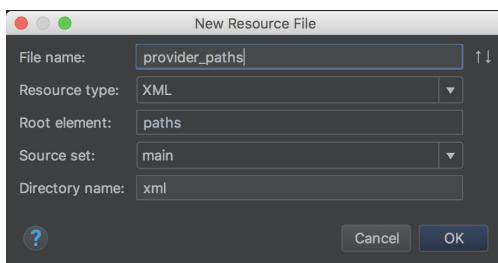
</application>

</manifest>

```

O código acima não vai compilar.

- Ainda é necessário criarmos o arquivo `provider_paths`. Para isso, vá na pasta `res`, clique com o botão direito e escolha a opção `new -> Android Resource File`. Depois escolha como `Resource type` a opção `xml` e como `Root element` o `paths`:



```

<?xml version="1.0" encoding="utf-8"?>
<paths>
 <external-path
 name="external_files"
 path=". " />
</paths>

```

- Vamos colocar a dependência do `CardView` na aplicação, para isso abra o `build.gradle` e adicione a seguinte linha:

```

dependencies {
 // demais dependencias
 implementation 'androidx.cardview:cardview:1.0.0'
}

```

- Agora precisaremos fazer uma util modificação em nosso `layout`, portanto abra o arquivo `activity_tweet.xml` no modo de texto e vamos adicionar o `ScrollView`, e colocar `CardView` com um `ImageView` dentro:

```

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

 <androidx.constraintlayout.widget.ConstraintLayout
 android:layout_width="match_parent"
 android:layout_height="match_parent">

```

```

 android:paddingBottom="40dp">

 <EditText
 android:id="@+id/tweet_mensagem"
 android:layout_width="0dp"
 android:layout_height="wrap_content"
 android:layout_marginEnd="2dp"
 android:layout_marginStart="2dp"
 android:layout_marginTop="10dp"
 android:hint="O que está acontecendo ?"
 android:inputType="textMultiLine"
 android:maxLength="250"
 app:layout_constraintEnd_toEndOf="parent"
 app:layout_constraintHorizontal_bias="0.0"
 app:layout_constraintStart_toStartOf="parent"
 app:layout_constraintTop_toTopOf="parent" />

 <androidx.cardview.widget.CardView
 android:layout_width="0dp"
 android:layout_height="300dp"
 android:layout_margin="5dp"
 android:paddingBottom="5dp"
 app:cardCornerRadius="5dp"
 app:layout_constraintEnd_toEndOf="parent"
 app:layout_constraintStart_toStartOf="parent"
 app:layout_constraintTop_toBottomOf="@+id/tweet_mensagem">

 <ImageView
 android:id="@+id/tweet_foto"
 android:layout_width="match_parent"
 android:layout_height="match_parent" />

 </androidx.cardview.widget.CardView>

</androidx.constraintlayout.widget.ConstraintLayout>
</ScrollView>

```

## 7. Agora precisamos tratar a volta da foto no TweetActivity :

```

override fun onResume() {
 super.onResume()
 if (localFoto != null) {
 carregaFoto()
 }
}

private fun carregaFoto() {

 val bitmap = BitmapFactory.decodeFile(localFoto)

 val bm = Bitmap.createScaledBitmap(bitmap, 300, 300, true)

 tweet_foto.setImageBitmap(bm)

 tweet_foto.scaleType = ImageView.ScaleType.FIT_XY
}

```

## 8. Tire uma foto e veja o que acontece.

## 5.6 AVISANDO A CÂMERA QUE QUEREMOS UMA RESPOSTA

Agora o usuário pode clicar no ícone da câmera que está na tela do formulário, ir para o aplicativo de câmera, tirar uma foto e a foto vai aparecer na tela do formulário. Se o usuário desistir de tirar uma foto enquanto estiver no aplicativo de câmera, ele deveria voltar para a tela do formulário, mas sem a imagem. Porém, a aplicação acaba sendo reiniciada, em vez de ir para a tela de formulário em que ele estava criando o *tweet*. Isso porque pedimos para a `Activity` de tirar foto ser criada independentemente da aplicação que existia antes.

Precisamos avisar o Android que a nossa `Activity` de formulário está esperando um retorno da `Activity` da câmera, ou seja, quando a `Activity` de tirar foto for finalizada, queremos que volte para a `Activity` que a iniciou, informando se a ação que ela deveria fazer, de tirar foto, foi bem sucedida ou não. Para avisar o Android que queremos um resultado da `Activity` que pedimos para iniciar, tratando essa `Activity` como uma espécie de ***sub-activity***, usamos o método `startActivityForResult` ao invés do método `startActivity`. Esse método também recebe a `Intent` representando a `Activity` que temos a intenção de iniciar. Mas podemos pedir para iniciar uma mesma `Activity` e querer um resultado dela de vários pontos diferentes do nosso código. Na hora de devolver o resultado, o Android precisa saber a qual chamada do método `startActivityForResult` o resultado está relacionado. Por isso, o método recebe também um código de requisição, que definimos o valor que quisermos, bastando ser positivo:

```
val intentVaiPraCamera = Intent(MediaStore.ACTION_IMAGE_CAPTURE)
startActivityForResult(intentVaiPraCamera, 123)
```

Quando a ação na `Activity` de tirar foto for finalizada, agora ela vai mandar o resultado da ação para a nossa `Activity` de formulário, informando se foi sucesso ou não, por exemplo. Recebemos o resultado de qualquer `Intent` que foi iniciada pelo método `onActivityResult`, que já existe em qualquer classe que seja uma `Activity`. Podemos então, dependendo do resultado, decidir uma lógica diferente para ser executada sobrescrevendo o método `onActivityResult`. Nesse método recebemos o código de requisição, aquele que definimos no método `startActivityForResult`, mas, além disso, recebemos um código informando o resultado da ação, de valor inteiro, e algumas informações que a `Activity` possa ter nos enviado, guardadas no meio que usamos para trafegar de uma `Activity` para outra, a `Intent`:

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

Por fim, dentro desse método agora podemos verificar se o `requestCode` foi o mesmo que definimos no `startActivityForResult` e se `resultCode` foi de "sucesso", comparando o valor da

variável com `Activity.RESULT_OK` para só então carregar a imagem na tela do formulário:

```
override fun onActivityResult(requestCode: Int,
 resultCode: Int,
 data: Intent?) {
 super.onActivityResult(requestCode, resultCode, data)
 if (requestCode == 123) {
 if (resultCode == Activity.RESULT_OK) {
 // carrega foto na tela do formulário
 }
 }
}
```

## 5.7 EXERCÍCIO: PREVENDO QUANDO O USUÁRIO CANCELAR A AÇÃO

### Objetivo

Faça a chamada da câmera saber que estamos esperando um resultado e trate esse resultado de maneira correta.

### Passo a passo

1. Altere o método `tiraFoto()` para usar o método `startActivityForResult` no lugar do `startActivity`:

```
private fun tiraFoto() {

 val vaiPraCamera = Intent(MediaStore.ACTION_IMAGE_CAPTURE)

 val caminhoFoto = defineLocalDaFoto()

 vaiPraCamera.putExtra(MediaStore.EXTRA_OUTPUT, caminhoFoto)

 startActivityForResult(vaiPraCamera, 123)

}
```

2. Remova o método `onResume`:

```
override fun onResume() {
 super.onResume()
 if (localFoto != null) {
 carregaFoto()
 }
}
```

3. Implemente o método `onActivityResult` para tratar a resposta da activity de camera.

```
override fun onActivityResult(requestCode: Int,
 resultCode: Int,
 data: Intent?) {
 super.onActivityResult(requestCode, resultCode, data)
 if (requestCode == 123) {
 if (resultCode == Activity.RESULT_OK) {
```

```
 carregaFoto()
 }
}
}
```

4. Rode novamente o aplicativo e veja se o problema foi resolvido.

### Saber inglês é muito importante em TI



Na **Alura Língua** você reforça e aprimora seu inglês! Usando a técnica *Spaced Repetitions* o aprendizado naturalmente **se adapta ao seu conhecimento**. Exercícios e vídeos interativos fazem com que você pratique em situações cotidianas. Além disso, todas as aulas possuem explicações gramaticais, para você entender completamente o que está aprendendo. Aprender inglês é fundamental para o profissional de tecnologia de sucesso!

[Pratique seu inglês na Alura Língua.](#)

## 5.8 SALVANDO A FOTO NO BANCO DE DADOS

Quando o usuário clicar no ícone para criar um *tweet*, agora o *tweet* precisa conter a mensagem e o id, mas pode ou não ter a imagem, ou seja, um objeto *tweet*, ao ser criado, pode conter a mensagem e o id ou a mensagem, o id e a imagem. Para definirmos o que um objeto vai possuir ao ser construído, usamos o construtor. Como já tem um construtor na classe `Tweet` que recebe a mensagem e o id, podemos alterá-lo para receber também a imagem, passando a localização da imagem. Mas pode ser que o usuário queira criar um *tweet* sem foto, ou seja, pode ser que seja passado um valor **nulo** para esse parâmetro. Como já vimos, se uma variável pode ter um valor nulo, precisamos deixar isso explícito na tipagem da variável, usando o operador `?:`

```
data class Tweet(val mensagem: String,
 val localFoto: String?,
 @PrimaryKey(autoGenerate = true) val id: Int = 0)
```

Com o objeto `Tweet` criado e preenchido com uma mensagem e uma foto, queremos salvá-lo no banco. Quando executarmos nossa aplicação em um dispositivo que ainda não tem ela instalada, vai dar certo salvar o *tweet*. Mas, quando executarmos nossa aplicação em um dispositivo que já tinha ela instalada, vai dar erro, pois já existe um bando nosso no dispositivo, e a tabela `Tweet` existente não possui a coluna `localFoto`.

## Atualizando tabela no banco

Para os dispositivos que já tem *tweets* salvos no banco de dados da nossa aplicação, não seria legal deletar tudo que já está salvo nele para recriar o banco. Podemos, em vez disso, **atualizar** a tabela *Tweet*.

Mas, como o dispositivo vai saber quais atualizações devem ser feitas? A gente pode fazer várias alterações nos modelos do projeto de tempos em tempos e o dispositivo precisa saber quais alterações ele precisa fazer para chegar no estado mais recente do banco.

Sempre vamos para uma situação diferente das tabelas no banco de dados, dizemos que estamos com uma versão diferente do banco. O `Android` usa então essas versões para checar se o banco está em uma versão desatualizada em relação ao aplicativo. Inicialmente, nosso banco estava definido com a versão 1, mas agora precisamos avisar aos dispositivos que já possuem nossa aplicação que estamos com uma nova versão. Fazemos isso alterando o número da versão para, por exemplo, 2:

```
@Database(entities = [Tweet::class], version = 2)
abstract class TwittelumDatabase : RoomDatabase() {
 // resto do código
}
```

Agora precisamos avisar ao dispositivo quais alterações devem ser feitas de uma versão para outra no banco. Sempre que queremos atualizar algo no banco de dados, dizemos que fazemos uma **migração** no banco. Para adicionar uma migração, chamamos o método `addMigration` durante o processo de criação do banco. Esse método, por sua vez, recebe um objeto do tipo da classe abstrata `Migration`. Precisamos então criar esse objeto do tipo `Migration`, e podemos tirar proveito da sintaxe do `Kotlin` para isso, criando um `object` que herde de `Migration`. Mas, sempre que fazemos uma migração, estamos saindo de uma versão do banco de dados e indo para outra. Por isso, o único construtor da classe abstrata `Migration` recebe o número da versão do banco antes da migração e a versão do banco após a migração:

```
object Migration1Para2 : Migration(1, 2)
```

Essa classe abstrata nos obrigará a implementar o método `migrate`, que recebe um objeto do tipo `SupportSQLiteDatabase` e define a alteração que deve ser feita no banco durante transição das versões que foram definidas no construtor. No nosso caso, queremos adicionar a coluna `localFoto` na tabela `Tweet` e usaremos o método `execSQL` do `SupportSQLiteDatabase` para isso:

```
object Migration1Para2 : Migration(1, 2) {

 override fun migrate(database: SupportSQLiteDatabase) {

 val sql = "alter table Tweet add column localFoto text"
 database.execSQL(sql)

 }
}
```

Agora que temos nosso objeto do tipo `Migration` criado, podemos passá-lo ao método `addMigrations` durante a criação do banco:

```
Room.databaseBuilder(context, TwittelumDatabase::class.java, DATABASE)
 .allowMainThreadQueries()
 .addMigrations(Migration1Para2)
 .build()
```

Agora até os dispositivos que já tinham nossa aplicação instalada conseguem salvar um tweet com imagem no banco de dados!

Porém, essa imagem está em um arquivo separado. Pode ser que o usuário esteja vendo esse arquivo em alguma outra tela ou que o arquivo esteja acessível a outras aplicações como a galeria de fotos e que o usuário exclua o arquivo por outra aplicação. Quando o usuário acessar de novo o `tweet`, não vai mais ver a foto, mas não é o comportamento que ele esperava, pois na nossa aplicação ele não queria ter deletado a imagem do `tweet`.

Uma possível solução para esse problema é salvarmos a imagem no banco de dados. Podemos salvar a imagem em `bytes` mesmo, utilizando o formato **BLOB** que o banco `SQLite` possui. Com esse formato, para poder armazenar todos os `bytes`, a coluna `localFoto` guarda uma referência para outra tabela, que possuirá apenas esse `bytes`. Isso acaba deixando as buscas e persistências na tabela `Tweet` mais demoradas. Outra possibilidade, muito utilizada, é salvar os `bytes` como **texto**.

Um codificador de `bytes` para `String` muito utilizado é a **Base64**. Podemos usar o método estático `encodeToString` da classe `Base64` que é responsável por esse processo de codificação. Ele recebe o `array` de `bytes` que sofrerá a codificação. Para fazer a transformação de `bytes` em caracteres, há uma espécie de tabela que é utilizada e, dependendo do local onde for usado o texto, alguns caracteres são considerados especiais. Por exemplo, se avisarmos que o texto criado vai ser utilizado em uma URL, sempre que a tabela for transformar no caractere `+`, que é especial para uma URL, ela colocará outro caractere no lugar. Precisamos então informar se vamos usar a tabela padrão ou alguma específica, como essa que não utiliza o `+`. Informamos isso passando uma constante para o método `encodeToString`. Como no nosso caso não temos nenhuma restrição de caracteres, pediremos para usar a tabela padrão com a constante `Base64.DEFAULT`:

```
Base64.encodeToString(byteArray, Base64.DEFAULT)
```

Agora sabemos transformar em `bytes` em `String`, mas ainda não temos a imagem no formato de `bytes`.

## Acessando os bytes da imagem

Para comprimir uma imagem em outros formatos, um objeto `Bitmap` tem o método `compress`, que escreve uma versão comprimida da imagem utilizando algum fluxo de saída. O fluxo de saída receberá os `bytes` e enviará para algum local. Como queremos apenas os `bytes`, utilizaremos o fluxo de

saída `ByteArrayOutputStream`, que enviará os *bytes* para um *array* de *bytes*. Além disso, o método `compress` precisa saber qual o formato da imagem que está dentro dele, para fazer a compressão correta, e a qualidade da compressão a ser feita, de 0 para qualidade mínima até 100 para qualidade máxima:

```
val bitmap = // decodifica bitmap
bitmap.compress(Bitmap.CompressFormat.JPEG, 100, ByteArrayOutputStream())
```

Agora precisamos pedir para o objeto `ByteArrayOutputStream` que passamos como parâmetro nos dar o *array* de *bytes*, chamando o método `toByteArray` dele:

```
val byteArrayOutputStream = ByteArrayOutputStream()
bitmap.compress(Bitmap.CompressFormat.JPEG, 100, byteArrayOutputStream)
val byteArray: ByteArray = byteArrayOutputStream.toByteArray()
```

## Criando funções de extensão

Conforme vimos acima, precisamos de várias linhas de código para conseguir fazer a lógica de transformar um `Bitmap` em `String` usando a `Base64`. Para esse código não ficar misturado com outras lógicas, podemos isolá-lo em um método:

```
fun decodificaParaBase64(bitmap: Bitmap): String {

 val byteArrayOutputStream = ByteArrayOutputStream()
 bitmap.compress(Bitmap.CompressFormat.JPEG, 100, byteArrayOutputStream)
 val byteArray = byteArrayOutputStream.toByteArray()

 return Base64.encodeToString(byteArray, Base64.DEFAULT)
}
```

Para essa lógica funcionar, precisamos passar um `Bitmap` para a função. Olhando com mais cuidado, na verdade a lógica de "decodificar para a Base64" é uma lógica que queremos fazer a partir de um objeto `Bitmap`, ou seja, queremos que um objeto `Bitmap` tenha esse comportamento. Mas essa classe não foi criada pela gente, os métodos dela já estão criados. Felizmente, o `Kotlin` veio com uma solução para essa necessidade de criarmos novos métodos para classes criadas por outras pessoas ou empresas: as **funções de extensão (extension functions)**.

Ao invés de criar um método solto, criaremos a partir da classe `Bitmap`. Com isso, não precisaremos mais pedir um `Bitmap` como parâmetro. Agora, quando quisermos acessar o objeto `Bitmap` dentro desse método, faremos exatamente como fazemos quando criamos um método em uma classe e queremos acessar o objeto "deste" momento: com o `this`. O código final ficará assim:

```
fun Bitmap.decodificaParaBase64(): String {

 val byteArrayOutputStream = ByteArrayOutputStream()
 this.compress(Bitmap.CompressFormat.JPEG, 100, byteArrayOutputStream)
 val byteArray = byteArrayOutputStream.toByteArray()
```

```
 return Base64.encodeToString(byteArray, Base64.DEFAULT)
 }


```

## Atrelando informações a um ImageView

Com isso tudo que fizemos, conseguimos transformar um `Bitmap` no formato `String`, que usaremos pra salvar no banco. Mas, quando pegamos a imagem pela tela do formulário, temos acesso à `ImageView` e, apesar de conseguirmos passar pra `ImageView` um `Bitmap`, não conseguimos recuperar o `Bitmap` depois. Precisamos então de alguma forma adicionar alguma informação à `ImageView` pra recuperá-la depois.

A componente `ImageView` possui a propriedade `tag`, que tem justamente a função de guardar qualquer objeto junto com uma `ImageView`, ou seja, é do tipo `Any`. Usaremos ela para já passar direto a imagem na `Base64`:

```
val fotoNaBase64: String = // codifica imagem para a Base64
tweet_foto.tag = fotoNaBase64
```

Para depois recuperá-la na hora de criar o `tweet`, lembrando que precisamos dizer para o `Kotlin` que o que está na `tag` é do tipo `String` e pode ser `nulo`:

```
val mensagem: String = // pega mensagem do EditText
val foto: String? = tweet_foto.tag as String?

val tweet: Tweet = Tweet(mensagemDoTweet, foto)
```

## 5.9 EXERCÍCIO: SALVANDO A FOTO

### Objetivo

Salvar a foto no banco ao cadastrar um tweet

### Passo a passo

- Precisamos fazer o `Tweet` possuir um atributo para armazenar a foto, mas não é todo `Tweet` que irá possuir uma foto, fazendo com que esse atributo possa ter valor `null`. Por isso, usaremos o operador de opcional (`?`):

```
@Entity
data class Tweet(val mensagem: String,
 val foto: String?,
 @PrimaryKey(autoGenerate = true) val id: Int = 0) {
}
```

- Agora precisamos falar que nosso banco teve uma mudança alterando a versão dele.

```
@Database(entities = [Tweet::class], version = 1-2)
abstract class TwittelumDatabase : RoomDatabase() {
```

```
// resto do código
}
```

3. Agora precisamos deixar claro o que ocorreu de mudança no banco nessa atualização:

```
@Database(entities = [Tweet::class], version = 2)
abstract class TwittelumDatabase : RoomDatabase() {
 // demais códigos
 companion object {

 // restante do código

 private fun criaBanco(context: Context): TwittelumDatabase {
 return Room.databaseBuilder(context, TwittelumDatabase::class.java, DATABASE)
 .allowMainThreadQueries()
 .addMigrations(Migration1Para2)
 .build()
 }
 }
}
```

O código acima ainda não vai compilar.

4. Agora precisamos criar esse objeto `Migration1Para2` no pacote `bancodedados`:

```
object Migration1Para2 : Migration(1, 2) {

 override fun migrate(database: SupportsSQLiteDatabase) {

 val sql = "alter table Tweet add column foto text"
 database.execSQL(sql)

 }
}
```

5. Agora precisamos ter a string que representa a foto. Para isso usaremos um formato bem conhecido no mundo android, chamado Base64. Contudo, precisamos ensinar o `Bitmap` a se converter para uma string nesse formato e, para isso, usaremos um recurso que o Kotlin provê, chamado `extension function`. Crie um arquivo chamado `BitmapExtension.kt` em um novo pacote, chamado `extensions`, e adicione o método `decodificaParaBase64`:

```
fun Bitmap.decodificaParaBase64(): String {

 val byteArrayOutputStream = ByteArrayOutputStream()
 this.compress(Bitmap.CompressFormat.JPEG, 100, byteArrayOutputStream)
 val byteArray = byteArrayOutputStream.toByteArray()

 return Base64.encodeToString(byteArray, Base64.DEFAULT)
}
```

6. Vamos usar esse método no momento em que temos o resultado da camera e criamos o `Bitmap`:

```
private fun carregaFoto() {

 val bitmap = BitmapFactory.decodeFile(localFoto)
```

```

 val bm = Bitmap.createScaledBitmap(bitmap, bitmap.width, bitmap.height, true)

 tweet_foto.setImageBitmap(bm)

 val fotoNaBase64 = bm.decodeToString()

 tweet_foto.scaleType = ImageView.ScaleType.FIT_XY

}

```

7. Agora será necessário salvar essa string. Para isso, a colocaremos dentro do `ImageView` com atributo `tag`:

```

private fun carregaFoto() {

 val bitmap = BitmapFactory.decodeFile(localFoto)

 val bm = Bitmap.createScaledBitmap(bitmap, bitmap.width, bitmap.height, true)

 tweet_foto.setImageBitmap(bm)

 val fotoNaBase64 = bm.decodeToString()

 tweet_foto.tag = fotoNaBase64

 tweet_foto.scaleType = ImageView.ScaleType.FIT_XY

}

```

8. Vamos fazer uma pequena alteração no nosso método `publicaTweet`, pensando na semântica:

```

private fun publicaTweet() {

 val tweet = criaTweet()

 viewModel.salva(tweet)

 Toast.makeText(this, "$tweet foi salvo com sucesso :D", Toast.LENGTH_LONG).show()
}

```

O código acima não vai compilar.

9. Precisamos implementar o método `criaTweet` para que nosso código compile:

```

fun criaTweet(): Tweet {

 val campoDeMensagemDoTweet = findViewById<EditText>(R.id.tweet_mensagem)

 val mensagemDoTweet: String = campoDeMensagemDoTweet.text.toString()

 val foto: String? = tweet_foto.tag as String?

 return Tweet(mensagemDoTweet, foto)
}

```

10. Rode novamente o aplicativo e tente salvar um novo tweet com e sem foto.

# PERSONALIZANDO NOSSA LISTA

## 6.1 O QUE SABEREI FAZER NO FIM DESSE CAPÍTULO ?

- Criar lista com layouts não criados pelo Android
- Trabalhar com Layout Inflater

Aprenda se divertindo na Alura Start!



Você conhece alguém que tem potencial para tecnologia e programação, mas que nunca escreveu uma linha de código? Pode ser um filho, sobrinho, amigo ou parente distante. Na **Alura Start** ela vai poder criar games, apps, sites e muito mais! É o **começo da jornada com programação e a porta de entrada para uma possível carreira de sucesso**. Ela vai estudar em seu próprio ritmo e com a melhor didática. A qualidade da conceituada Alura, agora para Starters.

[Conheça os cursos online da Alura Start!](#)

## 6.2 CRIANDO INTERFACE DO ITEM PERSONALIZADA

Agora que temos imagens dos *tweets*, vamos melhorar o *layout* de nossa listagem, customizando a aparência das linhas da `ListView` que apresenta a lista de *tweets*.

Atualmente, a lista de *tweets* é vinculada à `ListView` pelo *adapter* usando um *layout* do próprio Android:

```
val adapter = ArrayAdapter<Tweet>(this, android.R.layout.simple_list_item_1, tweets)
```

Esse layout encontra-se dentro do jar da SDK do Android.

O que queremos é deixar de usar o `android.R.layout.simple_list_item_1` e passar a usar um *layout* customizado nosso. Primeiro, temos que criar esse *layout* que irá definir a interface de um item da

lista. Criaremos um arquivo XML dentro da pasta res/layout , assim como fizemos pros layouts de tela.

## Alinhando componentes com LinearLayout

Até agora vimos um tipo de contêiner de layout que é baseado em restrições, o ConstraintLayout . Mas, se quisermos apenas alinhar duas ou mais componentes em um layout, podemos utilizar um outro contêiner: LinearLayout . Precisamos apenas informar para ele qual o sentido do alinhamento, se é vertical ou horizontal por meio do atributo android:orientation . Por exemplo, se quisermos em algum ponto do layout deixar um texto embaixo de outro, podemos adicionar o seguinte código:

```
<LinearLayout
 android:layout_width="match_parent"
 android:layout_height="match_parent"
 android:orientation="vertical">

 <TextView
 android:id="@+id/texto1"
 android:layout_width="wrap_content"
 android:layout_height="wrap_content" />

 <TextView
 android:id="@+id/texto2"
 android:layout_width="wrap_content"
 android:layout_height="wrap_content" />

</LinearLayout>
```

## Mostrando imagens redondas

Vamos querer mostrar a foto do usuário que criou o tweet da mesma forma que são mostrados os usuários em diversos aplicativos: em formato circular.

Há várias formas de fazer isso, mas uma bem simples é utilizar a componente CardView , que nos permite deixar as bordas arredondadas. A ideia é arredondar tanto as bordas que o CardView resultante terá o formato de um círculo. Para conseguirmos esse efeito, temos que arredondar a borda com um raio de metade do tamanho do CardView :

```
<androidx.cardview.widget.CardView
 android:layout_width="50dp"
 android:layout_height="50dp"
 app:cardCornerRadius="25dp">
 <!-- ImageView com imagem -->
</androidx.cardview.widget.CardView>
```

## Colocando as informações do tweet no layout

Agora teremos um layout definindo um item da lista com ImageView s e TextView s e queremos colocar uma informação específica do tweet para cada componente. O responsável por criar uma View a partir de um objeto qualquer é o Adapter e estamos utilizando a classe ArrayAdapter para isso até

o momento. Porém, com ela só conseguimos criar uma `View` apenas com um texto, pelo modo que essa classe foi definida.

Precisaremos então criar nosso próprio `Adapter`. Para isso, o `Android` disponibiliza a classe abstrata `BaseAdapter`, que nos força a implementar os métodos que, por exemplo, a `ListView` vai chamar. Assim, basta herdarmos da classe `BaseAdapter` e automaticamente seremos obrigados a implementar os métodos `getView`, `getItemId`, `getCount` e `getItem`:

```
class TweetAdapter() : BaseAdapter() {

 override fun getView(position: Int, convertView: View?, parent: ViewGroup?): View {
 return ???
 }

 override fun getItemId(position: Int): Long {
 return ???
 }

 override fun getItem(position: Int): Any {
 return ???
 }

 override fun getCount(): Int {
 return ???
 }
}
```

Com o método `getCount` precisamos explicar quantos itens vamos mostrar na `ListView`; a partir desse método o `Android` consegue determinar, por exemplo, o tamanho inicial da `ListView` (lembre-se que temos diversos tamanhos possíveis de tela!). Para informar o número de itens, precisamos ter a quantidade de elementos da lista de `tweets`, portanto vamos recebê-la no construtor de nossa classe e então chamar o método `size` da lista:

```
class TweetAdapter(private val tweets: List<Tweet>) : BaseAdapter() {

 override fun getCount(): Int {
 return tweets.size
 }

}
```

No método `getItemId` precisamos criar um identificador único para o item na tela. Usaremos o próprio valor do atributo `id` do `tweet`. Para pegar um `tweet` na posição `posicao` da lista de `tweets`, o `Kotlin` usa a sintaxe `tweets[posicao]`. Além disso, como o `id` do `Tweet` é do tipo `Int`, precisamos transformá-lo no tipo `Long` com o método `toLong`:

```
override fun getItemId(posicao: Int): Long {
 return tweets[posicao].id.toLong()
}
```

No método `getItem` precisamos retornar o objeto que se encontra na posição selecionada da nossa `ListView`. Como desejamos que as posições da lista de `tweets` e da `ListView` sejam as mesmas,

vamos apenas retornar o elemento da lista no índice `posicao` :

```
override fun getItem(posicao: Int): Any {
 return tweets[posicao]
}
```

Por fim, precisamos implementar o método `getView`, que recebe a posição na lista da `View` que será retornada, uma `View` que pode ser reaproveitada, se houver, e o `ViewGroup` que será pai dessa `View`, que no nosso caso seria a `ListView`. O esqueleto do método deve ser:

```
override fun getView(position: Int, convertView: View?, parent: ViewGroup?): View {
}
```

Esse método deve retornar uma `View`, que será a linha da lista apresentada na tela de nosso dispositivo. Queremos que a linha siga um *layout* personalizado nosso.

Mas como **preencher** nosso *layout* com os dados corretos? Precisamos em primeiro lugar transformar nosso *layout*, que está definido em um `xml`, para um objeto do tipo `View`. Da mesma forma que fizemos para conseguir um objeto `Menu` a partir de um `xml`, faremos para conseguir uma `View` do `xml` de *layout*: usaremos um **Inflater**. Para inflar o *layout* o Android disponibiliza o `LayoutInflater`. Para obtermos uma instância de um `LayoutInflater`, basta chamarmos o método estático `from` a partir da classe `LayoutInflater`, passando um objeto `Context`. Como não temos acesso a nenhum `Context` diretamente, vamos pedir ao pai dessa `View` que será criada, se ele não for nulo:

```
val inflater = LayoutInflater.from(parent?.context)
```

Para **inflarmos** um *layout*, basta invocarmos o método `inflate` passando o atributo da classe `R` correspondente ao arquivo de *layout* a ser inflado, quem será o pai dessa `View` e se já queremos prender a `View` no pai:

```
val view = inflater.inflate(R.layout.item, parent, false)
```

Agora basta pegarmos da lista de `tweets` o objeto `Tweet` que está na mesma posição que a `View` que acabamos de criar e colocarmos os dados desse `tweet` nas componentes dessa `View`. Para localizar um elemento da `View`, podemos fazer o importe estático pelo `id` do elemento. Por exemplo, podemos colocar a mensagem do `tweet` em um `TextView` com o `id` "item\_mensagem" com o código:

```
view.item_mensagem.text = tweet.mensagem
```

Para colocar a foto do `Tweet` em uma `ImageView` com o `id` "item\_foto", vamos chamar o método `setBitmap` do `ImageView`, mas precisamos primeiro ter um objeto `Bitmap` para passar a esse método e nosso objeto `Tweet` só possui a imagem como texto na Base64.

## Transformando texto para Bitmap usando a Base64

---

Para **decodificar** o texto que está na Base64 para *bytes*, podemos usar o método estático `decode` da classe `Base64` importada do `Kotlin`. Esse método recebe o texto que será decodificado e a indicação de quais caracteres foram utilizados na codificação. Como no nosso caso não precisamos de nenhuma restrição, avisamos com a constante `Base64.DEFAULT` :

```
val byteArray: ByteArray = Base64.decode(foto, Base64.DEFAULT)
```

Agora precisamos criar um `Bitmap` a partir desse objeto `ByteArray`. Faremos isso utilizando a fábrica de `Bitmap` s, `BitmapFactory`, e chamando seu método estático `decodeByteArray`. Passaremos para esse método o *array* de *bytes*, a posição do *array* que será o início da decodificação e a quantidade de *bytes* que serão decodificados:

```
val bitmap = BitmapFactory.decodeByteArray(byteArray, 0, byteArray.size)
```

Finalmente, podemos colocar esse `Bitmap` na `ImageView`:

```
view.item_foto.setImageBitmap(Carregador.decoder(it))
```

## Deixando ImageView visível se tiver foto

Como a foto pode ser nula, só queremos transformar em `Bitmap` e adicionar à `ImageView` se a foto não for nula. Com o `Kotlin`, conseguimos de forma prática, sem uso de `if`s, executar um código apenas se alguma variável não for nula. Fazemos isso usando o operador `let`:

```
tweet.foto?.let {
 // transforma foto em Bitmap e coloca na ImageView
}
```

Com o código acima, a lógica dentro das chaves {} só será executada se o valor de `foto` não for nulo. Se for nulo, a lógica é ignorada.

Mesmo não colocando uma foto na `ImageView`, a componente ocupa um espaço na tela. Para qualquer `View`, há um atributo chamado `android:visibility`, que podemos definir como `visible`, indicando que veremos a componente na tela; `invisible`, indicando que não veremos a componente na tela, mas o espaço dela fica reservado no *layout*; `gone`, indicando que a componente está completamente escondida, como se ela não estivesse na tela.

Podemos então adicionar no *layout* o atributo `android:visibility` como `gone` para a `ImageView` da foto do `tweet` e, se a foto não for nula, mudar a visibilidade para `visible` quando estiver colocando a foto na `ImageView`:

```
tweet.foto?.let {
 view.item_foto.visibility = View.VISIBLE
 view.item_foto.setImageBitmap(Carregador.decoder(it))
}
```

#### **PARA SABER MAIS: O MÉTODO INFLATE**

Ao inflar um *layout*, é uma boa prática passar como argumento quem será o pai dessa *view*, para que o Android possa calcular as suas dimensões. Omitir esse `parent` pode causar alguns comportamentos bem estranhos.

Veja mais detalhes neste *post* no blog da Caelum, que está com código em Java, mas explica melhor o método: <http://blog.caelum.com.br/conhecendo-melhor-o-metodo-inflate/>

## 6.3 EXERCÍCIO: EXIBINDO A FOTO NA LISTA

### Objetivo

Criar um adapter personalizado e usá-lo para exibir a lista principal.

### Passo a passo

1. Vamos criar um *layout* que represente nossos *tweets*. Para isso, crie o arquivo `item_tweet.xml` na pasta `layout` e um ícone, de nome `ic_pessoa`, que vai aparecer caso o *tweet* não tenha foto:

```
<?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"
 android:layout_width="match_parent"
 android:layout_height="wrap_content">

 <LinearLayout
 android:layout_width="match_parent"
 android:layout_height="wrap_content"
 android:layout_margin="10dp"
 android:orientation="horizontal">

 <androidx.cardview.widget.CardView
 android:layout_width="50dp"
 android:layout_height="50dp"
 app:cardCornerRadius="25dp">

 <ImageView
 android:id="@+id/item_perfil"
 android:layout_width="match_parent"
 android:layout_height="match_parent"
 android:src="@drawable/ic_pessoa" />
 </androidx.cardview.widget.CardView>

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

 <TextView>
```

```

 android:id="@+id/item_dono"
 android:layout_width="wrap_content"
 android:layout_height="wrap_content"
 android:layout_gravity="start"
 android:layout_margin="10dp"
 android:text="Seu nome de usuário"
 android:textColor="#0b0b0b"
 android:textStyle="bold" />

 <TextView
 android:id="@+id/item_tweet_texto"
 android:layout_width="match_parent"
 android:layout_height="wrap_content"
 android:layout_margin="8dp"
 android:text="Seu tweet vem aqui" />

 <androidx.cardview.widget.CardView
 android:layout_width="match_parent"
 android:layout_height="wrap_content"
 app:cardCornerRadius="5dp">

 <ImageView
 android:id="@+id/item_tweet_foto"
 android:layout_width="match_parent"
 android:layout_height="wrap_content"
 android:adjustViewBounds="true"
 android:maxHeight="100dp"
 android:scaleType="fitXY"
 android:visibility="gone" />
 </androidx.cardview.widget.CardView>

</LinearLayout>
</LinearLayout>
</LinearLayout>

```

2. Agora vamos criar nosso Adapter personalizado. Para isso, crie uma nova classe no novo pacote adapter e extenda de BaseAdapter :

```
class TweetAdapter() : BaseAdapter() {}
```

3. Precisamos implementar alguns métodos quando extendemos de BaseAdapter . O primeiro é o método getCount , que vai ser responsável por ditar quantos itens a lista possuirá, portanto precisaremos também possuir a lista para termos essa informação:

```
class TweetAdapter(private val tweets: List<Tweet>) : BaseAdapter() {

 override fun getCount(): Int {
 return tweets.size
 }

}
```

4. Próximo passo será implementar o método getItem , que é responsável por nos devolver o item de determinada posição:

```
class TweetAdapter(private val tweets: List<Tweet>) : BaseAdapter() {

 // restante do código
```

```

 override fun getItem(position: Int): Any {
 return tweets[position]
 }
 }

```

5. Ainda precisaremos definir qual é o id da view que está sendo adicionada na tela, esse comportamento é feito no método `getItemId` que também precisaremos sobrescrever:

```

class TweetAdapter(private val tweets: List<Tweet>) : BaseAdapter() {

 // restante do código

 override fun getItemId(position: Int): Long {
 return tweets[position].id.toLong()
 }

}

```

6. Por fim precisamos fazer o trabalho mais importante do nosso adapter que é adaptar nosso objeto para ser exibido como uma view na tela, por isso temos que sobrescrever o método `getView`:

```

class TweetAdapter(private val tweets: List<Tweet>) : BaseAdapter() {

 // restante do código

 override fun getView(position: Int, convertView: View?, parent: ViewGroup?): View {
 }

}

```

7. Vamos recuperar qual tweet estamos querendo exibir:

```

class TweetAdapter(private val tweets: List<Tweet>) : BaseAdapter() {

 // restante do código

 override fun getView(position: Int, convertView: View?, parent: ViewGroup?): View {
 val tweet = tweets[position]
 }

}

```

8. Agora temos que fazer nosso xml virar um objeto para podermos populá-lo com os dados de nosso Tweet. Pensando nisso, o android possui uma classe chamada `LayoutInflater` que a partir de um xml consegue criar uma instância de `View`:

---

```

class TweetAdapter(private val tweets: List<Tweet>) : BaseAdapter() {

 // restante do código

 override fun getView(position: Int, convertView: View?, parent: ViewGroup?): View {
 val tweet = tweets[position]

 val inflater = LayoutInflater.from(parent?.context)

```

```
 }
}
}
```

9. Agora com a instância de inflater podemos inflar o nosso layout:

```
class TweetAdapter(private val tweets: List<Tweet>) : BaseAdapter() {

 // restante do código

 override fun getView(position: Int, convertView: View?, parent: ViewGroup?): View {

 val tweet = tweets[position]

 val inflater = LayoutInflater.from(parent?.context)

 val view = inflater.inflate(R.layout.item_tweet, parent, false)

 return view
 }
}
```

10. Dado que temos a view, basta popularmos os campos dela:

```
class TweetAdapter(private val tweets: List<Tweet>) : BaseAdapter() {

 // restante do código

 override fun getView(position: Int, convertView: View?, parent: ViewGroup?): View {

 val tweet = tweets[position]

 val inflater = LayoutInflater.from(parent?.context)

 val view = inflater.inflate(R.layout.item_tweet, parent, false)

 view.item_tweet_texto.text = tweet.mensagem

 tweet.foto?.let {
 view.item_tweet_foto.visibility = View.VISIBLE
 view.item_tweet_foto.setImageBitmap(Carregador.decode(it))
 }

 return view
 }
}
```

O código acima ainda não vai compilar.

11. Precisamos apenas definir esse **Objeto** chamado **Carregador** . Vamos criá-lo no pacote extensions :

```
object Carregador {

 fun decode(foto: String): Bitmap {

 val decode: ByteArray = Base64.decode(foto, Base64.DEFAULT)
```

```
 val bitmap = BitmapFactory.decodeByteArray(decode, 0, decode.size)

 return bitmap
 }
}
```

2. Por fim precisamos alterar nossa lista de tweets para usar nosso novo adapter, mas apenas se a lista de tweets não for nula:

```
class ListaActivity : AppCompatActivity() {

 // restante do código

 private fun observer(): Observer<List<Tweet>> {

 return Observer { tweets ->
 tweets?.let {
 lista_tweet.adapter = TweetAdapter(tweets)
 }
 }
 }
}
```

3. Rode o aplicativo e veja se a lista agora mostra a foto e outros detalhes.

# DESACOPLANDO COMPORTAMENTOS COM FRAGMENTS

## 7.1 O QUE SABEREI FAZER NO FIM DESSE CAPÍTULO ?

- Dividir seu aplicativo de uma forma mais coesa
- Views de navegações mais interessantes
- Aprender a trabalhar com Fragments

**Seus livros de tecnologia parecem do século passado?**



Conheça a **Casa do Código**, uma **nova** editora, com autores de destaque no mercado, foco em **ebooks** (PDF, epub, mobi), preços **imbatíveis** e assuntos **atuais**.

Com a curadoria da **Caelum** e excelentes autores, é uma abordagem **diferente** para livros de tecnologia no Brasil.

[Casa do Código, Livros de Tecnologia.](https://www.casadocodigo.com.br/livros/tecnologia)

## 7.2 EXERCÍCIO: PEGANDO O PROJETO

### Objetivo

Dar início a um novo aplicativo de *tweet*, mas aproveitando tudo que já foi criado antes. O projeto do github no link [https://github.com/caelum/TwittelumWeb\\_](https://github.com/caelum/TwittelumWeb_) contém as classes que já criamos até o momento no curso e vamos utilizar daqui pra frente. Basta então importar este projeto pro Android Studio.

### Passo a passo

1. Entre no github da caelum [https://github.com/caelum/TwittelumWeb\\_](https://github.com/caelum/TwittelumWeb_) e baixe o projeto na sua

máquina clicando em `Clone or download` e, em seguida, em `Download ZIP`.

2. Com o projeto baixado, precisaremos adicioná-lo no *Android Studio*, para isso escolha a opção de importar um projeto já existente na tela inicial.
3. Aceite todas as permissões que forem solicitadas para alteração do projeto.
4. Rode o aplicativo e cheque se aparece uma tela em branco.

## 7.3 COLOCANDO UM MENU DE NAVEGAÇÃO COM BOTTOMNAVIGATIONVIEW

Muitas vezes queremos deixar as principais telas da nossa aplicação acessíveis com apenas um toque a partir de qualquer outra tela. Para conseguir essa funcionalidade, podemos criar um menu que terá um item para cada destino que queremos. A única forma que vimos de mostrar um menu na tela é pelo **Menu de Opções**, mostrado geralmente na **Barra de Ações**, mas o menu mostrado na **ActionBar** é focado para ações relacionadas à tela atual. Se queremos disponibilizar acesso direto a outras telas, talvez nada relacionadas à que estamos vendo no momento, o **Menu de Opções** não é muito indicado. É muito comum em vários aplicativos, como Google Fotos, Google Maps e Spotify, vermos listados na parte de baixo da tela ícones junto com uma palavra explicando para qual ela cada ícone nos levará. E cada ícone nos leva justamente para uma tela principal da aplicação. Para essa funcionalidade, o **Material Design** sugere o uso de um **Bottom Navigation**, que o *Android* nos oferece pela componente `BottomNavigationView`.

Pelo *Material Design*, a altura da componente deve ser de `56dp`, para conseguirmos clicar com o dedão nos itens, e a largura deve ser a componente toda. Também é recomendado que a cor de fundo seja a cor primária (principal) do tema da aplicação, para ficar parecido com a Barra de Ações. Definimos isso com o atributo `android:background`. Por fim, como queremos mostrar vários itens, que nos levarão para diferentes destinos, precisamos criar um menu, como já fizemos, definindo um `xml` de menu e passando ao atributo `app:menu`. A recomendação do *Material Design* é usar o *Bottom Navigation* se quisermos mostrar de 3 a 5 destinos. Se quisermos criar um `BottomNavigationView` dentro de um `ConstraintLayout` e o menu estiver no arquivo `menu_bottom_navigation.xml`, o código ficará parecido com:

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.ConstraintLayout xmlns:android="http://schemas.android.com/apk/res/android"
 android:layout_width="match_parent"
 android:layout_height="match_parent"
 xmlns:app="http://schemas.android.com/apk/res-auto">

 <com.google.android.material.bottomnavigation.BottomNavigationView
 android:id="@+id/bottom_navigation"
 android:layout_width="0dp"
 android:layout_height="56dp"
 android:background="@color/colorPrimary"
 app:layout_constraintBottom_toBottomOf="parent"
 app:layout_constraintEnd_toEndOf="parent"
 app:layout_constraintStart_toStartOf="parent"
 app:layout_constraintTop_toTopOf="parent" />
```

```
 app:layout_constraintStart_toStartOf="parent"
 app:menu="@menu/menu_bottom_navigation" />

</androidx.constraintlayout.widget.ConstraintLayout>
```

Com o *layout* acima, ao ver essa tela em um dispositivo, os itens e respectivos nomes vão aparecer, mas o que estiver selecionado não vai aparecer. Na verdade, todos os itens estão lá, mas, quando o usuário seleciona um item, esse item está com a mesma cor que o fundo. Podemos mudar a cor do ícone e do texto pelos atributos `app:itemIconTint` e `app:itemTextColor`, respectivamente.

Porém, quando alteramos esses atributos, todos os itens passam a ficar com as cores que definirmos para sempre, independente se o item está selecionado ou não. O tamanho do item muda levemente, mas a cor continua a mesma. Enquanto o usuário estiver navegando entre as telas, ele vai querer em determinado momento checar em qual das telas ele está, ou confirmar se o clique que ele fez no item foi percebido pela tela e ativou corretamente o item. Portanto, o comportamento inicial de mudar a cor do ícone e do texto para identificar o item que está selecionado, junto com mudar o tamanho do ícone e do texto, facilita muito a percepção do usuário.

## Mudando uma imagem de acordo com seu estado

Queremos no fundo definir uma cor para o ícone se ele estiver selecionado e outra cor se o item não estiver selecionado, ou seja, mudar a imagem (cor no ícone, por exemplo) que aparece no item dependendo do estado dele. Podemos fazer isso com um `drawable` de lista de estados, criando um arquivo `xml` do tipo `drawable`. Nele vamos usar a tag `selector` e, dentro dela, definiremos um `item` para cada estado que queremos configurar uma imagem diferente. Por exemplo, para definir a cor branca se o item estiver ativo, usamos o atributo `android:state_checked` para verificar se o item está selecionado e o atributo `android:color` para usar a cor branca:

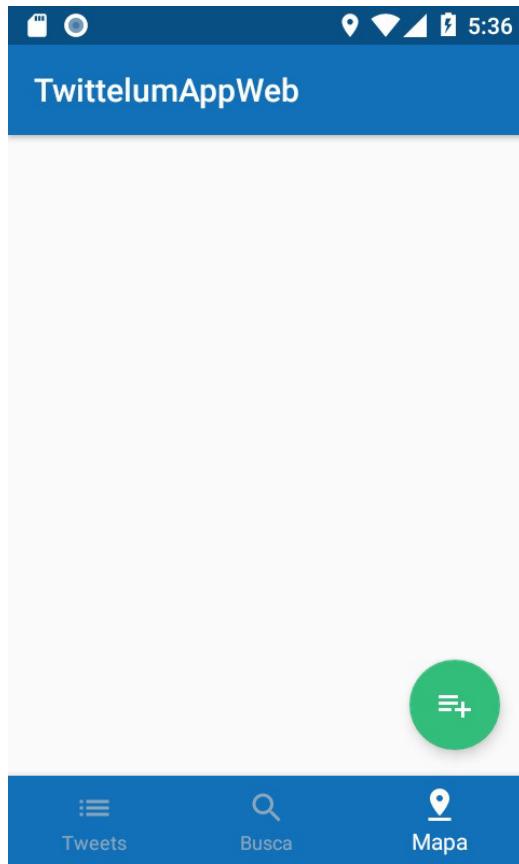
```
<?xml version="1.0" encoding="utf-8"?>
<selector xmlns:android="http://schemas.android.com/apk/res/android">
 <item android:state_checked="true"
 android:color="@color/branco"/>
</selector>
```

Uma vez que definimos todos os estados que queremos no arquivo `drawable`, chamamos esse arquivo ao invés de passar diretamente a cor para os atributos que queremos que variem de acordo com o estado. No nosso caso, eram o `app:itemIconTint` e o `app:itemTextColor`.

## 7.4 EXERCÍCIO: DEIXANDO NOSSO LAYOUT MAIS REAL

### Objetivo

Fazer nossa `Activity` possuir `BottomNavigationView` e um `FloatingActionButton` com os respectivos ícones.



## Passo a passo

1. Vamos adicionar o `BottomNavigationView` em nosso `activity_main.xml`. Você pode fazer via xml ou via editor visual, vamos deixar aqui como deveria ficar seu xml:

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.ConstraintLayout xmlns:android="http://schemas.android.com/apk/res/android"
 android:layout_width="match_parent"
 android:layout_height="match_parent"
 xmlns:app="http://schemas.android.com/apk/res-auto">

 <com.google.android.material.bottomnavigation.BottomNavigationView
 android:id="@+id/bottom_navigation"
 android:layout_width="0dp"
 android:layout_height="56dp"
 android:background="@color/colorPrimary"
 app:itemIconTint="@drawable/item_selecionado"
 app:itemTextColor="@drawable/item_selecionado"
 app:layout_constraintBottom_toBottomOf="parent"
 app:layout_constraintEnd_toEndOf="parent"
 app:layout_constraintStart_toStartOf="parent"
 app:menu="@menu/menu_bottom_navigation" />

</androidx.constraintlayout.widget.ConstraintLayout>
```

- Você deve estar com alguns erros agora, não se preocupe, nos próximos itens vamos resolvê-los.

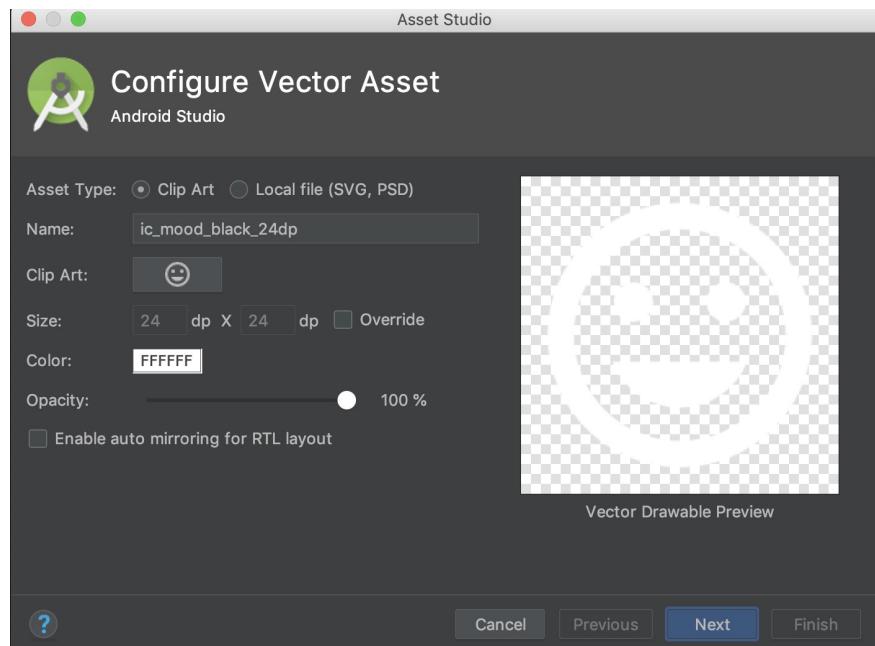
2. Vamos resolver o primeiro item em vermelho, criando um arquivo que se chama `item_selecionado`. Na pasta `drawable`, com o botão direito, crie um `drawable resource file` e seu conteúdo deverá ficar desta forma:

```
<?xml version="1.0" encoding="utf-8"?>
<selector xmlns:android="http://schemas.android.com/apk/res/android">
 <item android:state_checked="true"
 android:color="@color/branco"/>
 <item android:state_checked="false"
 android:color="@color/cinza"/>
</selector>
```

3. Agora precisamos criar o arquivo `menu_bottom_navigation`, que será um arquivo de menu:

```
<?xml version="1.0" encoding="utf-8"?>
<menu xmlns:android="http://schemas.android.com/apk/res/android">
 <item
 android:id="@+id/menu_tweets"
 android:icon="@drawable/lista"
 android:title="Tweets" />
 <item
 android:id="@+id/menu_busca"
 android:icon="@drawable/busca"
 android:title="Busca" />
 <item
 android:id="@+id/menu_mapa"
 android:icon="@drawable/mapa"
 android:title="Mapa" />
</menu>
```

4. Precisamos criar esse ícones agora, para isso, vá na pasta `drawable` e com o botão direito vá em *new*  
-> *vector asset*



- Crie um novo vetor para cada imagem chamada:
  - lista
  - mapa
  - busca

5. Agora vamos adicionar um `FloatingActionButton`, também em nosso `activity_main.xml`:

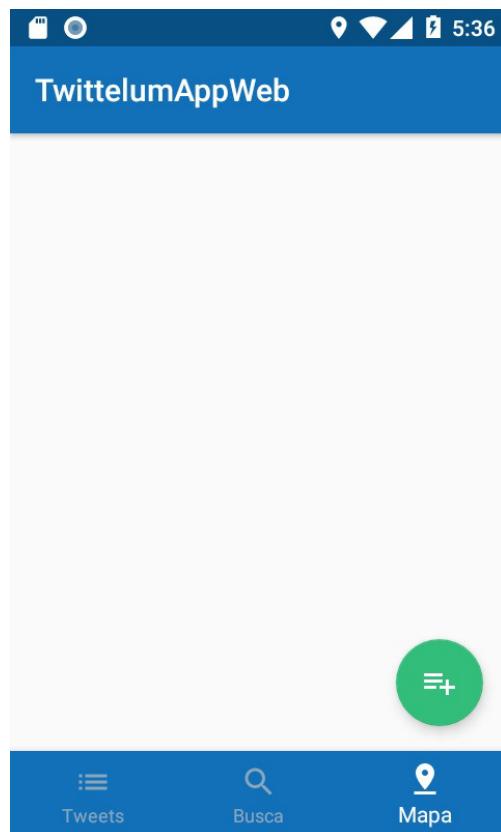
```
<?xml version="1.0" encoding="utf-8"?>
<aandroidx.constraintlayout.widget.ConstraintLayout>
 <com.google.android.material.floatingactionbutton.FloatingActionButton
 android:id="@+id/main_fab"
 android:layout_width="wrap_content"
 android:layout_height="wrap_content"
 android:layout_margin="16dp"
 android:src="@drawable/lista_novo"
 app:fabSize="normal"
 app:layout_constraintBottom_toTopOf="@id/bottom_navigation"
 app:layout_constraintEnd_toEndOf="parent" />

 <!-- BottomNavigationView -->

</aandroidx.constraintlayout.widget.ConstraintLayout>
```

6. Crie um ícone da mesma forma que fizemos no item 4 para representar o ícone `lista_novo`.

7. Rode o aplicativo e veja se seu aplicativo está assim:



**Agora é a melhor hora de respirar mais tecnologia!**



Se você está gostando dessa apostila, certamente vai aproveitar os **cursos online** que lançamos na plataforma **Alura**. Você estuda a qualquer momento com a **qualidade** Caelum. Programação, Mobile, Design, Infra, Front-End e Business! Ex-aluno da Caelum tem 15% de desconto, siga o link!

[Conheça a Alura Cursos Online.](#)

## 7.5 REAPROVEITANDO PEDAÇOS DA TELA

Muitas pessoas preferem utilizar *tablets* ao invés de *smartphones* para ter uma maior riqueza nos detalhes das imagens ou para ter acesso mais direto a algumas opções. Por exemplo, em um *tablet* aparecem mais ícones na Barra de Ações que no *smartphone*.

Mas, geralmente, a principal diferença no *layout* aparece quando vemos alguns aplicativos no modo paisagem. Por exemplo, quando estamos na listagem de emails do Gmail e clicamos em um email, geralmente a tela toda é preenchida pelo email que foi clicado. Mas, no caso de um *tablet* em modo paisagem, a lista continua aparecendo em parte da tela, dividindo a tela com o email selecionado.

O pessoal do Gmail precisou então definir um *layout* para cada tela: a de listagem, a de um email específico e a de listagem mostrando um email específico. No entanto, o *layout* da listagem e a parte com listagem do *layout* que mostra os dois é exatamente igual. Vamos pensar em como eles devem ter feito essa parte do código. Se eles copiarem e colarem código, a chance de esquecerem de atualizar um dos *layout* e deixá-los diferentes um do outro é bem grande.

Além disso, há várias lógicas que estão associadas às componentes do *layout* só com a listagem que também devem ser iguais às lógicas atreladas às componentes da parte da listagem no *layout* com a listagem e o email específico. Se atualizarmos uma lógica, teremos que atualizar a outra lógica. O ideal seria isolar esse pedaço do *layout* que está igual e essas lógicas que também são iguais para alterarmos o código em um único lugar.

Com a criação de telas cada vez mais complexas e com pedaços delas se repetindo cada vez mais em várias telas, essa necessidade ficou bem grande, fazendo com que o *Android* criasse uma **API** para facilitar esse processo de fragmentação das telas: a **API de Fragments**. Essa API está disponível a partir da versão *Honeycomb* (*Android* 3.0).

### Fragments

*Fragments* são pedaços de tela com *layout* e lógicas próprias que ficam dentro de uma *Activity*, que consegue definir a tela para o usuário. Precisamos então ter o *layout* da *Activity* e deixar nele um espaço reservado para cada fragmento que quisermos incluir, pensando na posição que queremos que ele apareça na tela. No *layout* de *Activity* que quisermos deixar espaço reservado para outras componentes, podemos usar o *FrameLayout*, uma *view* justamente com essa função.

Os fragmentos possuem ciclo de vida bem parecido com o da *Activity* e já tem vários outros comportamentos existentes, definidos na classe *Fragment*. Para criarmos um pedaço de tela, precisamos criar uma classe que herde de *Fragment*. Lembrando que, se quisermos que nossa aplicação seja compatível com dispositivos mais antigos e continue funcionando com aqueles que não dão suporte a fragmentos, devemos importar a classe *Fragment* do pacote de suporte:

```
class MeuFragmento : Fragment() {}
```

Como o fragmento precisa ser inserido no *layout* de uma *Activity*, um objeto *Fragment* precisa ser capaz de disponibilizar a *view* que estará no *layout*. Por isso, um *Fragment* possui o método *onCreateView*.

Para definir o *layout* do fragmento, faremos como fazemos para definir qualquer *layout*: criaremos um *xml*. Como já vimos, para se conseguir um objeto *View* a partir de um *xml*, precisamos de um *LayoutInflater*. Felizmente, o método *onCreateView* já recebe o *inflater* que precisamos.

Às vezes precisamos de informações do contêiner da *view* que será criada, como a largura e altura se definimos que a *view* deverá ter o mesmo tamanho que o contêiner. Por isso, o método *onCreateView* recebe quem será o contêiner da *view* que será criada.

Por fim, o *onCreateView* recebe, como o método *onCreate* de uma *Activity*, um *Bundle* com informações que foram salvas em algum estado anterior do fragmento.

Se quisermos criar um fragmento de lista, faríamos um código parecido com:

```
class ListaFragment : Fragment() {

 override fun onCreateView(inflater: LayoutInflater,
 container: ViewGroup?,
 savedInstanceState: Bundle?): View? {

 val view = inflater.inflate(R.layout.fragment_lista, container, false)

 return view
 }
}
```

## 7.6 TROCANDO FRAGMENTOS NA MESMA TELA

---

Em muitas situações, queremos aproveitar a mesma estrutura de tela e trocar apenas um pedaço da tela por outro dependendo da ação do usuário. Por exemplo, quando é uma aplicação que usa *bottom navigation*, em que a parte de baixo da tela e outras componentes vão se manter, mas o meio da tela vai ser trocado de acordo com o item que o usuário selecionar. Utilizando fragmentos fica muito fácil de separar os pedaços de tela que podem mudar e deixar as outras componentes no próprio *layout* da *Activity*.

Durante o processo de troca de um fragmento por outro, porém, pode ser que aconteça uma falha, e aí o ideal é voltar para o fragmento anterior e manter a consistência para o usuário. Além disso, o ciclo de vida dos dois fragmentos é alterado. Como já comentamos, não somos nós, desenvolvedores, que gerenciam o ciclo de vida de qualquer objeto no *Android*. Também é bem trabalhoso gerenciar se o estado das componentes que estão visíveis na tela é consistente e não tem parte de um fragmento e parte de outro. Por isso, o *Android* disponibiliza pra gente um gerenciador de fragmentos. A classe responsável por gerenciar fragmentos que vem da biblioteca de suporte é o *SupportFragmentManager*.

## Gerenciando fragmentos

A classe *AppCompatActivity* já possui uma propriedade do tipo *SupportFragmentManager*, chamada de *supportFragmentManager*. Como queremos trocar um fragmento por outro, mas queremos que volte para um estado anterior caso aconteça qualquer erro, precisamos pedir para o *supportFragmentManager* iniciar uma transação para ele saber qual é esse estado anterior. Fazemos isso com o método *beginTransaction*, que nos devolve um *FragmentTransaction*:

```
val fragmentTransaction: FragmentTransaction = supportFragmentManager.beginTransaction()
```

Com o objeto *FragmentTransaction*, podemos fazer as trocas de fragmento que quisermos com o método *replace*. Precisamos passar para ele uma instância do fragmento que queremos que ele coloque na tela e o local onde esse fragmento ficará, ou seja, o *id* da *view* que será preenchida pelo fragmento. Se queremos substituir o estado atual do *FrameLayout* com *id* "frame\_centeral" pelo fragmento *ListaFragment*, podemos escrever o seguinte código:

```
fragmentTransaction.replace(R.id.frame_centeral, ListaFragment())
```

Depois de realizarmos todas as trocas de fragmento que queremos nessa transação, precisamos avisar que será encerrada a transação com o método *commit*:

```
fragmentTransaction.commit()
```

## Usando fragmentos em uma tela com Bottom Navigation

A maioria dos aplicativos que utilizam o *BottomNavigationView* mantém essa componente na tela e trocam só um espaço na tela de acordo com o item que é selecionado. Esse espaço de tela é preenchido com um *Fragment* por meio do *SupportFragmentManager*. Mas essa troca só é feita quando o

usuário clica em um item. Para associarmos uma ação à seleção de um item do `BottomNavigationView`, há um *listener* chamado `BottomNavigationView.OnNavigationItemSelectedListener`, que define o método `onNavigationItemSelected`. Este método recebe o objeto `MenuItem` referente ao item que foi clicado e seu retorno indica se o item deve ser mostrado como selecionado na tela.

Podemos definir esse *listener* com o método `setOnNavigationItemSelectedListener` de um objeto `BottomNavigationView`. Para um `BottomNavigationView` com id "bottom\_navigation", podemos escrever o código:

```
bottom_navigation.setOnNavigationItemSelected { item ->
 when (item.itemId) {
 R.id.lista_tweets -> {
 // faz a troca para o fragmento de lista usando o FragmentManager
 true
 }
 else -> {
 false
 }
 }
}
```

Agora, quando o usuário clicar em um item do `BottomNavigationView`, o *listener* será executado e será trocado um pedaço da tela para o `Fragment` correspondente ao item.

## Iniciando a tela com um item selecionado

No momento em que a tela é aberta, o usuário ainda não clicou em nenhum item. Por isso, não será ativado o *listener* definido pelo `setOnNavigationItemSelectedListener` e o usuário não verá nada no espaço reservado para os `Fragment`s, justamente porque originalmente ele é apenas um `FrameLayout`, uma componente que reserva um espaço na tela.

Se quisermos que a tela já inicie com algum item selecionado, podemos definir o `id` de um item para a propriedade `selectedItemId`. Por exemplo, para iniciar a tela com o item de `id` "item\_lista\_tweet" selecionado, usamos o código:

```
bottom_navigation.selectedItemId = R.id.menu_tweets
```

## 7.7 EXERCÍCIO: CRIANDO O PRIMEIRO FRAGMENT

### Objetivo

Criar o fragment de listagem e o exibir ao clicar no item de lista do `BottomNavigationView`

### Passo a passo

- Vamos criar nosso *fragment*, para isso crie uma nossa classe no novo pacote `fragment` e extenda de `androidx.fragment.app.Fragment`:

```
class ListaTweetsFragment : Fragment() {}
```

- Agora precisamos criar a *view* que irá representar esse nosso *fragment*, por esse motivo, sobrescreveremos o método `onCreateView`:

```
class ListaTweetsFragment : Fragment() {

 override fun onCreateView(inflater: LayoutInflater,
 container: ViewGroup?,
 savedInstanceState: Bundle?): View? {

 }
}
```

- Agora precisamos criar a *view* que será o retorno desse método, faremos uso do `LayoutInflater` para criação da *view*:

```
class ListaTweetsFragment : Fragment() {

 override fun onCreateView(inflater: LayoutInflater,
 container: ViewGroup?,
 savedInstanceState: Bundle?): View? {

 val view = inflater.inflate(R.layout.lista_tweets_fragment, container, false)

 return view
 }
}
```

O código ainda não vai compilar.

- Agora precisamos definir o *layout* `lista_tweets_fragment`:

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.ConstraintLayout xmlns:android="http://schemas.android.com/apk/res/android"
 xmlns:app="http://schemas.android.com/apk/res-auto"
 android:layout_width="match_parent"
 android:layout_height="match_parent">

 <ListView
 android:id="@+id/lista_tweets"
 android:layout_width="0dp"
 android:layout_height="0dp"
 app:layout_constraintBottom_toBottomOf="parent"
 app:layout_constraintEnd_toEndOf="parent"
 app:layout_constraintStart_toStartOf="parent"
 app:layout_constraintTop_toTopOf="parent" />

</androidx.constraintlayout.widget.ConstraintLayout>
```

- Agora precisamos pegar os dados para popular nosso fragment, já tivemos algo bem similar quando

trabalhamos com Room , precisamos pegá-los de nosso ViewModel :

```
class TweetViewModel(private val repository: TweetRepository) : ViewModel() {

 fun salva(tweet: Tweet) = repository.salva(tweet)

 fun tweets(): List<Tweet> = listOf(
 Tweet("bla", null),
 Tweet("ble", null),
 Tweet("bli", null),
 Tweet("blo", null),
 Tweet("blu", null)
)
}
```

6. Agora precisamos recuperar o ViewModel em nosso Fragment e usar essa listagem:

```
class ListaTweetsFragment : Fragment() {

 private val viewModel: TweetViewModel by lazy {
 ViewModelProviders.of(activity!!, ViewModelFactory).get(TweetViewModel::class.java)
 }

 override fun onCreateView(inflater: LayoutInflater, container: ViewGroup?, savedInstanceState: Bundle?): View? {
 val view = inflater.inflate(R.layout.lista_tweets_fragment, container, false)

 val lista = viewModel.tweets()

 view.lista_tweets.adapter = TweetAdapter(lista)

 return view
 }
}
```

7. Agora precisamos definir que quando o item de lista for clicado ele exiba nosso fragment, para isso precisaremos definir um listener para essa ação:

```
class MainActivity : AppCompatActivity() {

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

 bottom_navigation.selectedItemId = R.id.menu_tweets

 listenerBottomNavigation()
 }

 private fun listenerBottomNavigation() {
 bottom_navigation.setOnNavigationItemSelected { item ->

 when (item.itemId) {
 R.id.menu_tweets -> {
 exibe(ListaTweetsFragment())
 true
 }
 else -> {
 false
 }
 }
 }
 }
}
```

```

 }
 }
}

}

```

O código ainda não vai compilar.

#### 8. Ainda fica faltando o método `exibe`, para criá-lo:

```

class MainActivity : AppCompatActivity() {
 //demais códigos

 private fun exibe(fragment: Fragment) {

 val transaction = supportFragmentManager.beginTransaction()

 transaction.replace(R.id.frame_principal, fragment)

 transaction.commit()
 }
}

```

O código ainda não vai compilar.

#### 9. Por fim, precisamos definir esse id que colocamos para ser trocado, alteramos o arquivo `activity_main.xml` adicionando um `FrameLayout`:

```

<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.ConstraintLayout xmlns:android="http://schemas.android.com/apk/res/android"
 xmlns:app="http://schemas.android.com/apk/res-auto"
 android:layout_width="match_parent"
 android:layout_height="match_parent">

 <FrameLayout
 android:id="@+id/frame_principal"
 android:layout_width="0dp"
 android:layout_height="0dp"
 app:layout_constraintBottom_toTopOf="@+id/bottom_navigation"
 app:layout_constraintEnd_toEndOf="parent"
 app:layout_constraintStart_toStartOf="parent"
 app:layout_constraintTop_toTopOf="parent" />

 <!-- FloatingActionButton -->
 <!-- BottomNavigationView -->

```

```
</androidx.constraintlayout.widget.ConstraintLayout>
```

#### 10. Rode o aplicativo e veja se nosso fragment é exibido.

## Editora Casa do Código com livros de uma forma diferente



Editoras tradicionais pouco ligam para ebooks e novas tecnologias. Não dominam tecnicamente o assunto para revisar os livros a fundo. Não têm anos de experiência em didáticas com cursos.

Conheça a **Casa do Código**, uma editora diferente, com curadoria da **Caelum** e obsessão por livros de qualidade a preços justos.

[Casa do Código, ebook com preço de ebook.](#)

## 7.8 MOSTRANDO CAMPO DE BUSCA NA TELA

Quase sempre que vemos uma lista muito grande em uma aplicação, com a lista de contatos no aplicativo padrão de contatos ou a lista de emails no aplicativo do Gmail, esses aplicativos disponibilizam um campo de busca para o usuário achar mais fácil o item que ele quer. Se começarmos a ter muitos tweets na nossa aplicação, começa a ficar interessante disponibilizarmos uma busca ao nosso usuário.

Podemos mostrar um ícone de lupa na Barra de Ações. Para isso, usamos o bom e velho *OptionsMenu* e podemos criar um xml de menu para definir o item da lupa. Porém, quando clicamos em um ícone de lupa em qualquer aplicativo, esperamos que apareça do lado da lupa um campo para preenchermos com o texto que será usado para filtrar a lista. O Android criou a partir da API 7 uma ferramenta que fornece justamente uma boa interface ao usuário em casos de busca: a classe `SearchView`. Se quisermos dar suporte às versões anteriores à API 7, precisamos usar a classe do pacote de suporte. Para associá-la ao item do menu da lupa, podemos passar o nome completo da classe para o atributo `app:actionViewClass` no item da lupa no xml :

```
<?xml version="1.0" encoding="utf-8"?>
<menu xmlns:android="http://schemas.android.com/apk/res/android"
 xmlns:app="http://schemas.android.com/apk/res-auto">

 <item
 android:id="@+id/barra_busca"
 android:icon="@drawable/lupa"
 android:title="Buscar"
 app:actionViewClass="androidx.appcompat.widget.SearchView"
 app:showAsAction="always" />

</menu>
```

Agora quando o usuário clicar na lupa, aparecerá um campo para ele digitar, mas o campo aparecerá

em apenas parte da Barra de Ações. Para o campo cobrir a Barra de Ações inteira enquanto o usuário estiver interagindo com a ferramenta de busca, podemos passar para o atributo `app:showAsAction` do item da lupa o valor `collapseActionBar`, além do valor que ele já tiver:

```
<item
 android:id="@+id/barra_busca"
 android:icon="@drawable/lupa"
 android:title="Buscar"
 app:actionViewClass="androidx.appcompat.widget.SearchView"
 app:showAsAction="always|collapseActionBar" />
```

## 7.9 FILTRANDO TWEETS

Quando o usuário escreve algo em um campo de busca, o filtro pode ser feito enquanto a pessoa digita ou depois que ela "envia" o texto. A ferramenta `SearchView` permite realizarmos alguma lógica para cada um desses momentos com o *listener* `SearchView.OnQueryTextListener`. Essa interface possui dois métodos: `onQueryTextChange`, que retorna `true` se tratamos a ação e `false` se a ferramenta `SearchView` deve fazer o comportamento padrão, de mostrar sugestões; e `onQueryTextSubmit`, que retorna `true` se tratamos a ação e `false` se a ferramenta `SearchView` deve fazer o comportamento padrão, de tratar qualquer *intent* ou processo associado.

Se quisermos fazer a busca dos *tweets* enquanto a pessoa digita o texto, vamos criar uma implementação da interface `SearchView.OnQueryTextListener` e colocar lógica no método `onQueryTextChange`:

```
class BuscaListener : SearchView.OnQueryTextListener {
 override fun onQueryTextSubmit(texto: String?): Boolean {
 return true
 }

 override fun onQueryTextChange(texto: String?): Boolean {
 // buscar tweets contendo o texto
 return true
 }
}
```

Uma maneira de fazer essa filtragem é percorrer toda a lista de *tweets*, procurar um a um qual contém o texto que queremos e adicionar os que tiverem o texto em outra lista. Felizmente, a linguagem de `Kotlin` incluiu vários códigos de programação funcional e, assim, entre outras coisas evita que a gente escreva código com muitos `for`. Para filtrar uma lista, ele possui o método `filter`, que recebe uma lógica que precisa devolver `true` ou `false`. O método `filter` vai adicionar em uma nova lista, que ele mesmo vai criar, todo elemento da lista que a lógica devolver `true`. No final do processo, ele nos devolve essa lista nova com os elementos que foram filtrados pela lógica.

Por fim, para verificar se uma palavra contém em um texto, podemos usar o método `contains` de um objeto `String`, avisando com `true` ou `false` se queremos ou não ignorar ou não a diferenciação

entre maiúsculas e minúsculas. Por exemplo, se temos uma lista de Tweet chamada tweets e queremos filtrar por todos os tweets que tenham a palavra "eu", faremos um código parecido com:

```
val tweetsFiltrados = tweets.filter { tweet -> tweet.mensagem.contains(texto, true) }
```

## Acessando o SearchView

O *listener* precisa ser passado para um objeto SearchView. Como o item da lupa está no **Menu de Opções**, podemos atrelar o *listener* a ele durante a criação do *Options Menu*, no método `onCreateOptionsMenu`. Nesse método, depois de atrelar o xml ao objeto `Menu`, podemos procurar por um item usando seu `id` com o método `findItem`. Lembrando que, como o menu pode ser nulo, usamos o operador `?` logo depois da variável `menu` pra só executar o método se o valor da variável não for `null`:

```
override fun onCreateOptionsMenu(menu: Menu?, inflater: MenuInflater?) {
 inflater?.inflate(R.menu.buscador_menu, menu)

 val itemLupa = menu?.findItem(R.id.barra_busca)
}
```

Agora, precisamos acessar especificamente a `SearchView` de dentro do item da lupa, pedindo a propriedade `actionView` e informando o `Kotlin` que sabemos que é do tipo `SearchView`:

```
val searchView = itemLupa?.actionView as SearchView
```

## Outra forma de fazer classe anônima com Kotlin

Com o objeto `SearchView` acessível pelo código, podemos chamar o método `setOnQueryTextListener` a partir dele e passar uma instância da classe que implementa a interface `SearchView.OnQueryTextListener`. As formas que fizemos antes serviam bem quando havia apenas um método na interface. Agora, com uma interface que possui mais de um métodos, vamos pensar um pouco diferente. Uma forma de pensarmos é que precisamos de um **objeto** que implemente a interface. Utilizando a expressão `object` do `Kotlin`, podemos definir essa implementação:

```
searchView.setOnQueryTextListener(object : SearchView.OnQueryTextListener {

 override fun onQueryTextSubmit(texto: String?): Boolean {
 return false
 }

 override fun onQueryTextChange(texto: String?): Boolean {
 //filtra tweets

 return false
 }
})
```

## 7.10 DEFININDO OPTIONSMENU EM FRAGMENTS

Pensando em aplicações que possuem o **Bottom Navigation**, cada item do **Bottom Navigation** pode levar a destinos completamente diferentes, sem relação nenhuma com o outro. Como já vimos, esses destinos podem ocupar apenas parte da tela com as **Fragment**s, mantendo na tela o **Bottom Navigation** e outras estruturas que fazem parte de todos os destinos. Como esses fragmentos de tela são independentes uns dos outros, é bem capaz que cada fragmento tenha seu próprio **Menu de Opções**, definindo ações específicas para aquele fragmento.

Vemos isso por exemplo no Google Maps, em que, clicando no item **Explorar** aparece um ícone de microfone e clicando no item **Para você** aparece um ícone de configurações.

Para casos como esses, podemos criar um **OptionsMenu** em cada **Fragment** que for necessário. Em um **Fragment** temos acesso aos mesmos métodos que usamos para criar e manipular um **OptionsMenu** em uma **Activity** : `onCreateOptionsMenu` e `onOptionsItemSelected`.

Mas, na hora de aparecer na tela, esse **Menu de Opções** aparece na **Barra de Ações**, que faz parte da **Activity**, portanto o **Menu de Opções** deve vir da **Activity**. Só o fato de configurar o **Menu de Opções** em um **Fragment** então não é suficiente. O **Fragment** precisa avisar a **Activity** que for usá-lo que pode contribuir com itens do **Menu de Opções**. Para isso, ele precisa chamar o método `setHasOptionsMenu` durante sua criação, ou seja, dentro do método `onCreate`. O método `setHasOptionMenu` recebe um booleano para avisar se esse fragmento tem itens de menu para contribuir com o **Menu de Opções** da **Activity**.

```
class OptionsMenuFragment : Fragment() {
 override fun onCreate(savedInstanceState: Bundle?) {
 super.onCreate(savedInstanceState)
 setHasOptionsMenu(true)
 }
}
```

Já conhece os cursos online Alura?



A Alura oferece centenas de **cursos online** em sua plataforma exclusiva de ensino que favorece o aprendizado com a **qualidade** reconhecida da Caelum. Você pode escolher um curso nas áreas de Programação, Front-end, Mobile, Design & UX, Infra e Business, com um plano que dá acesso a todos os cursos. Ex aluno da Caelum tem 15% de desconto neste link!

[Conheça os cursos online Alura.](#)

## 7.11 EXERCÍCIO: FAZENDO A PARTE DE BUSCA

### Objetivo

Criar o fragment de busca que seja responsável por filtrar Tweets com determinado conteúdo.

### Passo a passo

1. Crie um novo fragment chamado `BuscadorDeTweetsFragment` :

```
class BuscadorDeTweetsFragment : Fragment() {}
```

2. Vamos usar o mesmo *layout* que usamos no fragment de listagem, dado que apenas aplicaremos um filtro:

```
class BuscadorDeTweetsFragment : Fragment() {

 override fun onCreateView(inflater: LayoutInflater, container: ViewGroup?, savedInstanceState: Bundle?): View? {

 return inflater.inflate(R.layout.lista_tweets_fragment, container, false)
 }

}
```

3. Agora vamos definir a barra de busca, para isso vamos criar um novo arquivo de menu `buscador_menu` :

```
<?xml version="1.0" encoding="utf-8"?>
<menu xmlns:android="http://schemas.android.com/apk/res/android"
 xmlns:app="http://schemas.android.com/apk/res-auto">

 <item
 android:id="@+id/barra_busca"
 android:icon="@drawable/busca"
 android:title="Buscar"
 app:actionViewClass="androidx.appcompat.widget.SearchView"
 app:showAsAction="always|collapseActionView" />

</menu>
```

4. Agora precisamos *inflar* esse arquivo em nosso fragment:

```
class BuscadorDeTweetsFragment : Fragment() {

 //demais métodos

 override fun onCreateOptionsMenu(menu: Menu?, inflater: MenuInflater?) {

 inflater?.inflate(R.menu.buscador_menu, menu)

 }

}
```

5. Agora precisamos deixar claro para a `Activity` que esse fragment terá um menu, faremos isso no

onCreate do fragment:

```
class BuscadorDeTweetsFragment : Fragment() {

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

 // outros métodos
}
```

6. Agora precisamos definir o comportamento ao clicarmos no item do menu:

```
class BuscadorDeTweetsFragment : Fragment() {

 //demais métodos

 override fun onCreateOptionsMenu(menu: Menu?, inflater: MenuInflater?) {
 inflater?.inflate(R.menu.buscador_menu, menu)

 val botaoBusca = menu?.findItem(R.id.barra_busca)

 val search = botaoBusca?.actionView as SearchView

 search.setOnQueryTextListener(object : SearchView.OnQueryTextListener {
 override fun onQueryTextSubmit(texto: String?): Boolean {
 return false
 }

 override fun onQueryTextChange(texto: String?): Boolean {
 if (!texto.isNullOrEmpty()) {

 val filtrados = filtraTweetsPelo(texto)

 lista_tweets.adapter = TweetAdapter(filtrados)

 }
 return false
 }
 })
 }
}
```

O código ainda não vai compilar.

7. Vamos definir o método `filtraTweetsPelo` :

```
private fun filtraTweetsPelo(texto: String?): List<Tweet> {
 val tweets = viewModel.tweets()
 val tweetsFiltrados = tweets.filter { tweet -> tweet.mensagem.contains(texto!!, true) }
 return tweetsFiltrados
}
```

O código ainda não vai compilar.

8. Precisamos agora ter o `ViewModel` em nosso *fragment* para que essa busca possa funcionar:

```
class BuscadorDeTweetsFragment : Fragment() {

 private val viewModel: TweetViewModel by lazy {
 ViewModelProviders.of(activity!!, ViewModelFactory).get(TweetViewModel::class.java)
 }

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

 }

 // demais métodos
}
```

9. Por fim precisamos colocar esse novo fragment em ação ao escolhermos a opção de busca, para isso vamos adicionar uma nova condição em nosso listener na `MainActivity`:

```
class MainActivity : AppCompatActivity() {
 //demais códigos
 private fun listenerBottomNavigation() {
 bottom_navigation.setOnNavigationItemSelected { item ->

 when (item.itemId) {
 R.id.menu_tweets -> {
 exibe(ListaTweetsFragment())
 true
 }
 R.id.menu_busca -> {
 exibe(BuscadorDeTweetsFragment())
 true
 }
 else -> {
 false
 }
 }
 }
 }
}
```

10. Rode o aplicativo e tente buscar por algum conteúdo de um tweet.

# MANIPULANDO DADOS REAIS

## 8.1 O QUE SABEREI FAZER NO FIM DESSE CAPÍTULO ?

- Fazer requisições usando Retrofit
- Swipe to refresh
- High order fuctions

Saber inglês é muito importante em TI



Na **Alura Língua** você reforça e aprimora seu inglês! Usando a técnica *Spaced Repetitions* o aprendizado naturalmente **se adapta ao seu conhecimento**. Exercícios e vídeos interativos fazem com que você pratique em situações

cotidianas. Além disso, todas as aulas possuem explicações gramaticais, para você entender completamente o que está aprendendo. Aprender inglês é fundamental para o profissional de tecnologia de sucesso!

[Pratique seu inglês na Alura Língua.](#)

## 8.2 CRIANDO TELA DE LOGIN

Quando os *tweets* são listados, junto com a informação da mensagem e da foto do *tweet*, mostramos quem criou o *tweet*. Para nossa aplicação saber quem criou cada *tweet*, o usuário precisará realizar um *login* no nosso aplicativo. Para isso, teremos que criar um *layout* de formulário.

Utilizando o `TextEditor`, podemos ajudar o usuário a saber o que digitar avisando com o atributo `android:hint`. Porém, quando o usuário clica no campo de texto, ele perde a dica que estava sendo dada pelo atributo `android:hint`. Pensando nessa e em outras melhorias para um campo de texto, foram definidas algumas diretrizes sobre campos de texto no **Material Design**.

O Android implementa elas com a componente `TextInputLayout`, presente na mesma biblioteca que o `FloatingActionButton`. O `TextInputLayout` geralmente é utilizado em conjunto com outra componente, a `TextInputEditText`:

```
<com.google.android.material.textfield.TextInputLayout
 android:layout_width="match_parent"
 android:layout_height="wrap_content">

 <com.google.android.material.textfield.TextInputEditText
 android:layout_width="match_parent"
 android:layout_height="wrap_content"
 android:hint="Username" />

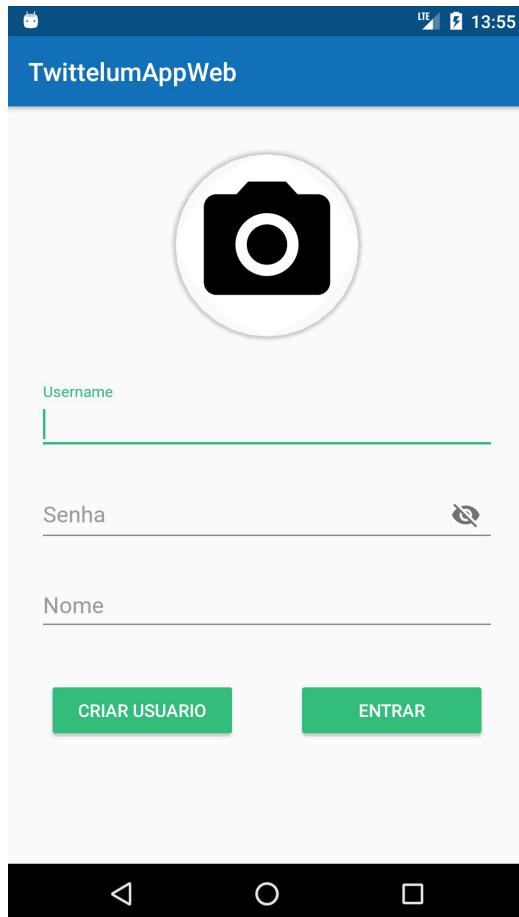
</com.google.android.material.textfield.TextInputLayout>
```

A `TextInputEditText` funciona bem parecido com a `EditText`, mas fornece mais suporte para acessibilidade em um campo de texto e mais controle do `TextInputLayout` sobre os aspectos visuais do campo de texto.

## 8.3 EXERCÍCIO: CRIANDO NOSSO USUÁRIO

### Objetivo

- Criar a tela de login do aplicativo, similar a essa:



- Criar a classe usuário para pegar as informações da tela
- Estruturar o código usando as seguintes Architecture Components: ViewModel e Repository

## Passo a passo

1. Crie a Activity para o usuário se logar e implemente o método onCreate :

```
class LoginActivity : AppCompatActivity() {

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

2. Vamos criar o layout, que chamamos de activity\_login , pelo **Editor de Layout** ou direto pelo xml . Não esquecendo de criar o ícone da câmera, de nome ic\_menu\_camera :

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

```

<androidx.constraintlayout.widget.ConstraintLayout
 android:layout_width="match_parent"
 android:layout_height="match_parent">

 <androidx.cardview.widget.CardView
 android:id="@+id/cardView"
 android:layout_width="150dp"
 android:layout_height="150dp"
 android:layout_marginStart="8dp"
 android:layout_marginTop="35dp"
 android:layout_marginEnd="8dp"
 app:cardCornerRadius="100dp"
 app:layout_constraintEnd_toEndOf="parent"
 app:layout_constraintStart_toStartOf="parent"
 app:layout_constraintTop_toTopOf="parent">

 <ImageView
 android:layout_width="match_parent"
 android:layout_height="match_parent"
 android:layout_margin="15dp"
 android:src="@android:drawable/ic_menu_camera" />

 </androidx.cardview.widget.CardView>

 <com.google.android.material.textfield.TextInputLayout
 android:id="@+id/textInputLayoutUsername"
 android:layout_width="0dp"
 android:layout_height="wrap_content"
 android:layout_marginStart="24dp"
 android:layout_marginTop="35dp"
 android:layout_marginEnd="24dp"
 app:layout_constraintEnd_toEndOf="parent"
 app:layout_constraintHorizontal_bias="1.0"
 app:layout_constraintStart_toStartOf="parent"
 app:layout_constraintTop_toBottomOf="@+id/cardView">

 <com.google.android.material.textfield.TextInputEditText
 android:id="@+id/login_campoUsername"
 android:layout_width="match_parent"
 android:layout_height="wrap_content"
 android:hint="Username" />
 </com.google.android.material.textfield.TextInputLayout>

 <com.google.android.material.textfield.TextInputLayout
 android:id="@+id/textInputLayoutSenha"
 android:layout_width="0dp"
 android:layout_height="wrap_content"
 android:layout_marginStart="24dp"
 android:layout_marginTop="15dp"
 android:layout_marginEnd="24dp"
 app:layout_constraintEnd_toEndOf="parent"
 app:layout_constraintStart_toStartOf="parent"
 app:layout_constraintTop_toBottomOf="@+id/textInputLayoutUsername"
 app:passwordToggleEnabled="true">

 <com.google.android.material.textfield.TextInputEditText
 android:id="@+id/login_campoSenha"
 android:layout_width="match_parent"
 android:layout_height="wrap_content"
 android:hint="Senha"
 android:inputType="textPassword" />
 </com.google.android.material.textfield.TextInputLayout>

```

```

<com.google.android.material.textfield.TextInputLayout
 android:id="@+id/textInputLayoutNome"
 android:layout_width="0dp"
 android:layout_height="wrap_content"
 android:layout_marginStart="24dp"
 android:layout_marginTop="15dp"
 android:layout_marginEnd="24dp"
 app:layout_constraintEnd_toEndOf="parent"
 app:layout_constraintHorizontal_bias="0.0"
 app:layout_constraintStart_toStartOf="parent"
 app:layout_constraintTop_toBottomOf="@+id/textInputLayoutSenha">

 <com.google.android.material.textfield.TextInputEditText
 android:id="@+id/login_campoNome"
 android:layout_width="match_parent"
 android:layout_height="wrap_content"
 android:hint="Nome" />
</com.google.android.material.textfield.TextInputLayout>

<Button
 android:id="@+id/login_criar"
 android:layout_width="150dp"
 android:layout_height="wrap_content"
 android:layout_marginStart="24dp"
 android:layout_marginTop="36dp"
 android:layout_marginEnd="20dp"
 android:backgroundTint="@color/colorAccent"
 android:text="Criar Usuário"
 android:textColor="@color/branco"
 app:layout_constraintEnd_toStartOf="@+id/login_entrar"
 app:layout_constraintHorizontal_bias="0.509"
 app:layout_constraintStart_toStartOf="parent"
 app:layout_constraintTop_toBottomOf="@+id/textInputLayoutNome" />

<Button
 android:id="@+id/login_entrar"
 android:layout_width="150dp"
 android:layout_height="wrap_content"
 android:layout_marginStart="20dp"
 android:layout_marginTop="36dp"
 android:layout_marginEnd="24dp"
 android:backgroundTint="@color/colorAccent"
 android:text="Entrar"
 android:textColor="@color/branco"
 app:layout_constraintEnd_toEndOf="parent"
 app:layout_constraintStart_toEndOf="@+id/login_criar"
 app:layout_constraintTop_toBottomOf="@+id/textInputLayoutNome" />

</androidx.constraintlayout.widget.ConstraintLayout>
</ScrollView>

```

3. Faça com que o aplicativo comece pela Activity de login, **alterando** o arquivo `AndroidManifest.xml`:

```

<application>
 <activity
 android:name=".activity.LoginActivity"
 android:windowSoftInputMode="stateAlwaysHidden">
 <intent-filter>
 <action android:name="android.intent.action.MAIN" />
 <category android:name="android.intent.category.LAUNCHER" />

```

---

```

 </intent-filter>
 </activity>

 <activity android:name=".activity.MainActivity" />

 // demais tags
</application>
```

4. Vamos criar o método que cria um usuário pegando os dados da tela:

```

class LoginActivity : AppCompatActivity() {

 //demais métodos

 private fun usuarioDaTela(): Usuario {

 val nome = login_campoNome.text.toString()
 val senha = login_campoSenha.text.toString()
 val username = login_campoUsername.text.toString()

 return Usuario(nome, senha, username)

 }

}
```

O código ainda não vai compilar.

5. Crie a classe `Usuario` para o código acima compilar:

```

data class Usuario(val nome: String,
 val username: String,
 val senha: String,
 val foto: String? = null,
 val id: Long = 0)
```

6. Vamos dar comportamento aos botões, para isso criaremos dois listeners e um `ViewModel` para eles usarem:

```

class LoginActivity : AppCompatActivity() {

 private val viewModel: UsuarioViewModel by lazy {
 ViewModelProviders.of(this, ViewModelFactory).get(UsuarioViewModel::class.java)
 }

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

 login_criar.setOnClickListener { viewModel.cria(usuarioDaTela()) }

 login_entrar.setOnClickListener { viewModel.loga(usuarioDaTela()) }
 }
}
```

O código acima ainda não vai compilar.

7. Precisamos agora criar esse `ViewModel`, portanto crie a classe `UsuarioViewModel` no pacote

```

viewmodel :

class UsuarioViewModel(private val repository: UsuarioRepository) : ViewModel() {

 fun cria(usuario: Usuario) = repository.cadastra(usuario)

 fun loga(usuario: Usuario) = repository.entra(usuario)

}

```

O código acima ainda não vai compilar.

- Para esse código compilar, precisamos criar esse Repository , portanto crie a classe `UsuarioRepository` no pacote `data` :

```

class UsuarioRepository {

 fun entra(usuario: Usuario) {
 Log.i("loginConta", "$usuario")
 }

 fun cadastra(usuario: Usuario) {
 Log.i("criaConta", "$usuario")
 }

}

```

- Por fim, precisaremos fazer uma pequena alteração em nosso `ViewModelFactory` :

```

object ViewModelFactory : ViewModelProvider.NewInstanceFactory() {

 private fun getTweetRepository() = TweetRepository()
 private fun getUsuarioRepository() = UsuarioRepository()

 override fun <T : ViewModel?> create(modelClass: Class<T>): T = when (modelClass) {
 TweetViewModel::class.java ->
 TweetViewModel(getTweetRepository()) as T
 else ->
 UsuarioViewModel(getUsuarioRepository()) as T
 }
}

```

- Rode o aplicativo e veja os logs sendo chamados quando clicamos nos botões.

## 8.4 DESAFIO OPCIONAL: PERMITINDO QUE O USUÁRIO TIRE UMA FOTO

### Objetivo

Fazer com que, quando o usuário clicar no campo de foto, abra o aplicativo de câmera e o usuário possa tirar uma foto. Se ele aceitar a foto, ela deverá aparecer no campo de foto no formulário.

**Aprenda se divertindo na Alura Start!**



Você conhece alguém que tem potencial para tecnologia e programação, mas que nunca escreveu uma linha de código? Pode ser um filho, sobrinho, amigo ou parente distante. Na **Alura Start** ela vai poder criar games, apps, sites e muito mais! É o **começo da jornada com programação e a porta de entrada para uma possível carreira de sucesso**. Ela vai estudar em seu próprio ritmo e com a melhor didática. A qualidade da conceituada Alura, agora para Starters.

[Conheça os cursos online da Alura Start!](#)

## 8.5 ACESSANDO DADOS DE OUTRAS PESSOAS

Pensando em um aplicativo como o *Twitter*, em que há muitos *tweets* para mostrar na *timeline* de cada pessoa, é inviável armazenar todos esses dados no celular de cada usuário. Primeiro porque um usuário precisa ter acesso no seu dispositivo a uma quantidade gigantesca de *tweets*; segundo porque ele precisa ter acesso aos *tweets* de outras pessoas, que foram criados em outros dispositivos. Para isso, é necessário que os dados sejam salvos em outro lugar, geralmente um servidor.

### Comunicando com um servidor

Para pedir ou enviar alguma informação a um servidor, enviamos uma requisição com uma URL e esperamos uma resposta em texto. Essa resposta pode vir em diversos formatos, como JSON, CSV, XML, texto puro. A mais utilizada atualmente é o JSON (*JSON Object Notation*), pois é uma estrutura muito leve e simples de trabalhar, exatamente o que precisamos num desenvolvimento *mobile*. Com ele podemos representar uma entidade ou objeto em um arquivo de formato fácil, que poderá ser lido e absorvido por diversas plataformas sem muito esforço para parseá-lo.

Neste curso, não falaremos do servidor, mas sim do Android. A Caelum tem uma apostila aberta à sua disposição sobre como criar um servidor com Java, o curso **Java para Desenvolvimento Web**: <https://www.caelum.com.br/curso-java-web/>.

Além disso, temos um treinamento dedicado a integração, o curso **SOA na prática: Integração com Web Services e Mensageria**: <https://www.caelum.com.br/curso-ee-soa-web-services-mensageria/>.

Há também a opção de criar um servidor web utilizando Javascript e, para isso, temos os cursos **Curso Angular para construção de Web Apps** (<https://www.caelum.com.br/curso-angular>) e **Curso**

## **React e Redux para construção de Web Apps** (<https://www.caelum.com.br/curso-react-redux>).

Para conseguirmos enviar uma requisição a um servidor, precisamos antes de tudo estabelecer uma conexão com o servidor. A partir do `Android 6`, podemos utilizar uma classe do Java chamada de `HttpURLConnection`. Mas, com essa classe, criamos a conexão na mão, e por isso, precisamos gerenciar todos os detalhes dela. Por exemplo, precisamos definir o *Content Type* e o *Accept*, para avisar que tipo de dado estamos enviando e queremos receber, respectivamente. Também precisamos definir se será enviado algum dado e se será recebido algum dado no conteúdo da requisição. Se quisermos enviar algum dado, temos que escrevê-lo manualmente na conexão, da mesma forma que temos que ler pela conexão algum dado que quisermos da resposta. Não podemos esquecer de fechar a conexão no final de todo esse processo.

O processo acima é muito trabalhoso e quase sempre são os mesmos passos que precisam ser feitos. Ao invés de todos os programadores terem que realizar esses passos toda hora, algumas empresas criaram bibliotecas que já gerenciam a conexão por nós. Precisamos apenas usá-la no projeto e avisá-la de alguns detalhes específicos da nossa requisição. Usaremos a biblioteca `Retrofit` para isso. Primeiro, precisamos adicioná-la ao nosso projeto:

```
dependencies {
 //demais dependências
 implementation 'com.squareup.retrofit2:retrofit:2.5.0'
}
```

A ideia é que toda a parte trabalhosa de gerenciar e configurar a conexão será implementada pela biblioteca. Mas, para isso, ela precisa saber o que vai implementar. Primeiro, avisaremos durante a criação do objeto do tipo `Retrofit` qual a URL base das requisições que queremos fazer:

```
val retrofit: Retrofit = Retrofit.Builder()
 .baseUrl(urlBaseDoServiçoEmHttps)
 .build()
```

Esse será o objeto que criará as implementações para a gente. Para ele saber que configurações ele deve implementar para cada requisição, ele usa a mesma ideia do `Room`, de definirmos com uma interface apenas os métodos que deverão ser implementados com anotações para informar as configurações a serem feitas. Por exemplo, se quisermos que seja configurada uma requisição do tipo `post` para a URL `user`, criamos uma interface com um método anotado com `@Post`, passando pra anotação a URL "`user`":

```
private interface UsuarioService {
 @POST("/user")
 fun insere()
}
```

Além de configurar a URL e o método da requisição, muitas vezes precisamos enviar alguns dados para o servidor processar essas informações. Para pedir pro servidor criar um usuário, por exemplo,

precisamos informar os dados do usuário para o servidor salvar os dados desse novo usuário no banco. Na nossa aplicação, podemos criar uma classe para representar os dados do usuário e facilitar a manipulação deles. Mas ainda precisamos converter o objeto para o formato JSON. A classe `JSONStringer` do Android nos permite fazer essa conversão, definindo passo a passo o JSON a partir das informações do objeto que queremos converter. No entanto, para a maioria dos objetos, esse processo de conversão é bem simples, só são muitos passos. A empresa que criou o `Retrofit` já possui vários conversores de JSON para objeto:

- Gson: com.squareup.retrofit2:converter-gson
- Jackson: com.squareup.retrofit2:converter-jackson
- Moshi: com.squareup.retrofit2:converter-moshi
- Protobuf: com.squareup.retrofit2:converter-protobuf
- Wire: com.squareup.retrofit2:converter-wire
- Simple XML: com.squareup.retrofit2:converter-simplexml
- Scalars (primitives, boxed, and String): com.squareup.retrofit2:converter-scalars

Basta escolhermos um e adicionar sua dependência ao projeto.

Precisamos agora pedir para o objeto do tipo `Retrofit` usar o conversor que escolhemos, passando a ele uma fábrica de conversores. Avisamos isso durante a criação do objeto, chamando o método `addConverterFactory` antes do método `build`. Para conseguir uma fábrica conversores usando o módulo do `Gson`, chamamos o método estático `create` da classe `GsonConverterFactory`:

```
val retrofit: Retrofit = Retrofit.Builder()
 .baseUrl(urlBaseDoServiçoEmHttps)
 .addConverterFactory(GsonConverterFactory.create())
 .build()
```

Agora que o objeto `Retrofit` consegue converter nossos objetos para JSON, precisamos informar pra ele que para uma requisição em específico queremos enviar um objeto nosso. Para o caso do login, por exemplo, queremos enviar um objeto do tipo `Usuario`, que possua `username` e `senha`. Qualquer configuração específica de uma requisição, informamos pela interface que criamos. Vamos informar que o método precisará receber um objeto do tipo `Usuario` e, para o `Retrofit` saber que esse parâmetro deve ser enviado pelo **corpo** da requisição, o anotamos com `@Body`:

```
private interface UsuarioService {

 @POST("/user")
 fun insere(@Body usuario: Usuario)

}
```

Depois de todas as configurações que quisermos fazer para montar a requisição, queremos efetivamente fazer a chamada para o servidor, ou seja, depois de tudo ser configurado, queremos ter à nossa disposição alguém que saiba realizar a chamada para o servidor. Por isso, pediremos como retorno

do método um objeto do tipo `Call`. Se o servidor for devolver alguma informação para a gente na resposta, vamos querer que o `Retrofit` já converta o JSON da resposta para um objeto de algum tipo. Por exemplo, se ele for devolver as informações do usuário existente no banco, podemos pedir pra ele pegar o JSON de resposta e converter para o objeto do tipo `Usuario`, que será preenchido nesse momento com o `id` do banco, além das outras informações. Definimos qual tipo será utilizado para converter o JSON de resposta utilizando o *generics* do `Call`:

```
private interface UsuarioService {

 @POST("/user")
 fun insere(@Body usuario: Usuario): Call<Usuario>

}
```

Com toda a interface definida, precisamos pedir para o retrofit criar uma implementação a partir dessa interface. Fazemos isso chamando o método `create` do objeto `Retrofit` e informando a esse método qual interface será implementada com a referência da interface em Java:

```
val retrofit: Retrofit = //cria retrofit com builder
val service: UsuarioService = retrofit.create(UsuarioService::class.java)
```

A partir da implementação da nossa interface, podemos executar o método `insere` que definimos na interface, utilizando a implementação criada pelo `Retrofit`:

```
val usuario: Usuario = //cria e popula usuário
val chamadaPraInserir: Call<Usuario> = service.insere(usuario)
```

Com a chamada pronta para criar um usuário, podemos pedir para efetivamente executar a chamada para o servidor com o método `execute`. No entanto, esse método realiza a chamada na mesma *thread* que está gerenciando a tela, a *UI Thread*. Da mesma forma que para acessar o banco de dados, ao realizar uma requisição, precisamos de uma *thread* secundária para não travar a tela para o usuário. Por isso, se tentarmos utilizar o método `execute` na *thread* principal, o Android lançará a exceção `NetworkOnMainThreadException`.

Felizmente, o `Retrofit` já disponibiliza um outro método, o `enqueue`, que realiza a requisição de forma assíncrona, em outra *thread*. Esse método pede uma implementação da interface `Callback`, para definirmos a lógica que será executada em caso de falha ou de sucesso. Essa interface usa a ideia do *generics* para especificarmos os tipos dos parâmetros dos métodos, por isso informamos a ela qual o tipo do objeto que queremos que o JSON de resposta seja convertido. Para não criar uma classe separada que implemente a interface, podemos fazer uso da classe anônima com o `object`:

```
chamadaPraInserir.enqueue(object : Callback<Usuario> {
 override fun onFailure(call: Call<Usuario>, t: Throwable) {
 // lógica a ser executada em caso de falha do servidor
 }

 override fun onResponse(call: Call<Usuario>, response: Response<Usuario>) {
 // lógica a ser executada no caso de recebermos uma resposta do servidor
 }
}
```

```
}
```

Quando pedirmos para executarmos esse método `enqueue`, ainda vai dar um erro na nossa aplicação. Isso porque, para realizar uma requisição, o usuário precisa estar ou conectado em alguma rede Wi-Fi ou estar com dados ativos. Seja qual for a opção, precisamos pedir permissão ao usuário para utilizar a rede ou consumir seus dados. Como é uma configuração sobre nossa aplicação ao usuário, pedimos essa permissão no `AndroidManifest.xml` com a tag `uses-permission`.

```
<manifest ...>
 <uses-permission android:name="android.permission.INTERNET" />
 // restante
</manifest>
```

## 8.6 PENSANDO NA ESTRUTURA DO PROJETO

Agora que sabemos como utilizar o `Retrofit` para facilitar o processo de requisição, precisamos ver em que momento chamarmos seus métodos dentro do nosso projeto.

Nosso projeto já está separado pelas classes `ViewModel`, que vão gerenciar as informações da tela com base no ciclo de vida das `Activity`s e pelas classes `Repository`, que acessam ou alteram os dados de algum lugar e passam esses dados às classes `ViewModel`. Quando usamos o banco de dados para armazenar os dados, encapsulamos a lógica de gerenciar os dados no `Dao` para não deixar muita lógica na classe `Repository`, de forma que ela só pedia para salvar ou deletar um dado do banco ao `Dao` e a `Repository` não sabia como a lógica tava implementada. Seguiremos a mesma ideia para a lógica de implementar o acesso aos dados no servidor: enquanto a função da classe `Repository` será definir de onde os dados virão ou serão alterados, a lógica de como serão acessados os dados no servidor, ou seja, de utilizar o `Retrofit`, ficará encapsulada em outra classe, que podemos chamar de `WebClient`. Colocaremos a lógica de executar a requisição ao servidor em um método, que podemos chamar de `registra`, na `WebClient` e chamaremos apenas esse método na `Repository`. Portanto, a `Repository` precisará ter acesso a um objeto do tipo `WebClient`. Além disso, como queremos cadastrar um usuário, a `Repository` precisa passar as informações do usuário pro método `registra` da `WebClient`:

```
class UsuarioRepository(private val client: WebClient) {
 fun cadastra(usuario: Usuario) = client.registra(usuario)
}
```

Nossa classe `WebClient` pode ficar com a seguinte implementação do método `registra`:

```
class WebClient() {
 fun registra(usuario: Usuario) {
```

```

 val retrofit: Retrofit = //cria retrofit com builder
 val service: UsuarioService = retrofit.create(UsuarioService::class.java)
 val chamadaPraInserir = service.cria(usuario)

 chamadaPraInserir.enqueue(object : Callback<Usuario> {
 override fun onFailure(call: Call<Usuario>, t: Throwable) {
 // lógica a ser executada em caso de falha do servidor
 }

 override fun onResponse(call: Call<Usuario>, response: Response<Usuario>) {
 // lógica a ser executada no caso de recebermos uma resposta do servidor
 }
 })
 }
}

```

Falta definirmos a lógica a ser executada dependendo da resposta do servidor. Se a requisição for bem sucedida, queremos redirecionar o usuário para a tela principal da nossa aplicação, mas, para sabermos no resto das telas que o usuário já está logado, precisamos armazenar essa informação com uma *flag*. Sempre que queremos recuperar dados para a nossa aplicação, a classe responsável por saber de onde pegar os dados é a `Repository`. Pensando nas responsabilidades de cada classe, a `WebClient` só tem a responsabilidade de fazer a requisição para o servidor, portanto quer pedir para quem chamar seu método passar a lógica que deve ser executada em caso de sucesso ou de falha.

Com o `Kotlin` conseguimos passar como parâmetro uma lógica, ou seja, podemos definir em um método um parâmetro que armazene uma lógica. Quando armazenamos um bloco de lógica, ou seja, uma função, em uma variável, chamamos essa função de ***high order function (função de alto nível)***. Quando definimos um parâmetro de um método, especificamos o tipo de cada variável. No caso de a variável guardar uma função, o tipo dela deve conter tanto o tipo dos parâmetros da função quanto o tipo de retorno. Por exemplo, se queremos que o método `registra` receba dois parâmetros, o `usuario`, que guardará um objeto do tipo `Usuario` e o `sucesso`, que guardará uma função que precise receber um objeto do tipo `Usuario` e não devolver nada (tipo `Unit` pro `Kotlin`), a estrutura do método `registra` fica assim:

```

fun registra(usuario: Usuario, sucesso: (usuario: Usuario) -> Unit) {
}

```

Agora que o método `registra` da `WebClient` consegue receber a lógica de sucesso como parâmetro, podemos fazer o mesmo para conseguir a lógica de falha, que deverá receber um objeto `Throwable` pra conseguir ter acesso ao erro que aconteceu e não deverá retornar nada:

```

fun registra(usuario: Usuario, sucesso: (Usuario) -> Unit, falha: (Throwable) -> Unit) {
}

```

Dentro do método `registra` conseguimos chamar esses parâmetros que guardam funções no

momento que precisarmos da lógica. Podemos chamar esses parâmetros do `registra` como se fossem métodos mesmo, passando entre parênteses as informações que eles pedem. Por exemplo, pra usar o parâmetro `falha`, escrevemos o código `falha(t)`, onde `t` deve ser um objeto do tipo `Throwable`:

```
fun registra(usuario: Usuario, sucesso: (Usuario) -> Unit, falha: (Throwable) -> Unit) {
 val retrofit: Retrofit = //cria retrofit com builder
 val service: UsuarioService = retrofit.create(UsuarioService::class.java)
 val chamadaPraInserir = service.cria(usuario)

 chamadaPraInserir.enqueue(object : Callback<Usuario> {
 override fun onFailure(call: Call<Usuario>, t: Throwable) {
 falha(t)
 }

 override fun onResponse(call: Call<Usuario>, response: Response<Usuario>) {
 // lógica a ser executada no caso de recebermos uma resposta do servidor
 }
 })
}
```

Para a lógica de resposta, só chamaremos a lógica de sucesso se a resposta for bem sucedida. Verificamos isso pela propriedade `isSuccessful` da variável `response`, que é booleana. Além disso, a lógica de sucesso precisa receber um objeto do tipo `Usuario`, que está vindo do corpo da resposta. Para conseguirmos o conteúdo do corpo da resposta, chamamos o método `body` da `response`, que nos retornará já um objeto do tipo `Usuario`, o que precisamos para passar para a lógica de sucesso:

```
fun registra(usuario: Usuario, sucesso: (Usuario) -> Unit, falha: (Throwable) -> Unit) {
 // criando as variáveis

 chamadaPraInserir.enqueue(object : Callback<Usuario> {
 override fun onFailure(call: Call<Usuario>, t: Throwable) {
 falha(t)
 }

 override fun onResponse(call: Call<Usuario>, response: Response<Usuario>) {
 if (response.isSuccessful) {
 val usuarioLogado: Usuario = response.body()
 sucesso(usuarioLogado)
 }
 }
 })
}
```

Porém, o valor retornado pelo método `body` pode ser nulo. Precisamos avisar o `Kotlin` que só queremos executar a lógica de sucesso se o retorno do método `body` não for nulo. Podemos fazer isso utilizando o operador `let`:

```
override fun onResponse(call: Call<Usuario>, response: Response<Usuario>) {
 if (response.isSuccessful) {
 response.body()?.let(sucesso)
 }
}
```

Voltando pra `Repository`, ela que armazenará a informação se o usuário está logado ou não, dependendo do sucesso ou falha da requisição, ou seja, ela que determinará qual será a lógica executada em caso de sucesso ou falha. Portanto, criaremos um método para cada uma dessas lógicas na `Repository`. Como essa lógica só será executada na `WebClient`, não precisamos definir um nome para a função na `Repository`, apenas guardaremos em uma variável, que passaremos na chamada do método `registra`:

```
class UsuarioRepository(private val client: WebClient) {

 fun cadastra(usuario: Usuario) = client.registra(usuario, sucesso, falha)

 private val sucesso = fun (usuario: Usuario) {
 Log.i("criou usuário", "sucesso! $usuario")
 }

 private val falha = fun (excecao: Throwable) {
 Log.i("falha na requisição", "erro: $excecao")
 }
}
```

Como só estamos definindo a lógica que será guardada em cada variável, podemos usar apenas a sintaxe de `{}` para delimitar o início e fim da lógica a ser guardada, informando apenas o parâmetro que ela deve receber:

```
class UsuarioRepository(private val client: WebClient) {

 fun cadastra(usuario: Usuario) = client.registra(usuario, sucesso, falha)

 private val sucesso = {usuario: Usuario ->
 Log.i("criou usuário", "sucesso! $usuario")
 }

 private val falha = { excecao: Throwable ->
 Log.i("falha na requisição", "erro: $excecao")
 }
}
```

Na lógica de sucesso, vamos querer avisar que o usuário está logado, e, na de falha, avisar que o usuário não está logado, portanto precisamos criar uma variável do tipo `Boolean` e alterar seu valor em cada lógica:

```
class UsuarioRepository(private val client: WebClient) {
 val estaLogado: Boolean

 // outros métodos

 private val sucesso = {usuario: Usuario ->
 estaLogado = true
 Log.i("criou usuário", "sucesso! $usuario")
 }

 private val falha = { excecao: Throwable ->
 estaLogado = false
 Log.i("falha na requisição", "erro: $excecao")
 }
}
```

```
}
```

Pensando na nossa aplicação, assim que conseguirmos a informação de que o usuário está logado, precisamos levar ele para a tela principal, independente se ele está abrindo o aplicativo e já estava logado ou se ele está na tela de login e as informações que ele digitou foram processadas com sucesso. Para conseguirmos redirecioná-lo assim que a `flag estaLogado` ficar com o valor `true`, o objeto `UsuarioViewModel` pode observá-la da `Activity` de login. No entanto, um objeto `ViewModel` só pode observar objetos que são do tipo `LiveData`. Precisamos mudar o tipo do nosso `estaLogado` para ser um `LiveData` de `Boolean`:

```
val estaLogado: LiveData<Boolean>
```

Porém, o tipo `LiveData` é uma classe abstrata que não permite alterarmos seus valores. Para alterarmos os valores de algum `LiveData` na mão, precisamos utilizar um subtipo de `LiveData`, o `MutableLiveData`. Já criaremos um objeto do tipo `MutableLiveData` na declaração da propriedade `estaLogado`:

```
val estaLogado: MutableLiveData<Boolean> = MutableLiveData()
```

Como o tipo `Boolean` está encapsulado no tipo `MutableLiveData`, para alterarmos o valor do `Boolean` em `estaLogado`, precisamos usar a propriedade `value` do `MutableLiveData`:

```
class UsuarioRepository(private val client: WebClient) {
 val estaLogado: MutableLiveData<Boolean> = MutableLiveData()

 // outros métodos

 private val sucesso = {usuario: Usuario ->
 estaLogado.value = true
 Log.i("criou usuário", "sucesso! $usuario")
 }

 private val falha = { excecao: Throwable ->
 estaLogado.value = false
 Log.i("falha na requisição", "erro: $excecao")
 }
}
```

## 8.7 EXERCÍCIO: CRIANDO O USUÁRIO NA API

### Objetivo

Enviar um pedido de criação do usuário para a API usando o Retrofit e, em caso de sucesso, redirecioná-lo para a `ListaActivity`.

### Passo a passo

1. Vamos já adicionar a dependência do retrofit e do conversor de JSON na aplicação:

```

dependencies {
 //demais dependências
 implementation 'com.squareup.retrofit2:retrofit:2.5.0'
 implementation 'com.squareup.retrofit2:converter-gson:2.5.0'
}

```

2. Queremos que, em vez de mostrar um *Log*, seja feita uma requisição ao clicar no botão **CRIAR USUÁRIO**. Faremos a lógica dessa requisição em uma nova classe, `UsuarioWebClient`, para separarmos as responsabilidades. Vamos primeiro alterar a lógica do método `cadastra` do `UsuarioRepository` para chamar um método dessa classe que iremos criar. Já adicionaremos também as implementações das lógicas que serão executadas em caso de sucesso ou de falha da requisição e que passaremos ao método `registra`:

```

class UsuarioRepository(private val client: UsuarioWebClient) {

 fun cadastra(usuario: Usuario) = client.registra(usuario, sucesso, falha)

 fun entra(usuario: Usuario) {
 Log.i("loginConta", "$usuario")
 }

 private val sucesso = fun (usuario: Usuario) {
 Log.i("cria", "sucesso! $usuario")
 }

 private val falha = fun (excecao: Throwable) {
 Log.i("cria", "falha!", excecao)
 }
}

```

O código acima ainda não compila.

3. Para o código acima compilar, precisamos criar a classe `UsuarioWebClient`, que será responsável por fazer as requisições do usuário. Faremos isso em um novo pacote `webservices`. Dentro dela, criaremos a estrutura do método `registra`:

```

class UsuarioWebClient(retrofit: Retrofit) {

 fun registra(usuario: Usuario, sucesso: (usuario: Usuario) -> Unit, falha: (Throwable) -> Unit)
 {
 }
}

```

4. Quem vai implementar a lógica da requisição para a gente é o Retrofit, mas ele pede para informarmos com uma interface quais serviços ele deverá implementar. Criaremos então, dentro da `UsuarioWebClient` mesmo, essa interface. Como queremos apenas criar um usuário no momento, vamos adicionar apenas o método `cria`, que indicará ao Retrofit que queremos uma requisição *post* para a URL `/usuario`:

```

class UsuarioWebClient(retrofit: Retrofit) {

 fun registra(usuario: Usuario, sucesso: (usuario: Usuario) -> Unit, falha: (Throwable) -> Unit)

```

```

{
}

private interface UsuarioService {

 @POST("/usuario")
 fun cria(@Body usuario: Usuario): Call<Usuario>
}

}

```

5. Agora que listamos quais serviços queremos que sejam implementados, vamos pedir para um objeto da classe `Retrofit` criar uma implementação dessa interface e nos fornecer uma instância dessa implementação:

```

class UsuarioWebClient(retrofit: Retrofit) {

 private val service: UsuarioService by lazy {
 retrofit.create(UsuarioService::class.java)
 }

 fun registra(usuario: Usuario, sucesso: (usuario: Usuario) -> Unit, falha: (Throwable) -> Unit)
 {

 private interface UsuarioService {

 @POST("/usuario")
 fun cria(@Body usuario: Usuario): Call<Usuario>
 }
 }
}

```

6. Por fim, vamos implementar a lógica do método `registra`, usando a instância que acabamos de receber:

```

class UsuarioWebClient(retrofit: Retrofit) {

 private val service: UsuarioService by lazy {
 retrofit.create(UsuarioService::class.java)
 }

 fun registra(usuario: Usuario, sucesso: (usuario: Usuario) -> Unit, falha: (Throwable) -> Unit)
 {

 val chamadaPraCriar = service.cria(usuario)

 chamadaPraCriar.enqueue(object : Callback<Usuario> {
 override fun onFailure(call: Call<Usuario>, t: Throwable) {
 falha(t)
 }

 override fun onResponse(call: Call<Usuario>, response: Response<Usuario>) {
 response.body()?.let(sucesso)
 }
 })
 }
 //interface já criada
}

```

```
}
```

7. Precisamos agora atualizar nosso `ViewModelFactory` para criar o `UsuarioRepository` com o `UsuarioWebClient`:

```
object ViewModelFactory : ViewModelProvider.NewInstanceFactory() {

 private fun getTweetRepository() = TweetRepository()

 private fun getUsuarioRepository(): UsuarioRepository {

 val retrofit = InicializadorDoRetrofit.retrofit
 val usuarioWebClient = UsuarioWebClient(retrofit)

 return UsuarioRepository(usuarioWebClient)
 }

 //resto do código
}
```

O código ainda não vai compilar.

8. Para o código acima compilar, precisamos criar uma instância da classe `Retrofit` para poder passar ao nosso `UsuarioWebClient`. Informaremos a essa instância o domínio da API que queremos acessar e quem se preocupará com as conversões de JSON para objeto. Vamos criar um novo **Object** no pacote `webservices`:

```
object InicializadorDoRetrofit {

 val retrofit: Retrofit = Retrofit.Builder()
 .baseUrl("https://twittelum-server.herokuapp.com")
 .addConverterFactory(GsonConverterFactory.create())
 .build()
}
```

9. Nesse momento podemos testar criar um usuário e verificar se aparece um *Log* de sucesso ou de falha no *Logcat*.

10. Para podermos redirecionar para a `MainActivity` apenas quando um usuário estiver logado, nossa `LoginActivity` deverá observar se o usuário está ou não logado:

```
class LoginActivity : AppCompatActivity() {

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

 login_criar.setOnClickListener { viewModel.cria(usuarioDaTela()) }
 login_entrar.setOnClickListener { viewModel.loga(usuarioDaTela()) }

 viewModel.usuarioEstaLogado().observe(this, Observer { estaLogado ->
 estaLogado?.let {
 if (estaLogado) {
 vaiParaMain()
 }
 }
 })
 }
}
```

```

 }
 }
}

viewModel.falha().observe(this, Observer {
 Toast.makeText(this, it?.message, Toast.LENGTH_LONG).show()
 Log.i("Login", "falha ao logar", it)
})
}

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

//restante do código
}

```

O código ainda não vai compilar.

- Para o código acima compilar, implementaremos no nosso `ViewModel` as lógicas de `usuarioEstaLogado()` e de `falha()`, que apenas delegarão a responsabilidade para nosso `Repository`:

```

class UsuarioViewModel(private val repository: UsuarioRepository) : ViewModel() {

 // demais métodos

 fun usuarioEstaLogado() = repository.estalogado

 fun falha() = repository.erro

}

```

O código ainda não vai compilar.

- Voltando ao `UsuarioRepository`, precisamos criar os atributos que chamamos no `UsuarioViewModel` e **alterar** as propriedades `sucesso` e `falha`:

```

class UsuarioRepository(private val client: UsuarioWebClient) {

 val usuarioDaSessao: MutableLiveData<Usuario> = MutableLiveData()
 val estalogado: MutableLiveData<Boolean> = MutableLiveData()
 val erro: MutableLiveData<Throwable> = MutableLiveData()

 fun cadastra(usuario: Usuario) = client.registra(usuario, sucesso, falha)

 fun entra(usuario: Usuario) {
 Log.i("loginConta", "$usuario")
 }

 private val sucesso = { usuario: Usuario ->

 estalogado.value = true
 usuarioDaSessao.value = usuario
 }

 private val falha = { excecao: Throwable ->

```

```
 estaLogado.value = false
 erro.value = excecao
}
}
```

3. Precisamos também adicionar a permissão de internet no nosso manifest:

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
 xmlns:tools="http://schemas.android.com/tools"
 package="br.com.caelum.twittelumappweb">

 <uses-permission android:name="android.permission.INTERNET" />

 <!-- restante -->
</manifest>
```

4. Antes de testar a criação do usuário, confirme se o servidor está rodando em algum lugar. Lembrando que, dependendo de onde for, é necessário atualizar a URL. Execute a aplicação e tente criar um usuário no sistema.



#### Seus livros de tecnologia parecem do século passado?

Conheça a **Casa do Código**, uma **nova** editora, com autores de destaque no mercado, foco em **ebooks** (PDF, epub, mobi), preços **imbatíveis** e assuntos **atuais**.

Com a curadoria da **Caelum** e excelentes autores, é uma abordagem **diferente** para livros de tecnologia no Brasil.

[Casa do Código, Livros de Tecnologia.](#)

## 8.8 EXERCÍCIO: LOGANDO O USUÁRIO NA API

### Objetivo

Enviar um pedido de login de um usuário já cadastrado para a API usando o Retrofit e, em caso de sucesso, redirecioná-lo para a `MainActivity`.

### Passo a passo

- Queremos que, ao clicar no botão `ENTRAR`, seja feita uma requisição em vez de mostrar um *Log*. Faremos essa lógica em um novo método da classe `UsuarioWebClient`, que receberá o usuário e as lógicas para sucesso e falha na requisição. Primeiro **alteraremos** a lógica do método `entra` do

`UsuarioRepository` para chamar esse método que criaremos logo mais:

```
class UsuarioRepository(private val client: UsuarioWebClient) {

 // outros códigos

 fun entra(usuario: Usuario) = client.fazLogin(usuario, sucesso, falha)
}
```

2. Vamos criar o método `fazLogin` na classe `UsuarioWebClient` para o código acima compilar:

```
class UsuarioWebClient(retrofit: Retrofit) {

 private val service: UsuarioService by lazy {
 retrofit.create(UsuarioService::class.java)
 }

 fun fazLogin(usuario: Usuario, sucesso: (usuario: Usuario) -> Unit, falha: (Throwable) -> Unit)
 {
 //interface UsuarioService criada
 }
}
```

3. Para termos a lógica de fazer uma requisição, vamos usar a instância da nossa interface `UsuarioService` que o `Retrofit` já nos passou. Como queremos fazer uma requisição de login na API, teremos também que adicionar na interface `UsuarioService` um método para avisarmos ao `Retrofit` que queremos uma requisição do tipo `post` para a URL `/usuario/login`:

```
class UsuarioWebClient(retrofit: Retrofit) {

 private val service: UsuarioService by lazy {
 retrofit.create(UsuarioService::class.java)
 }

 //método registra já criado

 fun fazLogin(usuario: Usuario, sucesso: (usuario: Usuario) -> Unit, falha: (Throwable) -> Unit)
 {
 val chamadaPraLogar = service.loga(usuario)

 chamadaPraLogar.enqueue(object : Callback<Usuario> {
 override fun onFailure(call: Call<Usuario>, t: Throwable) {
 falha(t)
 }

 override fun onResponse(call: Call<Usuario>, response: Response<Usuario>) {
 response.body()?.let(sucesso)
 }
 })
 }

 private interface UsuarioService {

 @POST("/usuario")
 fun cria(@Body usuario: Usuario): Call<Usuario>

 @POST("/usuario/login")
 }
}
```

```

 fun loga(@Body usuario: Usuario): Call<Usuario>
 }

}

```

4. Antes de testar o login, confirme se o servidor está rodando em algum lugar. Lembrando que dependendo de onde for, é necessário atualizar a URL. Execute a aplicação e tente fazer o login no sistema.

## 8.9 EXERCÍCIO OPCIONAL: FAZENDO O LOGOUT DO USUÁRIO

### Objetivo

Permitir que o usuário possa deslogar da aplicação

### Passo a passo

1. Vamos na nossa `MainActivity` fazer o processo de deslogar, para isso vamos criar um menu nela:

```

class MainActivity : AppCompatActivity() {
 private val viewModel: UsuarioViewModel by lazy {
 ViewModelProviders.of(this, ViewModelFactory).get(UsuarioViewModel::class.java)
 }

 //demais códigos

 override fun onCreateOptionsMenu(menu: Menu?): Boolean {
 menuInflater.inflate(R.menu.main_menu, menu)
 return true
 }

 override fun onOptionsItemSelected(item: MenuItem?): Boolean {
 if (item?.itemId == R.id.menu_sair) {
 viewModel.desloga()
 voltaProLogin()
 }
 return super.onOptionsItemSelected(item)
 }

 private fun voltaProLogin() {
 finish()
 startActivity(Intent(this, LoginActivity::class.java))
 }
}

```

2. O código não está compilando porque ainda falta criarmos o método `desloga` no `UsuarioViewModel`:

```

class UsuarioViewModel(private val repository: UsuarioRepository) : ViewModel() {

 //demais métodos

 fun desloga() = repository.desloga()
}

```

```
}
```

3. Precisamos também criar o método `desloga` no nosso `UsuarioRepository`:

```
class UsuarioRepository(private val client: UsuarioWebClient) {

 val usuarioDaSessao: MutableLiveData<Usuario> = MutableLiveData()
 val estaLogado: MutableLiveData<Boolean> = MutableLiveData()
 val erro: MutableLiveData<Throwable> = MutableLiveData()

 //demais códigos

 fun desloga() {
 estaLogado.value = false
 usuarioDaSessao.value = Usuario()
 }
}
```

4. Ficou faltando apenas criarmos o xml `main_menu` que representa esse menu:

```
<menu xmlns:android="http://schemas.android.com/apk/res/android"
 xmlns:app="http://schemas.android.com/apk/res-auto">

 <item
 android:id="@+id/menu_sair"
 android:title="Sair"
 app:showAsAction="never" />
</menu>
```

5. Rode o aplicativo novamente.

## 8.10 EXERCÍCIO: SALVANDO OS TWEETS NA API

### Objetivo

Quando o usuário clicar no botão flutuante na tela principal, deve ir para uma tela de cadastro que, por sua vez, vai enviar um pedido para a API de criação de um tweet com as informações contidas no formulário.

### Passo a passo

1. Precisamos adicionar ao botão flutuante que está na tela principal a lógica de ir para a `TweetActivity`:

```
class MainActivity : AppCompatActivity() {
 // outros códigos

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

 listenerBottomNavigation()

 main_fab.setOnClickListener {
```

```

 val intent = Intent(this, TweetActivity::class.java)
 startActivity(intent)

 }
}
}

```

- Quando o usuário digitar as informações do tweet e apertar pra salvar, vai ser executado o método `publicaTweet` da `TweetActivity`, já implementado e mostrado abaixo. Ele, por sua vez, vai chamar o método `criaTweet`, também já implementado. Vamos **alterar** o `criaTweet` adicionando a informação do **dono** ao tweet:

```

class TweetActivity : AppCompatActivity() {
 //demais atributos
 private val usuarioViewModel: UsuarioViewModel by lazy {
 ViewModelProviders.of(this, ViewModelFactory).get(UsuarioViewModel::class.java)
 }

 private fun publicaTweet() {
 val tweet = criaTweet()

 viewModel.salva(tweet)

 Toast.makeText(this, "$tweet foi salvo com sucesso :D", Toast.LENGTH_LONG).show()
 }

 fun criaTweet(): Tweet {
 val campoDeMensagemDoTweet = findViewById<EditText>(R.id.tweet_mensagem)

 val mensagemDoTweet: String = campoDeMensagemDoTweet.text.toString()

 val foto: String? = tweet_foto.tag as String?

 val dono = usuarioViewModel.usuarioDaSessao().value

 return Tweet(mensagemDoTweet, foto, dono!!)
 }

 // outros códigos
}

```

Nesse momento o código acima ainda não vai compilar.

- Precisamos primeiro criar o método `usuarioDaSessao` na classe `UsuarioViewModel`, recebendo o usuário da sessão do `UsuarioRepository`:

```

class UsuarioViewModel(private val repository: UsuarioRepository) : ViewModel() {

 // demais métodos

 fun usuarioDaSessao() = repository.usuarioDaSessao

}

```

- Ainda precisamos **alterar** o construtor do modelo `Tweet`, para um tweet ser criado já com seu

dono:

```
data class Tweet(val mensagem: String,
 val foto: String?,
 val dono: Usuario
) {
 // métodos já criados
}
```

5. Depois de termos o objeto tweet, já estamos pedindo pro ViewModel salvar, e o ViewModel já está passando essa responsabilidade para o Repository . Porém, até o momento, o Repository não está fazendo nada. Ele irá apenas delegar a necessidade de salvar um tweet para uma nova classe TweetWebClient , que terá como única responsabilidade fazer requisições relacionadas a tweets. No TweetRepository , altere o método salva para chamar o método insere do TweetWebClient e crie os métodos erro e sucesso para passar ao método insere e tratar o sucesso e a falha da requisição, respectivamente. O código final ficará assim:

```
class TweetRepository(private val client: TweetWebClient) {

 val excecao: MutableLiveData<Throwable> = MutableLiveData()
 val tweetCriado: MutableLiveData<Tweet> = MutableLiveData()

 fun salva(tweet: Tweet) = client.insere(tweet, sucesso(), erro())

 private fun erro() = { erro: Throwable ->
 excecao.value = erro
 }

 private fun sucesso() = { tweet: Tweet ->
 tweetCriado.value = tweet
 }
}
```

Nesse momento o código ainda não vai compilar.

6. Vamos criar a classe TweetWebClient no pacote webservices e dentro dela já criar a interface de serviço TweetService e já pedir uma implementação dessa interface a um objeto do tipo Retrofit :

```
class TweetWebClient(private val retrofit: Retrofit) {

 private val tweetService = retrofit.create(TweetService::class.java)

 private interface TweetService {
 @POST("/tweet")
 fun salva(@Body tweet: Tweet): Call<Tweet>
 }
}
```

7. Agora vamos usar essa implementação da interface para efetivamente fazer a requisição de salvar um tweet, criando o método insere que chamamos pela TweetRepository :

```
class TweetWebClient(private val retrofit: Retrofit) {
```

```

private val tweetService = retrofit.create(TweetService::class.java)

fun insere(
 tweet: Tweet,
 sucesso: (tweet: Tweet) -> Unit,
 falha: (erro: Throwable) -> Unit
) {
 tweetService.salva(tweet).enqueue(object : Callback<Tweet> {
 override fun onFailure(call: Call<Tweet>, t: Throwable) {
 falha(t)
 }

 override fun onResponse(call: Call<Tweet>, response: Response<Tweet>) {
 response.body()?.let(sucesso)
 }
 })
}
// interface
}

```

Nesse momento algum código do nosso projeto não vai compilar.

- Precisamos alterar o nosso `ViewModelFactory` passando um `TweetWebClient` na criação do `TweetRepository`. Vamos aproveitar para refatorar o `ViewModelFactory` criando outro **Object**, `Injetor`, no mesmo arquivo, `ViewModelFactory.kt`. Esse `Injetor` irá isolar as criações dos objetos do tipo `Repository`.

```

object ViewModelFactory : ViewModelProvider.NewInstanceFactory() {
 override fun <T : ViewModel?> create(modelClass: Class<T>): T = when (modelClass) {
 TweetViewModel::class.java -> {
 TweetViewModel(Injetor.getTweetRepository) as T
 }
 else -> {
 UsuarioViewModel(Injetor.getUsuarioRepository) as T
 }
 }
}

object Injetor {
 private val getRetrofit = InicializadorDoRetrofit.retrofit

 private val getTweetWebClient = TweetWebClient(getRetrofit)
 val getTweetRepository = TweetRepository(getTweetWebClient)

 private val getUsuarioWebClient = UsuarioWebClient(getRetrofit)
 val getUsuarioRepository = UsuarioRepository(getUsuarioWebClient)
}

```

- Como queremos observar se houve uma falha na requisição ou se o tweet salvo foi retornado, vamos adicionar as lógicas de `falha` e de `novoTweet` no `TweetViewModel`. Vamos aproveitar para adicionar o dono na mão nos tweets que criamos para a lógica de lista, fazendo o código compilar:

```
class TweetViewModel(private val repository: TweetRepository) : ViewModel() {
```

```

fun salva(tweet: Tweet) = repository.salva(tweet)

fun falha() = repository.excecao

fun novoTweet() = repository.tweetCriado

private val dono: Usuario = Usuario("Maria", "Maria", "123")

fun tweets(): List<Tweet> = listOf(
 Tweet("bla", null, dono),
 Tweet("ble", null, dono),
 Tweet("bli", null, dono),
 Tweet("blo", null, dono),
 Tweet("blu", null, dono)
)
}

```

10. Por fim, precisamos definir na `TweetActivity` qual será a lógica feita em caso de sucesso ou de falha da criação de um tweet, usando o `viewModel` para observar o que foi alterado:

```

class TweetActivity : AppCompatActivity() {
 // outros códigos

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

 //outros códigos

 viewModel.falha().observe(this, Observer { excecao ->
 Toast.makeText(this, "erro: ${excecao?.message}", Toast.LENGTH_LONG).show()
 Log.e("TWEET", "falha na requisição", excecao)
 })

 viewModel.novoTweet().observe(this, Observer { tweet ->
 Toast.makeText(this, "Tweet salvo: ${tweet?.mensagem}", Toast.LENGTH_LONG).show()
 Log.i("TWEET", "$tweet criado na API")
 })
 }
}

```

11. Rode o aplicativo, tente salvar um tweet e verifique se apareceu um `Toast` e um `Log` no `LogCat` com a mensagem do tweet criado.

**Agora é a melhor hora de respirar mais tecnologia!**



Se você está gostando dessa apostila, certamente vai aproveitar os **cursos online** que lançamos na plataforma **Alura**. Você estuda a qualquer momento com a **qualidade** Caelum. Programação, Mobile, Design, Infra, Front-End e Business! Ex-aluno da Caelum tem 15% de desconto, siga o link!

[Conheça a Alura Cursos Online.](#)

## 8.11 ATUALIZANDO LISTA COM INFORMAÇÕES DO SERVIDOR

### Pegando informações do servidor

Para pegarmos todos os *tweets* do servidor, precisamos definir as configurações para a requisição que queremos fazer na interface `TweetService`, que será implementada pelo `Retrofit`. Diferentemente das requisições que já configuramos para o `Retrofit`, essa será com o método `get`, pois apenas está *pegando* informações do lado do servidor. Para definir que a requisição será do tipo `get`, usamos a anotação `@Get`, passando a URL que queremos acessar. Queremos que a resposta da requisição seja uma lista de *tweets*, portanto, avisamos isso com o retorno do método da interface, que deve ser um objeto que possa fazer uma chamada para o servidor e nos dar uma resposta do tipo `List<Tweet>`:

```
private interface TweetService {
 // demais métodos
 @GET("/tweet")
 fun lista(): Call<List<Tweet>>
}
```

### Atualizando a lista da tela

Queremos que, assim que chegar a resposta da requisição, a lista que está na tela seja atualizada com os dados que vieram da requisição. Para as `Activity`s de lista perceberem que teve uma alteração na lista e atualizarem a tela no momento da alteração, a lista precisa ser observável pelo `ViewModel`, ou seja, precisa ser do tipo `LiveData`. Então, no `Repository` do *tweet*, vamos criar uma propriedade do tipo `MutableLiveData`, porque a gente que vai atualizar o valor desse `LiveData` na mão:

```
class TweetRepository(private val client: TweetWebClient) {
 private val lista: MutableLiveData<List<Tweet>> = MutableLiveData()
```

```
// demais métodos
}
```

Precisamos agora chamar um método do `TweetWebClient` que deverá fazer para a gente a lógica de buscar os `tweets` no servidor. Precisamos passar para esse método que lógica deve ser realizada no caso de sucesso e no caso de falha da requisição:

```
class TweetRepository(private val client: TweetWebClient) {
 private val lista: MutableLiveData<List<Tweet>> = MutableLiveData()

 // demais métodos

 fun buscaLista() = client.buscaTweets(sucessoParaLista, falha)

 private val sucessoParaLista = { tweets: List<Tweet> ->
 //lógica de sucesso
 }

 private val falha = { t: Throwable ->
 //lógica de sucesso
 }

}
```

Para a lógica de sucesso, queremos alterar o valor da propriedade `lista`. Para isso, podemos alterar a propriedade `value`:

```
class TweetRepository(private val client: TweetWebClient) {
 private val lista: MutableLiveData<List<Tweet>> = MutableLiveData()

 // demais métodos

 private val sucessoParaLista = { tweets: List<Tweet> ->
 lista.value = tweets
 }

}
```

No entanto, se houver outra alteração na mesma propriedade `value` logo depois, queremos garantir que a alteração feita na lógica do `sucessoParaLista` será a última a ser executada no objeto do tipo `MutableLiveData`. Conseguimos isso com o método `postValue`, que define uma lógica para ser executada por último se houverem várias alterações na propriedade `value`:

```
class TweetRepository(private val client: TweetWebClient) {
 private val lista: MutableLiveData<List<Tweet>> = MutableLiveData()

 // demais métodos

 private val sucessoParaLista = { tweets: List<Tweet> ->
 lista.postValue(tweets)
 }

}
```

## 8.12 EXERCÍCIO: LISTANDO OS TWEETS INSERIDOS NA API

---

## Objetivo

- Quando o usuário for para a tela de listagem, deverá ver uma lista com todos os tweets presentes na API.
- Quando o usuário for para a tela de busca de tweets, deverá poder filtrar pelos tweets presentes na API.

## Passo a passo

1. Queremos carregar a lista de tweets da API assim que o usuário entrar na tela principal. Para isso, na criação da `MainActivity`, vamos pedir à camada `ViewModel` para carregar os tweets para a gente chamando o método `carregaTweets`. Não se esqueça de **pegar o viewModel** pela `ViewModelFactory`:

```
class MainActivity : AppCompatActivity() {
 private val tweetViewModel: TweetViewModel by lazy {
 ViewModelProviders.of(this, ViewModelFactory).get(TweetViewModel::class.java)
 }

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

 tweetViewModel.carregaLista()

 // demais códigos
 }

 // demais métodos
}
```

Nesse momento o código ainda não compila.

2. Vamos criar esse método `carregaLista` na `TweetViewModel`, que vai delegar a responsabilidade de conseguir a lista para a camada `Repository`, chamando o método `buscaLista` da `TweetRepository`:

```
class TweetViewModel(private val repository: TweetRepository) : ViewModel() {

 // demais códigos

 fun carregaLista() = repository.buscaLista()
}
```

Nesse momento o código ainda não vai compilar.

3. Precisamos criar o método `buscaLista` na `TweetRepository`, que vai pedir à classe `TweetWebClient` a lista com os tweets da API, chamando o novo método `buscaTweets`. Esse método vai receber a função `sucessoParaLista`, que será executada em caso de sucesso da

requisição e atualizará uma lista com os tweets que chegarem da requisição. Também vai receber a função `falha`, já criada, para sabermos se houve erro na requisição:

```
class TweetRepository(private val client: TweetWebClient) {
 private val lista: MutableLiveData<List<Tweet>> = MutableLiveData()

 // demais métodos

 fun buscaLista() = client.buscaTweets(sucessoParaLista(), erro())

 private fun sucessoParaLista() = { tweets: List<Tweet> ->
 lista.postValue(tweets)
 }

}
```

Nesse momento o código ainda não vai compilar.

4. Por fim, vamos criar o método `buscaTweets` na classe `TweetWebClient` para fazer a requisição efetivamente. Teremos também que informar à interface de serviço que precisaremos da implementação de mais um método, que chamaremos de `busca`:

```
class TweetWebClient(private val retrofit: Retrofit) {

 private val tweetService = retrofit.create(TweetService::class.java)

 fun buscaTweets(
 sucesso: (tweets: List<Tweet>) -> Unit,
 falha: (erro: Throwable) -> Unit
) {

 tweetService.busca().enqueue(object : Callback<List<Tweet>> {
 override fun onFailure(call: Call<List<Tweet>>, t: Throwable) {
 falha(t)
 }

 override fun onResponse(call: Call<List<Tweet>>, response: Response<List<Tweet>>) {
 response.body()?.let(sucesso)
 }
 })

 // demais métodos

 private interface TweetService {
 // demais métodos

 @GET("/tweet")
 fun busca(): Call<List<Tweet>>
 }
 }
}
```

5. Como queremos observar se a lista de tweets mudou para atualizar a tela, vamos **alterar** o método `tweets` na `TweetViewModel` para pedir a lista de `tweets` ao `TweetRepository` e nos devolver um

`LiveData` de lista de tweets. Não se esqueça de tirar o *dono* que havíamos criado na mão. O código final ficará assim:

```
class TweetViewModel(private val repository: TweetRepository) : ViewModel() {

 fun salva(tweet: Tweet) = repository.salva(tweet)

 fun tweets(): LiveData<List<Tweet>> = repository.pegaLista()

 fun falha() = repository.excecao

 fun novoTweet() = repository.tweetCriado

 fun carregaLista() = repository.buscaLista()

}
```

Nesse momento o código ainda não vai compilar.

6. Vamos criar esse método `pegaLista` no `TweetRepository`. Ele deve retornar algo que seja um `LiveData` e, nesse caso, retornará a lista que está guardando os tweets:

```
class TweetRepository(private val client: TweetWebClient) {

 // demais atributos e métodos

 fun pegaLista(): LiveData<List<Tweet>> = lista

}
```

Nesse momento algum código no projeto não vai compilar.

7. Agora o método `tweets` do `viewModel` não devolve mais uma lista de tweets simples, devolve um `LiveData`:

- No `ListaTweetsFragment`, vamos alterar a chamada do método `tweets` para observar quando há mudanças na lista de tweets:

```
class ListaTweetsFragment : Fragment() {
 private val viewModel: TweetViewModel by lazy {
 ViewModelProviders.of(activity!!, ViewModelFactory).get(TweetViewModel::class.java)
 }

 override fun onCreateView(inflater: LayoutInflater, container: ViewGroup?, savedInstanceState: Bundle?): View? {

 val view = inflater.inflate(R.layout.lista_tweets_fragment, container, false)

 viewModel.tweets().observe(this, Observer { lista ->
 lista?.let{ tweets ->
 view.lista_tweets.adapter = TweetAdapter(tweets)
 }
 })

 return view
 }
}
```

```
}
```

- No `BuscadorDeTweetsFragment`, vamos pegar o valor que está contido dentro do `LiveData`:

```
class BuscadorDeTweetsFragment : Fragment() {
 // outros métodos

 private fun filtraTweetsPelo(texto: String?): List<Tweet> {
 viewModel.tweets().value?.let { tweets ->
 val tweetsFiltrados = tweets.filter { tweet -> tweet.mensagem.contains(texto!!) }
 return tweetsFiltrados
 }
 return emptyList()
 }

}
```

8. Rode o aplicativo e veja que os tweets são listados e filtrados. Porém, ainda não está mostrando o nome do dono de cada tweet. Precisamos adicionar essa informação à tela de listagem, pelo `TweetAdapter`, **alterando** o método `getView`:

```
class TweetAdapter(private val tweets: List<Tweet>) : BaseAdapter() {
 override fun getView(position: Int, convertView: View?, parent: ViewGroup?): View {

 val tweet = tweets[position]

 val inflater = LayoutInflater.from(parent?.context)
 val view = inflater.inflate(R.layout(tweet_item, parent, false)

 view.item_conteudo.text = tweet.mensagem

 tweet.foto?.let {
 view.item_foto.visibility = View.VISIBLE
 view.item_foto.setImageBitmap(Carregador.decodeImage(it))
 }

 view.item_dono.text = tweet.dono.nome

 return view
 }
 // demais métodos
}
```

9. Rode o aplicativo e veja se os tweets são listados e filtrados e se estão mostrando o nome do dono de cada tweet.

## 8.13 ATUALIZANDO A LISTA COM SWIPE

Quando vemos uma lista em qualquer aplicativo e queremos garantir que os dados na lista são os mais recentes, arrastamos a tela para baixo, com a intenção de atualizar o conteúdo da lista. Esse conceito é chamado de **swipe to refresh**. Como isso ficou muito comum em aplicativos, o Android já possui uma componente com esse comportamento, a `SwipeRefreshLayout`, da biblioteca de suporte.

Vamos deixar nossa lista com a função de *swipe*. Como o comportamento dessa componente deve

valer para a tela toda, ela é um `Layout`, ou seja, podemos trocar a componente pai que estiver no `layout` de lista pela componente `SwipeRefreshLayout`:

```
<androidx.swiperefreshlayout.widget.SwipeRefreshLayout
 xmlns:android="http://schemas.android.com/apk/res/android"
 android:layout_width="match_parent"
 android:layout_height="match_parent">

 <ListView
 android:id="@+id/lista_tweets"
 android:layout_width="match_parent"
 android:layout_height="match_parent" />

</androidx.swiperefreshlayout.widget.SwipeRefreshLayout>
```

Só com essa mudança, quando arrastarmos a tela para baixo, vai aparecer um ícone redondo girando, dando a ideia de que alguma lógica está sendo processada.

No entanto, ele ficará eternamente girando e visível na tela. Precisamos avisar à componente quando o ícone deve sair da tela. Fazemos isso setando para `false` a propriedade `isRefreshing` da componente. Se dermos o `id swipe` pra componente no XML, no momento em que quisermos avisar que a atualização terminou sumindo com o ícone, podemos escrever o código:

```
swipe.isRefreshing = false
```

Além de avisar à componente quando a lógica terminou, precisamos avisar *que lógica* será feita quando o `swipe` for ativado, ou seja, quando a tela for arrastada para baixo. Fazemos isso definindo um *listener* com o método `setOnRefreshListener` da componente `SwipeRefreshLayout`:

```
swipe.setOnRefreshListener {
 //lógica de pegar os tweets do servidor
}
```

Com essas lógicas da componente `SwipeRefreshLayout`, conseguimos carregar os *tweets* do servidor assim que o usuário fizer um `swipe` e faer com que o ícone desapareça quando o processo finalizar. Podemos customizar a componente, por exemplo mudando as cores do ícone com o método `setColorSchemeColors`. Para o ícone ficar com as cores azul, vermelho e verde, intercaladas, podemos escrever o código:

```
swipe.setColorSchemeColors(Color.BLUE, Color.RED, Color.GREEN)
```

## Editora Casa do Código com livros de uma forma diferente



Editoras tradicionais pouco ligam para ebooks e novas tecnologias. Não dominam tecnicamente o assunto para revisar os livros a fundo. Não têm anos de experiência em didáticas com cursos.

Conheça a **Casa do Código**, uma editora diferente, com curadoria da **Caelum** e obsessão por livros de qualidade a preços justos.

[Casa do Código, ebook com preço de ebook.](#)

## 8.14 EXERCÍCIO: ATUALIZANDO OS TWEETS

### Objetivo

Atualizar os tweets com o conceito swipe to refresh

### Passo a passo

1. Vamos atualizar o layout `lista_tweets_fragment.xml` para passar a ter o `SwipeRefreshLayout`:

```
<androidx.swiperefreshlayout.widget.SwipeRefreshLayout
 xmlns:android="http://schemas.android.com/apk/res/android"
 android:layout_width="match_parent"
 android:id="@+id/swipe"
 android:layout_height="match_parent">

 <ListView
 android:id="@+id/lista_tweets"
 android:layout_width="match_parent"
 android:layout_height="match_parent" />

</androidx.swiperefreshlayout.widget.SwipeRefreshLayout>
```

2. Agora em nosso fragment, precisaremos atualizar a criação dele:

```
class ListaTweetsFragment : Fragment() {

 private val viewModel: TweetViewModel by lazy {
 ViewModelProviders.of(activity!!, ViewModelFactory).get(TweetViewModel::class.java)
 }

 override fun onCreateView(inflater: LayoutInflater, container: ViewGroup?, savedInstanceState: Bundle?): View? {

 val view = inflater.inflate(R.layout.lista_tweet_fragment, container, false)

 viewModel.tweets().observe(this, Observer { lista ->
```

```
 view.swipe.isRefreshing = false

 lista?.let{ tweets ->
 view.lista_tweets.adapter = TweetAdapter(tweets)
 }
 })

view.swipe.setOnRefreshListener { viewModel.carregaLista() }
view.swipe.setColorSchemeColors(Color.BLUE, Color.RED, Color.GREEN)

return view
}

}
```

3. Crie um tweet e atualize a lista novamente.

# TRABALHANDO COM MAPS

## 9.1 O QUE SABEREI FAZER NO FIM DESSE CAPÍTULO ?

- Usar o GoogleMaps dentro da aplicação
- Adicionar comportamentos no mapa



Conheça a **Casa do Código**, uma **nova** editora, com autores de destaque no mercado, foco em **ebooks** (PDF, epub, mobi), preços **imbatíveis** e assuntos **atuais**.

Com a curadoria da **Caelum** e excelentes autores, é uma abordagem **diferente** para livros de tecnologia no Brasil.

[Casa do Código, Livros de Tecnologia.](#)

## 9.2 MOSTRANDO UM MAPA PRO USUÁRIO

Atualmente, em diversos aplicativos é importante mostrar um mapa com a localização do que está sendo oferecido. Alguns exemplos são um aplicativo de farmácia, para mostrar as farmácias daquela rede mais próximas do usuário, e um aplicativo de banco, para mostrar a localização de cada agência. Nossos aplicativos de *tweet* também irão mostrar um mapa, com a localização de cada *tweet* que foi publicado. Mas como faremos para ter a tela de um mapa?

Cada um desses aplicativos que mostram o mapa tem a tela customizada para o tema do próprio aplicativo, mas, para não precisarem criar o mapa do zero, muitos reutilizam um mapa já criado de alguma empresa que já disponibiliza mapas, como a **Google**.

Para utilizarmos as classes que a **Google** disponibiliza para o mapa, precisamos adicionar algumas dependências no `build.gradle`. Além disso, quando mostramos um mapa para o usuário, geralmente ele vai querer acessar a localização dele mesmo, para isso, precisamos pedir permissão para o acesso da sua

localização. Por fim, para utilizarmos qualquer serviço da Google , precisamos dizer à empresa quem está acessando o serviço. Fazemos isso com uma chave pra ligar a aplicação com alguma conta da Google. Essa chave é gerada pelo *Console de API* da Google e precisamos inserí-la em um arquivo de configuração do Google Maps .

Esse processo todo era feito manualmente há um tempo atrás, mas a Google começou a facilitar a vida dos programadores permitindo que toda essa configuração seja feita pelo próprio Android Studio , bastando apenas pedir para criar uma Google Maps Activity que as permissões e dependências necessárias serão adicionadas nos arquivos corretos. Também já será criado o arquivo de configuração do Google Maps , chamado de google\_maps\_api.xml e ele terá uma explicação de como conseguir a chave para a API. Se não quisermos utilizar a Activity e o arquivo de layout gerados, basta apagá-los no final do processo. O exercício a seguir tem o passo a passo a ser feito.

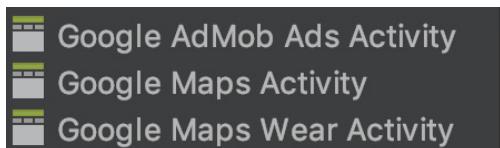
## 9.3 EXERCÍCIO: CONFIGURANDO O GOOGLE MAPS

### Objetivo

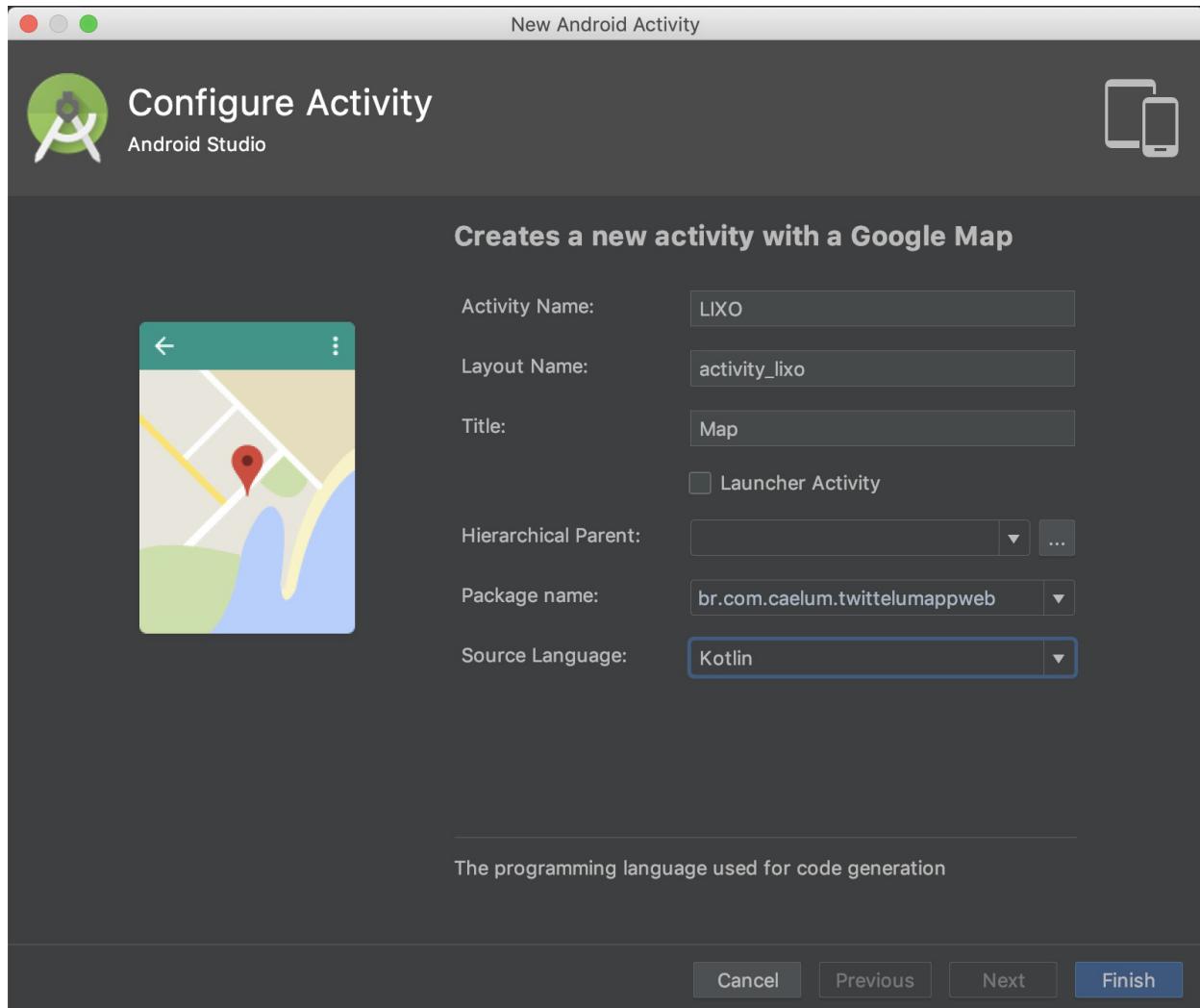
Conseguir através do Android Studio configurar a biblioteca do google maps e habilitar o acesso na Api do google

### Passo a passo

1. Vá na pasta raiz do projeto e clique em new com o botão direito do mouse. Procure por Gooogle e escolha a opção: Google Maps Activity.



2. Altere o nome da Activity que será gerada por LIXO e dê ok.



3. Delete a activity e o layout xml que foram gerados com o nome lixo.
4. Abra o arquivo `google_maps_api.xml` que foi gerado quando fizemos esse procedimento. Nele haverá um link em comentário logo nas primeiras linhas, pegue a linha completa e abra em qualquer navegador.
5. Siga o passo a passo no sistema web e pegue o token que será gerado e coloque-o no arquivo `google_maps_api.xml` onde está pedindo.

## 9.4 USANDO O MAPA DA GOOGLE

Na nossa aplicação, assim como em diversas outras aplicações que utilizam o mapa da Google, não queremos mostrar a tela inteira como sendo da Google. Queremos mostrar uma tela da nossa aplicação, com nosso tema e algumas outras componentes além do mapa. Para isso, queremos mostrar o mapa em um **pedaço de tela**, ou seja, em um `Fragment`. Felizmente, o Google Maps já fornece um fragmento

de mapa, o `SupportMapFragment`. Para utilizá-lo, basta criarmos uma classe nossa que tenha os mesmos comportamentos e características que o fragmento de mapa da Google, herdando dessa classe:

```
class MeuMapaFragment : SupportMapFragment()
```

Com isso, ao exibir o `MeuMapaFragment` em alguma tela, já veremos o mapa da Google no pedaço de tela que disponibilizamos para o `Fragment`.

**Agora é a melhor hora de respirar mais tecnologia!**



Se você está gostando dessa apostila, certamente vai aproveitar os **cursos online** que lançamos na plataforma **Alura**. Você estuda a qualquer momento com a **qualidade** Caelum. Programação, Mobile, Design, Infra, Front-End e Business! Ex-aluno da Caelum tem 15% de desconto, siga o link!

[Conheça a Alura Cursos Online.](#)

## 9.5 EXERCÍCIO: EXIBINDO O MAPA

### Objetivo

Ao clicar na aba do mapa exiba nosso mapa

### Passo a passo

1. Crie a classe `MapaFragment` que herde de `SupportMapFragment`:

```
class MapaFragment : SupportMapFragment() {}
```

2. Agora precisamos exibi-lo assim que o clique sobre o botão do nosso `BottomNavigationView` for clicado em nossa `MainActivity`:

```
private fun listenerBottomNavigation() {
 bottom_navigation.setOnNavigationItemSelected { item ->

 when (item.itemId) {
 R.id.menu_tweets -> {
 exibe(ListaTweetsFragment())
 true
 }

 R.id.menu_busca -> {
 exibe(BuscadorDeTweetsFragment())
 true
 }
 }
 }
}
```

```
 }

 R.id.menu_mapa -> {
 exibe(MapaFragment())
 true
 }
 else -> {
 false
 }
}
}
```

# TRABALHANDO COM GPS

## 10.1 O QUE SABEREI FAZER NO FIM DESSE CAPÍTULO ?

- Manipular o GPS
- Manipular o mapa
- Elvis operator

Aprenda se divertindo na Alura Start!



Você conhece alguém que tem potencial para tecnologia e programação, mas que nunca escreveu uma linha de código? Pode ser um filho, sobrinho, amigo ou parente distante. Na **Alura Start** ela vai poder criar games, apps, sites e muito mais! É o **começo da jornada com programação e a porta de entrada para uma possível carreira de sucesso**. Ela vai estudar em seu próprio ritmo e com a melhor didática. A qualidade da conceituada Alura, agora para Starters.

[Conheça os cursos online da Alura Start!](#)

## 10.2 ACESSANDO A LOCALIZAÇÃO

Nosso aplicativo vai permitir que um *tweet* seja salvo com a localização da pessoa que escreveu o *tweet*. Para isso, precisamos utilizar API da *Google Play Services* de localização, a [Google Location Services API](#). A primeira coisa que precisamos fazer para conseguir utilizar essa API é incluir sua dependência no projeto:

```
implementation 'com.google.android.gms:play-services-location:16.0.0'
```

Para conseguir pegar a localização atual do dispositivo, precisamos estabelecer uma conexão com o GPS. A *Google Location Services API* possui uma classe responsável por se comunicar com o GPS, a `FusedLocationProviderClient` . Para obtermos uma instância dessa classe, pedimos à classe

`LocationServices` com o método estático `getFusedLocationProviderClient`, que pede um objeto do tipo `Context` como parâmetro:

```
val client: FusedLocationProviderClient = LocationServices.getFusedLocationProviderClient(context)
```

Podemos agora pedir a esse `client`, responsável por se comunicar com o GPS, a localização atual. Como o usuário pode se locomover, geralmente as telas que mostram a localização atual precisam pedir várias vezes a localização ao serviço do GPS, de forma frequente. Portanto, para pedir a localização, usamos o método `requestLocationUpdates` do objeto `client`, do tipo `FusedLocationProviderClient`. Precisamos informar vários detalhes sobre o pedido da localização ao GPS, como quão precisa queremos que a informação seja e qual a frequência na repetição dos pedidos. Definimos essas e outras configurações com um objeto do tipo `LocationRequest`. Também precisamos definir que lógica queremos fazer no resultado de cada requisição. Essa lógica fica em um método definido pela interface `LocationCallback`. O método `requestLocationUpdates`, portanto, recebe um objeto `LocationRequest` e uma implementação de `LocationCallback`. Ele pede um terceiro parâmetro que basicamente vai definir em que *thread* será executada a lógica do *callback*. Como a lógica vai ser de apenas atualizar valores ao *tweet*, podemos executá-la na *thread* que o método `requestLocationUpdates` será chamado, ou seja, na *thread* principal. Definimos isso passando `null` para o terceiro parâmetro:

```
val client: FusedLocationProviderClient = //pega client da LocationServices
val request: LocationRequest
val locationCallback: LocationCallback
client.requestLocationUpdates(request, locationCallback, null)
```

Para cancelar as atualizações frequentes da localização, o objeto `FusedLocationProviderClient` possui um método chamado `removeLocationUpdates`, que recebe uma implementação de `LocationCallback`:

```
client.removeLocationUpdates(locationCallback)
```

## Configurando o `LocationRequest`

Para termos um objeto do tipo `LocationRequest`, basta instanciá-lo:

```
val request: LocationRequest = LocationRequest()
```

Há uma série de configurações que podemos passar para um pedido de localização, como a frequencia entre cada pedido, que definimos pela propriedade `interval` o tempo em milissegundos entre cada requisição. Por exemplo, para fazer uma nova requisição a cada 1 segundo, passamos o valor `1000` para o `interval`:

```
request.interval = 1000
```

Outra configuração muito importante é a prioridade com que a localização é procurada, e está intimamente ligada com a precisão e com a origem da informação. Esse valor deve ser passado por

constantes pré-definidas, como `LocationRequest.PRIORITY_HIGH_ACCURACY` para a maior precisão possível. Essa configuração é definida pela propriedade `priority`:

```
request.priority = LocationRequest.PRIORITY_HIGH_ACCURACY
```

Podemos passar várias outras informações, como qual a distância mínima de locomoção para a localização ser efetivamente atualizada. Essa configuração é definida pela propriedade `smallestDisplacement` com um valor do tipo `float`, calculado em metros:

```
request.smallestDisplacement = 10.0f
```

## Utilizando o operador apply

Juntando essas três configurações, temos o código:

```
request.interval = 1000
request.priority = LocationRequest.PRIORITY_HIGH_ACCURACY
request.smallestDisplacement = 10.0f
```

Só estamos chamando propriedades do objeto em `request`, mas temos que escrever essa variável todas as vezes. Para evitar isso, o Kotlin criou o operador `apply`, que permite executarmos uma lógica que opera principalmente nas propriedades do objeto que chamou o `apply` e é usado mais comumente em configurações de objetos, justamente o nosso caso. Com o operador `apply`, o código acima pode ser reescrito da seguinte maneira:

```
request.apply {
 interval = 1000
 priority = LocationRequest.PRIORITY_HIGH_ACCURACY
 smallestDisplacement = 10.0f
}
```

## Definindo a lógica de callback

Para termos uma implementação de `LocationCallback`, basta criarmos uma classe que implemente a interface e o método `onLocationResult` que ela define:

```
class RespostaDaLocalizacao : LocationCallback {
 override fun onLocationResult(result: LocationResult?) {
 // lógica com o resultado da localização
 }
}
```

Esse método recebe um objeto do tipo `LocationResult`. Conseguimos a informação da última localização calculada pela propriedade `lastLocation` desse objeto. Como esse `LocationResult` pode vir nulo, só acessaremos a propriedade se houver realmente um objeto na variável `result`:

```
class RespostaDaLocalizacao : LocationCallback {
 override fun onLocationResult(result: LocationResult?) {
 val lastLocation: Location = result?.lastLocation
 }
}
```

## 10.3 PASSANDO COORDENADAS PARA O TWEET

Com a implementação do `LocationCallback`, conseguimos um objeto do tipo `Location`. Podemos pedir esse objeto na hora de criar um *tweet* chamando um método `getLocation` da nossa `RespostaDaLocalizacao`. Para isso, precisamos deixar a `Location` acessível por todos os métodos da `RespostaDaLocalizacao` e permitir que seu valor seja nulo, caso o método `onLocationResult` ainda não tenha sido chamado. A classe `RespostaDaLocalizacao` terá um código parecido com:

```
class RespostaDaLocalizacao : LocationCallback {
 private var location: Location? = null

 override fun onLocationResult(result: LocationResult?) {
 location = result?.lastLocation
 }

 fun getLocation(): Location? = location
}
```

Chamaremos esse código na hora de criar um *tweet*, dentro da `TweetActivity`:

```
class TweetActivity : AppCompatActivity() {
 private val respostaDaLocalizacao: RespostaDaLocalizacao = RespostaDaLocalizacao()

 fun criaTweet(): Tweet {

 val localizacao: Location = respostaDaLocalizacao.getLocation()

 val tweet = // pega os outros os dados do tweet e cria instancia de tweet com todos os dado
 s
 return tweet
 }
}
```

Com o código acima, estamos deixando disponível na `Activity` um código da *Google Location Services API*. Se mudarmos o modo como acessamos o GPS, a `Activity` será impactada. Para evitar isso, vamos passar para a `Activity` apenas as informações que ela precisa: a **latitude** e a **longitude**, informações que o objeto `Location` possui. Na classe `RespostaDaLocalizacao`, trocaremos o método `getLocation` pelo `getCoordenadas`, que irá pegar essas informações da `Location`:

```
class RespostaDaLocalizacao : LocationCallback {
 private var location: Location? = null

 override fun onLocationResult(result: LocationResult?) {
 location = result?.lastLocation
 }

 fun getCoordenadas() {
 val latitude = lastLocation?.latitude
 val longitude = lastLocation?.longitude
 }
}
```

Esse método precisará agora devolver essas duas informações, mas o retorno de um método deve ser só um valor. Podemos criar uma classe nossa para armazenar esses dois valores em um único tipo, mas

só vamos usar essa classe nesse momento. Também podemos devolver um *array*, mas precisaremos saber que valor está em que posição do *array*. No fundo, só estamos querendo agrupar esses valores para devolver os dois de uma vez em um método. Ao pegar o retorno do método, não vamos precisar das duas informações juntas. Para agrupar dois valores momentaneamente, o *Kotlin* disponibiliza a classe *Pair*, que sempre é definida para dois tipos específicos com o *generics*. Para criar um objeto do tipo *Pair<Double, Double>*, usamos seu construtor, que receberá dois valores do tipo *Double*, conforme especificamos pelo *generics*. Podemos então retornar um objeto do tipo *Pair<Double, Double>* no método *getCoordenadas*. Esse objeto será criado com a *latitude* e *longitude*, ou *0.0* para cada valor que for nulo:

```
fun getCoordenadas(): Pair<Double, Double> {
 val latitude = lastLocation?.latitude
 val longitude = lastLocation?.longitude
 return Pair(latitude ?: 0.0, longitude ?: 0.0)
}
```

Na hora de criar um *tweet*, agora chamaremos o método *getCoordenadas*, que devolve um *Pair* com dois valores:

```
class TweetActivity : AppCompatActivity() {
 private val respostaDaLocalizacao: RespostaDaLocalizacao = RespostaDaLocalizacao()

 fun criaTweet(): Tweet {
 val coordenadas: Pair<Double, Double> = respostaDaLocalizacao.getCoordenadas()

 val tweet = // pega os outros os dados do tweet e cria instancia de tweet com todos os dado
 s
 return tweet
 }
}
```

Para passar para o construtor do *Tweet*, precisaremos chamar cada uma das informações de coordenadas separadamente. Podemos acessá-las com as propriedades *first* e *second* de *Pair*:

```
val coordenadas: Pair<Double, Double> = respostaDaLocalizacao.getCoordenadas()
val latitude: Double = coordenadas.first
val longitude: Double = coordenadas.second
```

Uma outra forma de fazer exatamente a mesma lógica que o código acima é pedindo para o *Kotlin* já guardar para a gente os dois valores retornados pelo *getCoordenadas* em variáveis diferentes, utilizando a sintaxe de ***destructuring declaration***, em que conseguimos justamente "decompor" um objeto em várias variáveis. Usando o código acima como base mas usando sintaxe de *destructuring declaration*, teremos o código:

```
val (latitude, longitude): Pair<Double, Double> = respostaDaLocalizacao.getCoordenadas()
```

## 10.4 EXERCÍCIO: FAZENDO NOSSO TWEET TER POSIÇÃO

## Objetivo

Pegar a localização atual do aparelho a cada 5s e passar a latitude e longitude no construtor do Tweet .

## Passo a passo

1. Adicione no gradle a dependencia do gps:

```
implementation 'com.google.android.gms:play-services-location:16.0.0'
```

2. Crie a classe GPS:

```
class GPS(context: Context) : LocationCallback() {

 private val client: FusedLocationProviderClient = LocationServices.getFusedLocationProviderClient(context)

 var lastLocation: Location? = null

 override fun onLocationResult(result: LocationResult?) {
 lastLocation = result?.lastLocation
 }

 fun fazBusca() {
 val requisicao = LocationRequest()

 requisicao.apply {
 interval = 5000
 priority = LocationRequest.PRIORITY_HIGH_ACCURACY
 smallestDisplacement = 10.0f
 }

 client.requestLocationUpdates(requisicao, this, null)
 }

 fun cancela() {
 client.removeLocationUpdates(this)
 }

 fun coordenadas(): Pair<Double, Double> {
 val latitude = lastLocation?.latitude
 val longitude = lastLocation?.longitude
 return Pair(latitude ?: 0.0, longitude ?: 0.0)
 }
}
```

3. Vamos adicionar as propriedades de local na classe Tweet :

```
data class Tweet(val mensagem: String,
 val foto: String?,
 val dono: Usuario,
 val latitude: Double,
 val longitude: Double,
 val id: Long = 0) {
 //demais códigos
}
```

}

4. No momento em que estivermos criando um tweet será necessário passarmos a latitude e longitude:

```
class TweetActivity : AppCompatActivity() {

 //demais códigos

 private lateinit var gps: GPS

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

 supportActionBar?.setDisplayHomeAsUpEnabled(true)

 gps = GPS(this)
 gps.fazBusca()

 // demais códigos
 }

 override fun onDestroy() {
 super.onDestroy()
 gps.cancela()
 }

 fun criaTweet(): Tweet {

 val campoDeMensagemDoTweet = findViewById<EditText>(R.id.tweet_mensagem)

 val mensagemDoTweet: String = campoDeMensagemDoTweet.text.toString()

 val foto: String? = tweet_foto.tag as String?

 val dono = usuarioViewModel.usuarioDaSessao().value

 val (latitude, longitude) = gps.coordenadas()

 return Tweet(mensagemDoTweet, foto, dono!!, latitude, longitude)
 }

 // restante do código
}
```

5. Rode novamente o aplicativo e veja se tudo está funcionando de acordo.

**Seus livros de tecnologia parecem do século passado?**



Conheça a **Casa do Código**, uma **nova** editora, com autores de destaque no mercado, foco em **ebooks** (PDF, epub, mobi), preços **imbatíveis** e assuntos **atuais**.

Com a curadoria da **Caelum** e excelentes autores, é uma abordagem **diferente** para livros de tecnologia no Brasil.

[Casa do Código, Livros de Tecnologia.](#)

## 10.5 ADICIONANDO MARCADORES NO MAPA

Queremos mostrar os *tweets* no mapa, mas só podemos adicionar os marcadores para a posição de cada *tweet* depois que o mapa estiver renderizado.

Como depende de conexão, processamento, entre outras coisas, para o mapa ficar pronto, não sabemos quanto tempo esse processo vai levar, podendo ser demorado. Por isso, esse processo de renderizar o mapa é feito assincronamente. Para falarmos que queremos adicionar qualquer informação no mapa assim que ele ficar pronto, chamamos o método `getMapAsync` do `SupportMapFragment`. Pensando no ciclo de vida do nosso fragmento de mapa, o melhor momento para pedir o mapa pronto é quando o fragmento já está quase pronto para ser mostrado na tela, ou seja, quando está sendo renderizado o fragmento. Esse momento acontece no método `onResume` :

```
class MapaFragment : SupportMapFragment() {

 override fun onResume() {
 super.onResume()
 getMapAsync()
 }

}
```

O método `getMapAsync` , no entanto, vai pedir como parâmetro a lógica a ser realizada quando o mapa estiver pronto por meio da interface `OnMapReadyCallback` , que nos forçará a implementar o método `onMapReady` . Precisaremos então de uma implementação dessa interface:

```
class AdicionaMarcador : OnMapReadyCallback {
 override fun onMapReady(googleMap: GoogleMap?) {
 }
}
```

O método `onMapReady` recebe um objeto do tipo `GoogleMap` , que é justamente a representação do

mapa já renderizado.

Agora podemos adicionar quantos marcadores quisermos chamando o método `addMarker` do objeto `GoogleMap` para cada novo marcador. Esse método recebe um objeto do tipo `MarkerOptions`, que é a representação de um marcador com todas as opções que quisermos configurar para ele. Para obter um objeto `MarkerOptions`, basta chamar seu construtor `default`:

```
val markerOptions = MarkerOptions()
googleMap?.addMarker(markerOptions)
```

## Customizando marcadores

A partir desse objeto chamamos métodos pra fazer as configurações, como `position`, para definir a posição do marcador. Para definir a posição, precisamos passar a latitude e longitude juntos com uma classe que representa essas duas informações, a `LatLng`. Criamos então um objeto `LatLng` com a latitude e a longitude específicas e passamos esse objeto para o método `position`:

```
val latitude = // pega latitude de algum lugar
val longitude = // pega longitude de algum lugar
markerOptions.position(LatLng(latitude, longitude))
```

Outra informação que podemos setar é um título para o marcador, com o método `title`:

```
markerOptions.title("Caelum")
```

Uma coisa interessante de se observar é que todos esses métodos de configuração devolvem o próprio objeto `MarkerOptions`, então dá pra chamar um método em sequência do outro:

```
markerOptions.position(LatLng(latitude, longitude)).title("Caelum")
```

## Outras maneiras de iterar por uma lista

Agora que sabemos criar, customizar e adicionar marcadores no mapa, precisamos iterar pelos `tweets` para fazer esse processo para cada `tweet`. A maneira que mais estamos acostumados é com a ideia de `for each`. A sintaxe do `for each` no Kotlin é a seguinte:

```
val tweets: List<Tweet> = // pega lista do servidor
for(tweet in tweets) {
 // cria, customiza marcador com dados do tweet e adiciona no mapa
}
```

Porém, o `for` no Kotlin só compila se a variável com a lista não puder ser nula. Então, se o valor dentro da variável `tweets` puder ser nulo, teremos que pedir pra só executar o `for` se o valor não for nulo:

```
val tweets: List<Tweet> = // pega lista do servidor
tweets?.let {
 for(tweet in tweets) {
 // cria, customiza marcador com dados do tweet e adiciona no mapa
 }
}
```

```
}
```

O código já começa a ficar com muitas chaves abrindo e fechando. Para reduzir o quanto escrevemos, podemos chamar o método `forEach` a partir de qualquer tipo que guarda vários elementos, apenas verificando com o operador `? se o valor da variável é nulo antes:`

```
val tweets: List<Tweet> = // pega lista do servidor
tweets?.forEach {
 // cria, customiza marcador com dados do tweet e adiciona no mapa
}
```

## 10.6 EXERCÍCIO: EXIBINDO NO MAPA OS TWEETS

### Objetivo

Adicionar um marcador no mapa para cada `tweet` que estiver no servidor.

### Passo a passo

1. Vamos colocar os pontos no mapa, para isso será necessário termos a lista de nosso viewmodel em nosso fragment:

```
class MapaFragment : SupportMapFragment() {

 private val tweetViewModel: TweetViewModel by lazy {
 ViewModelProviders.of(activity!!, ViewModelFactory).get(TweetViewModel::class.java)
 }
}
```

2. Agora que precisamos possuir o mapa para podermos manipula-lo:

```
class MapaFragment : SupportMapFragment() {

 override fun onResume() {
 super.onResume()
 getMapAsync(this)
 }
}
```

3. Devido ao `this`, será necessário implementarmos a interface `OnMapReadyCallback` em nosso classe, onde colocaremos um marcador dentro do mapa para cada `Tweet` :

```
class MapaFragment : SupportMapFragment(), OnMapReadyCallback {

 override fun onMapReady(googleMap: GoogleMap?) {

 val lista = tweetViewModel.tweets().value

 lista?.forEach { tweet ->

 val markerOptions = MarkerOptions()
 markerOptions.position(LatLng(tweet.latitude, tweet.longitude))
 markerOptions.title(tweet.dono.nome)

 googleMap?.addMarker(markerOptions)
 }
 }
}
```

```
 }
}
}
```

4. Rode o aplicativo e veja os pontos serem exibidos.