Unidade 3

Aula 1 - Componentes gráficos

A interface gráfica do seu *app* é tudo o que o usuário vai se importar. É por meio dela que ele vai interagir com o seu *app*. O SDK do Android possui uma variedade de componentes de UI (***U****ser* ***I****nterface* ou Interface do Usuário). Esses componentes de UI podem ser estruturas de *layout* ou controladores. O Android também provê módulos prontos de UI como *Dialogs, Notifications* e *Menus*. Agora você vai entrar em uma das partes mais importantes do curso: os componentes gráficos.

Para que você tenha uma ideia do tamanho da importância dos componentes gráficos, saiba que eles são nada mais, nada menos que tudo que o usuário pode ver na tela. Os componentes gráficos são itens de interface do usuário, e nada chama mais a atenção em um *app* do que um bom visual, pois trata-se do cartão de entrada para seu aplicativo.

## Visão geral

Os botões, textos, campos de entrada, barras de navegação e tudo o mais que você pode ver em uma tela de aplicativo fazem parte de um conjunto bem estruturado de objetos. Todos os elementos de UI são criados a partir de objetos *View* e *ViewGroup*. Esses objetos fazem parte do SDK do Android e estão nos níveis mais altos, se colocados em uma hierarquia, dos componentes gráficos.

Uma *View* é um objeto que desenha algo na tela com o qual o usuário do *app* pode interagir. Uma *ViewGroup* é um objeto que contém um agrupamento de outras *Views* (ou de *ViewGroups*).

O Android fornece uma coleção de subclasses *View* e *ViewGroup* que oferecem controles de entrada comuns — como botões e campos de texto — e vários modelos de *layout*, como um *LinearLayout* ou *RelativeLayout*.

### A hierarquia das *Views*

Em geral, a interface do usuário é definida por uma hierarquia de *Views* e *ViewGroups*. Cada *ViewGroup* é um recipiente ou camada invisível que agrega e organiza *Views* ou outras *ViewGroups* de nível inferior, e essas *Views* de nível inferior podem ser controles de entrada ou outros elementos que desenham a UI.



Figura - Hierarquia da relação entre Views e ViewGroups

Para definir essa hierarquia e começar a compor uma interface de usuário, basta instanciar objetos *View* no código e começar a criar uma árvore hierárquica, mas a maneira mais fácil para isso é utilizando um XML de *layout.*

Todas as subclasses de *View* do XML tem uma classe respectiva de mesmo nome no SDK do Android, ou seja, o *TextView* usado no código Java representa o *TextView* usado no *layout* XML.

A listagem a seguir mostra um *layout* definido que possui uma *ViewGroup* estrutural que agrega outros dois objetos de *View* de controle. Mais precisamente falando, têm-se o componente de estrutura chamado *LinearLayout* e, em um nível inferior a ele, dois componentes de controle: o *TextView* e o *Button*.

<?xml version="1.0" encoding="utf-8"?>  
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"  
 android:layout\_width="fill\_parent"  
 android:layout\_height="fill\_parent"  
 android:orientation="vertical" >  
 <TextView android:id="@+id/text"  
 android:layout\_width="wrap\_content"  
 android:layout\_height="wrap\_content"  
 android:text="Eu sou um texto" />  
 <Button android:id="@+id/button"  
 android:layout\_width="wrap\_content"  
 android:layout\_height="wrap\_content"  
 android:text="Eu sou um botão" />  
</LinearLayout>

Listagem 1 – Código XML para criação de um layout com um Texto e um Botão

O resultado da tela será o seguinte:

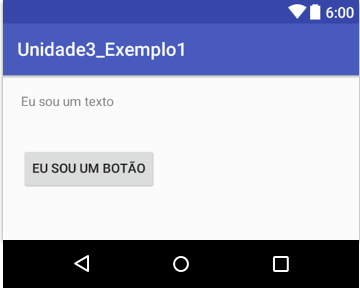


Figura - Imagem em altura reduzida do primeiro exemplo sobre XML

Nesta aula você entenderá melhor o que é um componente de estrutura de *layout* e componentes de controle, mas para você entender de vez como o *layout* é estruturado, você vai iniciar aprendendo sobre o que é um **Padrão de Projeto**.

## Padrão de projeto - *Composite*

Você já ouviu falar daquele termo: **“Não reinvente a roda”**? Pois é, na programação, principalmente na *mobile*, isso é uma lei sagrada.

Os recursos de um *smartphone* são bem mais limitados e escassos do que em um computador *desktop*. Em *mobile* há menos poder de processamento, menos poder gráfico, menos memória de armazenamento e menos memória de acesso randômico (RAM) se for comparar ao *desktop*, então gerenciar esses recursos é primordial.

Qualquer decisão errada tomada no desenvolvimento de um *app* pode acarretar em um funcionamento lento, alto consumo de memória e bateria, o que é problemático, pois o usuário ficaria frustrado com o uso do *app*. Também poderia acarretar em altos prejuízos caso ocorresse alguma falha técnica, por exemplo, um *app* de banco debitar na conta do cliente e não avisar que a transação foi bem-sucedida por causa de um travamento ou uma falha no Wi-Fi. Muitos desses problemas podem ser evitados se você **não tentar reinventar a roda**. Mas o que significa isso?

Os padrões de projeto foram criados para solucionar problemas comuns e corriqueiros no mundo do desenvolvimento de *software*, e conhecendo-os bem, ao se deparar com um problema de programação, você já saberá como resolver. Basta aplicar um padrão que o resolva.

Na aula seguinte será explicado mais a fundo o intuito dos padrões de projeto. Nesta aula você aprenderá o padrão ***Composite*** para entender como funciona a estrutura hierárquica das *Views* e *ViewGroups*.

### O padrão *Composite*

Todos os componentes de UI são *Views*, ou melhor, são objetos que herdam a classe *View*, conforme já visto. Além disso, eles também podem ser compostos por outras *Views*, por exemplo, um *Button* é uma *View* e ao mesmo tempo pode conter uma *View* dentro de sua estrutura. Veja a imagem a seguir:

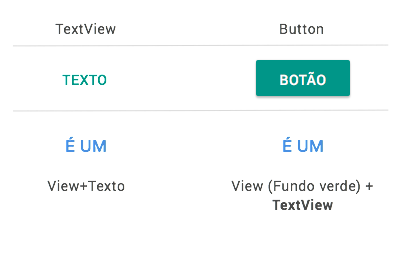


Figura - Diferenças e semelhanças entre um TextView e um Button

O que essa imagem indica é que um *TextView* é uma *View* que contém um texto. Do outro lado tem um Button, que nada mais é que uma *View* com fundo verde que contém um *TextView* dentro de si, que por sua vez também é uma *View*. Então o botão é uma *View* que contém uma *View* dentro.

Então, entendido o problema. Agora imagine que você foi contratado pela Google® para resolver um problema que acontece nas *Views* reportado por muitos usuários. Como você daria manutenção e corrigiria esse problema?

Conhecendo o padrão de projeto usado no desenvolvimento desses componentes você vai tirar de letra.

O comportamento dessas *Views* é descrito pelo padrão de projeto ***Composite*.** Agora você vai formalizá-lo.

#### Problema

Uma aplicação precisa manipular uma coleção de objetos “primitivos” e “compostos”. O processamento dos objetos primitivos é manipulado um a um e o processamento dos objetos compostos é manipulado diferentemente. Nesse caso, ter um “identificador de tipo” de cada objeto em uma tentativa de diferenciar seu processamento não é desejável.

#### Objetivo

O objetivo do *Composite* é:

* compor objetos em uma estrutura de árvore que representa uma hierarquia todo-parte.
* permitir que o usuário (objeto que o usa) trate objetos individuais e as composições desses objetos uniformemente.
* pode ser feita uma analogia com o sistema de pastas do seu computador: “uma pasta contém alguns itens, sendo que esses itens podem tratar-se de um documento qualquer ou também uma outra pasta”.

#### Discussão

Tecnicamente falando, para implementar o padrão *Composite*, define-se uma classe abstrata (chamada *Component*) que especifica o comportamento a ser exercido uniformemente por meio dos objetos primitivos e compostos. É necessário também fazer subclasses de *Component* que definam uma classe *Primitive* e uma *Composite.*

Use esse padrão sempre que você tiver “composições que contêm componentes, sendo que cada um deles pode ser uma composição”. Veja o diagrama:



Figura - Diagrama de classes do padrão **Composite**

Se você se lembrar da sua aula sobre diagramas de classe em UML no curso de POO, verá que o objeto *Composite* tem dois tipos de relação com o objeto *Component:* ***uma agregação e uma herança.*** Você se recorda do que são relações de agregação e de herança? Relembre:

Herança é um dos conceitos fundamentais da Orientação a Objetos. Com herança é possível reaproveitar as características e funcionalidades de uma classe em outra classe. Quando se define uma herança, todas as características e funcionalidades de uma classe são estendidas para a outra, desse modo a classe que faz a extensão se torna uma especialização da classe estendida.

Agregação é um conceito mais peculiar. Em uma relação entre dois objetos, a existência do objeto todo tem sentido independente da presença do objeto parte, sendo que o objeto parte está contido no objeto todo.

Dado isso, a figura ilustra composições que possuem componentes (agregação) que podem ser vistos pela analogia “*component* e *children*” em uma das relações entre *Component* e *Composite*. Ao mesmo tempo, os componentes podem ser uma composição também — pela relação de herança — e dessa forma se tem uma estrutura recursiva entre *Component* e *Composite*, visto que “um agrega o outro ao mesmo tempo em que um é o outro”. Entenda o que cada entidade representa:

***Component* (Componente)**

* É uma interface para os objetos (tanto parte como todo);
* Ajuda a gerenciar os objetos da hierarquia (adicionando, removendo etc.).

***Leaf* (Folha)**

* Representa objetos folha, ou seja, não possuem filhos em uma estrutura hierárquica;
* São os chamados objetos primitivos, mencionados anteriormente.

***Composite* (Composição)**

* São objetos compostos de componentes;
* Seus “filhos” podem ser outros objetos *Composite;*
* Tem estrutura recursiva.

***Client* (Cliente)**

* Não está no diagrama, mas representa os objetos que manipulam toda a estrutura;
* Chamará os métodos addElement() ou doThis(), como na nota do diagrama.

#### Exemplo

No Android, a classe *View* trabalha como o ***Component*,** enquanto o gerenciamento dos objetos “filhos” que é designado ao ***Composite*** é a classe *ViewGroup***.**

As classes ***Leaf,***  *Button***,** *ImageView***,** *TextView*,entre outras, são subclasses criadas diretamente da *View* (***Component****)* ou de subclasses dela, por exemplo, a classe *Button* deriva do *TextView* que por sua vez deriva diretamente da *View*. A classe ***Composite*** (*ViewGroup*) é deduzida estendendo diretamente uma *View* e implementando duas interfaces chamadas *ViewParent* — que define uma interface de um objeto composto — e *ViewManager* — que define a interface para adicionar ou remover componentes na árvore.

Como esperado, o método getParent(), que é necessário para se obter o “pai” do componente, é colocado na classe *View* (o ***Component***).

O objeto *ViewGroup* (***Composite***) tem um *array* que mantém seus objetos “filhos”.

Agora veja o diagrama de classes da versão simplificada das *View* e *Widgets* (os objetos folha — *Button*, *TextView* etc. — serão chamados de *Widget*s) do Android.

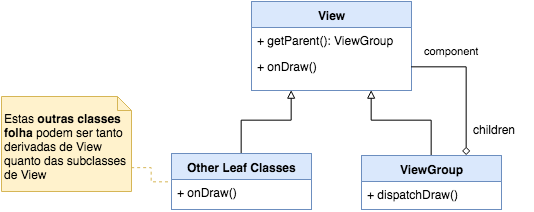


Figura - Estrutura das Views representada no padrão Composite

Agora serão feitas as últimas considerações. O diagrama de classe apresentado no começo desta discussão apresenta o método chamado operation(). Na implementação do Android, o método onDraw() na *View* (***Component****)* é o substituto para o operation(). O método dispatchDraw(), que é chamado quando o *Component* é desenhado, no objeto *ViewGroup* *(****Composite***) navega entre a lista de objetos e chama onDraw() em cada objeto “filho”.

Dessa maneira, pode-se finalmente dizer que *as Views e os Widgets* do Android são implementados utilizando o **Padrão de Projeto *Composite*.**

## Controles de entrada

Pode ser difícil se adaptar aos padrões de projeto no início, mas eles são de grande valia, uma vez que entendendo os padrões de projeto você estará apto a entender qualquer tipo de código e/ou melhorar a qualidade do seu. Agora que você viu como é feita a implementação das *Views* e dos *Widgets* do Android, serão detalhados alguns *Widgets*: os controles de entrada.

Controles de entrada são os componentes de UI passíveis de interação pelo usuário. O Android oferece uma ampla variedade de *widgets* que podem ser usados no seu *layout*, como botões, campos de texto, barras de busca, caixas de seleção, botões de *zoom*, botões de alternância e muito mais.

Adicionar um controle de entrada à UI é tão simples quanto adicionar um elemento XML ao *layout* XML. Por exemplo, a seguir apresenta-se um *layout* com um campo de texto e um botão:

<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"  
 android:layout\_width="fill\_parent"  
 android:layout\_height="fill\_parent"  
 android:orientation="vertical" >  
 <EditText  
 android:layout\_width="fill\_parent"  
 android:layout\_height="wrap\_content"  
 android:hint="Digite um texto"  
 android:layout\_margin="20dp"/>/>  
 <Button android:id="@+id/button"  
 android:layout\_width="fill\_parent"  
 android:layout\_height="wrap\_content"  
 android:text="Eu sou um botão"  
 android:layout\_margin="20dp"/>  
</LinearLayout>

Listagem 2 – Código XML com dois controles de entrada (um Button e um EditText)

Por enquanto, ignore os elementos de *layout* (*LinearLayout* nesse exemplo). Isso será explicado na próxima aula.

Veja o resultado desse XML:

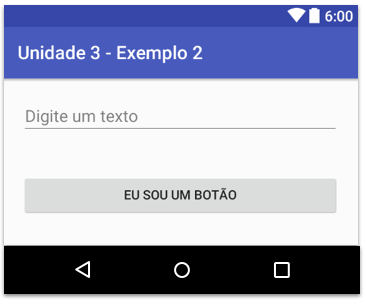


Figura - Tela simplificada com o resultado gerado pelo código da Listagem 2

Aqui há dois *Widgets* (ou *Views*, ou *Components*, ou objetos *Leaf*): o *Button* e o *EditText*.

Cada controle de entrada tem suporte para um conjunto específico de eventos de entrada, portanto, você pode tratar eventos como quando o usuário digita um texto ou toca em um botão.

### Controles comuns

A seguir há uma lista de alguns controles comuns que você pode usar no aplicativo.

|  |  |  |
| --- | --- | --- |
| **Botões** | | |
| 4.png |  | Um botão que pode ser pressionado pelo usuário para realizar uma ação. |
|  | *Button* |
|  |  |  |
| **Campos de texto** | | |
| 5.png |  | Um campo de texto editável. É possível usar o *widget* *AutoCompleteTextView* para criar uma entrada de texto que forneça sugestões para preenchimento automático. |
|  | *EditText, AutoCompleteTextView* |
|  |  |  |
| **Caixas de seleção** | | |
| 6.png |  | Uma chave liga/desliga que pode ser alternada pelo usuário. Use caixas de seleção ao apresentar aos usuários um grupo de opções selecionáveis que não sejam mutuamente exclusivas. |
|  | *CheckBox* |
|  |  |  |
| **Botões de opções** | | |
| 7.png |  | Similar às caixas de seleção, mas somente uma opção pode ser selecionada no grupo, ou seja, é mutuamente exclusiva. |
|  | *RadioGroup, RadioButton* |
|  |  |  |
| **Botões de alternância** | | |
| 8.png |  | Um botão liga/desliga, como um interruptor de luz. |
|  | *ToggleButton* |
|  |  |  |
| **Controle giratório** | | |
| 10.png |  | Uma lista que permite que os usuários escolham um valor dentro de um conjunto. |
|  | *Spinner* |
|  |  |  |
| **Seletores** | | |
| 9.png |  | Uma caixa de diálogo para que os usuários selecionem um valor para um conjunto usando botões. Seleção de data ou hora pode ser feita por meio deste *widget.* |
|  | *DatePicker,TimePicker* |

## Eventos de entrada

Cada controle de entrada não funciona sozinho. Não seria possível obter as interações do usuário sem os **eventos de entrada**.

No Android há uma maneira de interceptar eventos de interação de um usuário com o aplicativo. A classe *View* fornece meios para fazer isso.

Mas por que eventos de interação com o usuário devem ser interceptados?

Imagine uma tela de *login*, na qual o usuário precisa fornecer seu *e-mail* e senha. Uma prática comum é validar se os valores fornecidos nos campos estão corretos, por exemplo, o *e-mail* tem um formato particular que pode ser validado via *app* antes que se inicie uma tentativa de *login* em si.

Dentro dos vários *widgets* do Android, é possível notar vários métodos públicos de retorno de chamada que parecem ser úteis para os eventos de UI. Esses métodos têm nomes das ações que eles representam. Por exemplo, uma *View* (como um botão) possui o método chamado onTouch(), que é disparado no momento em que o usuário toca na respectiva *View*, no entanto, para interceptar esse evento, é necessário estender a classe e substituir o evento para cada ocasião. Isso seria pouco prático, não é? Por isso a classe *View* também contém uma coleção de interfaces aninhadas com retornos de chamadas que podem ser definidos com muito mais facilidade. Essas interfaces são chamadas de *Event Listeners* (escutas de evento) e serão utilizadas para interceptar os seus eventos.

## *Event listeners*

Antes de implementar um *listener*, serão listados os mais importantes.

onClick()

De interface OnClickListener, é chamado quando o usuário toca na *View*.

onLongClick()

De interface *OnLongClickListener*, é chamado quando o usuário toca e mantém a *View* pressionada.

onFocusChange()

De interface *OnFocusChangeListener*, é chamado quando o usuário navega de uma *View* para outra, usando as teclas de navegação ou o cursor de bola (*smartphones* com teclas).

onKey()

De interface *OnKeyListener*, é chamado quando o usuário está com foco no item ou solta uma tecla do dispositivo.

onTouch()

De interface *OnTouchListener*, é chamado quando o usuário realiza uma ação qualificada como um evento de toque, incluindo o pressionamento, a liberação ou qualquer outro gesto de movimento na tela (dentro dos limites do item).

onCreateContextMenu()

De interface *OnCreateContextMenuListener*, é chamado quando um menu de contexto está sendo construído (como resultado de um "clique longo").

**DICA:** Procure sobre os Menus na documentação do Android para mais informações.

Esses métodos são únicos de suas respectivas interfaces. Para definir um desses métodos e lidar com seus eventos, implemente a interface aninhada na *Activity* ou defina-a como uma classe anônima. Em seguida, passe uma instância da implementação para o respectivo método *View.set...Listener*(). Exemplo: chame setOnClickListener()e passe-o à implementação de *OnClickListener*.

O exemplo abaixo mostra como registrar uma escuta de clique para um botão.

*//Cria uma implementação anônima do evento OnClickListener*  
private OnClickListener myClickListener = new OnClickListener() {  
 public void onClick(View v) {  
 *// faz alguma coisa quando o botão é clicado*  
 }  
};  
  
protected void onCreate(Bundle savedValues) {  
  *...*  
 *// Captura o botão do seu layout*  
 Button button = (Button)findViewById(R.id.button);  
 *// Registra o onClick listener com a implementação abaixo*  
 button.setOnClickListener(myClickListener);  
 *...*  
}

Você também pode achar mais conveniente implementar *OnClickListener* como parte da *Activity*. Por exemplo:

public class LoginActivity extends Activity implements OnClickListener {  
 protected void onCreate(Bundle savedValues) {  
 *...*  
 Button button = (Button)findViewById(R.id.button);  
 button.setOnClickListener(this);  
 }  
  
 *// Implementação da interface OnClickListener*  
 public void onClick(View v) {  
 *// faz alguma coisa quando o botão é clicado*  
 }  
 *...*  
}

Para entender do que se trata uma implementação anônima, na próxima aula você aprenderá o padrão de projeto ***Decorator***.Vá agora para um exemplo prático.

Agora que você sabe compor uma tela com controles de entrada e manipular seus controles de evento, crie um exemplo simples e que se encontra na maioria dos aplicativos hoje em dia: uma tela de *login*.

Como ponto de partida, foi iniciado o projeto para você com algumas coisas prontas para agilizar o processo. O projeto iniciado está disponível no arquivo **Unidade\_3\_-\_Aula\_1\_-\_Exemplo\_1.zip.**

## *App MailList*

A partir desta aula, você começará a desenvolver um *app* profissional. Agora que aprendeu um pouco sobre os componentes gráficos, você tem total capacidade para iniciar esta jornada.

O *app* que você vai desenvolver é o ***MailList*.** Ele será uma versão bacana do Gmail, um *app* de *e-mail* que você deve conhecer.

No *MailList*, você terá uma tela de ***Login fake*** (existirá uma tela de *login*, mas a autenticação em si não será feita), uma tela **Entrada**, na qual aparecerá a lista de *e-mails* recebidos, uma tela **Escrever**, que é destinada à composição de *e-mails*, e um menu lateral, com diversas opções para você escolher e posteriormente incrementar seu *app*.

### Tela *login*

**O que você já tem pronto?**

Se você se lembrar das suas aulas de POO, nas quais falou-se de MVC (*Model View Controller*), vai se lembrar de que sempre se modelavam os *models* antes do início do desenvolvimento do aplicativo. Nesse caso, a camada foi modelada e deixada pronta para você. Primeiro, veja o diagrama:

User.png

Figura - Diagrama de classe da Classe User

Para esse exemplo foi criado um único *model*, que não se relacionará com nenhum outro *model*. Esse é o *User*, e ele possui apenas os atributos *e-mail* e *password.*

O motivo de se utilizar essa classe é o método isValid(,), que verificará se o e-mail e a senha são válidos ou não. Entre no arquivo **User.java** e leia o código para que você tenha um melhor entendimento.

Para dar início a este projeto, certifique-se de que a **API 22** já está instalada (ela foi instalada na unidade 2, lembra? Mas confira se está mesmo instalada), pois foi a API o alvo para este projeto. Se o seu *smartphone* tem uma versão do Android menor que a 5.1, mude o valor do item minSdkVersion do arquivo **build.app (Module:app)** com o número da API desejada. Sincronize o *Gradle* e rode o *app*.

Para instalar alguma API diferente da API 22, basta seguir os passos dados na Unidade 2 sobre a configuração do Android Studio.

Foi deixado pronto também um método que mostrará um alerta na tela. Esse método encontra-se na classe **LoginActivity**, mas não se preocupe com o conteúdo dele, pois serão abordados os **diálogos de alerta** nas aulas seguintes.

Também foram inseridas neste projeto inicial algumas *strings* predefinidas e algumas medidas no arquivo dimens.xml. O arquivo *dimens* é um *resource* e está no mesmo diretório que o strings.xml. Nele, em vez de se inserir textos, devem ser inseridas medidas que se deseja reutilizar em mais de um *widget*. Enfim, siga para o passo a passo.

**DICA**: R***esource****,* traduzido do inglês, significa recurso. Em Android, um recurso é tudo o que foge da linguagem Java. O seu *layout* XML é um recurso, os arquivos de *strings* são recursos, o arquivo *dimens* é um recurso e as imagens que serão colocadas no *app* também são.

OBS.: A unidade utilizada para espaçamentos e tamanhos mais apropriada é a **dp**. Utilize sempre medidas em **dp** nos seus xmls.

Com o projeto inicial aberto, primeiro você vai compor o *layout*:

1. Abra o arquivo **activity\_login.xml**. O objetivo é a seguinte tela:

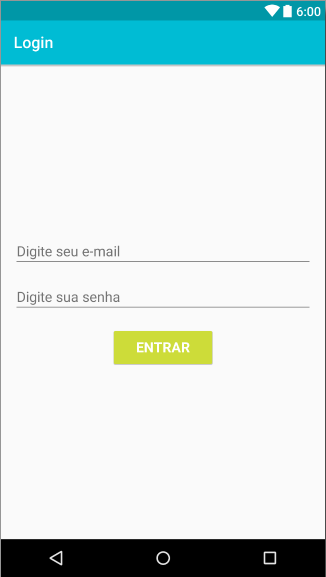


Figura - Resultado esperado para a tela Login

1. Seu *layout* já está predefinido. Em vez de utilizar o <RelativeLayout> que é o padrão, você utilizará o <LinearLayout> para que os *widgets* se empilhem um sobre o outro. O código inicial é o seguinte:

<?xml version="1.0" encoding="utf-8"?>  
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"  
 xmlns:tools="http://schemas.android.com/tools"  
 android:layout\_width="match\_parent"  
 android:layout\_height="wrap\_content"  
 android:padding="@dimen/activity\_general\_margin"  
 tools:context=".UI.LoginActivity"  
 android:orientation="vertical"  
 android:layout\_gravity="center\_vertical">  
</LinearLayout>

Listagem 3 – Configurando o LinearLayout

**Entendendo o código:** no atributo layout\_width foi utilizado o valor "match\_parent" para que a *ViewGroup* (<LinearLayout>) tenha a mesma largura que a tela (sua parente). Em layout\_height utilizou-se o valor "wrap\_content" para que a altura da <LinearLayout> corresponda à soma das alturas de suas *subviews*. Para o conteúdo não ficar grudado na borda da tela, foi utilizada a propriedade padding e dado um espaçamento geral. Para que as *subviews* contidas no <LinearLayout> se alinhem uma abaixo da outra, como se estivessem empilhadas, modificou-se o atributo orientation para "vertical". Como você pode ver na imagem “Resultado esperado para a tela *Login*”, o conteúdo está concentrado no centro da tela. Para isso, foi configurado o atributo layout\_margin para "center\_vertical".

1. Dentro do <LinearLayout> insira o <AutoCompleteTextView> para seu campo de *e-mail*:

<AutoCompleteTextView  
 android:id="@+id/email"  
 android:layout\_width="match\_parent"  
 android:layout\_height="wrap\_content"  
 android:hint="@string/prompt\_email"  
 android:inputType="textEmailAddress"  
 android:maxLines="1"  
 android:singleLine="true"  
 android:fontFamily="@string/abc\_font\_family\_caption\_material" />

Listagem 4 – Inserindo e configurando o AutoCompleteTextView

**Entendendo o código:** utilizando o <AutoCompleteTextView>, há sugestões de correções ortográficas no momento em que se preenche o campo. A lógica de layout\_width e layout\_height é a mesma do <LinearLayout>. O campo hint representa o texto de ajuda que fica no campo enquanto ele está vazio e, neste caso, estará escrito "Digite seu e-mail". O campo inputType indica que o campo receberá um *e-mail*, e isso faz com que o *layout* do teclado mude. Os campos maxLines e singleLine limitam o número de linhas que o <AutoCompleteTextView> aceitará. Por fim, o campo fontFamily troca a fonte do texto.

1. Abaixo insira um <EditText> para seu campo de senha.

<EditText  
 android:id="@+id/password"  
 android:layout\_width="match\_parent"  
 android:layout\_height="wrap\_content"  
 android:hint="@string/prompt\_password"  
 android:inputType="textPassword"  
 android:maxLines="1"  
 android:layout\_marginTop="@dimen/activity\_general\_margin"  
 android:singleLine="true"  
 android:fontFamily="@string/abc\_font\_family\_caption\_material" />

Listagem 5 – Inserindo e configurando o EditText

**Entendendo o código:** desta vez o inputType é "textPassword" e isso faz com que os caracteres ocultem-se no campo. O restante dos campos representam espaçamento e formatação, como visto nos elementos anteriores.

1. Por fim, insira um <Button> que receberá um *event listener* futuramente.

<Button  
 android:id="@+id/email\_sign\_in\_button"  
 style="?android:textAppearanceMedium"  
 android:layout\_width="wrap\_content"  
 android:layout\_height="wrap\_content"  
 android:layout\_marginTop="@dimen/activity\_general\_margin"  
 android:paddingLeft="32dp"  
 android:paddingRight="32dp"  
 android:paddingTop="@dimen/activity\_general\_margin"  
 android:paddingBottom="@dimen/activity\_general\_margin"  
 android:textStyle="bold"  
 android:layout\_gravity="center\_horizontal"  
 android:backgroundTint="@color/colorAccent"  
 android:textColor="#FFF"  
 android:text="@string/action\_sign\_in"/>

Listagem 6 – Inserindo e configurando o Button

**Entendendo o código:** o <Button> também possui uma série de atributos puramente para definição do *layout*.

**DICA:** Para saber o que cada um desses elementos representa, pesquise na documentação do Android.

1. A **activity\_login.xml** deverá corresponder exatamente à listagem a seguir:

<?xml version="1.0" encoding="utf-8"?>  
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"  
 xmlns:tools="http://schemas.android.com/tools"  
 android:layout\_width="match\_parent"  
 android:layout\_height="wrap\_content"  
 android:padding="@dimen/activity\_general\_margin"  
 tools:context=".UI.LoginActivity"  
 android:orientation="vertical"  
 android:layout\_gravity="center\_vertical">  
  
 <AutoCompleteTextView  
 android:id="@+id/email"  
 android:layout\_width="match\_parent"  
 android:layout\_height="wrap\_content"  
 android:hint="@string/prompt\_email"  
 android:inputType="textEmailAddress"  
 android:maxLines="1"  
 android:singleLine="true"  
 android:fontFamily="@string/abc\_font\_family\_caption\_material" />  
  
 <EditText  
 android:id="@+id/password"  
 android:layout\_width="match\_parent"  
 android:layout\_height="wrap\_content"  
 android:hint="@string/prompt\_password"  
 android:inputType="textPassword"  
 android:maxLines="1"  
 android:layout\_marginTop="@dimen/activity\_general\_margin"  
 android:singleLine="true"  
 android:fontFamily="@string/abc\_font\_family\_caption\_material" />  
  
 <Button  
 android:id="@+id/email\_sign\_in\_button"  
 style="?android:textAppearanceMedium"  
 android:layout\_width="wrap\_content"  
 android:layout\_height="wrap\_content"  
 android:layout\_marginTop="@dimen/activity\_general\_margin"  
 android:paddingLeft="32dp"  
 android:paddingRight="32dp"  
 android:paddingTop="@dimen/activity\_general\_margin"  
 android:paddingBottom="@dimen/activity\_general\_margin"  
 android:textStyle="bold"  
 android:layout\_gravity="center\_horizontal"  
 android:backgroundTint="@color/colorAccent"  
 android:textColor="#FFF"  
 android:text="@string/action\_sign\_in"/>  
</LinearLayout>

Listagem 7 – Código de **activity\_login.xml** completo

Você tem o seu *layout* pronto. Veja como ele ficou:

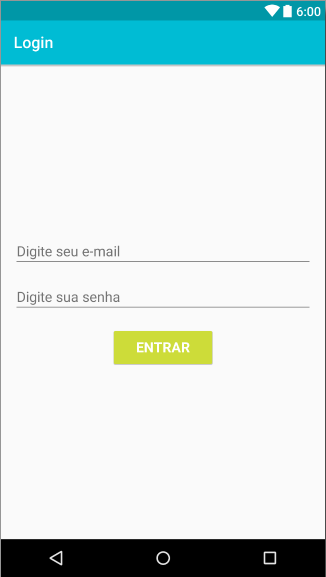


Figura - Tela de Login pronta

A próxima etapa é dar vida a ele. Abra o arquivo LoginActivity.java e siga estes últimos passos:

* Crie 3 atributos que armazenarão seu campo de e-mail, de senha e botão. Veja como:

public class LoginActivity extends AppCompatActivity {  
 private Button loginButton;  
 private EditText passwordEditText;  
 private AutoCompleteTextView emailAutoCompleteTextView;  
}

Listagem 8 – Definindo atributos em LoginActivity

* Como nos exemplos das aulas anteriores, é preciso capturar as *Views* por meio da classe R. Para isso, localize o método onCreate() e insira as seguintes linhas de código:

@Override  
 protected void onCreate(Bundle savedInstanceState) {  
 super.onCreate(savedInstanceState);  
 setContentView(R.layout.activity\_login);  
  
 loginButton = (Button)findViewById(R.id.email\_sign\_in\_button);  
 passwordEditText = (EditText)findViewById(R.id.password);  
 emailAutoCompleteTextView = (AutoCompleteTextView)findViewById(R.id.email);  
  
 }

Listagem 9 – Relacionando os elementos do XML (activity\_login) aos atributos da Listagem 8

* Agora, finalmente será implementado seu primeiro *EventListener:* o clique no botão de *login*. Existem duas maneiras de implementar o *EventListener* (onClick()): criar uma implementação anônima ou implementar a interface *OnClickListener*. Como tem apenas um botão, foi escolhida a segunda opção, então faça com que sua *Activity* implemente a interface *OnClickListener*.

public class LoginActivity extends AppCompatActivity implements View.OnClickListener {

…

}

Listagem 10 – Fazendo com que a classe LoginActivity implemente o OnClickListener

* Ao realizar o passo anterior, você é obrigado a implementar os métodos da interface e, como você já sabe, as interfaces de *EventListeners* possuem apenas um método a ser implementado. No caso da interface *OnClickListener*, é preciso implementar o método onClick(). Veja abaixo:

@Override  
 public void onClick(View v) {  
 User user = new User(emailAutoCompleteTextView.getText(), passwordEditText.getText());  
  
 if (user.isValid()){  
 showAlert("Logado com sucesso! ");  
 }else{  
 showAlert("E-mail ou senha inválidos.");  
 }  
 }

Listagem 11 – Validando os dados inseridos nos campos de e-mail e senha

* Por fim, atribua o *EventListener,* que agora é a própria Activity, ao seu botão.

@Override  
 protected void onCreate(Bundle savedInstanceState) {  
 super.onCreate(savedInstanceState);  
 setContentView(R.layout.activity\_login);  
  
 loginButton = (Button)findViewById(R.id.email\_sign\_in\_button);  
 passwordEditText = (EditText)findViewById(R.id.password);  
 emailAutoCompleteTextView = (AutoCompleteTextView)findViewById(R.id.email);  
  
 loginButton.setOnClickListener(this);  
 }

Listagem 12 – Especificando no botão quem é o seu OnClickListener

Sua escuta de evento está finalizada. Essa é uma lógica simples de *Login* que se aplica à maioria dos projetos. Aproveite e brinque bastante com a criação do *layout*. É muito importante que você explore e teste os diversos tipos de atributos e *widgets* para aprimorar seu conhecimento. Crie e divirta-se.

## Resumo

Nesta aula entendemos como funciona a hierarquia das Views e ViewGroups através do padrão Composite. Aprendemos também o que são os componentes de entrada e seus eventos. Por fim, colocamos tudo isto em prática criando a primeira tela do nosso aplicativo MailList.

## Exercícios

## TDP

## 

# Aula 2

Aula 2 – *Layouts*

Na aula anterior, você viu uma breve introdução à criação de *layouts* no Android Studio. Você pôde explorar e entender a estrutura em que as *Views* e *ViewGroups* se organizam por meio do padrão de projeto ***Composite*.** Agora você vai aprender com mais profundidade o funcionamento dos *Layouts*.

Como já é sabido, o *layout* define a estrutura visual do *app*. É possível defini-lo de duas maneiras:

* **declarar os elementos de UI via XML**: o Android oferece um vocabulário XML que corresponde às classes e subclasses de *View*, como as de *widgets* e *layouts*.
* **instanciar e manipular os elementos do *layout* em tempo de execução**: você pode criar objetos *View* e *ViewGroup* e manipulá-los **por meio de programação** (Java).

A notícia boa é que é possível utilizar as duas maneiras ao mesmo tempo. No arquivo XML é possível criar o estado inicial das telas, ou seja, como elas devem ser a princípio. Com Java, você pode dar vida a essas telas, dando a possibilidade ao programador de trocar textos, manipular eventos de clique, navegação entre telas, animações e etc.

A vantagem em priorizar a declaração da UI no XML é que se pode separar melhor a **apresentação do aplicativo** do código que **controla seu comportamento**. Além disso, a declaração de *layout* em XML facilita a depuração de problemas de UI. Então nesta aula você vai se concentrar em aprender a declarar o *layout* em XML.

Como você pode ter percebido, os nomes dos elementos do XML (<Button>, <EditText>) correspondem aos nomes das classes e as propriedades correspondem aos métodos delas. Isso se torna tão intuitivo que é possível supor qual propriedade de um elemento XML corresponde a determinado método da classe ou qual classe corresponde a certo elemento XML. Por exemplo, o elemento *EditText* tem o atributo *text* que corresponde ao EditText.setText() (para atribuir um texto) ou EditText.getText() (para obter o texto atribuído). Viu como é fácil supor? São poucas as diferenças.

## *Layouts* comuns

Como visto anteriormente, uma *ViewGroup* é um agrupamento de *Views*. Cada subclasse da *ViewGroup* fornece um modo exclusivo de exibir as vistas aninhadas dentro dela. Veja agora alguns dos tipos mais comuns criados no Android.

Pelo padrão *Composite*, provou-se que é possível aninhar um ou mais tipos de *layouts* (*ViewGroups*) em outro *layout* para obter a aparência desejada, mas o mais adequado é manter uma hierarquia de *layouts* o menos profunda possível. Quanto menos *layouts* aninhados, haverá menos complexidade no código e sua tela carregará mais rápido.

***LinearLayout***: *layout* que organiza as *Views* filhas em uma única linha horizontal ou vertical. Ele cria uma barra de rolagem se o comprimento da janela exceder o comprimento da tela.

***RelativeLayout*:** permite especificar a localização de objetos filhos relativos entre si (exemplo: filho A à esquerda do filho B) ou relativos aos pais (exemplo: alinhados no topo do pai).

***WebView***: exibe páginas da *web*.

### *LinearLayout*

*LinearLayout* é uma *ViewGroup* que alinha todas as *subviews* em uma direção única, verticalmente ou horizontalmente. Você pode especificar a direção do *layout* utilizando o atributo *orientation*.



Figura - Representação gráfica utilizada pela Google de um LinearLayout.

Uma *subview* de um *LinearLayout* é empilhada uma depois da outra, então se você estiver em um *LinearLayout* com orientação vertical, terá uma *View* por linha, caso esteja na orientação horizontal, tem uma *View* por coluna. Um *LinearLayout* respeitará também as margens entre as *subviews* e a gravitação (*grativity*) de cada uma delas.

#### *Layout Weight*

No *LinearLayout* também podem ser definidos pesos para cada *subview* com o atributo layout\_weight. Esse atributo dita a prioridade do quanto de espaço o elemento deve ocupar na tela. *Subviews* podem especificar um peso, e todo espaço restante na *ViewGroup* é ocupado por essas *subviews* proporcionalmente aos respectivos pesos. Por padrão eles serão “0dp”.

Por exemplo, se há três *TextViews* e dois deles tem peso “1” enquanto o outro não tem peso definido. O terceiro TextView, sem peso definido, ocupará apenas o tamanho do seu conteúdo ou o espaço que você definir. Os outros dois *TextViews* preencherão o espaço restante e dividirão esse espaço igualmente entre eles.

### *RelativeLayout*

Este *layout* é um pouco diferente do anterior. Ele mostra as *subviews* em posições relativas. A posição de cada *View* pode ser especificada como relativa à View irmã (por exemplo, à esquerda de ou abaixo de alguma *View*) ou como posição relativa ao *RelativeLayout* mãe (será chamada de *parent*).



Figura - Representação gráfica utilizada pela Google de um RelativeLayout.

Um *RelativeLayout* é realmente poderoso para criação de interfaces mais complexas, pois ele elimina o aninhamento das *Views* e as mantém *flat,* ou seja, mantém uma estrutura de *layout* menos profunda e melhora a *performance* do *app*. Praticamente tudo que você pode compor usando um *LinearLayout* você poderia substituir pelo *RelativeLayout*, a única desvantagem quanto a isso é que a complexidade da criação dessas interfaces aumenta consideravelmente.

#### Posicionamento

O *RelativeLayout* oferece alguns atributos que auxiliam no posicionamento das *Views*.

layout\_alignParentTop

Se "*true*", faz com que o topo da *View* se encoste no topo da *View* *parent*.

layout\_centerVertical

Se "*true*", faz com que a *View* fique no centro da *View* *parent.*

layout\_below

Posiciona a *View* encostando seu topo na parte de baixo da *View* irmã.

layout\_toRightOf

Posiciona a *View* encostando seu lado esquerdo no lado direito da *View* irmã.

Os valores para esses atributos podem ser um booleano ou um ID. Usando um booleano, a *interface builder* tentará posicionar automaticamente as *Views*, mas utilizando o ID, é possível explicitar qual *View* é relativa à outra.

## A Biblioteca ButterKnife

Agora que você aprendeu a criar seus *layouts*, você entrará no mundo das bibliotecas externas. Lembra-se de que foi dito que o *Gradle* serve como gerenciador de dependências? Pois bem, ele será utilizado agora.

Para fechar esta sessão sobre os *layouts,* será apresentada uma biblioteca que facilitará a referenciação dos elementos de *layout* no código.

Em programação, o ato de acessar um elemento de *layout* por meio do *controller* (no seu caso, a *Activity*) é chamado de *Binding*. Você deve ter sentido que referenciar os *widgets* e implementar os *EventListeners* é um tanto entediante e por vezes percebe-se que o código torna-se poluído quando há um *layout* mais complexo.

Para lhe convencer que a utilização de bibliotecas de terceiros é bastante produtiva, será apresentada a biblioteca ButterKnife.

**DICA**: Em geral, na programação, tem-se o costume de reaproveitar alguns pedaços de *softwares* bem-sucedidos distribuindo-os como componentes. Existe outro grupo grande de programadores que se dedica a criar esses componentes, sem necessariamente ter um vínculo com um *software*. Esses componentes são chamados de **bibliotecas**, ou ***frameworks***, e são feitos para ajudar o trabalho de outras pessoas cumprindo a premissa principal da programação orientada a objetos: o reuso.

Criada por Jakew Harton, a ButterKnife é capaz de facilitar todos esses procedimentos de pegar referência de *View* e *resources* ou até mesmo implementar um *listener*.  
  
Mas como é possível adicioná-la no seu projeto?

Para adicionar a biblioteca ButterKnife siga os seguintes passos:

1. Esteja certo de que você utilizará a última versão da biblioteca. Para isso visite o *site* oficial: <http://jakewharton.github.io/butterknife/>. Esse site possui instruções de instalação ao final da página.
2. Copie e cole ambas as linhas. Agora abra o projeto ***Login*.**
3. Entre no arquivo **build.gradle (Module: app)** e cole o trecho copiado ao final do item *dependencies*:

dependencies {

compile fileTree(**dir**: **'libs'**, **include**: [**'\*.jar'**])

testCompile **'junit:junit:4.12'**

compile **'com.android.support:appcompat-v7:24.2.0'**

*// ButterKnife*

compile **'com.jakewharton:butterknife:8.4.0'**

apt **'com.jakewharton:butterknife-compiler:8.4.0'**

}

Listagem 13 – Adicionando a biblioteca ButterKnife com o arquivo **build.gradle (Module: app)**

1. Fora do item *dependencies*, adicione o seguinte comando:

apply **plugin**: **'android-apt'**

1. Seu arquivo build.gradle (Module: app) deverá estar parecido com a seguinte listagem:

apply **plugin**: **'com.android.application'**

apply **plugin**: **'android-apt'**

android {

compileSdkVersion 24

buildToolsVersion **"24.0.2"**

defaultConfig {

applicationId **"br.com.pearson.login"**

minSdkVersion 22

targetSdkVersion 24

versionCode 1

versionName **"1.0"**

}

buildTypes {

release {

minifyEnabled **false**

proguardFilesgetDefaultProguardFile(**'proguard-android.txt'**), **'proguard-rules.pro'**

}

debug {

debuggable **true**

}

}

}

dependencies {

compile fileTree(**dir**: **'libs'**, **include**: [**'\*.jar'**])

testCompile **'junit:junit:4.12'**

compile **'com.android.support:appcompat-v7:24.2.0'**

*// ButterKnife*

compile **'com.jakewharton:butterknife:8.4.0'**

apt **'com.jakewharton:butterknife-compiler:8.4.0'**

}

Listagem 14 – Arquivo **build.gradle (Module: app)** completo

1. Agora entre no **build.gradle (Module: Login)** e insira o seguinte comando ao final do item *dependencies*:

classpath **'com.neenbedankt.gradle.plugins:android-apt:1.8'**

Listagem 15 – Adicionando componentes necessários pelo arquivo **build.gradle (Module: Login)**

1. Seu código **build.gradle (Module: Login)** deverá estar parecido com este:

buildscript {

repositories {

jcenter()

}

dependencies {

classpath **'com.android.tools.build:gradle:2.1.3'**

classpath **'com.neenbedankt.gradle.plugins:android-apt:1.8'**

}

}

allprojects {

repositories {

jcenter()

}

}

task clean(**type**: Delete) {

delete rootProject.buildDir

}

Listagem 16 – Arquivo **build.gradle (Module: Login)** completo

1. Sincronize seu *Gradle* e, pronto, a biblioteca já pode ser utilizada.

Agora veja como realmente a ButterKnife funciona.

Foi criado um ponto de partida para este exemplo prático. Descompacte o arquivo **Unidade\_3\_-\_Aula\_2\_-\_Exemplo\_1.zip** , abra o projeto contido nele e, no seu Android Studio, entre no arquivo **activity\_login.xml**:

<?xml version="1.0" encoding="utf-8"?>  
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"  
 xmlns:tools="http://schemas.android.com/tools"  
 android:layout\_width="match\_parent"  
 android:layout\_height="match\_parent"  
 android:padding="@dimen/activity\_general\_margin"  
 tools:context=".UI.LoginActivity"  
 android:orientation="vertical">  
  
 <TextView  
 android:layout\_width="match\_parent"  
 android:layout\_height="0dp"  
 android:layout\_weight="2"  
 android:gravity="center\_vertical"  
 android:text="Login"  
 android:textAlignment="center"  
 android:fontFamily="@string/abc\_font\_family\_title\_material"  
 android:forceHasOverlappingRendering="false"  
 android:textSize="36dp"  
 android:textIsSelectable="true" />  
  
 <AutoCompleteTextView  
 android:id="@+id/email"  
 android:layout\_width="match\_parent"  
 android:layout\_height="wrap\_content"  
 android:hint="@string/prompt\_email"  
 android:inputType="textEmailAddress"  
 android:maxLines="1"  
 android:singleLine="true"  
 android:fontFamily="@string/abc\_font\_family\_caption\_material" />  
  
 <EditText  
 android:id="@+id/password"  
 android:layout\_width="match\_parent"  
 android:layout\_height="wrap\_content"  
 android:hint="@string/prompt\_password"  
 android:inputType="textPassword"  
 android:maxLines="1"  
 android:layout\_marginTop="@dimen/activity\_general\_margin"  
 android:singleLine="true"  
 android:fontFamily="@string/abc\_font\_family\_caption\_material" />  
  
 <Button  
 android:id="@+id/email\_sign\_in\_button"  
 style="?android:textAppearanceMedium"  
 android:layout\_width="wrap\_content"  
 android:layout\_height="wrap\_content"  
 android:layout\_marginTop="@dimen/activity\_general\_margin"  
 android:text="@string/action\_sign\_in"  
 android:paddingLeft="32dp"  
 android:paddingRight="32dp"  
 android:paddingTop="@dimen/activity\_general\_margin"  
 android:paddingBottom="@dimen/activity\_general\_margin"  
 android:textStyle="bold"  
 android:layout\_gravity="center\_horizontal"  
 android:backgroundTint="@color/colorAccent"  
 android:textColor="#ddffffff"/>  
  
 <Space  
 android:layout\_width="match\_parent"  
 android:layout\_height="0dp"  
 android:layout\_weight="1"/>  
  
</LinearLayout>

Listagem 17 – Novo arquivo **activity\_login.xml** completo

Primeiro foi dado um acabamento melhor nas *views* que já existiam e, para exemplificar o *LinearLayout* com seu atributo layout\_weight, foram inseridos dois novos elementos:

* Um <TextView> no topo da tela, apenas com a palavra "Login":

<TextView  
 android:layout\_width="match\_parent"  
 android:layout\_height="0dp"  
 android:layout\_weight="0.7"  
 android:gravity="center\_vertical"  
 android:text="Login"  
 android:textAlignment="center"  
 android:fontFamily="@string/abc\_font\_family\_title\_material"  
 android:forceHasOverlappingRendering="false"  
 android:textSize="36dp"  
 android:textIsSelectable="true" />

Listagem 18 – Inserindo um novo texto no topo da tela

* Um elemento <Space> no rodapé da página. Esse elemento é utilizado quando se deseja ocupar uma lacuna da tela.

<Space  
 android:layout\_width="match\_parent"  
 android:layout\_height="0dp"  
 android:layout\_weight="0.3"/>

Listagem 19 – Inserindo um espaço em branco para ajustar o layout

Veja que ambos possuem o atributo layout\_weight e uma particularidade quando se utiliza esse atributo é que se a *orientation* do *LinearLayout* é vertical, o layout\_height do elemento deve ser "0dp", e quando a *orientation* é horizontal, o layout\_width do elemento é quem deve ter o valor "0dp". Isso porque é o atributo de peso que assumirá a responsabilidade de configurar a altura ou a largura do elemento, nesse caso.

A melhor maneira de se distribuir o peso é formando uma soma de até "1". Como há dois elementos sendo coordenados pelos pesos, definiu-se que o <TextView> terá o peso "0.7" e o <Space> terá o peso "0.3", que representa 30% e 70% das lacunas respectivamente. Sendo assim, seu *layout* assumirá esta forma:

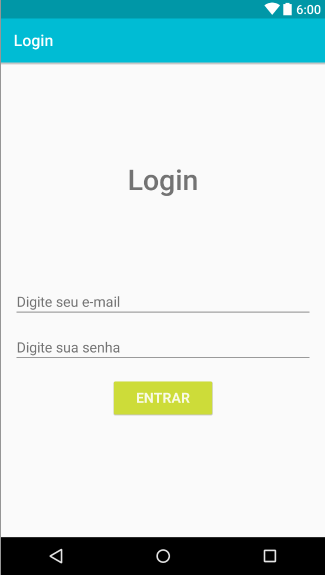


Figura - Segunda versão da tela de Login

Antes de implementar o ButterKnife, veja que foi criada também uma segunda *Activity* com mais exemplos de *LinearLayout* para você explorar. Ela tenta reproduzir uma tela de composição de *e-mail*, mas não possui nenhuma funcionalidade. Essa tela trata-se da **ComposeActivity.java** com seu *layout* **activity\_compose.xml**. Fique livre para modificá-la, adicionar um *Model*, validações e etc. Veja como ela é:



Figura - Tela escrever e-mail

### Usando a ButterKnife

Agora que você deixou sua tela mais bonita, é hora de limpar o código utilizando a nova biblioteca.

1. Na declaração das *Views,* substitua isto:

private AutoCompleteTextView emailAutoCompleteTextView;

private EditText passwordEditText;

por isto:

@BindView(R.id.email) AutoCompleteTextView emailAutoCompleteTextView;

@BindView(R.id.password) EditText passwordEditText;

Listagem 20 – Declaração de atributos com ButterKnife

**Entendendo o código:** veja que a ButterKnife utiliza *Annotations*. *Annotations* é uma forma avançada de programação que pode modificar a sintaxe do código adicionando extensões. Neste caso, a Annotation @BindView realiza uma série de coisas internamente, como referenciar a *View*, fazer *cast* para o tipo apropriado e etc.

DICA: Quando se fala em *cast****,*** refere-se à conversão de um objeto de uma classe para outra.

1. No método onCreate() não é preciso mais utilizar o findViewById. Remova-o.
2. Ainda no mesmo método, insira o seguinte comando:

ButterKnife.bind(this);

1. Substitua o método onClick() por:

@OnClick(R.id.email\_sign\_in\_button)  
 public void submit() {  
  
 User user = new User(emailAutoCompleteTextView.getText(), passwordEditText.getText());  
  
 if (!user.isValidEmail()){  
 emailAutoCompleteTextView.setError("E-mail inválido");  
 }  
  
 if (!user.isValidPassword()){  
 passwordEditText.setError("Senha inválida. Sua senha deve conter pelo menos 4 caracteres.");  
 }  
  
 if (user.isValid()) {  
 *// navega para a próxima tela*  
 startNextActivity();  
 }  
 }

Listagem 21 – Nova forma de definir Listeners utilizando a ButterKnife

1. Agora adicione o método fieldFocusChanged(). Esse método possibilita pegar o evento de quando o *EditText* perde ou ganha o foco, assim você vai validar o *e-mail* e a senha quando o respectivo campo perde o foco. Veja:

@OnFocusChange({R.id.email, R.id.password})  
 public void fieldFocusChanged(View v, boolean hasFocus) {  
  
 User user = new User(emailAutoCompleteTextView.getText(), passwordEditText.getText());  
  
 if (!hasFocus) {  
 if (v.getClass() == emailAutoCompleteTextView.getClass()) {  
 if (!user.isValidEmail()){  
 emailAutoCompleteTextView.setError("E-mail inválido");  
 }  
 }else{  
 if (!user.isValidPassword()){  
 passwordEditText.setError("Senha inválida. Sua senha deve conter pelo menos 4 caracteres.");  
 }  
 }  
 }  
 }

Listagem 22 – Detectando quando um campo muda de foco e validando-o

Já que foram substituídos os métodos acima, remova a declaração de implementação das interfaces *EventListeners*:

public class LoginActivity extends AppCompatActivity ~~implements View.OnClickListener~~ {

…

}

Listagem 23 – Remove a interface OnClickListener, pois ela não é mais útil aqui

O código completo da sua *Activity* é este:

|  |
| --- |
| public class LoginActivity extends AppCompatActivity {   @BindView(R.id.email) AutoCompleteTextView emailAutoCompleteTextView;  @BindView(R.id.password) EditText passwordEditText;   @Override  protected void onCreate(Bundle savedInstanceState) {  super.onCreate(savedInstanceState);  setContentView(R.layout.activity\_login);  ButterKnife.bind(this);  }   @OnFocusChange({R.id.email, R.id.password})  public void fieldFocusChanged(View v, boolean hasFocus) {   User user = new User(emailAutoCompleteTextView.getText(), passwordEditText.getText());   if (!hasFocus) {  if (v.getClass() == emailAutoCompleteTextView.getClass()) {  if (!user.isValidEmail()){  emailAutoCompleteTextView.setError("E-mail inválido");  }  }else{  if (!user.isValidPassword()){  passwordEditText.setError("Senha inválida. Sua senha deve conter pelo menos 4 caracteres.");  }  }  }  }   @OnClick(R.id.email\_sign\_in\_button)  public void submit() {   User user = new User(emailAutoCompleteTextView.getText(), passwordEditText.getText());   if (!user.isValidEmail()){  emailAutoCompleteTextView.setError("E-mail inválido");  }   if (!user.isValidPassword()){  passwordEditText.setError("Senha inválida. Sua senha deve conter pelo menos 4 caracteres.");  }   if (user.isValid()) {  *// navega para a próxima tela*  startNextActivity();  }  }   public void startNextActivity() {  Intent intent = new Intent(this, ComposeActivity.class);  startActivity(intent);  }  } |

Listagem 24 – LoginActivitity completa

1. Você tem agora a sua *Activity* totalmente refatorada com a ButterKnife. Viu como o código ficou mais limpo? Execute-o e teste para ver o efeito.

**DICA**: No mundo da computação existem termos peculiares e um deles é o termo “refatorar”, que se trata do ato de fazer uma reforma, alteração grande e de forma disciplinada em um sistema de *software* sem que seu comportamento mude. Não é uma versão 2.0 do seu *software*, é apenas uma melhoria interna no seu código.

Repare que foi removido o método de *Alerts* que você tinha na aula passada. Em vez disso, você passou a utilizar o método de erro dos próprios *EditTexts*. Eles basicamente mostram uma mensagem vermelha quando são invocados. Agora quando *e-mail* e senha estão válidos, navega-se para a nova tela **CompositeActivity.** Nas aulas seguintes, serão dados mais exemplos de navegação entre *Activities* por meio dos *Intents*.

## Resumo

Nesta aula aprendemos um dos conceitos fundamentais do Android: Seus *layouts*. Existem muitos outros ainda, mas com estes conseguimos fazer praticamente tudo. Se você quiser saber mais sobre os tipos de *layout*, dê uma pesquisada na documentação oficial do Android. Nós aprendemos também como utilizar código de terceiros e descobrimos como eles realmente podem facilitar nossa vida, usando o exemplo da biblioteca Butterknife.