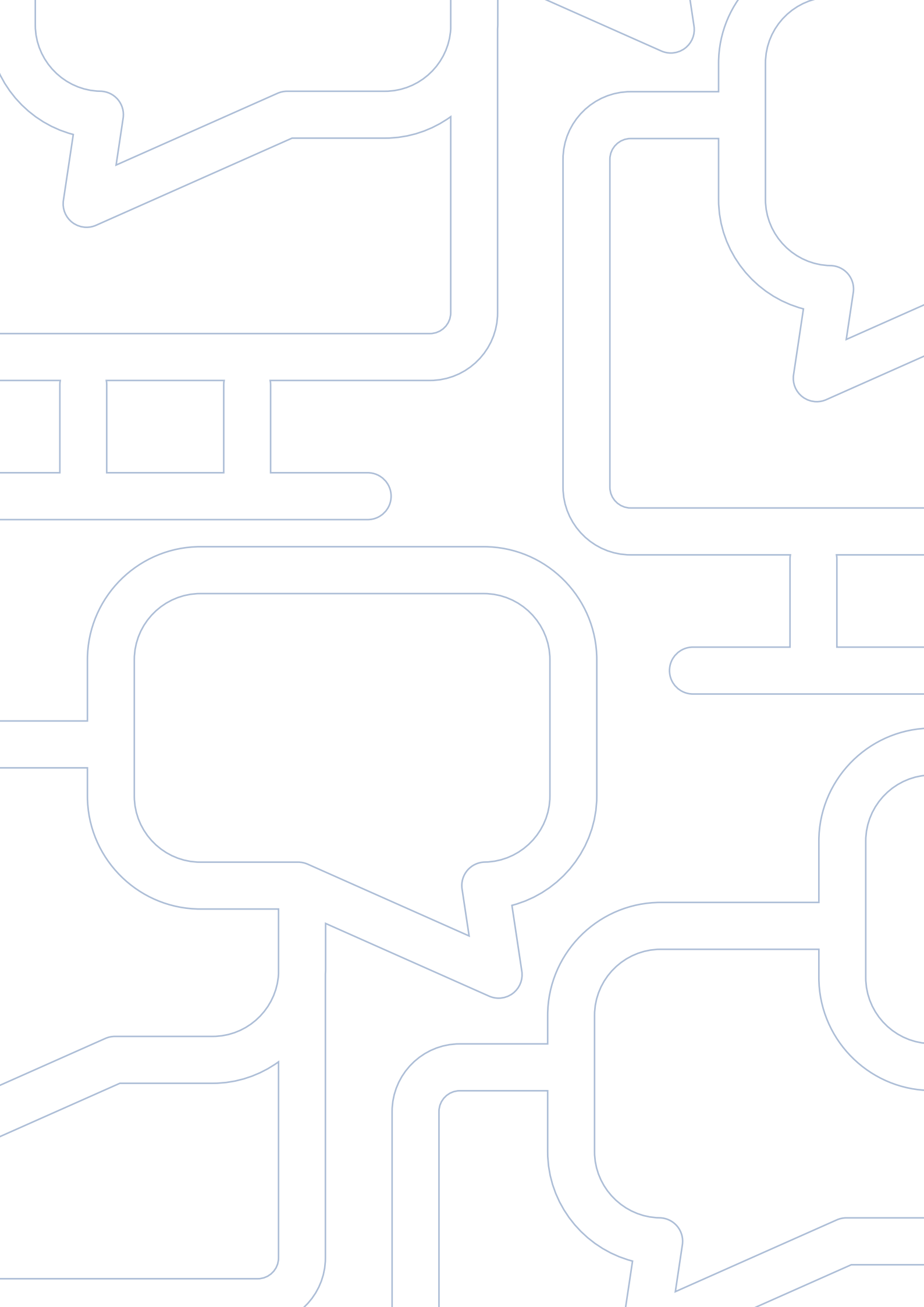
Unifametro
Formar para transformar

Imagem: Freepik

Programação para Dispositivos Móveis

UNIDADE II



Unifametro
Formar para transformar

Apresentação

Seja bem-vindo à disciplina **Programação para Dispositivos Móveis**. A metodologia adotada pela **UNIFAMETRO** na modalidade a distância conta com aulas ministradas no Ambiente Virtual de Aprendizagem Moodle, favorecendo atividades colaborativas, aproximando professores e alunos distantes fisicamente.

Para a melhor compreensão do assunto, é importante que, além da leitura desse conteúdo, você participe das atividades e tarefas propostas no cronograma e consulte dicas e complementos sugeridos no decorrer da disciplina. Este material também está disponível para impressão ou para download em formato pdf **clicando no botão abaixo:**

educacaoonline.unifametro.edu.br

É importante ressaltar que, nessa metodologia, o seu compromisso com a aprendizagem é fundamental para que ela aconteça. Portanto, participe, lendo os textos antes de cada aula para interagir melhor nos momentos de comunicação virtual e presencial, nas discussões com os tutores, professores e colegas de curso.

BONS ESTUDOS!

Fornecimento de recursos

Objetivos de aprendizagem

Ao final deste texto, você deve apresentar os seguintes aprendizados:

- Identificar os principais recursos consumidos por um aplicativo Android.
- Explicar o funcionamento da pasta "res" e sua relação com os recursos.
- Demonstrar o uso da pasta "res".

Introdução

Neste capítulo você aprenderá sobre os recursos que a plataforma Android fornece para apoio na construção de aplicativos.

Na construção de um aplicativo, assim como na construção de um *software*, é importante sabermos quais as funcionalidades que a API do sistema operacional oferece, o que também é válido para o desenvolvimento de um aplicativo. Sendo assim, apresentar e entender o fornecimento de recursos pelo uso da pasta ″res″ e qual sua relação com a camada de fornecimento de recursos da plataforma Android é o objetivo deste capitulo.

Conhecendo os *resources* em projetos Android

Desenvolver um aplicativo é muito mais do que escrever linhas de código. O bom desenvolvedor precisa conhecer os recursos que o Android oferece e saber trabalhar com eles na construção de seus aplicativos. Na estrutura do Android, os recursos disponíveis para o desenvolvedor ficam localizados na pasta “res”. A seguir, vamos listar brevemente os recursos que podemos encontrar nesse diretório.

Principais recursos consumidos

Conhecendo e entendendo os pilares de um aplicativo, vamos falar especificamente sobre recursos. Todo aplicativo Android trabalha com alguns recursos em comum. Entre eles se destacam, por exemplo, os recursos gráficos e o *layout*. No Android, todos os recursos consumidos ficam na estrutura de recursos, conhecida pelo diretório "res".

Os recursos padrão já disponíveis para consumo de qualquer aplicativo e sua finalidade são descritos a seguir.

- **Animações :** arquivos XML que definem as animações intermediárias.
- **Cores :** arquivos XML que definem uma lista de estado de cores.
- ***Drawable:*** os arquivos Bitmap (png, .9.png, .jpg, .gif) ou os arquivos XML são compilados nos seguintes subtipos de recurso *drawable*:
  - arquivos Bitmap;
  - 9-Patch (bitmaps redimensionáveis);
  - listas de estado;
  - formatos;
  - desenháveis de animação;
  - outros desenháveis;
  - arquivos *drawable* para diferentes densidades do ícone do inicializador.
- ***Layout:*** arquivos XML que definem um *layout* de interface do usuário.
- **Menu:** arquivos XML que definem os menus do aplicativo, como menu de opções, menu de contexto ou submenu.
- **Arquivos *raw:*** arquivos arbitrários para se salvar na forma bruta. No entanto, se for preciso acesso aos nomes e à hierarquia dos arquivos originais, considere salvar alguns recursos no diretório assets/ (em vez de res/raw/). Os arquivos em assets/ não recebem um código de recurso, portanto é possível lê-los usando apenas o AssetManager.
- **Valores:** arquivos XML que contêm valores simples, como *strings*, números inteiros e cores. Os arquivos de recurso podem estar em demais subdiretórios res/. Neste caso, é necessário definir um único recurso com base no nome do arquivo XML. Arquivos no diretório values/ descrevem outros recursos. Por exemplo, um elemento do tipo <string> cria um recurso R.string; e um elemento <color> cria um recurso R.color. Já que cada recurso é definido por seu próprio arquivo XML, podemos nomear o arquivo da forma que desejarmos e mesclar recursos em um único

arquivo. No entanto, o desenvolvedor pode desejar colocar arquivos de tipos únicos em arquivos separados. Veja algumas convenções de nomes que podemos criar nesse diretório:

- arrays.xml para matriz de recurso (matriz digitadas);
- colors.xml para valores de cor;
- dimens.xml para valores de dimensão;
- strings.xml para valores de *string*;
- styles.xml para estilos.

- **XML:** arquivos arbitrários XML que podem ser lidos em tempo de execução chamando Resources.getXML(). Vários arquivos de configuração XML devem ser salvos aqui, como uma configuração buscável.

Estrutura de diretório de recursos

Além dos recursos padrão, quase todos os aplicativos também devem fornecer recursos alternativos que possibilitam suportar características específicas de cada dispositivo. Por exemplo, podemos necessitar especificar recursos alternativos de *layout* de tela para um *tablet* ou para um celular, ou suportar idiomas diferentes em um mesmo aplicativo.

Como desenvolvedor, você sempre deverá utilizar os arquivos e as pastas de *resource* para armazenar os valores e as imagens do seu aplicativo. Dessa forma, além de organizado, será muito mais fácil manter e atualizar o seu código com maior facilidade. A pasta "res" é a pasta do projeto que guarda todos os *resources* da aplicação. A Figura 1 mostra a estrutura da pasta "res".

```
res/
    layout/
        main.xml  (Default layout)
    layout-ar/
        main.xml  (Specific layout for Arabic)
    layout-ldrtl/
        main.xml  (Any "right-to-left" language, except
                  for Arabic, because the "ar" language qualifier
                  has a higher precedence.)
```

Figura 1. Estrutura da pasta "res" com um *layout* de idioma diferente do *default*.

Fonte: App resources... (2019, documento *on-line*).

A pasta "res" deve ser subdividida em várias pastas com propósitos diferentes. Por exemplo, as pastas que começam com o sufixo *drawable* são utilizadas para armazenar as imagens da aplicação; as pastas *layout* servem para armazenar os arquivos XML usados para montar os *layouts* do aplicativo etc. Para que possamos utilizar os recursos disponíveis na pasta "res", é necessário definirmos o código que utilizará esse recurso na classe R, que é gerada quando o aplicativo é compilado.

O Quadro 1 ilustra essa estrutura e os tipos de recursos encontrados em cada subdiretório.

Quadro 1. Estrutura do diretório *resources*

Diretório	Recurso
animator/	Arquivos XML que definem animações de propriedades.
anim/	Arquivos XML que definem *tween animations* (animações de propriedades podem ser salvas aqui também, mas o diretório animator/ é preferível, por questão de organização).
color/	Arquivos XML que definem um estado de lista de cores.
drawable/	Arquivos Bitmap (png, .jpg, .gif) ou arquivos XML que são compilados nos seguintes *resources*: ■ arquivos Bitmap; ■ 9-Patch (Bitmaps redimensionáveis); ■ listas de estado; ■ *shapes* (formas); ■ animações (*frame animations*); ■ outros tipos.
mipmap/	Arquivos *drawable* especificamente para ícones de aplicações de diferentes tipos de densidade.
layout/	Arquivos XML que definem sua UI (user interface).
menu/	Arquivos XML que definem os menus da sua aplicação, como menu de opções, de contexto e submenus.
raw/	Pasta para salvar arquivos na sua forma natural (*raw*).

(Continua)

(Continuação)

Quadro 1. Estrutura do diretório *resources*

Diretório	Recurso
values/	Arquivos XML que contêm valores, como *strings*, números inteiros e cores. Dentro dessa pasta você deve nomear cada arquivo XML com o nome do seu *resource* e, dessa forma, irá acessá-lo com uma subclasse de R. Por exemplo, o arquivo string.xml será acessado por R.string. Abaixo algumas convenções: ■ arrays.xml para arrays; ■ colors.xml para valores de cores; ■ dimens.xml para dimensões; ■ strings.xml para todas as *strings*; ■ styles.xml para estilos.
xml/	Arquivos XML que podem ser lidos em tempo de execução chamando Resources.getXML(). Vários arquivos XML de configuração podem ser salvos aqui.

Fonte: Adaptado de Valney (2015).

Após disponibilizar um recurso no aplicativo, é possível referenciá-lo e utilizá-lo pelo seu ID. A classe R deve possuir um código para referenciar todos os recursos consumidos pelo aplicativo. Sendo assim, para cada tipo de recurso há uma subclasse R (por exemplo, R.drawable para todos os recursos *drawable*) e para cada recurso daquele tipo existe um número estatístico inteiro, que é o ID do recurso para recuperá-lo.

Fique atento

É fundamental para qualquer desenvolvedor Android conhecer bem os recursos disponíveis. Esse conhecimento gera melhor aproveitamento e possibilita o desenvolvimento de aplicativos com qualidade superior, aproveitando melhor os recursos dos dispositivos.

Apesar de o código dos recursos serem especificados na classe R, conforme já mencionado, esta classe não deve ser usada como fonte para consulta para descobrir o código de um recurso, pois este sempre é composto por:

- Tipo de recurso — cada recurso estará agrupado em um tipo, como por exemplo *string*, *drawable* e *layout*.
- Nome do recurso — na verdade, é o nome do arquivo, excluindo a extensão ou o valor no atributo android.name do XML se o recurso for um valor simples como uma *string*.

Existem duas formas de acessarmos um recurso:

- Pelo código: usar um número inteiro estático de uma subclasse R. Por exemplo: R.string.hello. *String* é o tipo de recurso e *hello*, seu nome. Existem diversas APIs do Android que podem acessar seus recursos quando você fornece um ID de recurso nesse formato.
- Você pode usar um recurso no código, passando o ID do recurso como parâmetro do método. Por exemplo, é possível definir uma ImageView para usar o recurso res/drawable/myimage.png, usando setImageResource():

```
ImageView imageView = (ImageView) findViewById(R.
id.myimageview);
imageView.setImageResource(R.drawable.myimage);
```

Também é possível recuperar recursos individuais usando métodos em *resources*, dos quais é possível obter uma instância com getResources().

Saiba mais

Não é recomendável realizar a alteração do arquivo R.java manualmente. Este é gerado pela ferramenta aapt quando o projeto é compilado. As alterações serão sobrepostas em uma próxima compilação.

Link

Acesse o *link* a seguir para ler mais sobre recursos.

https://qrgo.page.link/HQyjS

Podemos definir valores para alguns atributos do XML utilizando uma referência a um recurso existente. Isso será feito com frequência ao criar arquivos de *layout* para fornecer *strings* e imagens para os *widgets*.

Por exemplo, se você adicionar um *button* ao *layout*, deverá usar um recurso de *string* para o texto do botão:

```
<Button
  android:layout _ width="fill _ parent"
  android:layout _ height="wrap _ content"
  android:text="@string/submit" />
```

Esta é a sintaxe para referenciar um recurso em um recurso XML:

```
@[<package _ name>:]<resource _ type>/<resource _ name>
```

- <package_name> é o nome do pacote no qual o recurso está localizado (não é obrigatório ao referenciar recursos do mesmo pacote);
- <resource_type> é a subclasse R do tipo de recurso;
- <resource_name> é o nome do arquivo do recurso sem a extensão ou o valor do atributo android:name no elemento XML (para valores simples).

Consulte tipos de recursos para obter mais informações sobre cada tipo de recurso e como referenciá-los.

Referência a atributos de estilo

Os recursos referentes a atributo de estilo servem para que possamos referenciar o atributo em temas. Referenciar atributos de estilo possibilita trabalharmos com *layouts* personalizados, diversificando a aparência de elementos da interface, deixando-os em um estilo que corresponda ao padrão visual desejado.

A sintaxe para referenciar um atributo de estilo se assemelha ao formato do recurso, mas ao invés do símbolo (@), se usa uma interrogação (?). A parte relacionada ao tipo de recurso nesse caso é opcional. Confira no exemplo a seguir.

```
?[<package _ name>:][<resource _ type>/]<resource _ name>
```

Definindo recurso de *string*

Para todos os textos que precisamos exibir no aplicativo, como o rótulo de um botão ou um texto informativo em uma tela, definiremos o texto no arquivo string.xml, localizado em res/values. Cada entrada é uma chave que representa o ID do texto e depois o valor em si.

Por exemplo, se quisermos que um botão exiba a palavra "Enviar", procederemos adicionando o seguinte recurso.

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
  <string name="hello">Hello!</string>
  <string name="submit _ label">Enviar</string>
</resources>
```

Após realizada a referência do recurso de *string* para o submit_label, o padrão a ser exibido será sempre "Enviar". Deveremos então criar os *qualified resource files*, ou seja, qualificar esse recurso para que este valor possa ser alterado para os diferentes dispositivos ou países. Também poderemos armazenar *strings* mais complexas (com html ou caracteres especiais) usando o CDATA:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
 <string name="feedback _ label">
 <![CDATA[
  Please <a href="http://highlight.com">let us know</a> if you
have feedback on this or if
   you would like to log in with another identity service.
Thanks! This is a longer string!
 ]]>
 </string>
</resources>
```

Link

Para mais detalhes sobre a definição, recomendamos a consulta ao guia disponível no *link* a seguir.

https://qrgo.page.link/YtVCq

Consulte mais sobre o fornecimento de recursos alternativos no *link* a seguir.

https://qrgo.page.link/9cZRq

Referências

APP RESOURCES overview. *Android Developers Docs: Guides*, [*S. l.*], 2019. Disponível em: https://developer.android.com/guide/topics/resources/providing-resources. Acesso em: 13 jun. 2019.

VALNEY, M. A pasta "res" e os resources em um projeto Android. *Mário Valney*, [*S. l.*], 4 abr. 2015. Disponível em: https://mariovalney.com/a-pasta-res-e-os-resources-em-um--projeto-android/. Acesso em: 13 jun. 2019.

Leituras recomendadas

ANDROID Developers. [*S. l.: S. n.*], 2019. Disponível em: https://developer.android.com. Acesso em: 13 jun. 2019.

DEITEL, P.; DEITEL, H.; DEITEL, A. *Android*: como programar. 2. ed. Porto Alegre: Bookman, 2015. 690 p.

DEITEL, P.; DEITEL, H.; WALD, A. *Android 6 para programadores*: uma abordagem baseada em aplicativos. 3. ed. Porto Alegre: Bookman, 2016. 618 p.

DICAS de segurança. *Android Developers Docs: Guides*, [*S. l.*], 2019. Disponível em: https://developer.android.com/training/articles/security-tips. Acesso em: 13 jun. 2019.

Recursos e a classe R.java

Objetivos de aprendizagem

Ao final deste texto, você deve apresentar os seguintes aprendizados:

- Reconhecer a classe R.java em projetos Android.
- Explicar o uso da classe R.java como ferramenta de acesso a recursos.
- Descrever as etapas para o uso da classe R.java.

Introdução

Em projetos de desenvolvimento de aplicativos para Android, é importante ressaltar a finalidade da classe R.java. Essa classe tem como objetivo principal realizar a ligação entre o arquivo XML e seu respectivo código na aplicação. Neste capítulo, conheceremos a classe R.java e aprenderemos como utilizá-la para trabalhar com os recursos.

Entendendo a classe R.java no Android

A classe `R` é responsável por fazer a ligação entre o arquivo XML, localizado no diretório res, e o código da aplicação. Essa classe é gerada automaticamente pelo *plugin* da AAPT, no momento em que o *build* é executado. Nunca deveremos realizar alterações diretamente nessa classe: a classe `R` é gerada automaticamente pelo *plugin* da AAPT e somente a AAPT pode realizar alterações nessa classe. A classe `R` representa os recursos do projeto Android, que correspondem a todos os conteúdos da pasta res.

Quando o aplicativo for compilado, conforme citado acima, a AAPT irá gerar a classe `R` sem que exista a necessidade de interferência. Essa classe contém um código que permite acesso a todos os recursos disponíveis no diretório res. Para cada tipo de recurso há uma subclasse `R` (por exemplo, `R.drawable` para todos os recursos *drawable*) e, para cada recurso daquele tipo, há um número inteiro estático (por exemplo, `R.drawable.icon`). Esse número inteiro é o ID do recurso e pode ser usado para recuperá-lo.

A Figura 1 mostra a sintaxe utilizada para acessar um tipo de recurso.

```
ImageView imageView = (ImageView) findViewById(R.id.myimageview);
imageView.setImageResource(R.drawable.myimage);
```

Figura 1. Exemplo de código para acessar um recurso.

Fonte: Acesso... (2019, documento *on-line*).

Como acessar os recursos

Sabemos que devemos organizar os arquivos e as pastas e que devemos sempre nos preocupar em criarmos um aplicativo com a melhor compatibilidade possível, mas como fazer para trabalharmos com os recursos?

O processo é relativamente simples. Para realizar este acesso, devemos utilizar o ID de cada recurso, e todos estes IDs são definidos em sua respectiva classe `R`, gerada automaticamente pela IDE.

Quando a aplicação é compilada, a IDE gera a classe `R` para os *resources*, que estão na pasta res, e as subclasses, para os tipos de *resource*. Além disso, para cada *resource* desse tipo é automaticamente gerado um inteiro estático, mas apesar da classe `R` conter todos esses IDs, não é necessário consultá-la para acessar um *resource*, Para isto, basta sabermos que um ID é sempre composto pelo tipo e pelo nome do *resource* (sem extensão). Sendo assim, o ID de um arquivo *activity _about.xml* dentro da pasta *layout*, por exemplo, é `layout.activity _ about`.

Portanto, existem duas formas de acessarmos um *resource* utilizando esse ID.

- No código, usando o respectivo identificador inteiro estático, provido pela classe `R`, assim `R.layout.activity _ about`.

Por exemplo:

```
@Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity _ about);
    }
```

- Em um arquivo XML, usando uma sintaxe especial (@ tipo/nome) e o ID provido pela classe `R`, assim: `@string/ list _ item _ textview _ title _ default`.

Por exemplo:

```
&<lt;TextView
    android:id="@+id/list _ item _ textview _ title"
    android:layout _ width="wrap _ content"
    android:layout _ height="wrap _ content"
    android:text="@string/app _ credits"
    style="@style/h1"&gt; />
```

Em suma, o identificador para acessar um recurso pela classe é o caminho para se encontrar o atributo.

Alguns recursos podem ser acessados pela utilização do ID do atributo. Por exemplo, se colocarmos um botão na tela e este tiver o `id button1`, usaríamos a referência ao respectivo ID para acessá-lo:

```
Int i = R.id.button1;
```

Lembramos que todos os recursos disponíveis ficam localizados na estrutura de diretórios da pasta *res*. A Figura 2 demonstra esta estrutura.

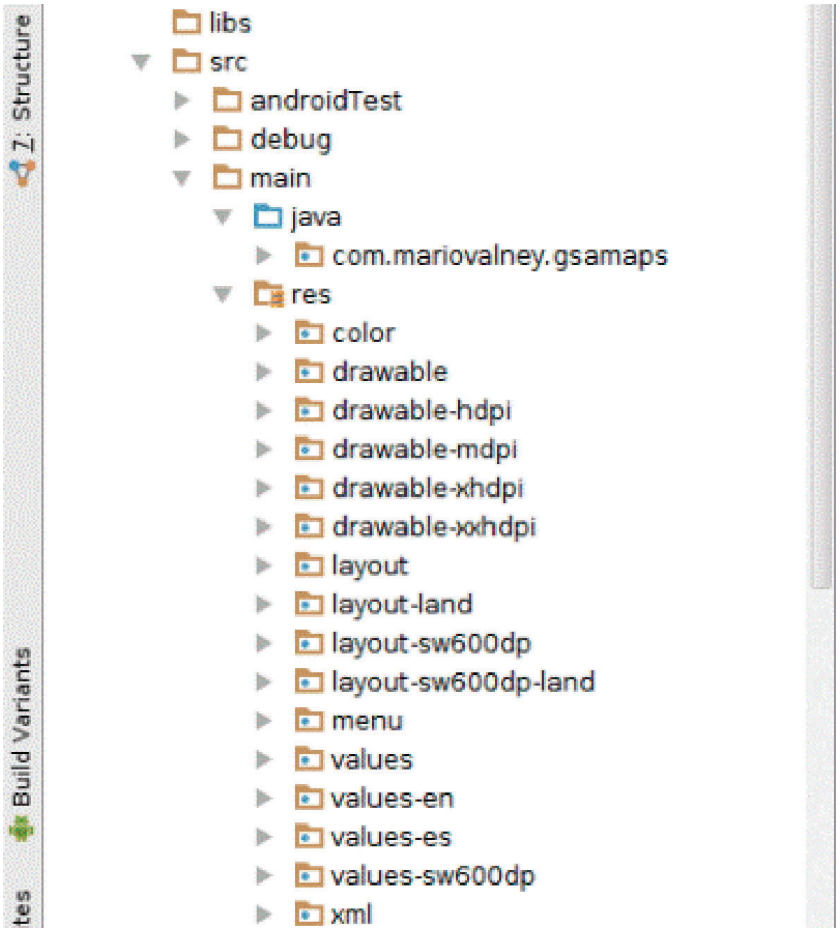


Figura 2. Pasta res na estrutura padrão do projeto Android.

Fonte: Valney (2015, documento *on-line*).

Sem a classe `R` não é possível realizar a busca de valores de variáveis estáticas definidas nos arquivos XML. Vamos analisar o código abaixo, um exemplo de classe R.java de um aplicativo de venda, lembrando que essa classe foi gerada automaticamente pela IDE de desenvolvimento.

```
/* AUTO-GENERATED FILE.  DO NOT MODIFY.
 *
 * This class was automatically generated by the
 * AAPT tool from the resource data it found.  It
 * should not be modified by hand.
 */
```

```
package com.example.ampla.vendaapp;

public final class R {
    public static final class attr {
    }
    public static final class dimen {
         /**  Default screen margins, per the Android Design
guidelines.

       Example customization of dimensions originally defined
in res/values/dimens.xml
         (such as screen margins) for screens with more than
820dp of available width. This
        would include 7" and 10" devices in landscape (~960dp
and ~1280dp respectively).

         */
    public static final int activity _ horizontal _ margin=0x7f040000;
     public static final int activity _ vertical _ margin=0x7f040001;
    }
    public static final class drawable {
        public static final int ic _ launcher=0x7f020000;
    }
    public static final class id {
        public static final int action _ settings=0x7f080000;
    }
    public static final class layout {
        public static final int activity _ main=0x7f030000;
    }
    public static final class menu {
        public static final int main=0x7f070000;
    }
    public static final class string {
        public static final int action _ settings=0x7f050002;
        public static final int app _ name=0x7f050000;
        public static final int hello _ world=0x7f050001;
    }
    public static final class style {
        /**
```

```
        Base application theme, dependent on API level. This
theme is replaced
       by AppBaseTheme from res/values-vXX/styles.xml on newer
devices.

           Theme customizations available in newer API levels
can go in
              res/values-vXX/styles.xml, while customizations
related to
            backward-compatibility can go here.


         Base application theme for API 11+. This theme com-
pletely replaces
          AppBaseTheme from res/values/styles.xml on API 11+
devices.

 API 11 theme customizations can go here.

         Base application theme for API 14+. This theme com-
pletely replaces
        AppBaseTheme from BOTH res/values/styles.xml and
        res/values-v11/styles.xml on API 14+ devices.

 API 14 theme customizations can go here.
         */
        public static final int AppBaseTheme=0x7f060000;
        /**  Application theme.
 All customizations that are NOT specific to a particular
API-level can go here.
         */
        public static final int AppTheme=0x7f060001;
    }
}
```

Fique atento

A classe R.java é gerada automaticamente na compilação e permite que a aplicação acesse qualquer recurso, como imagens e arquivos, utilizando as constantes dessa classe para construir a tela da aplicação. Não altere essa classe manualmente.

Etapas para utilização da classe R.java

Para demonstrar a utilização da classe `R` e referenciar o uso de recursos, vamos utilizar os arquivos de cores, que são arquivos utilizados para definir todas as cores em um aplicativo.

As cores devem ser definidas em *res/values/colors.xml* e o arquivo XML deverá ser parecido com o exemplo abaixo:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
   <color name="white">#FFFFFF</color>
   <color name="yellow">#FFFF00</color>
   <color name="fuchsia">#FF00FF</color>
</resources>
```

Definidas as cores, podemos acessá-las com um código semelhante a este:

```
// Use ContextCompatResources instead of getColor()
int color = ContextCompat.getColor(context, R.color.yellow);
```

É importante salientar que a forma mais usual de acesso aos recursos de cores (desde a API 24) requer o fornecimento do *context* para resolver quaisquer atributos de temas personalizados.

Deve-se referenciar o XML dentro de qualquer *view*, utilizando a sintaxe:

```
<TextView
    android:layout _ width="fill _ parent"
    android:layout _ height="wrap _ content"
    android:textColor="@color/fuchsia"
    android:text="Hello"/>
```

Esse seria um exemplo com tudo que precisamos para utilizar os recursos de cores. Certifique-se de manter todas as cores codificadas nos arquivos de *layout*.

Há muitos métodos que aceitam um parâmetro de ID de recurso. Os recursos podem ser recuperados pela utilização de métodos em `Resources`. É possível obter uma instância de `Resources` com `Context.getResources()`.

Abaixo, observamos alguns exemplos do acesso aos recursos no código:

```
getWindow().setBackgroundDrawableResource(R.drawable.my _ ba-
ckground _ image) ;

getWindow().setTitle(getResources().getText(R.string.
main _ title));

setContentView(R.layout.main _ screen);

mFlipper.setInAnimation(AnimationUtils.loadAnimation(this,
        R.anim.hyperspace _ in));

TextView msgTextView = (TextView) findViewById(R.id.msg);
msgTextView.setText(R.string.hello _ message);
```

Link

No *link* a seguir você pode acessar o guia do desenvolvedor Android para obter mais informações sobre o desenvolvimento desta aplicação.

https://qrgo.page.link/zFh86

Exemplo

Definido um recurso de *String*, podemos acessá-lo em nosso código Java ou em nossos *layouts* XML a qualquer momento. Para acessar o recurso no arquivo XML Layout, simplesmente use a sintaxe `@`, utilizada para acessar qualquer recurso:

```
<Button
    android:layout _ width="match _ parent"
    android:layout _ height="wrap _ content"
    android:text="@string/submit _ label" />
```

Para acessar o recurso diretamente em seu código Java, basta usar os métodos `getResources.getString` ou `getString` para acessar o valor dado ao ID do recurso:

```
String submitText = getResources().getString(R.string.
submit _ label)
```

Assim, o valor da *String* será recuperado. Essa sintaxe funciona para quase todos os recursos, desde imagens (*drawables*) até cores. O método `getResources()` retorna um objeto `Resources` com muitos métodos de busca de recursos. Cada recurso é definido em diferentes pastas e arquivos dentro do diretório res, dependendo do tipo.

Referências

ACESSO aos recursos. *Android Developers*, [*S. l.*], 2019. Disponível em: https://developer.android.com/guide/topics/resources/accessing-resources.html?hl=pt-br. Acesso em: 16 jun. 2019.

VALNEY, M. A pasta "res" e os resources em um projeto Android. *Mário Valney*, [*S. l.*], 4 abr. 2015. Disponível em: https://mariovalney.com/a-pasta-res-e-os-resources-em-um--projeto-android/. Acesso em: 16 jun. 2019.

Leituras recomendadas

ANDROID Developers. [*S. l.*: *S. n.*], 2019. Disponível em: https://developer.android.com. Acesso em: 16 jun. 2019.

DEITEL, P.; DEITEL, H.; DEITEL, A. *Android*: como programar. 2. ed. Porto Alegre: Bookman, 2015. 690 p.

DEITEL, P.; DEITEL, H.; WALD, A. *Android 6 para programadores*: uma abordagem baseada em aplicativos. 3. ed. Porto Alegre: Bookman, 2016. 618 p.

DICAS de segurança. *Android Developers Docs: Guides*, [*S. l.*], 2019. Disponível em: https://developer.android.com/training/articles/security-tips. Acesso em: 16 jun. 2019.

Componente *activity*

Objetivos de aprendizagem

Ao final deste texto, você deve apresentar os seguintes aprendizados:

- Definir uma *activity*.
- Descrever o ciclo de vida de uma *activity*.
- Reconhecer os principais métodos executados na sequência de uma *activity*.

Introdução

Neste capítulo, você conhecerá o componente *activity*, reponsável pela exibição de interfaces de usuário, ou seja, quando alteramos um contato da agenda telefonica, efetuamos uma ligação ou enviamos um SMS, estamos interagindo com o sistema por meio de uma *activity*.

Também compreenderá o ciclo de vida de um componente *activity*, um de seus mais importantes conceitos, uma vez que define o comportamento da *activity* nas eventuais situações que surgem enquanto um dispositivo é utilizado. Ainda, reconhecerá os principais métodos executados durante a execução de uma *activity*.

Definição de *activity*

Segundo Deitel, Deitel e Wald (2016), a *activity* é um componente de aplicativos Android que fornece uma interface de usuário com a qual este pode interagir para realizar determinada operação, discar um número no telefone, gravar um vídeo, enviar uma mensagem ou até mesmo visualizar uma rota em um mapa. Cada *activity* recebe uma janela que exibe a interface do usuário. Geralmente, uma *activity* preenche a tela integralmente, porém esse compmente pode ser menor que a tela do dispositivo e até mesmo flutuar sobre outras janelas.

Na Figura 1, você pode observar alguns exemplos de interfaces de usuários, com a tela de cadastro de usuário (*Sign Up*) ocupando a tela do dispositivo em sua totalidade, enquanto as telas de *login* e recuperação de senha (*Sign In* e *Forget Password*, respectivamente) não ocupariam a tela em sua totalidade flutuando sobre outras janelas.

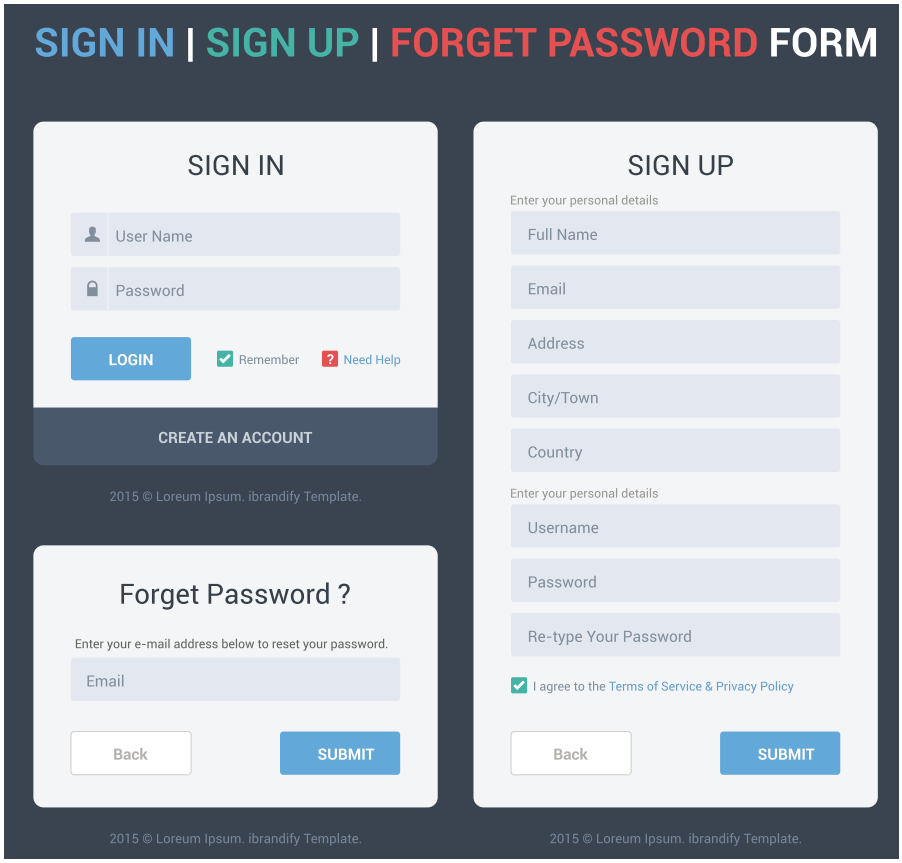


Figura 1. Exemplos de interfaces de usuário.

Fonte: ibrandify gallery/Shutterstock.com.

Para criar uma *activity*, é necessário implementar uma subclasse do tipo `Activity`, que pode ser ela mesma ou uma de suas implementações, como a classe `AppCompatActivity`. Na subclasse criada, deve-se executar um método de retorno de chamada do sistema quando da transição entre os diversos estados de seu ciclo de vida (mais adiante, abordaremos com mais detalhes o ciclo de vida de uma *activity*), como na criação, na interrupção, na retomada ou na destruição da *activity*. Os dois métodos mais importantes de retorno das chamadas mencionadas são:

- `onCreate()` — único de implementação obrigatória, uma vez que o sistema o chama ao criar a *activity*, devendo-se inicializar os componentes essenciais da atividade. É fundamental a chamada do método `setContentView()` para definir o leiaute da interface do usuário da *activity*.
- `onPause()` — indica o primeiro indício de que o usuário está saindo da *activity*, apesar de não apontar de maneira segura que é esta a ação em realização. Porém, normalmente, deve-se confirmar qualquer alteração, que, por sua vez, precisa ser salva, uma vez que o usuário pode não retomar à *activity* novamente.

Link

É muito importante conhecer a documentação da classe `Activity`, já que será o componente mais utilizado pelo desenvolvedor no início de sua curva de aprendizado. Para obter mais informações sobre a *activity*, acesse o *link* a seguir.

https://qrgo.page.link/nRz6p

A seguir, há um exemplo de uma *activity* chamada `MyFirstApp`, na qual podemos observar o método `onCreate` executando um `setContentView(R.layout.main`, que poderia estar configurando a visualização da uma interface, como a ilustrada na Figura 2.

```
public class MyFirstApp extends Activity {
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);
    }
}
```

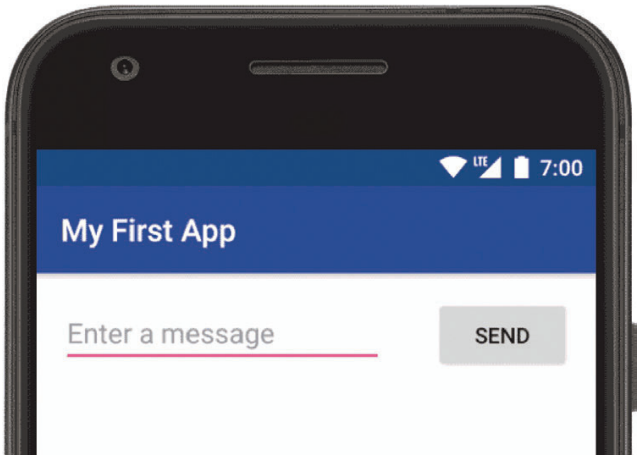


Figura 2. Exemplo de uma visualização de *activity*.
Fonte: Criar... (2019, documento *on-line*).

Ao idealizar um novo aplicativo com as ferramentas do Android SDK, a *activity* criada contém automaticamente um filtro de `intent` que declara que a atividade responde à ação `main` (principal), que deve ser colocada na categoria `launcher` (inicializador). O filtro de `intent` tem a seguinte aparência no manifesto do projeto:

```
<activity android:name=".ExampleActivity" android:icon="@
drawable/app _ icon">
    <intent-filter>
        <action android:name="android.intent.action.MAIN" />
        <category android:name="android.intent.category.LAUN-
CHER" />
    </intent-filter>
</activity>
```

O elemento `<action>` indica que esse é o "principal" ponto de entrada do aplicativo, e o elemento `<category>` que essa *activity* deve ser listada no inicializador do aplicativo do sistema (para permitir que quando os usuários executem o aplicativo, essa *activity* seja a primeira a ser chamada).

Fique atento

O Android Studio cria as estruturas descritas no manifesto e nas demais configurações sempre que você criar um projeto novo ou adicionar *activities* a um projeto. Porém, é muito importante que você conheça o manifesto do projeto, pois, por meio dele, poderá personalizar diversas características dos componentes *activities* que incluirá em seu projeto.

Para iniciar outra atividade, é possível chamar o método `startActivity()` passando uma *intent* que descreva a atividade que se deseja iniciar. O *intent* especifica a atividade exata que deve ser iniciada ou descreve o tipo de ação que ela deve executar (e o sistema seleciona a atividade adequada, que pode ser até mesmo de outro aplicativo), além de poder portar pequenas quantidades de dados a serem usados pela atividade iniciada.

No exemplo a seguir, temos a codificação de uma *intent* que iniciará a *activity* `ProductActivity` enviando a ela uma informação com o nome de `productId`, que, possivelmente, representa o código de determinado produto no qual, por exemplo, os dados do produto podem ser carregados e editados nessa *activity*:

```
Intent intent = new Intent(this, ProductActivity.class);
intent.putExtra("productId", varId);
startActivity(intent);
```

Existem casos em que se torna necessário receber um resultado de alguma *activity* iniciada, quando a *activity* deve ser iniciada utilizando o método `startActivityForResult()` (em vez de `startActivity()`, conforme exemplo anterior). Após a execução da *activity* indicada, o *activity* chamador receberá o resultado em uma *intent* no método `onActivityResult()`.

A Figura 3 ilustra esse processo: o método `pickContact()` cria uma *intent* para buscar um contato do dispositivo e retornar as informações deste para serem processadas pelo *activity* chamador, para o qual se utiliza o método `startActivityForResult()`. O resultado dessa seleção de contato é obtido no método `onActivityResult()`, em que se verifica se o resultado da *activity* chamada é `RESULT_OK` e se se trata da *activity* chamada verificando se o parâmetro `requestCode` é igual a `PICK_CONTACT_REQUEST` (observe que, ao chamar o método `startActivityForResult()`, o identificador `PICK_CONTACT_REQUEST` é enviado, que compreende um código qualquer, porém o único definido pelo desenvolvedor); após essas verificações, os dados do contato selecionado são tratados.

Observe que qualquer chamada realizada na *activity* utilizando o método `startActivityForResult()` será retornada no mesmo método `onActivityResult()` e que o parâmetro `requestCode` os diferencia para que cada um receba o tratamento adequado ao retornado solicitado.

```
private void pickContact() {
    // Create an intent to "pick" a contact, as defined by the content provider URI
    Intent intent = new Intent(Intent.ACTION_PICK, Contacts.CONTENT_URI);
    startActivityForResult(intent, PICK_CONTACT_REQUEST);
}

@Override
protected void onActivityResult(int requestCode, int resultCode, Intent data) {
    // If the request went well (OK) and the request was PICK_CONTACT_REQUEST
    if (resultCode == Activity.RESULT_OK && requestCode == PICK_CONTACT_REQUEST) {
        // Perform a query to the contact's content provider for the contact's name
        Cursor cursor = getContentResolver().query(data.getData(),
        new String[] {Contacts.DISPLAY_NAME}, null, null, null);
        if (cursor.moveToFirst()) { // True if the cursor is not empty
            int columnIndex = cursor.getColumnIndex(Contacts.DISPLAY_NAME);
            String name = cursor.getString(columnIndex);
            // Do something with the selected contact's name...
        }
    }
}
```

Figura 3. Exemplos de *activity* sendo chamada e retornando determinado resultado.

Fonte: Activity (2019, documento *on-line*).

Ciclo de vida das *activities*

Segundo Deitel, Deitel e Deitel (2015), em geral um aplicativo Android tem diversas *activities* pouco vinculadas entre si. Na maioria das vezes, uma *activity* de um aplicativo é definida como "principal", sendo apresentada ao usuário ao iniciar o aplicativo pela primeira vez. Assim, cada atividade pode iniciar outra atividade para executar diferentes ações. Na inicialização de uma nova atividade, a atividade anterior é interrompida, mas o sistema conserva a atividade em uma pilha (chamada "pilha de retorno").

Quando uma atividade inicia, ela é enviada para a pilha de retorno e obtém o foco do usuário. A pilha de retorno segue o mecanismo básico de pilha `Last In First Out` ou, simplesmente, `LIFO` (o último que entra é o primeiro que sai). Dessa forma, quando o usuário terminar a atividade atual e apertar o botão Voltar, a *activity* sairá da pilha (é destruída) e a atividade anterior será retomada. A Figura 4 ilustra esse comportamento com uma linha cronológica exibindo o progresso entre *activities* com a pilha de retorno atual em cada ponto no tempo, em que as *activities* são empilhadas até que a **Activity 3** é chamada e inserida na pilha; após a interação com o usuário ser concluída, este utiliza o botão Voltar para retornar à **Activity 2**, o que, consequentemente, destruirá a **Activity 3**.

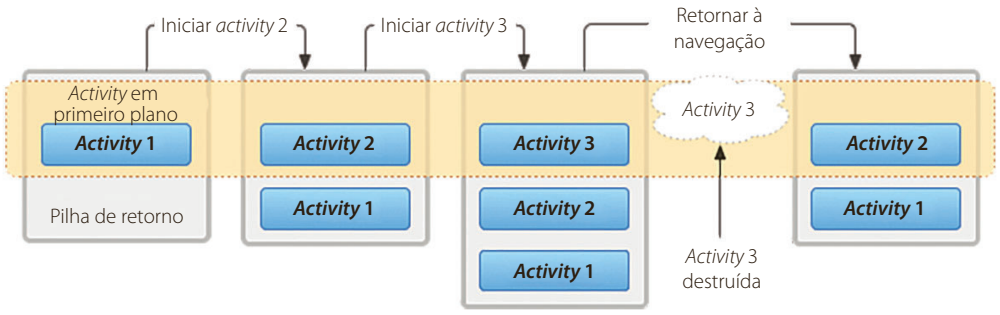


Figura 4. Exemplo de *activities* Android sendo empilhadas.

Fonte: Tarefas... (2019, documento *on-line*).

Quando uma *activity* é interrompida pelo início de uma nova *activity*, ela é notificada sobre essa alteração de estado por meio de métodos de retorno de chamada do ciclo de vida da atividade. Uma *activity* apresenta diversos métodos de retorno de chamada quando de uma alteração em seu estado, seja quando o sistema a está criando, interrompendo, retomando ou destruindo uma *activity*; e, a cada retorno de chamada, oferece uma oportunidade de executar trabalhos específicos adequados a essa alteração de estado.

Exemplo

Quando interrompida, a atividade deve liberar todos os objetos grandes, como conexões com a rede ou com um banco de dados. Quando a atividade for retomada, será possível readquirir os recursos necessários e retomar as ações interrompidas. Essas transições de estado fazem parte do ciclo de vida da atividade.

O gerenciamento do ciclo de vida das atividades se dá por meio da implementação de métodos de retorno de chamada, essenciais para desenvolver um aplicativo flexível. O ciclo de vida de uma atividade é diretamente afetado pela associação a outras atividades, pela tarefa e pela pilha de retorno. Uma *activity* pode apresentar três estados:

- `Resumed` — a *activity* está em primeiro plano e tem o foco do usuário, normalmente sendo chamado "em execução".
- `Paused` — uma *activity* assume esse estado quando ainda está visível, mas com outra *activity* em primeiro plano e com o foco, ou seja, outra *activity* está visível por cima da *activity* mencionada que está transparente ou não cobre a tela do dispositivo inteiramente.
- `Stopped` — a *activity* está em segundo plano, quando seu objeto ainda está em memória, mas não mais visível no dispositivo.

Uma *activity* `Paused` ou `Stopped` pode ser descartada pelo método `finish()` ou o sistema simplesmente a eliminar quando necessitar de memória; nesse caso, quando a *activity* for reaberta, será totalmente recriada. A *activity* pode transitar entre os diferentes estados descritos, sendo notificada por meio de vários métodos de retorno de chamada, os quais constituem "ganchos" que podem ser modificados para executar uma rotina adequada quando da mudança do estado da atividade. A seguir, temos uma lista dos métodos mencionados:

- `onCreate()`.
- `onStart()`.
- `onResume()`.
- `onPause()`.
- `onStop()`.
- `onDestroy()`.

Esse conjunto de métodos define o ciclo de vida da *activity*, e, ao implementá-los, é possível monitorar três *loops* alinhados a ele:

- **Tempo de vida total** — ocorre entre o método `onCreate()` (onde normalmente o laioute é definido) e o `onDestroy()` (onde os recursos são liberados).
- **Tempo de vida visível** — ocorre entre os métodos `onStart()` e `onStop()`; o sistema pode alternar a execução desses métodos várias vezes durante todo o tempo de vida de uma *activity* enquanto ela alterna entre visível e oculta ao usuário.
- **Tempo de vida em primeiro plano** — *loop* entre os métodos `onResume()` e `onPause()`, por exemplo, quando uma caixa de diálogo é chamada, o método `onPause()` é executado na *activity*, e no retorno da caixa de diálogo à *activity* o método `onResume()` é chamado.

A Figura 5 ilustra os *loops* e os caminhos que uma atividade pode tomar entre os seus estados. Os retângulos representam os métodos de retorno de chamada que podem ser implementados para executar operações quando a atividade transita entre os estados.

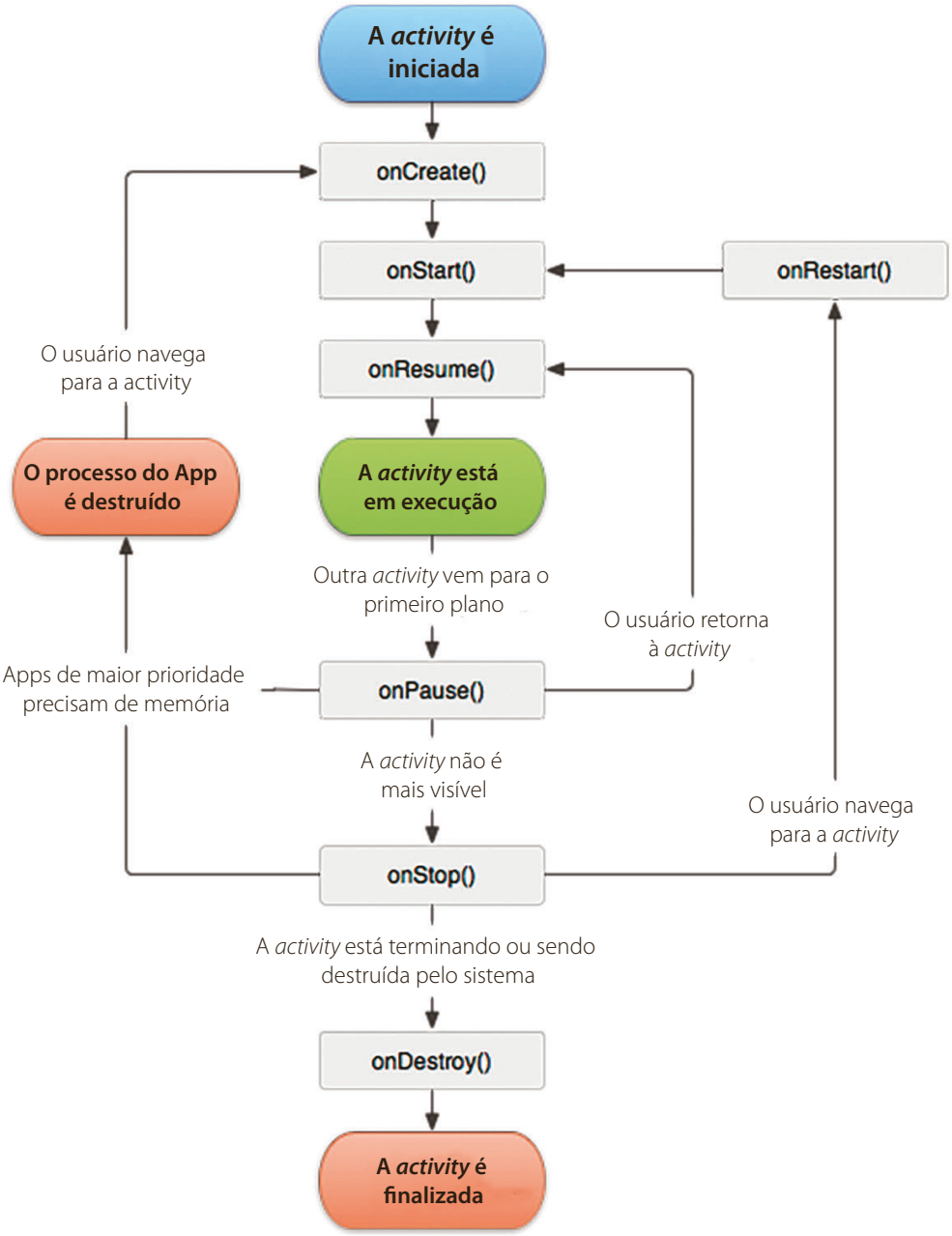


Figura 5. Ciclo de vida de uma *activity*.

Fonte: Atividades (2019, documento *on-line*).

No Quadro 1, há uma descrição dos métodos dentro do ciclo de vida geral da atividade, além de podermos observar se o sistema consegue eliminar a atividade depois da conclusão do método. A coluna "Pode ser eliminado depois?" indica se é possível eliminar o aplicativo ao qual a *activity* pertence, sem que nenhum código seja executado, logo após os métodos `onPause()`, `onStop()` e `onDestroy()`.

Quadro 1. Resumo dos métodos de retorno de chamada do ciclo de vida da atividade

Método	Descrição	Pode ser eliminado depois	Próximo
`onCreate()`	Chamado quando a atividade é criada pela primeira vez, no qual se deve fazer toda a configuração estática normal — criar exibições, vincular dados a listas, etc. Esse método recebe um objeto Bundle (pacote) contendo o estado anterior da atividade, caso esse estado tenha sido capturado. Sempre seguido de `onStart()`	Não	`onStart()`
`onRestart()`	Chamado depois da interrupção da atividade, logo antes de ser reiniciada. Sempre seguido de `onStart()`	Não	`onStart()`
`onStart()`	Chamado logo antes de a atividade se tornar visível ao usuário. Seguido de `onResume()`, se a atividade for para segundo plano, ou `onStop()`, se ficar oculta	Não	`onResume()` ou `onStop()`
`onResume()`	Chamado logo antes de a atividade iniciar a interação com o usuário. Nesse ponto, a atividade estará no topo da pilha de atividades com a entrada do usuário direcionada a ela. Sempre seguido de `onPause()`	Não	`onPause()`
`onPause()`	Chamado quando o sistema está prestes a retomar outra atividade. Esse método normalmente é usado para confirmar alterações não salvas a dados persistentes, animações interrompidas e outros elementos que talvez estejam consumindo CPU, e assim por diante. Ele sempre deve fazer tudo bem rapidamente porque a próxima atividade não será retomada até ela retornar. Seguido de `onResume()`, se a atividade retornar para a frente, ou de `onStop()`, se ficar invisível ao usuário.	**Sim**	`onResume()` ou `onStop()`

(Continua)

(Continuação)

Quadro 1. Resumo dos métodos de retorno de chamada do ciclo de vida da atividade

Método	Descrição	Pode ser eliminado depois	Próximo
`onStop()`	Chamado quando a atividade não está mais visível ao usuário. Isso pode acontecer porque ela está sendo destruída ou porque outra atividade (uma existente ou uma nova) foi retomada e está cobrindo-a. Seguido de `onRestart()`, se a atividade estiver voltando a interagir com o usuário, ou `onDestroy()`, se estiver saindo	**Sim**	`onRestart()` ou `onDestroy()`
`onDestroy()`	Chamado antes de a atividade ser destruída. Trata-se da última chamada que a atividade receberá. Pode ser chamado porque a atividade está finalizando (alguém chamou `finish()` nela) ou porque o sistema está destruindo temporariamente essa instância da atividade para poupar espaço. É possível distinguir entre essas duas situações com o método `isFinishing()`	**Sim**	Nada

Fonte: Atividades (2019, documento *on-line*).

Link

Quando o sistema destrói uma *activity* para liberar memória, esse objeto `Activity` que estava na pilha não pode ser exibido novamente de maneira intacta. Dessa forma, o sistema precisa recriar o objeto se o usuário navegar de volta a ele, de modo que o usuário não perceba que esse procedimento ocorreu e suas informações se encontrem intactas exatamente como antes. Você pode obter mais informações sobre esse processo, denominado estado transiente do aplicativo, no *link* a seguir.

https://qrgo.page.link/xDpps

Métodos executados na sequência de uma *activity*

Quando uma *activity* transita entre os diferentes estados e *loops* conforme descrito, ela é notificada por meio de vários métodos de retorno de chamada, os quais constituem espécies de "ganchos" que podem ser modificados para executar um trabalho adequado quando o estado da *activity* muda.

Na Figura 6, temos a ilustração de uma *activity* chamada `ExampleActivity`, em que podemos observar a implementação básica dos métodos que fazem parte do ciclo de vida da *activity* (métodos `onCreate()`, `onStart()`, `onResume()`, `onPause()`, `onStop()` e `onDestroy()`), além da chamada dos métodos `super.onCreate();`, `super.onStart();`, `super.onResume();`, `super.onPause();`, `super. onStop();` e `super. onDestroy();`, que pertencem à classe `Activity`, na qual a *activity* `ExampleActivity` está estendida. Veja que essa chamada é obrigatória logo após a ocorrência da chamado do método na *activity* `ExampleActivity` para que as rotinas internas da classe estendida sejam executadas e, posteriormente, as funcionalidades implementadas no método sejam executadas.

Fique atento

Utilizando a IDE Android Atudio, a criação de *activities* é automatizada, tornando-se possível escolher diversos modelos de *activities* predefinidos e o código-base ser inserido no projeto automaticamente. Nesse sentido, é necessário apenas implementar as funcionalidades previstas para a *activity* em questão.

Torna-se importante salientar que não é preciso implementar todos os métodos evidenciados na Figura 6, uma vez que somente o método `onCreate()` tem implementação obrigátória. Os demais são implementados de modo opcional conforme a necessidade de controle diante das necessidades de persistência de dados presentes na interface quando a *activity* for destruída pelo sistema ou algo do tipo. Dessa forma, uma *activity* com uma funcionalidade básica de, por exemplo, exibir uma informação na tela para o usuário, não conseguirá implementar muitos dos métodos citados.

```java
public class ExampleActivity extends Activity {
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        // The activity is being created.
    }
    @Override
    protected void onStart() {
        super.onStart();
        // The activity is about to become visible.
    }
    @Override
    protected void onResume() {
        super.onResume();
        // The activity has become visible (it is now "resumed").
    }
    @Override
    protected void onPause() {
        super.onPause();
        // Another activity is taking focus (this activity is about to be "paused").
    }
    @Override
    protected void onStop() {
        super.onStop();
        // The activity is no longer visible (it is now "stopped")
    }
    @Override
    protected void onDestroy() {
        super.onDestroy();
        // The activity is about to be destroyed.
    }
}
```

Figura 6. Implementação dos métodos de uma *activity*.

Fonte: Atividades (2019, documento *on-line*).

Conforme Simon (2011), além dos métodos que devem ser implementados, a *activity* dispõe de métodos que o desenvolvedor pode utilizar em diversas situações para várias funcionalidades, por exemplo, o método `setContentView` é responsável por configurar o leiaute exibido pela *activity*.

Dessa forma, em uma implementação básica, temos o exemplo ilustrado pela Figura 7, na qual está implementada a *activity* `CalendarActivity`. Em seu método `onCreate`, o formato do calendário utilizado pelo usuário é recuperado das preferências do usuário do aplicativo em questão, pela classe `SharedPreferences` (observando que temos dois formatos de visualização possíveis: `DAY_VIEW_MODE` e `WEEK_VIEW_MODE`); uma vez recuperada essa informação, a interface correspondente é exibida utilizando o método `setContentView` para definir o leiaute, quando da identificação do leiaute que será exibido.

Link

Normalmente, representa uma boa prática de usabilidade salvar os últimos dados que o usuário utilizou em determinada interface, como os dados de um filtro utilizado para gerar um relatório ou até mesmo uma configuração que o usuário modificou para adequar a visualização da interface à sua necessidade. A classe `SharedPreferences` fornece essa funcionalidade com uma implementação muito simples, conforme você poderá observar no *link* a seguir.

https://qrgo.page.link/jUqTH

Na interface de usuário da *activity* da Figura 7, temos uma opção para alterar essa visualização, cujo formato selecionado é armazenado na variável `mCurViewMode`. Dessa forma, na incidência do método `onPause` (que, conforme já mencionado, pode indicar que o usuário está saindo da interface), o formato da visualização armazenado em `mCurViewMode` é salvo utilizando a classe `SharedPreferences`. E, ao iniciar a *activity* `CalendarActivity`, novamente a visualização selecionada anteriormente pelo usuário estará disponível para o ele.

```
public class CalendarActivity extends Activity {
    ...

    static final int DAY_VIEW_MODE = 0;
    static final int WEEK_VIEW_MODE = 1;

    private SharedPreferences mPrefs;
    private int mCurViewMode;

    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        SharedPreferences mPrefs = getSharedPreferences();
        mCurViewMode = mPrefs.getInt("view_mode", DAY_VIEW_MODE);
        if (mCurViewMode == 1) {
            setContentView(R.layout.week);
        }
        else {
            setContentView(R.layout.day);
        }
    }

    protected void onPause() {
        super.onPause();

        SharedPreferences.Editor ed = mPrefs.edit();
        ed.putInt("view_mode", mCurViewMode);
        ed.commit();
    }
}
```

Figura 7. Exemplos de *activity* implementando os métodos mais utilizados.

Fonte: Activity (2019, documento *on-line*).

Outro método muito utilizado é o `findViewById`, empregado na busca de componentes da *activity*, por exemplo, um campo texto presente em uma interface configurada pelo método `setContentView`; abaixo, temos um exemplo do método em uso, quando ele busca o campo texto `R.id.message` (podemos fazer a relação do campo *message* presente nesse exemplo, como na Figura 7):

```
TextView tvMessage = (TextView) findViewById(R.id.message).
```

Além dos métodos mencionados relacionados à manipulação do leiaute, temos aqueles associados ao controle da *activity*, como o `finish()`, responsável por fechar e destruir (remover da memória do dispositivo) a *activity* corrente.

Link

Pela documentação da classe `Context` disponível no *link* a seguir, você pode obter mais informações sobre métodos disponíveis para executar as mais diversas funções em um projeto Android (observando que a classe `Activity` é estendida à classe `Context` e que, logo, a classe `Activity` disponibiliza a utilização de todos os métodos da `Context`).

https://qrgo.page.link/eRjtG

Referências

ACTIVITY. *Android Developers*, [*S. l.*], 2019. Disponível em: https://developer.android.com/reference/android/app/Activity.html?hl=pt-BR. Acesso em: 26 jun. 2019.

ATIVIDADES. *Android Developers*, [*S. l.*], 2019. Disponível em: https://developer.android.com/guide/components/activities.html?hl=pt-BR. Acesso em: 26 jun. 2019.

CRIAR uma interface do usuário simples. *Android Developers*, [*S. l.*], 2019. Disponível em: https://developer.android.com/training/basics/firstapp/building-ui?hl=pt-BR. Acesso em: 26 jun. 2019.

DEITEL, P.; DEITEL, H.; DEITEL, A. *Android*: como programar. 2. ed. Porto Alegre: Bookman, 2015. 690 p.

DEITEL, P.; DEITEL, H.; WALD, A. *Android 6 para programadores*: uma abordagem baseada em aplicativos. 3. ed. Porto Alegre: Bookman, 2016. 618 p.

SIMON, J. *Head first Android development*. Sebastopol: O'Reilly, 2011. 608 p.

TAREFAS e pilha de retorno. *Android Developers*, [*S. l.*], 2019. Disponível em: https://developer.android.com/guide/components/tasks-and-back-stack.html?hl=pt-BR. Acesso em: 26 jun. 2019.

Navegação entre telas

Objetivos de aprendizagem

Ao final deste texto, você deve apresentar os seguintes aprendizados:

- Descrever a utilização de mais de uma *activity*.
- Exemplificar o uso de navegação entre telas.
- Demonstrar a navegação em mais de uma *activity*.

Introdução

Neste capítulo, você aprofundará seus conhecimentos sobre as *activities*, compreendendo melhor suas características, funcionalidades e requisitos para a criação de aplicações usem duas ou mais *activities*.

Como você estudou nos capítulos anteriores, as *activities*, representadas por interfaces gráficas com o usuário, possibilitam que este interaja com a aplicação móvel desenvolvida para *tablets* e *smartphones*, a partir da adição de componentes de texto, botões, mapas, imagens e diversos outros, a fim de que a aplicação mantenha uma aparência agradável e intuitiva. Entretanto, em virtude do tamanho reduzido em geral dos dispositivos móveis, inevitavelmente há uma restrição do espaço disponível para disponibilizar as informações requeridas e exigidas do e pelo usuário.

São justamente as limitações impostas por esses tipos de dispositivos que tornam não somente importante, mas também necessário o desenvolvimento de aplicações que permitam a navegação por mais de uma tela, ou *activity*.

Apps mobile com duas ou mais *activities*

Todo desenvolvedor — de aplicações *desktop*, Web ou *mobile* — terá uma preocupação em comum: a de desenvolver uma aplicação que apresente uma interface gráfica agradável, de fácil utilização (intuitiva) e que atenda às expectativas dos usuários quanto aos dados que apresenta.

Entretanto, diferentemente do que ocorre no desenvolvimento de aplicações não direcionadas para dispositivos móveis, o desenvolvedor *mobile* depara-se com um complexo obstáculo, já que, em razão de suas características, dispositivos móveis como *tablets* e *smartphones* (ainda que muito evoluídos) ainda apresentam uma série de restrições relacionadas a espaço de armazenamento, poder de processamento, memória e resolução de tela, sendo o último o foco deste material.

Hoje, sabe-se que existem diversos modelos de dispositivos que operam a plataforma Android, cada um aspectos próprios, e que, felizmente, a própria plataforma de desenvolvimento e o ambiente Android garantem que as aplicações se adaptem às diferentes resoluções fornecidas pelos dispositivos disponíveis no mercado. Entretanto, a falta de espaço ainda é um problema, obrigando o desenvolvedor, em muitos casos, a desenvolver aplicações que atuem não somente com uma, mas duas ou mais telas, ou seja, *activities*, separando, assim, a informação em grupos temáticos e permitindo o acesso do usuário de forma gradativa na medida em que avança na utilização da aplicação *mobile* (GUIMARÃES; TAVARES, 2014).

Adicionar novas *activities* ao projeto

Após definido o projeto de navegação para a aplicação e o modelo das interfaces gráficas com o usuário, pode-se iniciar o efetivo desenvolvimento da aplicação *mobile*. A essa altura, entendemos que você já deve ter criado seu primeiro exemplo de aplicação *mobile* e que não apresentará dificuldade em iniciar projetos e compreender os passos a seguir.

Uma vez aberto o projeto e que você esteja na tela de desenvolvimento, navegue pelas pastas (pacotes) do projeto, geralmente dispostas no canto esquerdo da tela. Agora, que está localizado em sua IDE (ambiente de desenvolvimento integrado, do inglês *integrated development environment*), com o botão direito do *mouse* clique sobre a pasta raiz da aplicação; depois, no menu que será apresentado, escolha a opção *new* e, logo em seguida, a opção `Activity`. A seguir, será apresentada uma série de opções de *activities*, as mesmas exibidas quando você inicia um novo projeto para o Android, entre as quais pode escolher segundo a sua preferência, embora, para este material, sugiramos a `Empty Activity`. Todo o procedimento descrito pode ser observado na Figura 1.

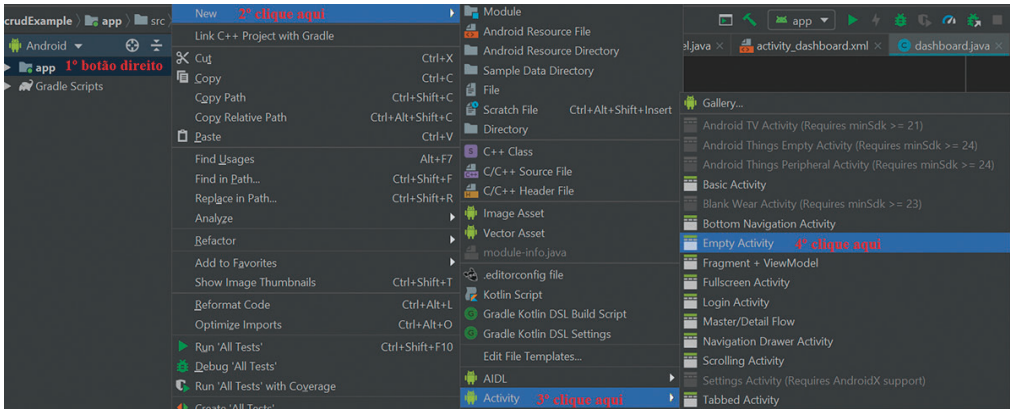


Figura 1. Adicionando uma nova *activity* ao projeto.

Antes que sua nova *activity* esteja pronta para utilização, você deverá escolher um nome para ela, definir se sua IDE de programação deverá ou não criar automaticamente o arquivo `layout.xml` dessa nova *activity* (o recomendado é que marque sim) e delimitar se ela deverá ou não ser lançada (*Laucher Activity*), o recomendado é deixar esta opção desmarcada. Feito isso, sua nova *activity* estará inserida em seu projeto e pronta para ser programada.

Nas versões mais recentes do Android Studio (ferramenta utilizada nos exemplos deste material), o programador pode escolher entre as linguagens Java e Kotlin.

- **Java:** pelo fato de todos os exemplos deste material estarem baseados na linguagem Java, sempre que iniciar um novo projeto e/ou adicionar uma nova *activity* ao seu projeto, lembre-se de selecioná-la como sua linguagem de programação.
- **Kotlin:** relativamente nova, tem se mostrado uma proposta da Google para tornar o desenvolvimento de aplicações para Android mais simples e amigável. Segundo a Google, é compatível com qualquer versão do Android a partir da 3.0, mas interoperável com Java, além de concisa, robusta, segura, portátil e fácil de utilizar e aprender.

Navegar entre duas ou mais *activities*

Agora que sua aplicação já tem mais que uma *activity*, você pode escrever códigos que permitam a navegação entre elas, ou seja, realizado determinado evento, o usuário será direcionado de uma tela A para uma tela B.

Para poder realizar a mudança de contexto, isto é, transferir o usuário de uma tela para outra, é preciso primeiro ter uma segunda tela construída: para facilitar o seu entendimento, supomos que seu projeto tenha uma *activity* a que chamaremos `MainActivity` e outra *activity* que denominaremos `telaSecundaria`, de modo que o código da classe Java da `MainActivity` seja semelhante ao código apresentado a seguir.

```
1. Import android.content.Intent;
2. public class MainActivity extends AppCompatAc-
   tivity {
3.  private Button btAbrirSegundaTela;
4.  @Override
5.  protected void onCreate(Bundle savedInstanceS-
   tate) {
6.  super.onCreate(savedInstanceState);
7.  setContentView(R.layout.activity_main);
8.  btAbrirSegundaTela = (Button) findViewById(R.
   id.btSegundaTela);
9.  btAbrirSegundaTela.setOnClickListener(new View.
   OnClickListener(){
10. @Override
11. public void onClick(View v) {
12. Intent segundaTela = new Intent(MainActivity.
   this, telaSecundaria.class);
13. startActivity(segundaTela);
14. }
15. });
16. }
17. }
```

Observe, no código apresentado, as linhas 12 e 13: isso é o que podemos chamar de "pulo do gato". Essas duas linhas têm a função de iniciar e apresentar a nova tela ao usuário, cujo componente responsável é o objeto `Intent`. *Intents* constituem objetos de mensagem que podem ser utilizados para solicitar ações de outros componentes do aplicativo (DEITEL, DEITEL; WALD, 2016).

Um `Intent` pode ser declarado de maneira explícita ou implícita, cujas principais diferenças são apresentadas no Quadro 1.

Quadro 1. *Intent* explícito x *Intent* implícito

INTENT	
Implícito	**Explícito**
Não especificam nenhum componente pelo nome; declaram uma ação geral a ser realizada; permitem que componentes de outros aplicativos processem as ações Por exemplo: se você deseja exibir ao usuário uma localização em um mapa, pode usar um *intent* implícito para solicitar que outro aplicativo exiba uma localização especificada no mapa	Especificam o componente a ser iniciado pelo nome; normalmente é utilizado para iniciar componentes no próprio aplicativo Por exemplo: para iniciar uma nova atividade em resposta a uma ação do usuário ou para iniciar um serviço para baixar um arquivo em segundo plano

Fonte: Adaptado de Android (2019, documento *on-line*).

No código apresentado no Quadro 1, optou-se pela utilização de um ***Intent* explícito**, tipo que sempre será utilizado quando for realizar chamadas para serviços internos à sua aplicação. Um *Intent* explícito é invocado com parâmetros que indicam o contexto e a classe da nova intenção; já em um ***Intent* implícito**, a passagem de parâmetros é um pouco diferente, com uma sintaxe que seria algo como: `Intent nomeDaIntent = new Intent`("caminho CompletoDoServiço"). Vamos deixar um pouco mais simples com um exemplo.

```
Intent novaTela = new Intent("br.com.exemplo.action.NOVATELA");
```

Feito dessa forma, a única informação que o programa tem é de que dentro do pacote `br.com.exemplo.action` existe uma possível classe `NOVATELA` que ele (o programa) deverá instanciar e iniciar.

Fique atento

Quando estiver desenvolvendo seu aplicativo e precisar utilizar um `Intent` para iniciar um serviço (*service*), busque sempre utilizá-lo na sua forma explícita. Ao empregar um `Intent` implícito para esse tipo de operação, cria-se uma brecha de segurança, pois não é possível determinar qual serviço que responderá ao `Intent`, além do fato de que o usuário não poderá ver qual serviço foi iniciado (ANDROID..., 2019, documento *on-line*).

No caso do código apresentado, foi utilizado o objeto `Intent` para a criação de uma instância de uma nova "intenção": o primeiro parâmetro de uma nova intenção, ou `Intent`, representa o contexto atual, ou seja, a classe ativa no momento — fato importante porque os *intents* são utilizados não somente para abertura de telas, mas também para outras atividades —; como segundo parâmetro de uma nova intenção, deve-se colocar o nome completo do arquivo `.class` da nova tela que será aberta. Utiliza-se o arquivo `.class`, pois, depois de compilados, os arquivos Java tornam-se arquivos `.class`. Por fim, na linha 13 é utilizado o método `startActivity` para iniciar a nova *activity*, tornando-a ativa em sua aplicação.

Enviar dados entre duas telas

Nesse momento, em que seu projeto tem duas ou até mesmo mais *activities*, você deve querer transmitir informações entre elas, o que é completamente normal, já que não existiria muita funcionalidade em uma aplicação com diversas *activities* se não fosse possível trocar informações entre elas.

Para essa troca de informações entre duas ou mais *activities*, você precisará usar, além de um objeto `Intent`, um objeto `Bundle`, uma classe que dá origem a objetos do mesmo tipo, que consiste em um mapeamento do tipo chave/valor (sendo a chave uma *string* e o valor qualquer tipo primitivo ou até mesmo objetos empacotáveis) e cuja função é possibilitar o envio e o recebimento de dados pela aplicação.

Assim, voltemos ao nosso exemplo anterior, imaginando agora que alterações foram realizadas na tela principal (no caso, na `MainActivity`), sendo adicionados a ela um campo de texto para o nome do usuário, um campo de texto para o e-mail do usuário e um campo de data para a data de nascimento do usuário, cada um com os seguintes nomes, respectivamente, `textoNome`, `textoEmail` e `dataDtNascimento`. Desenvolva sua tela

principal e sua `MainActivity` de modo que na *activity* sejam inseridos os campos de texto citados e, também, um botão, ao qual chamaremos de `btEnviaDados`. Ao terminar, sua classe `MainActivity` deve parecer-se com o código apresentado a seguir.

```
1. Import android.content.Intent;
2. Import android.os.Bundle;
3. public class MainActivity extends AppCompatAc-
   tivity {
4.  private Button btEnviaDados;
5.  private EditText textoNome;
6.  private EditText textoEmail;
7.  private EditText dataDtnascimento;
8.  @Override
9.  protected void onCreate(Bundle savedInstanceS-
   tate) {
10. super.onCreate(savedInstanceState);
11. setContentView(R.layout.activity_main);
12. btEnviaDados = (Button) findViewById(R.id.
   btEnviaDados);
13.  textoNome = (EditText) findViewById(R.id.
   textoNome);
14.  textoEmail = (EditText) findViewById(R.id.
   textoEmail);
15. dataDtnascimento = (EditText) findViewById(R.
   id. dataDtnascimento);
16. btEnviaDados.setOnClickListener(new View.On-
   ClickListener(){
17. @Override
18. public void onClick(View v) {
19. String nome = textoNome.getText().toString();
20. String email = textoEmail.getText().toString();
21.  String dataNascimento = dataDtnascimento.get-
   Text().toString();
22. Bundle parametros = new Bundle();
23. parametros.putString("nome", nome);
24. parametros.putString("email", email);
25. parametros.putString("dtNasc", dataNascimento);
```

```
26.  Intent segundaTela = new Intent(MainActivity.
     this, telaSecundaria.class);
27.  segundaTela.putExtras(parametros);
28.  startActivity(segundaTela);
29.  }
30.  });
31.  }
32. }
```

Diferentemente da primeira versão apresentada para esse código, neste temos a inserção do objeto parâmetros como instância da classe `Bundle`, objeto responsável pelo armazenamento da lista de dados que serão transferidos entre as atividades (telas). Nesse código, o propósito é que você não somente reforce seu aprendizado recuperando dados da tela e os armazene em variáveis, como também entenda como transferir tais dados de uma tela para outra.

Note que na linha 22 foi criado o objeto `parametros` do tipo *undle*, objeto que armazenará os valores recuperados, cuja sintaxe-padrão para inserção de itens em sua lista de dados é `<nome_do_objeto>.<comando_de_inserção_e_tipo_de_dado>`(“chave”, valor). Assim, como pode ser observado nas linhas 23, 24 e 25 — para inserir um objeto, por exemplo, você usaria o comando de inserção de dados `putParcelable` —, após inseridos todos os itens desejados, utilize o método `putExtas` do objeto `Intent` (nesse caso, `segundaTela.putExtas`) e, como parâmetro, insira seu objeto `Bundle`.

Desse modo, terminamos a primeira parte, mas não todo o código. Agora que você terminou a implementação da classe `MainActivity`, é hora de implementar a classe `Java` da *activity* `telaSecundaria`; depois de pronta, seu código deve ser parecido com o apresentado a seguir.

```
1.  import android.content.Intent;
2.  import android.os.Bundle;
3.  public class telaSecundaria extends AppCompa-
    tActivity {
4.   private TextView textonome;
5.   private TextView textoemail;
6.   private TextView textoNascimento
7.   @Override
8.   protected void onCreate(Bundle savedInstanceS-
    tate) {
```

```
9.  super.onCreate(savedInstanceState);
10. setContentView(R.layout.activity_dashboard);
11.  textonome = (TextView) findViewById(R.id.
    textonome);
12.  textoemail = (TextView) findViewById(R.id.
    textoemail);
13.  textoNascimento = (TextView) findViewById(R.
    id. textoemail);
14. Intent telaAnterior = getIntent();
15. if(telaAnterior!= null){
16. Bundle params = telaAnterior.getExtras();
17. if(params != null){
18. String nome = params.getString("nome");
19. String email = params.getString("email");
20. String nascimento = params,getString("dtNasc");
21. textonome.setText(nome);
22. textoemail.setText(email);
23. textoNascimento.setText("dtNasc");
24. }
25. }
26. }
27.}
```

Note que o código não é complexo e muitos dos comandos são até mesmo semelhantes àqueles da classe `MainActivity`. Observe então as linhas em negrito, iniciando pela linha 14, nas quais se cria um objeto `Intent`, mas que, diferentemente do que ocorre quando a intenção é iniciar uma nova atividade, você criará um objeto `Intent` que receberá uma referência de uma intenção preexistente, no caso uma cópia da instância do objeto `Intent` criado na classe `MainActivity`.

Na linha 16, é criado um objeto `Bundle`; assim como na classe `MainActivity` você criou um `Bundle` para concentrar os itens que deveriam ser transferidos entre as telas, nessa classe `telaSecundaria` cria-se novamente o objeto para receber e armazenar os dados. Observe que, dessa vez, ao criarmos o objeto, estamos utilizando o comando `telaAnterior.getExtras()`, ou seja, a aplicação entende que deve recuperar os dados provenientes da tela que fez a chamada para a tela atual.

Saiba mais

Para transferir dados entre telas, você não utilizará obrigatoriamente um objeto da classe `Bundle`, já que as próprias *intents*, por meio de técnicas de sobrecarga de métodos, são capazes de suprimir a necessidade dos Bundles.

Todo objeto `Intent` apresenta os métodos `putExtras`("chave", "valor") — `public Intent puxExtra (String name, int[] value)` — para enviar dados entre as *activities*, além de métodos `get` que variarão de acordo com o tipo de dado que se deseja recuperar.

Alguns possíveis métodos `get` são `getStringExtra`, `getDoubleExtra` e `getIntExtra`, os quais devem ter os mesmos parâmetros, ou seja, é preciso identificar qual é a chave do valor que se deseja recuperar com o método `get`. Por exemplo: `getStringExtra`("razao_social").

Para maiores informações sobre os *intents* e como enviar e receber dados diretamente da `Intent` sem a necessidade de se utilizar um objeto `Bundle`, acesse os *links* a seguir.

https://qrgo.page.link/c3uSn

https://qrgo.page.link/Wf9xZ

Aplicativos multitelas

Naturalmente, você pode estar se perguntando: se existem tantos *layouts* diferentes, alguns até mesmo com a possibilidade de adicionar algum tipo de *slider* ou apresentação de informações de modo que estejam escondidas e, depois, apareçam, por que criar aplicativos com múltiplas telas? A resposta é simples: a cada dia que passa, as aplicações para Android têm ficado mais sofisticadas, em parte porque os aparelhos vêm se modernizando, mas também pelo fato de que cada vez mais as pessoas trocam computadores por *smartphones* (DEITEL, DEITEL; DEITEL, 2015).

Link

Os melhores materiais para pesquisa e entendimento sobre os componentes e recursos de qualquer plataforma e/ou linguagem de programação sempre serão seu material oficial.

Certamente, outros livros, cursos e tutoriais condensam tais informações e as tornam mais didáticas, mas sempre vale a pena visitar o material oficial.

No *link* a seguir, você terá informações completas sobre interfaces gráficas com o usuário utilizando o Android.

https://qrgo.page.link/p86GH

São muitos os estudos que mostram que os usuários preferem navegar entre páginas a ficar "rolando tela" para encontrar a informação de que precisam, principalmente quando estão lidando com uma tela de tamanho reduzido, o que cria até mesmo certo desconforto.

Por isso, e por outros motivos, o desenvolvimento de aplicações com múltiplas telas mostra-se tão importante, motivo exatamente pelo qual você deve saber como desenvolver esse tipo de aplicação, tentando promover sua capacidade de desenvolver interfaces gráficas com o usuário amigáveis, intuitivas e práticas. A seguir, recomendamos que observe algumas imagens sobre um conjunto de telas de navegação de determinado aplicativo móvel.

Link

Cientes da importância das interfaces gráficas e das possibilidades trazidas pelo recurso de navegação entre telas, a própria Google lançou um componente capaz de facilitar e trazer novas experiências para a construção de interfaces gráficas: o Android Jetpack. Por suas extensas funcionalidades, sugerimos que entre no *link* da ferramenta para que possa conhecê-la melhor.

https://qrgo.page.link/kHAFc

Referências

ANDROID Developers. [*S. l.*: *S. n.*], 2019. Disponível em: https://developer.android.com. Acesso em: 19 jun. 2019.

DEITEL, P.; DEITEL, H.; DEITEL, A. *Android*: como programar. 2. ed. Porto Alegre: Bookman, 2015.

DEITEL, P.; DEITEL, H.; WALD, A. *Android 6 para programadores*: uma abordagem baseada em aplicativos. 3. ed. Porto Alegre: Bookman, 2016.

GUIMARÃES, A. P. N.; TAVARES, T. A. Avaliação de Interfaces de Usuário voltada à Acessibilidade em Dispositivos Móveis: Boas práticas para experiência de usuário. *In*: SIMPÓSIO BRASILEIRO DE SISTEMAS MULTIMÍDIA E WEB, 20., 2014, João Pessoa. *Anais* [...]. João Pessoa: Centro de Convenções, 2014. p. 22–29. Disponível em: http://www.lbd.dcc.ufmg.br/bdbcomp/servlet/Trabalho?id=21533. Acesso em: 19 jun. 2019.

Leitura recomendada

INTENTS e filtros de Intents. ANDROID Developers, [*S. l.*], 2019.Disponível em: https://developer.android.com/guide/components/intents-filters?hl=pt-br. Acesso em: 19 jun. 2019.

Persistência com SQLite

Objetivos de aprendizagem

Ao final deste texto, você deve apresentar os seguintes aprendizados:

- Conceituar a persistência no âmbito do sistema Android.
- Exemplificar os principais comandos de persistência.
- Aplicar *scripts* de persistência para futuros projetos.

Introdução

Neste capítulo, você aprenderá sobre a persistência de dados em aplicações Android utilizando o sistema de banco de dados SQLite. Esse recurso é um dos mais importantes de qualquer aplicação, graças ao qual os dados gerados por uma aplicação podem ser armazenados para futura utilização, ou seja, transferem-se os dados da memória principal (cache), também temporária, para a memória permanente.

A persistência de dados no desenvolvimento para Android pode se dar por diversos meios, e não necessariamente pelo SQLite, embora este seja, por diversos motivos que veremos ao longo do material, o mais indicado. Ainda, você se familiarizará com os comandos de persistência e *scripts* para aplicação em projetos futuros.

Persistência de dados no Android

Um dos recursos mais importantes de qualquer plataforma de programação refere-se à possibilidade de persistir dados. Em diversas ocasiões, será necessário o armazenamento permanente dos dados, principalmente em aplicações nas quais é preciso realizar cadastros (DEITEL; DEITEL; DEITEL, 2015).

A persistência de dados consiste basicamente em armazenar dados gerados pelo aplicativo — em memória secundária — que sejam ou que possam ser necessários no futuro, ou seja, a persistência dos dados refere-se à capacidade de armazenar dados (de qualquer tipo) na memória secundária do dispositivo, para futura utilização pelo aplicativo.

Link

Para saber mais sobre as formas e tecnologias de persistência de dados na plataforma Android, visite o próprio *site* da plataforma pelo *link*, a partir do qual você terá informações completas sobre armazenamento de dados no Android.

https://qrgo.page.link/eRyVX

Uma aplicação de contas pessoais, uma aplicação que lhe ajude a encontrar os melhores valores para determinados produtos, um aplicativo de gestão financeira, controle de consumo de combustível para um veículo, enfim, são várias as situações nas quais se verificará a utilidade da persistência de dados. E, mesmo com finalidades diferentes, todos os aplicativos sugeridos têm algo em comum: necessitam utilizar algum recurso de persistência de dados.

Em Android, a persistência de dados pode ocorrer das seguintes formas (OPÇÕES..., 2019, documento *on-line*):

- preferências compartilhadas — armazena dados primitivos privados em pares do tipo chave — valor;
- armazenamento interno — armazena dados privados na memória do dispositivo;
- armazenamento externo — armazena dados públicos no armazenamento externo compartilhado;
- banco de dados SQLite — armazena dados estruturados em um banco de dados privado;
- conexão de rede — armazena dados na *web* com seu próprio servidor de rede.

Entre as opções apresentadas, exploraremos neste capítulo o banco de dados SQLite, a melhor opção diante dos demais tipos, pois recebe suporte nativo da plataforma Android.

Android e SQLite

Apesar de ter total suporte no ambiente Android, o banco de dados SQLlite não é um produto desenvolvido para essa plataforma, tampouco para a Google. Ele é totalmente escrito na linguagem C, a mesma linguagem na qual se desenvolve a plataforma Android, tornando-se um dos principais motivos de seu suporte nativo (DEITEL; DEITEL; WALD, 2016).

O SQLite utiliza somente uma única classe sem qualquer configuração de SQL, além do fato de nem todas as funções estarem configuradas no SQLite. Trata-se de um banco de dados de domínio público, totalmente gratuito, cujas fontes podem ser obtidas por seu próprio site, sendo modificadas e compiladas (ABOUT..., 2019, documento *on-line*).

Em resumo, pode-se dizer que as principais características do SQLite são as seguintes:

- é totalmente suportado pelas principais versões do Android;
- qualquer banco de dados poderá ser acessado por seu nome a partir de qualquer classe da aplicação, mas não poderá ser acessado fora dela;
- trata-se de uma biblioteca ***open source*** implementada na linguagem C, um banco de dados embutido sem servidor e que realiza os procedimentos de leitura e escrita diretamente no arquivo;
- sua utilização não requer qualquer tipo de configuração ou processo administrativo — você somente precisará definir os comandos SQL para criar e atualizar o banco de dados. O SQLite é gerenciado automaticamente pela plataforma Android.

Interação com o SQLite

Por indicação da própria Google, para criar um novo banco de dados deve-se utilizar a subclasse `SQLiteOpenHelper` e modificar seu método `onCreate()`, a partir do qual é possível executar o comando SQLite que criará as tabelas no banco de dados.

O SQLite também possibilita a consulta de dados, para o qual se deve utilizar os métodos `query()` do `SQLiteDatabase`, que, por sua vez, aceitam diversos parâmetros de consulta, como tabela a ser consultada, projeção, seleção, agrupamentos, etc. Contudo, caso você deseje realizar uma consulta mais complexa, como aquelas com "***aliases***" de coluna, deverá utilizar o

`SQLiteQueryBuilder`, que fornece vários métodos para a sua criação (OPÇÕES..., 2019, documento *on-line*).

Todas as consultas do SQLite retornam um **cursor**, que aponta para as linhas que foram retornadas pela consulta. Esse **cursor** constituirá o mecanismo que você deverá utilizar para navegar pelos resultados gerados pela consulta dos dados, ou seja, será utilizado para ler as linhas e colunas.

A partir do construtor que você definiu, poderá obter uma instância de implementação de `SQLiteOpenHelper`. Para que consiga realizar operações de gravação e leitura no banco de dados, invoque, respectivamente, os métodos `getWritableDatabase()` e `getReadableDatabase()`, os quais retornarão um objeto `SQLiteDatabase`, que representa o banco de dados e fornece os métodos para as operações do `SQLite`.

O melhor modo de entender o funcionamento dos comandos de interação entre o Android e o SQLite se dá por meio da apresentação de códigos práticos, embora se deva atentar para o fato de que os códigos apresentados aqui são apenas parcelas de uma pequena aplicação.

Saiba mais

O portal iMasters é um dos mais completos portais de conteúdo relacionados à tecnologia da informação, oferendo, inclusive, certificados para programadores reconhecidos por órgãos federais (como a Força Aérea Brasileira). Neste portal, você encontrará um ótimo tutorial de como criar um CRUD (Cadastro, Recuperação, Updade e Delete) utilizando Android e SQLite.

Para ter acesso à primeira parte do artigo/tutorial, a partir da qual poderá navegar gratuitamente para as demais, acesse o *link* a seguir.

https://qrgo.page.link/pFcfP

Primeiro, crie a classe que será responsável pela conexão/criação do seu banco de dados, na qual colocaremos os comandos de interação com o banco de dados. A princípio, o código da sua classe deve se assemelhar ao que é apresentado a seguir.

```
public class MeuDatabase extends SQLiteOpenHelper {


    public MeuDatabase(@Nullable Context context) {
        super(context,  name: "agenda",  factory: null,  version: 1);
    }

    @Override
    public void onCreate(SQLiteDatabase db) {
        db.execSQL("create table contatos (id integer primary key autoincrement, nome varchar(40), numero varchar (11));");
    }

    @Override
    public void onUpgrade(SQLiteDatabase db, int oldVersion, int newVersion) {

    }
```

Nesse exemplo, a classe chama-se `MeuDatabase` e estende as funcionalidades da classe `SQLiteOpenHelper`, como sugerido anteriormente. O método construtor da classe envia como parâmetros o nome do banco de dados (nesse caso, **agenda**, um *factory* que deixamos como **null** e, na versão, colocamos 1).

Novamente como sugerido, sobrescrevemos o método `onCreate`, que executará o comando que criará a sua tabela. Em detalhes, foi criada a tabela **contatos**, com campo **id** do tipo **integer**, sendo **primary key** e **autoincrement**, além dos campos **nome** do tipo **varchar (40)** e o **número** do tipo **varchar (11)**. O método `onUpgrade` pode ficar vazio, quando se deseja atualizar o banco de dados.

Fique atento

Para poder realizar todas as operações necessárias/desejadas utilizando o SQLite, você deverá adicionar as seguintes classes à seção `import` de sua classe:

```
import android.content.ContentValues;
import android.content.Context;
import android.database.Cursor;
import android.database.sqlite.SQLiteDatabase;
import android.database.sqlite.SQLiteOpenHelper;
import android.support.annotation.Nullable.
```

Outros métodos inseridos na classe foram os métodos de inserção de dados e, também, o de recuperação de dados. Primeiro, analisaremos o método responsável pela inserção de dados no banco de dados.

```
27      public void insereContato(String nome, String numero){
28          SQLiteDatabase sqLiteDatabase = this.getWritableDatabase();
29
30          ContentValues contentValues = new ContentValues();
31
32          contentValues.put("nome", nome);
33          contentValues.put("numero", numero);
34
35          sqLiteDatabase.insert( table: "contatos", nullColumnHack: null,contentValues);
36      }
```

Para inserir novos registros no banco de dados, recomendamos que você utilize o já citado objeto `getWritableDatabase` associado a um objeto `ContentValues`. O objeto `ContentValues` será responsável por armazenar os valores inseridos no banco de dados pelo método `insert` objeto `getWritableDatabase`. O primeiro parâmetro do método `insert` será a tabela na qual os valores deverão ser inseridos, o segundo o deixamos como `null` (padrão) e o terceiro serão os valores.

A próxima operação que verificaremos consiste na remoção de um registro. O código pode ser observado logo a seguir.

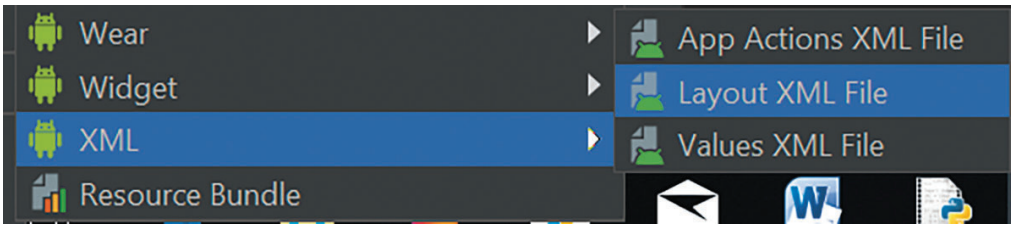
```
38      public int removeContato(String id){
39          SQLiteDatabase sqLiteDatabase = this.getWritableDatabase();
40          return sqLiteDatabase.delete( table: "contatos", whereClause: "id = ?",new String[]{String.valueOf(id)});
41      }
```

Sem dúvidas, o código para remoção de um registro é o mais simples, bastando somente invocar o método `delete` e informar o id do registro que se deseja remover. Por fim, veremos o método de atualização de registro. Observe o código a seguir.

```
43      public int updateContato(String nome, String numero, int id){
44          SQLiteDatabase sqLiteDatabase = this.getWritableDatabase();
45          ContentValues contentValues = new ContentValues();
46          contentValues.put("nome",nome);
47          contentValues.put("numero",numero);
48          return sqLiteDatabase.update( table: "contatos", contentValues, whereClause: "id = ?",new String[]{String.valueOf(id)});
49      }
```

Para atualizar um registro, você usará os mesmos recursos empregados para a inserção de um novo registro, entretanto utilizará o método `update` no lugar do método `insert`, além de informar a condição da cláusula `WHERE`, semelhantemente ao que é feito no método de remoção. Todas essas operações são básicas da interação entre o SQLite e o Android.

Nesse momento, em que observamos as operações de inserção e remoção de dados, seguiremos para as operações de busca, alteração e listagem dos dados. Iniciaremos com o conjunto de códigos e operações necessários para a apresentação da lista de registros armazenados no banco de dados. Para reproduzir a proposta desse material, você utilizará os seguintes componentes: uma nova `Activity`, na qual será inseria um `RecyclerView`; e um arquivo `Layout XML`, em que será inserido um componente `CardView`.



A nova `Activity` será chamada `activity_display` e o arquivo `Layout XML`, `lista_item.xml`. A seguir, observe como ficará o XML de cada um dos componentes.

ACTIVITY_DISPLAY.XML

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout  xmlns:android="http://schemas.android.com/apk/
res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout _ width="match _ parent"
    android:layout _ height="match _ parent"
    tools:context=".DisplayActivity">
```

```
    <android.support.v7.widget.RecyclerView
        android:id="@+id/rev"
        android:layout _ width="match _ parent"
        android:layout _ height="match _ parent">
    </android.support.v7.widget.RecyclerView>

</LinearLayout>
```

LAYOUT XML

```
<?xml version="1.0" encoding="utf-8"?>
<android.support.v7.widget.CardView xmlns:android="http://sche-
mas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout _ width="match _ parent"
    android:layout _ height="100dp"
    android:layout _ margin="20dp"
    app:cardMaxElevation="20dp">

    <android.support.constraint.ConstraintLayout
        android:layout _ width="match _ parent"
        android:layout _ height="match _ parent">

        <ImageButton
            android:id="@+id/btEdit"
            android:layout _ width="42dp"
            android:layout _ height="42dp"
            android:layout _ marginStart="8dp"
            android:layout _ marginTop="4dp"
          app:layout _ constraintStart _ toEndOf="@+id/tvNumero"
          app:layout _ constraintTop _ toBottomOf="@+id/tvNome"
          app:srcCompat="@android:drawable/ic _ menu _ edit" />
```

```
        <TextView
            android:id="@+id/tvNumero"
            android:layout _ width="250dp"
            android:layout _ height="40dp"
            android:layout _ marginStart="8dp"
            android:layout _ marginTop="4dp"
            android:layout _ marginEnd="8dp"
            android:text="TextView"
            app:layout _ constraintEnd _ toEndOf="parent"
            app:layout _ constraintHorizontal _ bias="0.0"
            app:layout _ constraintStart _ toStartOf="parent"
         app:layout _ constraintTop _ toBottomOf="@+id/tvNome" />

        <TextView
            android:id="@+id/tvNome"
            android:layout _ width="353dp"
            android:layout _ height="40dp"
            android:layout _ marginStart="8dp"
            android:layout _ marginTop="4dp"
            android:layout _ marginEnd="8dp"
            android:text="TextView"
            app:layout _ constraintEnd _ toEndOf="parent"
            app:layout _ constraintHorizontal _ bias="0.0"
            app:layout _ constraintStart _ toStartOf="parent"
            app:layout _ constraintTop _ toTopOf="parent" />

        <ImageButton
            android:id="@+id/btDelete"
            android:layout _ width="42dp"
            android:layout _ height="42dp"
            android:layout _ marginStart="8dp"
            android:layout _ marginTop="4dp"
           app:layout _ constraintStart _ toEndOf="@+id/btEdit"
          app:layout _ constraintTop _ toBottomOf="@+id/tvNome"
            app:srcCompat="@android:drawable/ic _ delete" />
    </android.support.constraint.ConstraintLayout>

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

A partir desse momento, em que as telas estão criadas, é preciso criar as condições para preencher esses componentes. Para isso, duas novas classes foram criadas, a `Adaptador.java`, responsável por criar o objeto `Adapter` com os dados para preencher a lista de `CardView`'s, que, por sua vez será inserido no `RecyclerView`; e a `GetterSetter.java`, que funcionará como intermediária entre a classe `Adaptador` e a classe `MeuDatabase`.

Observe a seguir a facilidade da implementação da classe `GetterSetter`.

```
public class GetterSetter {
    String nome;
    String numero;
    String id;

    public GetterSetter(String nome, String numero, String id){
        this.nome   = nome;
        this.numero = numero;
        this.id     = id;
    }
    public String getNome() { return nome; }

    public String getNumero() {
        return numero;
    }

    public void setNome(String nome) { this.nome = nome; }

    public void setNumero(String numero) { this.numero = numero; }

    public String getId() { return id; }

    public void setId(String id) { this.id = id; }
}
```

Como dito, a classe é apenas uma intermediária que facilitará as operações de recuperação e definição dos dados recuperados no banco de dados. Para inserir tais dados no `RecyclerView`, precisamos preencher um `Adapter` e, depois, associá-lo ao `RecyclerView`. Para isso, você terá que criar uma classe estendida de `RecyclerView.Adapter`, como demonstrado a seguir:

```
1.  public class Adaptador extends RecyclerView.
Adapter<Adaptador.MyRecycler>{
2.
3.      Context context;
4.      ArrayList<GetterSetter> lista;
5.      MeuDatabase mdb;
6.
```

```
7.          public Adaptador(Context context,
ArrayList<GetterSetter> lista) {
8.              this.context = context;
9.              this.lista = lista;
10.         }
11.
12.         @NonNull
13.         @Override
14.        public MyRecycler onCreateViewHolder(@Non-
Null ViewGroup parent, int i) {
15.           View v = LayoutInflater.from(parent.getCon-
text()).inflate(R.layout.lista_item,parent,false);
16.             return new MyRecycler(v);
17.         }
18.
19.         @Override
20.          public void onBindViewHolder(@NonNull
MyRecycler myRecycler, int position) {
21.          final GetterSetter gs = lista.get(position);
22.           myRecycler.tvNome.setText(gs.getNome());
23.                 myRecycler.tvNumero.setText(gs.
getNumero());
24.          myRecycler.btDelete.setOnClickListener(new
View.OnClickListener() {
25.                 @Override
26.                 public void onClick(View v) {
27.                         mdb = new MeuDatabase(v.
getContext());
28.                    mdb.removeContato(gs.getId());
29.                 }
30.             });
31.         }
32.
33.         @Override
34.         public int getItemCount() {
35.             return lista.size();
36.         }
37.
```

```
38.      public class MyRecycler extends Recycler-
View.ViewHolder{
39.          TextView tvNome, tvNumero;
40.          ImageButton btEdit, btDelete;
41.          public MyRecycler(View itemView) {
42.              super(itemView);
43.              tvNome   = itemView.findViewById(R.
id.tvNome);
44.              tvNumero = itemView.findViewById(R.
id.tvNumero);
45.              btEdit   = itemView.findViewById(R.
id.btEdit);
46.              btDelete = itemView.findViewById(R.
id.btDelete);
47.          }
48.      }
49.  }
```

Praticamente todos os métodos da classe são criados automaticamente, bastando somente preenchê-los. Na linha 15, cria-se um novo objeto View para ser inserido como parâmetro de inicialização para o objeto `MyRecycler`, referente à classe `MyRecycler` criada na linha 38. Nessa classe, é criada a ligação entre os componentes visuais que compõem o arquivo `lista_item.xml` e a classe `MyRecycler`, que, por sua vez, permitirá preencher o `RecyclerView`. O método `onBindViewHolder` é o responsável por preencher o `RecyclerView` na medida em que este se torna visível para o usuário. Nessa classe, você recuperará os dados de cada um dos itens a partir de um objeto da classe `GetterSetter`, que, por sua vez, foi preenchido como uma lista, como veremos posteriormente. É também nesse método que serão inseridos os métodos `onClickListener` dos botões de alteração e de exclusão de dados.

Agora, você pode seguir para o método que recupera os registros do banco de dados e preenche a lista de objetos `GetterSetter`. Veja o código a seguir.

```
57    public Cursor getDados(){
58        SQLiteDatabase sqLiteDatabase = this.getReadableDatabase();
59        Cursor cursor = sqLiteDatabase.rawQuery( sql: "select * from contatos", selectionArgs: null);
60        return cursor;
61    }
```

Nesse momento, você deve estar se perguntando: "mas em que momento a lista é preenchida?". Isso é feito justamente no método `onCreate` da classe `DisplayActivity.java`, criada com a `Activity display_activity.xml`. Vejamos o código.

Iniciaremos pela operação de busca, pois tanto a remoção já apresentada quanto a alteração ocorrem sobre um registro específico (na maioria dos casos). O comando é simples, como você pode ver a seguir.

```
1.  protected void onCreate(Bundle savedInstanceS-
tate) {
2.       super.onCreate(savedInstanceState);
3.      setContentView(R.layout.activity_display);
4.       rv = (RecyclerView)findViewById(R.id.rev);
5.       RecyclerView.LayoutManager layoutManager =
new LinearLayoutManager(DisplayActivity.this);
6.       rv.setLayoutManager(layoutManager);
7.      mdb = new MeuDatabase(DisplayActivity.this);
8.       cursor = mdb.getDados();
9.       if (cursor.getCount()>0){
10.          if(cursor.moveToFirst()){
11.              do{
12.                  id     = cursor.getString(0);
13.                  nome   = cursor.getString(1);
14.                  numero = cursor.getString(2);
15.                        GetterSetter gs = new
GetterSetter(nome, numero, id);
16.                  lista.add(gs);
17.              }while (cursor.moveToNext());
18.          }
19.      }
20.             Adaptador adaptador = new
Adaptador(DisplayActivity.this, lista);
21.      rv.setAdapter(adaptador);
22.  }
```

A classe `DisplayActivity` realiza a amarração entre as classes `Adaptador` e `GetterSetter`. Para finalizar, precisamos retornar à classe `MeuDatabase`. A seguir, veja como fica o código da operação de atualização.

```
42      public int updateContato(String nome, String numero, String id){
43          SQLiteDatabase sqLiteDatabase = this.getWritableDatabase();
44          ContentValues contentValues = new ContentValues();
45          contentValues.put("nome",nome);
46          contentValues.put("numero",numero);
47          return sqLiteDatabase.update( table: "contatos", contentValues,  whereClause: "id = ?",
48                                                  new String[]{String.valueOf(id)});
```

Exemplo de operação com SQLite

Nesse momento, em que já vimos um exemplo abstrato de criação de um banco de dados SQLite a partir do Android, podemos observar um exemplo prático, um pouco mais detalhado e que permitirá um melhor entendimento do funcionamento dos objetos de manipulação do banco de dados.

Scripts e limitações do SQLite

Como estudamos ao longo deste capítulo, não é por acaso que o SQLite se tornou o "queridinho" da Google para aplicações Android. Trata-se de um sistema de banco de dados extremamente leve, pequeno, robusto e com todos os recursos necessários para aplicações desenvolvidas para dispositivos móveis, ou seja, aplicações pequenas que executam preferencialmente em um ambiente local, o dispositivo do usuário.

Nesse ponto do material, você já aprendeu que o SQLite nada mais é que uma biblioteca escrita em C, e não um sistema gerenciador de banco de dados, o que significa dizer que o sistema oferece diversos recursos relacionados (seguindo o modelo) SQL 92, embora com limitações, algumas resolvidas pelo próprio Android, e outras não, devendo ser observadas quando se opta por utilizar o SQLite como um sistema de banco de dados.

Quadro 1. Comandos SQL suportados e não suportados pelo SQLite

SQLite	
COMANDOS SQL SUPORTADOS	**COMANDOS SQL NÃO SUPORTADOS**
SELECT UPDATE DELETE INSERT ALTER TABLE*	DELETE (VÁRIAS TABELAS) FOREIGN KEY TRIGGERS RIGHT JOIN FULL OUTER JOIN GRANT REVOKE

*Suporte parcial

Fonte: Query… (2019, documento *on-line*).

Apesar de suas limitações, o SQLite oferece diversos recursos observados nos sistemas de gerenciamento de bancos de dados mais modernos, como a propriedade ACID (Atomicidade, Consistência, Isolamento e Durabilidade).

As características provenientes da propriedade ACID são extremamente importantes para garantir a integridade dos dados, um problema que incomodou os programadores durante décadas.

Referências

ABOUT SQLite. *SQLite*, [*S. l.*], 2019. Disponível em: https://www.sqlite.org/about.html. Acesso em: 26 jun. 2019.

DEITEL, P.; DEITEL, H.; DEITEL, A. *Android*: como programar. 2. ed. Porto Alegre: Bookman, 2015. 690 p.

DEITEL, P.; DEITEL, H.; WALD, A. *Android 6 para programadores*: uma abordagem baseada em aplicativos. 3. ed. Porto Alegre: Bookman, 2016. 618 p.

OPÇÕES de Armazenamento. *Android Developers*, [*S. l.*], 2019. Disponível em: https://developer.android.com/guide/topics/data/data-storage?hl=pt-BR. Acesso em: 26 jun. 2019.

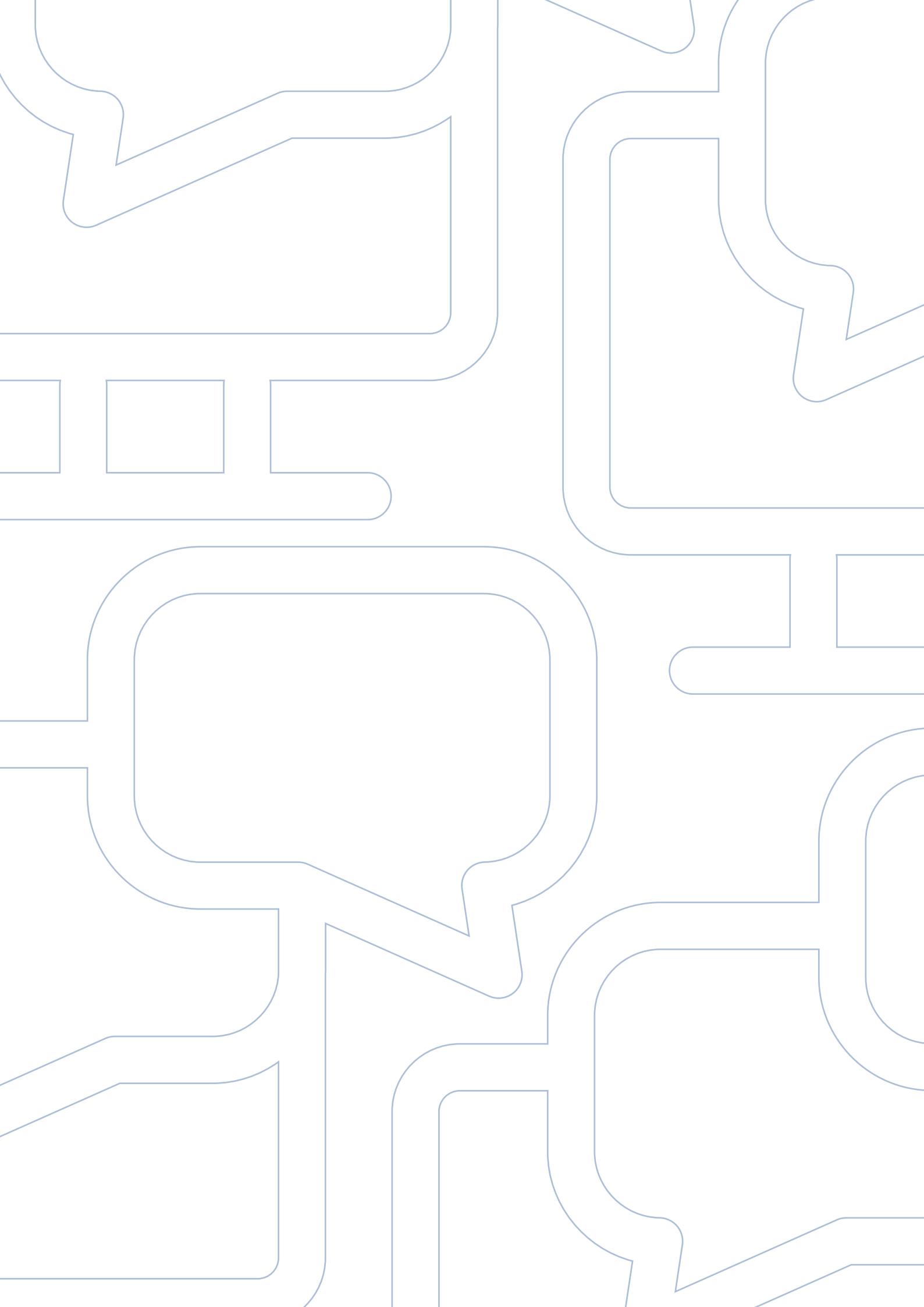
Leitura recomendada

ANDROID Developers. [*S. l.*: *S. n.*], 2019. Disponível em: https://developer.android.com. Acesso em: 26 jun. 2019.

ANOTAÇÕES

ANOTAÇÕES

ANOTAÇÕES



Unifametro

Formar para transformar

/unifametro @unifametro unifametro.edu.br