

Dispositivos Móveis

Aula 10 - Resources, Arquivos e Sharedpreferences

Apresentação

Nesta aula, abordaremos os estudos sobre armazenamento e manipulação de dados. Iniciaremos nossos estudos entendendo como o Android organiza os arquivos utilizados pela nossa aplicação (são chamados de *resources*), seguido da manipulação de arquivos e o uso de preferências (chamado de *SharedPreferences*).



Vídeo 01 - Apresentação

Objetivos

Ao final desta aula, você será capaz de:

- Entender a organização de recursos do Android.
- Acessar e manipular arquivos da aplicação.
- Criar e manter preferências da sua aplicação.

Resources

Como temos estudado até agora, sabemos que uma aplicação Android é composta por diversos recursos, tais como imagens, animações, arquivos de áudio e vídeo, estilos, etc. Por padrão, todos esses recursos são organizados dentro da pasta `res` e são organizados em subpastas, e, a seguir, veremos mais detalhes dessa estrutura. Além das pastas padrão de recursos, podemos, também, definir pastas específicas que serão utilizadas pelo Android, de acordo com as configurações do ambiente de execução da aplicação, bastando utilizar os qualificadores de recursos pré-definidos pela plataforma. A **Tabela 1** trás os principais modificadores de recursos relacionados a características de tela. A tabela contendo todos os modificadores disponíveis pode ser encontrada em <http://developer.android.com/guide/topics/resources/providing-resources.html>.

Perceba que alguns modificadores dependem de uma versão específica de API a ser utilizada, como indicado na própria tabela.

Característica da Tela	Qualificador	Descrição
Largura Disponível	w<N>dp	Especifica uma largura mínima de tela, em dp, para aquele recurso ser utilizado. Ex: w720dp para telas com no mínimo 720 dp de largura. Esse valor leva em consideração a orientação do dispositivos para determinar qual dimensão é a largura. API 13
Altura Disponível	h<N>dp	Especifica uma altura mínima de tela, em dp, para que o recurso seja utilizado. Ex: h720dp para telas com no mínimo 720 dp de altura. Esse valor também leva em consideração a orientação do dispositivo. API 13
Tamanho	small	Recursos para telas pequenas.

Característica da Tela	Qualificador	Descrição
	normal	Recursos para telas de tamanho normal. (Tamanho padrão)
	large	Recursos para telas grandes.
	xlarge	Recursos para telas extragrandes.
Aspecto da tela (AspectRatio)	long	Recursos para telas com proporções muito diferentes da altura e largura das medidas padrão.
	notlong	Recursos para telas com proporções similares à altura e à largura das medidas padrão.
Orientação	land	Recursos para telas em modo paisagem.
	port	Recursos para telas em modo retrato.
Densidade	ldpi	Recursos para telas de baixa densidade (~120dpi).
	mdpi	Recursos para telas de média densidade (~160dpi). (Densidade padrão)
	hdpi	Recursos para telas de alta densidade (~240dpi).
	xhdpi	Recursos para telas de densidade extra-alta (~320dpi).
	nodpi	Recursos para telas de todas as densidades. Devem ser imagens, independente de densidade, uma vez que o sistema não irá redimensioná-las.

Característica da Tela	Qualificador	Descrição
	tvdpi	Recursos para telas de TV. Sua densidade real ficaria em torno da média e da alta.

Tabela 1 – Qualificadores de Recursos do Android

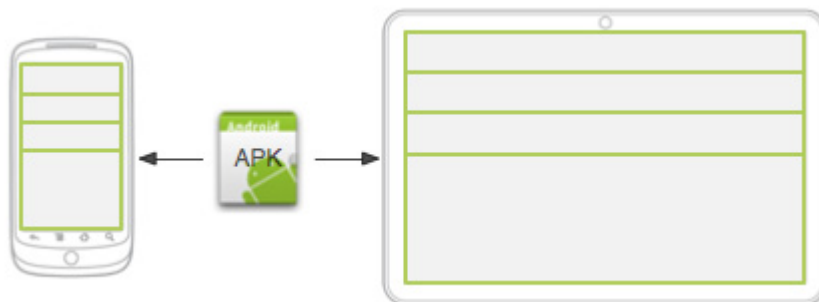
Para cada tipo de recurso, deve-se definir uma pasta padrão (sem nenhum qualificador), que será utilizada quando não existem pastas qualificadas ou ainda quando o ambiente de execução atual não se encaixa em nenhuma das condições impostas pelos qualificadores utilizados. Isso é importante principalmente quando se utilizam qualificadores que dependem de uma versão específica de API, para garantir que haja a disponibilidade de recursos em qualquer versão.

Um exemplo disso é o seguinte cenário: imagine que desejamos definir layouts independentes para as diferentes orientações de tela (retrato ou paisagem). Será que precisamos criar duas pastas classificadas, uma para **layout-land** e outra **layout-port**? A resposta é não. Como o Android utiliza a pasta **layout** como padrão, basta definirmos uma outra pasta **layout-land**, armazenando os layouts específicos da orientação paisagem. Nesse caso, haverá um conjunto de arquivos de layout genéricos, que serão usados sempre que não houver arquivos de layout específicos para uma determinada orientação.

Outro ponto importante é em relação à utilização de múltiplos qualificadores para uma única pasta. Nesse caso, deve-se separar os qualificadores por hífen e utilizá-los na ordem em que foram apresentados na tabela. Por exemplo: **layout-port-hdpi**. Tenha em mente também que não são diferenciadas maiúsculas de minúsculas e que só é possível ter um recurso do mesmo tipo em cada pasta, ou seja, não é possível ter um **layout-hdpi-ldpi**.

Para melhor exemplificar o uso dos qualificadores, observe a **Figura 1** a seguir.

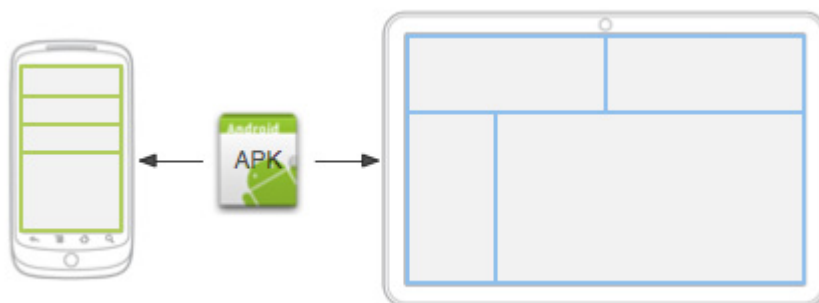
Figura 01 - Aplicação em diferentes aparelhos com o mesmo layout.



Fonte: Android Developers.

Na **Figura 1**, observamos que o layout definido na aplicação para telas pequenas, como a de um smartphone, é o mesmo utilizado para telas grandes, como a de um tablet. Nem sempre esse tipo de comportamento é otimizado para todos os casos e, por isso, caberia o uso de qualificadores do tamanho da tela, melhorando a usabilidade da aplicação. Após seu uso, a aplicação, que antes se comportava como na **Figura 1**, se comportará como na **Figura 2**, com diferentes telas.

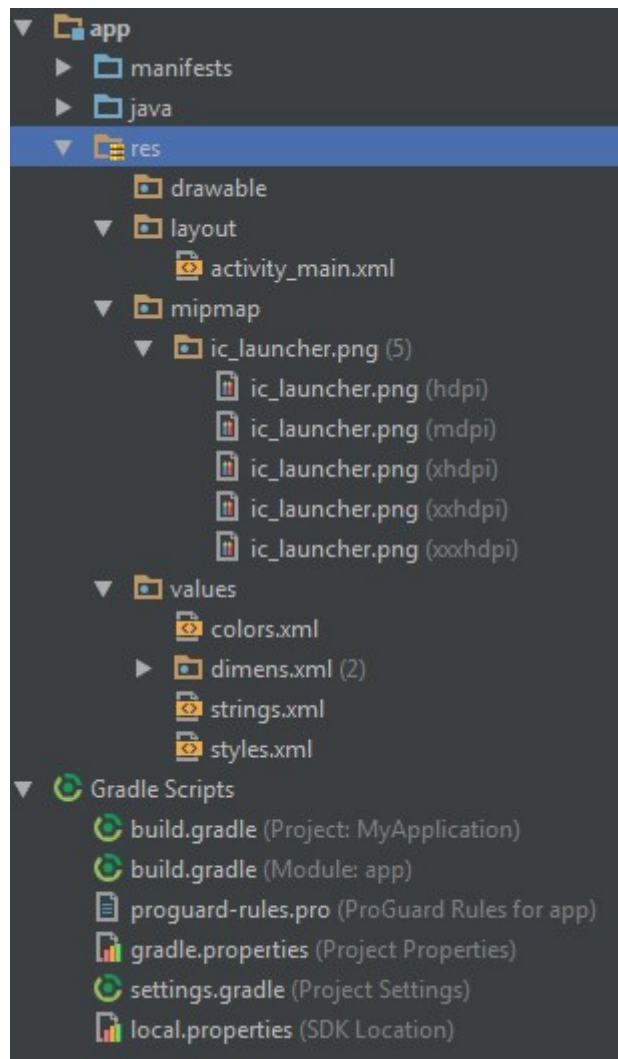
Figura 02 - Aplicação em diferentes aparelhos com layouts independentes para cada tamanho de tela.



Fonte: Android Developers.

A estrutura de organização dos recursos de nossa aplicação é bem simples e pode ser observada na **Figura 3**.

Figura 03 - Estrutura de organização dos recursos.



Como pode ser observado, a estrutura de um projeto é organizada de tal forma a realizar uma separação entre os arquivos de configuração, as classes (geradas automaticamente e criadas) e os recursos. Conforme já dito anteriormente, todos os recursos são armazenados na pasta res e são separados em subpastas de acordo com o tipo de recurso armazenado. As subpastas suportadas são mostradas no **Tabela 2**.

Directory

ResourceType

animator/

Arquivos XML para definir animações de propriedades.

Directory	ResourceType
anim/	Arquivos XML para definir <i>tweensanimations</i> (pode salvar arquivos de animações de propriedades, apesar de ser recomendado apenas para <i>tweenanimations</i>).
color/	Arquivos XML para listar cores.
drawable/	Arquivos de imagem (.png, .9.png, .jpg, .gif) ou XMLs que podem ser convertidos em imagens.
mipmap/	Arquivos de imagem para diferentes densidades de ícones da aplicação.
layout/	Arquivos que definem a interface com o usuário.
menu/	Arquivos de menu, seja de contexto ou de opções.
raw/	Arquivos em formato <i>raw</i> . Devem ser abertos utilizando-se um <code>InputStream</code> através da chamada do método <code>Resources.openRawResource</code> .
values/	<p>Arquivos XML contendo valores simples, como strings, inteiros, cores, etc. Diferentemente das outras pastas, os recursos mostrados nessa pasta são definidos pelos filhos da tag <code><resource></code> dos arquivos XML. Por exemplo, um <i>resource</i> que contenha uma tag <code><string></code> irá gerar um registro em <i>R.string</i>, e não em <i>R.values</i>. Apesar do nome dos arquivos não ser importante para a classe R, existem convenções em relação à nomeação deles:</p> <ul style="list-style-type: none"> - <i>arrays.xml</i> para arrays - <i>colors.xml</i> para cores - <i>dimens.xml</i> para dimensões - <i>strings.xml</i> para strings - <i>styles.xml</i> para estilos
xml/	Arquivos XML arbitrários, que podem ser lidos através do método <code>Resources.getXml</code> .

Tabela 2 - Subpastas de recursos suportados

Muitas vezes, precisamos acessar esses recursos que definimos dentro das classes do programa. Para isso, referenciamos os recursos utilizando seu identificador único, mais especificamente, o seu nome. Esse identificador único é criado automaticamente no arquivo R.java, durante o processo de compilação da aplicação.

A classe R.java armazena os IDs de todos os recursos adicionados na pasta res e segue uma estrutura bem simples. Para cada tipo de recurso utilizado no projeto, é adicionada uma subclasse estática à classe R (por exemplo, R.layout contém os ids de todos os layouts definidos) e para cada recurso adicionado àquele tipo, é criado um inteiro, também estático, cujo valor é o identificador único a ser utilizado para futuras referências. O valor desses IDs são criados e gerenciados automaticamente pelo ADT (*AndroidDevelopment Tools*).

Como já viemos estudando até agora, sabendo que o nome dos recursos na classe R é o nome do arquivo (sem sua extensão) ou o valor definido na propriedade android:name no XML do recurso, podemos facilmente acessá-los. Se quisermos, por exemplo, acessar o arquivo de layout main.xml através de uma classe, devemos referenciá-lo pelo nome R.layout.main.

Se necessitamos referenciar um recurso a partir de um arquivo XML, utilizamos uma estrutura similar. Para acessar uma imagem com o nome "minhaimagem", que foi adicionada na pasta drawable, fazemos a referência ao seu id através da estrutura @drawable/minhaimagem, como pode ser visto no trecho de definição de um ImageView na **Listagem 1**.

```
1 <ImageView android:id="@+id/imageld" android:src="@drawable/minhaimagem"/>
```

Listagem 1 – Referência a uma imagem definida em uma das pastas drawable

É importante perceber que as pastas qualificadas são utilizadas direto pelo sistema Android e não pelo programador para escolher qual recurso será exibido. Sendo assim, se temos um layout main.xml que possui três variações possíveis (ldpi, mdpi e hdpi, por exemplo), devemos apenas criar um arquivo main.xml em cada uma das pastas qualificadas (layout-ldpi, layout-mdpi e layout-hdpi, no caso) e referenciar o layout através de R.layout.main, deixando para o Android escolher, durante a execução, em qual pasta o recurso será buscado.



Vídeo 02 - Resources Android

Atividade 01

1. Responda qual a diferença prática entre os arquivos presentes nas pastas *drawable-ldpi* e *drawable-hdpi*?
2. Que nome você daria às pastas de *resources* que precisam armazenar arquivos de menus para os diferentes tamanhos de telas suportados pelo Android?

Arquivos

Muitas vezes, durante o desenvolvimento de uma aplicação, verificamos a necessidade de armazenar certas informações no aparelho para que possamos utilizá-las mais à frente. Uma das formas de armazenamento existente é a de armazenamento direto em arquivos, e será a primeira que iremos estudar.

Essa técnica nos permite ter grande controle sobre o armazenamento das informações, mas a sua manipulação pode ser muito trabalhosa. Por padrão, toda aplicação tem um espaço próprio disponibilizado pelo aparelho, e é nesse espaço que são armazenados os arquivos. Esse espaço é exclusivo de sua aplicação e, portanto, nenhum outro aplicativo poderá ler ou modificar os arquivos lá colocados.

O uso do armazenamento interno segue a seguinte ordem: primeiro devemos recuperar um objeto do tipo `FileOutputStream`, e a partir daí podemos usar todas as classes e métodos presentes nas APIs de Java para Entrada/Saída (`java.io.*` e `java.nio.*`). Após recuperar esse objeto, podemos escrever no arquivo usando o método *write* e, por fim, fechamos o *input* através do método *close*. Vamos observar a **Listagem 2** para melhor entender o uso de arquivos.

```

1 String nomeArquivo = "arquivo_hello";
2 String conteudo = "hello world!";
3 FileOutputStream fos;
4 try {
5     fos = openFileOutput(nomeArquivo, Context.MODE_PRIVATE);
6     fos.write(conteudo.getBytes());
7     fos.close();
8 } catch (FileNotFoundException e) {
9     //...
10 } catch (IOException e) {
11     //...
12 }

```

Listagem 2 - Escrita em arquivo

Dessa forma, será criado um arquivo chamado `arquivo_hello`, contendo a representação em bytes da string `"hello world!"`.

É possível, ainda, realizar a abertura e escrita do arquivo de outra forma. Da forma mostrada anteriormente, escrevemos no arquivo a representação em bytes da String desejada, mas e se quiséssemos escrever a String em si no arquivo? Uma das abordagens possíveis é a mostrada a seguir, na **Listagem 3**.

```

1 try {
2     String nomeArquivo = "arquivo_hello";
3     String conteudo = "hello world!";
4
5     File file = new File(getFilesDir() + "/" + nomeArquivo + ".teste");
6
7     BufferedWriter bw = new BufferedWriter(new FileWriter(file));
8
9     bw.write(conteudo);
10
11     bw.flush();
12     bw.close();
13 } catch (IOException ex) {
14     //...
15 }

```

Listagem 3 - Outra abordagem para escrita em arquivo

Conforme podemos observar, a criação de um arquivo mostrada na **Listagem 3** segue um outro modelo. Primeiramente, definimos uma variável String chamada `nomeArquivo`, que irá armazenar o nome do arquivo que iremos criar. Logo após, é criado um objeto do tipo `File` a partir do nome do arquivo, já passado com o caminho padrão disponibilizado pelo Android para a aplicação (`getFilesDir()`). Com o arquivo criado, basta criarmos um objeto do tipo `BufferedWriter` para poder

escrever nossa String desejada. Após finalizar a escrita, basta chamar os métodos `flush` e `close` do `BufferedWriter` para que as mudanças realizadas sejam escritas no arquivo. Fazendo isso, diferentemente do mostrado através da **Listagem 2**, os dados armazenados no arquivo serão exatamente os escritos, sem passar por qualquer tipo de transformação em bytes.

Para fazer a leitura dos dados armazenados usando uma das formas mostradas, usamos um processo bem simples, descrito na **Listagem 4**.

```
1  BufferedReader input = null;
2  try {
3      input = new BufferedReader(new InputStreamReader(openFileInput(nomeArquivo
4      + "_string.teste")));
5
6      String line;
7      StringBuffer buffer = new StringBuffer();
8
9      while ((line = input.readLine()) != null) {
10         buffer.append(line);
11     }
12
13     Log.d("Leitura de Arquivo", buffer.toString());
14 } catch (Exception e) {
15     e.printStackTrace();
16 } finally {
17     if (input != null) {
18         try {
19             input.close();
20         } catch (IOException e) {
21             e.printStackTrace();
22         }
23     }
24 }
```

Listagem 4 - Leitura de arquivos

O processo mostrado acima se inicia com a criação de um objeto do tipo *BufferedReader*, que será responsável pelo acesso ao conteúdo do arquivo. Para instanciá-lo, utilizamos o nome do arquivo armazenado, de forma que seu objeto de trabalho seja o arquivo passado. Logo após, tentamos ler o conteúdo do arquivo, linha a linha, utilizando a estrutura de repetição *while*. Dentro do laço, o conteúdo lido (uma linha) é armazenado em um objeto do tipo *StringBuffer*. Ao final, imprimimos o conteúdo encontrado. Devemos lembrar que, após finalizarmos nosso trabalho com o arquivo, devemos fechar a conexão existente no objeto *BufferedReader*, e assim o fizemos na implementação do *finally*.

Armazenar informações com um dos métodos vistos, é a garantia de que os arquivos ficarão na memória interna do aparelho do usuário e de que eles possam ser removidos a qualquer momento, caso o usuário limpe os dados da aplicação.

Outra abordagem que pode ser utilizada é o uso de cache. Ao utilizar o armazenamento em forma de cache, o Android pode fazer a remoção das informações, quando o sistema necessitar de espaço. Sua implementação pode ser feita muito semelhante à mostrada na Listagem 3, tendo como única diferença a chamada do método *getCacheDir()* no lugar de *getFilesDir()*. Dessa forma, qualquer arquivo armazenado utilizando essa estrutura, será tratado como arquivo de cache.

As formas que vimos até agora usam a memória interna do aparelho, mas podemos, também, guardar os arquivos, sejam de cache ou não, no chamado armazenamento externo do aparelho (pode ser um dispositivo fisicamente externo como um cartão SD ou mesmo uma região de armazenamento interna não removível). Nesse caso, os arquivos criados não serão removidos pelo Android automaticamente, nem mesmo quando a aplicação for desinstalada, quando o usuário limpar os dados da aplicação, ou mesmo quando o Android precisar de mais espaço de armazenamento.

Como o armazenamento externo pode se tratar de um tipo removível de mídia, uma das preocupações que devemos ter, antes de trabalhar com os arquivos, é verificar se a mídia é acessível no aparelho e se é possível realizar a escrita de dados, se esse for o caso. Para tal verificação, basta seguirmos o modelo apresentado na

Listagem 5.

```
1 boolean mExternalStorageAvailable = false;
2 boolean mExternalStorageWriteable = false;
3 String state = Environment.getExternalStorageState();
4 if (Environment.MEDIA_MOUNTED.equals(state)) {
5     mExternalStorageAvailable = mExternalStorageWriteable = true;
6 } else if (Environment.MEDIA_MOUNTED_READ_ONLY.equals(state)) {
7     mExternalStorageAvailable = true;
8     mExternalStorageWriteable = false;
9 }
```

Listagem 5 - Verificação de disponibilidade e acesso do armazenamento externo

Seguindo essa forma mostrada, após a execução desse bloco de código, as variáveis `mExternalStorageAvailable` e `mExternalStorageWriteable` terão seus valores verdadeiros apenas se o armazenamento externo estiver disponível no momento e for possível realizar a escrita de arquivos nele, respectivamente.

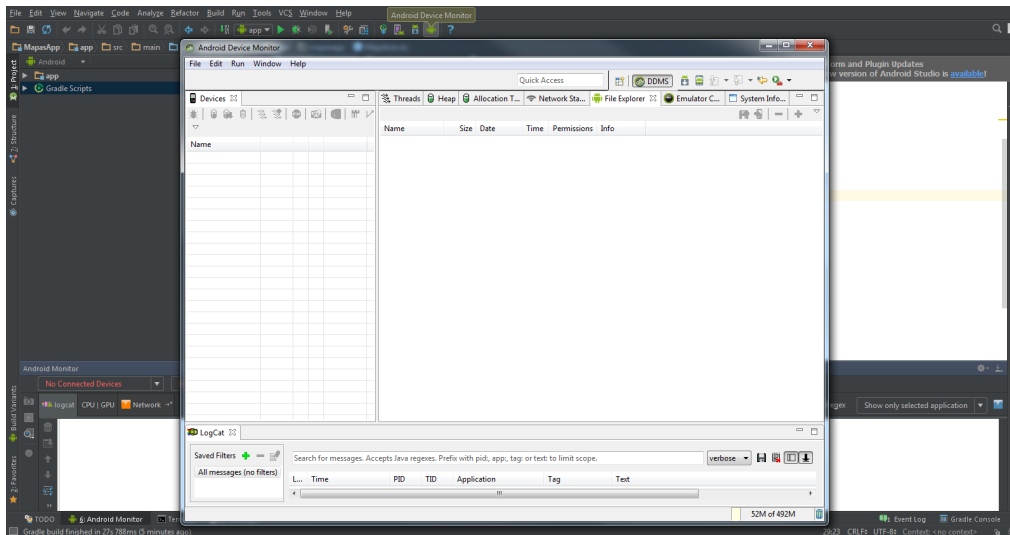
Tendo esses dados em mãos, então, podemos prosseguir com a inserção de arquivos de acordo com os valores definidos para as duas variáveis de controle.

```
1 if(mExternalStorageAvailable && mExternalStorageWriteable) {
2     try {
3         String filename = nomeArquivo + "_externo";
4
5         File file = new File(getExternalFilesDir(null) + "/" +
6             filename + ".teste");
7
8         BufferedWriter bw = new BufferedWriter(new FileWriter(file));
9
10        bw.write(conteudo);
11
12        bw.flush();
13        bw.close();
14    } catch (IOException ex) {
15        //..
16    }
17 }
```

Listagem 6 - Adicionando arquivos na estrutura de armazenamento externa

Conforme mostrado na **Listagem 6**, a estrutura utilizada ainda se assemelha bastante à mostrada na **Listagem 3**, com diferença apenas na criação do arquivo, uma vez que aqui é chamado o método `getExternalFilesDir`. Sobre esse método, é importante ressaltar o uso de seu parâmetro, que indica o tipo de subdiretório que queremos acessar, e seus valores, definidos como constantes da classe `android.os.Environment` (alguns exemplos são: `DIRECTORY_MUSIC` para acesso direto à pasta de músicas, `DIRECTORY_PICTURES` para acessar a pasta de fotos e assim por diante). No caso do exemplo mostrado na **Listagem 6**, foi passado *null* como parâmetro, visto que nosso arquivo não se encaixa em nenhuma das categorias já existentes.

O vídeo a seguir demonstra a visualização de arquivos no DDMS do Eclipse. No Android Studio, a mesma interface está presente. Para acessá-la, utilize a opção Android Device Monitor, como visto na imagem abaixo.



Vídeo 03 - Acessando Arquivos no Emulador ou Dispositivo

Atividade 02

1. Crie uma Activity que possui basicamente um EditText com múltiplas linhas e dois Botões (com os textos "Salvar" e "Recuperar"). Ela deve executar as seguintes ações:
 - a. O botão de Salvar, ao ser clicado, deve salvar o conteúdo do EditText em um arquivo.
 - b. O botão de Recuperar deve ler o conteúdo do arquivo e preencher o EditText. Caso o arquivo não exista, nada deve ser feito.

SharedPreferences

Outro tópico muito importante a ser estudado é a parte de preferências de usuário. Através do uso das preferências, ou *SharedPreferences*, deixamos o tratamento de armazenamento e recuperação com o Android, e ainda temos a

possibilidade de utilizar toda a estrutura disponibilizada para tratamento desses dados.

O armazenamento das preferências é feito baseado no conceito de pares chave-valor e podem ter diversos níveis de acesso, de acordo com a necessidade da aplicação. Por padrão, as preferências são definidas como privadas à aplicação, e é baseado nesse conceito que iremos desenvolver nossos exemplos.

Existem duas formas de se definir uma preferência, programaticamente ou através de arquivos XML. A escrita e leitura de preferências de forma programática são feitas de forma bem simples. No exemplo mostrado na **Listagem 7**, mostramos uma das formas de se fazer a leitura dos dados armazenados como preferências.

```
1 SharedPreferences preferences =  
2 PreferenceManager.getDefaultSharedPreferences(this);  
3  
4 String username = preferences.getString("username", "indefinido");
```

Listagem 7 – Recuperação de valor de preferência

Para a recuperação do valor de uma preferência precisamos apenas obter a referência ao arquivo de referências padrão, através da classe `PreferenceManager` e, logo após, utilizar o método `get` equivalente ao valor que queremos recuperar (os métodos possíveis são `getString`, `getBoolean`, `getFloat`, `getInt` e `getLong`; cada um desses métodos recebe como parâmetro a chave referente à preferência, e o valor padrão que deve ser retornado caso essa preferência não exista).

Para os casos de edição ou criação de uma nova preferência, utilizaremos a mesma variável `preferences` definida na **Listagem 7**, e incluiremos o trecho definido na **Listagem 8**.

```
1 SharedPreferences.Editor edit = preferences.edit();  
2 edit.putString("password", "123");  
3 edit.commit();
```

Listagem 8 - Adição de uma preferência

Como pode ser visto, recuperamos inicialmente um objeto do tipo *Editor*, que será o responsável por realizar nossas mudanças nas preferências; depois inserimos a preferência desejada (para edição de uma preferência, basta utilizar uma chave já

existente no arquivo) e, por fim, submetemos a mudança através do método commit. Após a execução desse trecho de código, uma nova preferência com nome "password" e valor "123" existirá nos registros de sua aplicação.

Como dito anteriormente, outra forma de tratar as preferências da aplicação é através de arquivos XML. Essa forma de abordagem é interessante quando a manipulação das preferências deve ser feita diretamente pelo usuário, uma vez que o próprio Android já disponibiliza toda a estrutura para o tratamento dos tipos de informação suportados pelas preferências.

Para criar uma tela de preferências da aplicação baseada em um XML, devemos definir o arquivo com as preferências que queremos trabalhar, conforme mostrado na **Listagem 9**.

```
1 <?xml version="1.0" encoding="utf-8" ?>
2 <PreferenceScreen xmlns:android="http://schemas.android.com/apk/res/android">
3 <EditTextPreference
4     android:key="username"
5     android:summary="Nome de usuario"
6     android:title="Username"/>
7 <EditTextPreference
8     android:inputType="textPassword"
9     android:key="password"
10    android:summary="Senha"
11    android:title="Password"/>
12 </PreferenceScreen>
```

Listagem 9 – Arquivo de preferências

O nome do arquivo é arbitrário, mas, nesse exemplo, diremos que ele se chama "preferences.xml". Esse arquivo deve ser criado dentro da pasta res/xml, como vimos anteriormente. A definição do arquivo começa com a tag PreferenceScreen, que, por sua vez, possui as tags filhas EditTextPreference (no caso utilizamos duas propriedades de texto). Importante notar a propriedade android:key de cada uma das preferências definidas. Essas chaves são exatamente as mesmas das utilizadas nas **Listagens 7** e **8**, o que faz com que esses códigos trabalhem com as mesmas preferências.

Definido o arquivo XML contendo as preferências, temos que criar uma Activity específica de preferências. Desde a API 11, o Android modificou a implementação para que esse tipo de comportamento fosse coberto por um Fragment, filho da

classe PreferenceFragment. Esse Fragment não necessita de nenhum tipo de tratamento diferente do padrão para funcionar corretamente, podendo ser adicionado a qualquer tipo de Activity que normalmente o receberia. Assim, sua implementação do Fragment pode seguir o modelo da **Listagem 10** e sua Activity principal poderia carregá-lo normalmente, como mostra a **Listagem 11**.

```
1 public static class MinhasPreferenciasFragment extends PreferenceFragment {  
2  
3     @Override  
4     public void onCreate(Bundle savedInstanceState) {  
5         super.onCreate(savedInstanceState);  
6  
7         // Load the preferences from an XML resource  
8         addPreferencesFromResource(R.xml.preferences);  
9     }  
10 }
```

Listagem 10 - Implementação do Fragment de preferências

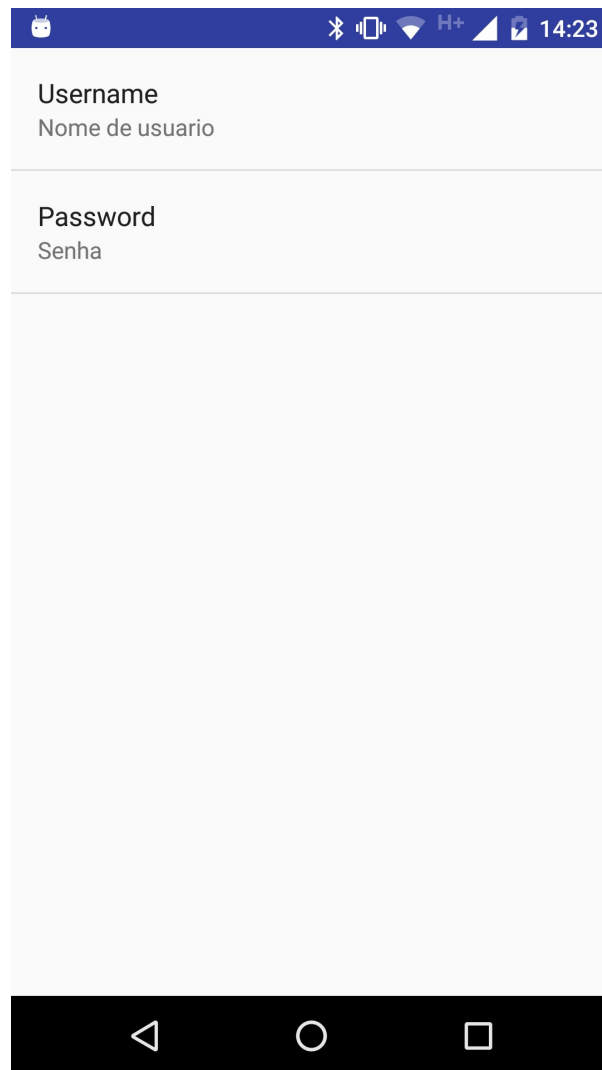
```
1 public class MainActivity extends Activity {  
2     @Override  
3     protected void onCreate(Bundle savedInstanceState) {  
4         super.onCreate(savedInstanceState);  
5  
6         // Display the fragment as the main content.  
7         getSupportFragmentManager().beginTransaction()  
8         .replace(android.R.id.content, new MinhasPreferenciasFragment())  
9         .commit();  
10    }  
11 }
```

Listagem 11 - Implementação da Activity utilizando o Fragment de preferências.

De acordo com as **Listagens 10** e **11**, quando o Fragment em questão for chamado, todas as preferências definidas no arquivo preferences.xml estarão prontas para edição. Tal comportamento é garantido pela chamada do método addPreferencesFromResource.

Pronto! Agora tudo que precisamos fazer é executar a Activity e o comportamento será o mostrado na **Figura 4**.

Figura 04 - Tela de preferências.



The screenshot shows an Android application interface with a blue header bar. The status bar at the top displays various icons (Bluetooth, vibration, Wi-Fi, H+, signal strength, battery) and the time 14:23. The main content area is light gray and contains two sections: 'Username' with the subtitle 'Nome de usuario' and 'Password' with the subtitle 'Senha'. Each section has a horizontal line below it. At the bottom, there is a black navigation bar with three white icons: a back arrow, a circle, and a square.



Vídeo 04 - Lendo o Conteúdo de Arquivos de Resources

Atividade 03

1. Crie uma Activity que possui basicamente um EditText e um Button. Ela deve executar as seguintes ações:
 - a. O Button, ao ser clicado, deve recuperar o valor do EditText e salvar o conteúdo em uma preferência.
 - b. A aplicação, ao ser iniciada (no método onCreate), deve recuperar o valor da preferência previamente armazenada e preencher o EditText. Caso a preferência ainda não exista, nada deve ser feito.

Leitura Complementar

DEVELOPERS. **Data Storage.** Disponível em: <<http://developer.android.com/guide/topics/data/data-storage.html>>. Acesso em: 22 jun. 2012.

Material *on-line* (em inglês) sobre aplicações Android.

Resumo

Nesta aula aprendemos como usar arquivos de recursos, como imagens, vídeos, etc. em nossas aplicações. Também estudamos como o Android trata o armazenamento de arquivos de um aparelho, seja interno ou externo, e mostramos como criar arquivos utilizando técnicas e abordagens diferentes. Estudamos também o conceito e a implementação de SharedPreferences, armazenando um conjunto de informações utilizando a estrutura já disponibilizada pela plataforma.

Autoavaliação

1. Explique o motivo de não ser recomendado o armazenamento de informações importantes da aplicação no formato de cache.
2. Descreva uma situação em que seria justificado o armazenamento de informações através de arquivos de cache.
3. É possível que uma mesma preferência seja compartilhada por diversos aplicativos? Explique.
4. É possível armazenar qualquer tipo de informação (nos diferentes formatos) em um arquivo no formato de preferências? Explique.

Referências

ANDROID DEVELOPERS. 2012. Disponível em: <<http://developer.android.com/index.html>>. Acesso em: 27 mai. 2015.

DIMARZIO, J. **Android: A Programmer's Guide**. New York: McGraw-Hill, 2008. Disponível em: <<http://books.google.com.br/books?id=hoFl5pxjGesC>>. Acesso em: 5 set. 2012.

HASEMAN, C. **Android Essentials**. Berkeley, CA. USA: Apress, 2008.

MEIER, R. **Professional Android 2 Application Development**. New York: John Wiley & Sons, 2010. Disponível em: <<http://books.google.com.br/books?id=ZthJlG4o-2wC>>. Acesso em: 5 set. 2012.