



Universidade Federal de São Carlos
Centro de Ciências Exatas e de Tecnologia
Departamento de Computação

**Introdução às tecnologias para desenvolvimento de
aplicações em plataformas móveis Android**

Processos: 23112.003595/2012-35 e 23112.004023/2013-54

Coordenadores:

Ricardo Menotti

Daniel Lucrédio

Responsável:

Caio Cesar Almeida Pegoraro

São Carlos - SP, 15 de outubro de 2014



Sobre este documento

Os seguintes autores contribuíram para a elaboração deste material, a revisão indicada é a última realizada por cada um deles:

- Caio Cesar Almeida Pegoraro r112
- José Eduardo da Silva Teixeira Junior r98
- Matheus Fernando Finatti r86

O código fonte do material, bem como de todos os exemplos usados, encontra-se disponível em <http://mobile.dc.ufscar.br/>.

Resumo

Esse material didático oferece uma visão geral de como programar para o sistema móvel Android e utilizar suas APIs nativas na criação de aplicativos. O material tentará cobrir desde o básico, como a configuração do ambiente de desenvolvimento, criação de layouts básicos e complexos, estrutura geral de um aplicativo e, finalmente, apresentar a programação de aplicativos mais complexos que utilizam APIs nativas.

O objetivo é fornecer noções sobre como utilizar as ferramentas do Android, introduzir os conceitos sem entrar em detalhes aprofundados do sistema operacional e assim disponibilizar uma visão básica sobre o assunto. Após a leitura desse material e realização da prática o leitor deverá estar preparado para construir seus próprios aplicativos nativos, e poderá até monetizá-los se desejar.

Sumário

Sobre	i
Resumo	ii
Lista de Figuras	vi
Lista de Tabelas	vii
Lista de Algoritmos	vii
1 Introdução	2
2 Configuração do Ambiente	6
3 Linguagem do Android	7
3.1 Linguagem	7
3.2 Entendendo a estrutura de uma aplicação Android	8
3.3 Alguns arquivos importantes:	9
4 Criando seu primeiro aplicativo	12
5 Design	27
5.1 Activity	27
5.2 Especifique a <i>activity</i> que inicia seu aplicativo	28
5.3 Tipos de <i>Layout</i>	29
5.4 Listas (<i>ListView</i>)	32
5.5 Listas Compostas	35
5.6 Listas expansíveis (<i>ExpandableListView</i>)	38
5.7 Grades (<i>GridView</i>) e imagens <i>ImageView</i>	45
5.8 Fragmentos	50
5.9 Abas (<i>Tabs</i>)	53

5.10	Trocar de página com gesto de arrastar usando <code>ViewPager</code>	60
5.11	Abas com gesto de arrastar	62
5.12	<i>ActionBar</i>	64
6	Comunicação	70
6.1	<i>Internet</i>	70
6.2	Telefone	75
6.3	<i>Short Message Service (SMS)</i>	76
7	Armazenamento	78
7.1	<i>Shared Preferences:</i>	78
7.2	Armazenamento interno	80
7.3	Armazenamento Externo	84
7.4	Banco de dados	85
8	Câmera	90
8.1	Usando a API	90
8.2	Gravando vídeos	99
8.3	Usando um <code>Intent</code>	101
9	Áudio	105
9.1	Gravando e tocando áudio	105
10	Localização e Mapas	109
10.1	Acessando a localização	109
10.2	<i>Google Maps</i>	111
11	Compartilhamento	117
12	Agenda e Contatos	122
12.1	Usando o <i>Contacts Provider</i>	122
13	Acelerômetro	132
14	Bluetooth	135
15	Internacionalização	146

Lista de Figuras

1.1	Distribuição das versões do Android	4
4.1	Primeira janela de criação de novo aplicativo	13
4.2	Segunda janela de criação de novo aplicativo	13
4.3	Terceira janela de criação de novo aplicativo	14
4.4	Quarta janela de criação de novo aplicativo	14
4.5	Quinta janela de criação de novo aplicativo	15
4.6	Selecionando o Hello world	17
4.7	<i>activity</i> com os elementos colocados na tela	18
4.8	Criando uma nova <i>activity</i>	22
4.9	Primeira tela do primeiro aplicativo	26
4.10	Primeira tela após escrever texto na caixa de texto	26
4.11	Segunda tela mostrando a mensagem enviada	26
5.1	Ciclo de vida de uma <i>activity</i>	27
5.2	LinearLayout vertical (à esquerda) e horizontal (à direita)	29
5.3	LinearLayout composto	29
5.4	Exemplo de RelativeLayout	30
5.5	FrameLayout com exemplo de posicionamento usando <code>layout_gravity</code>	31
5.6	Exemplo de TableLayout	31
5.7	Esquema de uma lista	32
5.8	Detalhes de um elemento da lista	32
5.9	Lista simples	35
5.10	Lista Composta	38
5.11	Exemplo de lista expansível rodando em um <i>smartphone</i>	44
5.12	Esquema de um GridView	45
5.13	Demonstração de um GridView	47
5.14	Exemplo GridView com imagem em tela cheia	50
5.15	Esquema da interface com abas	54

5.16	Figura mostrando as 3 abas criadas no exemplo	59
5.17	Exemplo de <i>ActionBar</i> no aplicativo Calendário	65
5.18	Exemplo de busca na <i>ActionBar</i>	69
10.1	Ativando <i>Maps API</i> no <i>Google API Console</i>	112
10.2	Registrando um <i>app</i> no <i>Google API Console</i>	112
10.3	Passo final para obter a <i>API Key</i>	113
10.4	Adicionando a <i>Google Play Services</i> como biblioteca do projeto	114

Lista de Tabelas

- 1.1 Tabela com as distribuições das versões do Android, coletada durante um período de 7 dias acabando em 1 de abril de 2014. Como esses dados são adquiridos através do novo app da play store, o qual dá apenas suporte para o Android 2.2 e superior, versões mais antigas não estão inclusas. Todavia, em Agosto de 2013, versões mais antigas que o Android 2.2 somavam um total de 1% dos dispositivos que logaram nos servidores da google 3

Lista de Algoritmos

4.1	Exemplo de configuração de versão do SDK no arquivo <code>AndroidManifest.xml</code>	17
4.2	Código da caixa de texto no arquivo <code>activity_main.xml</code>	18
4.3	Código do botão	19
4.4	Arquivo de strings com as duas strings adicionadas	19
4.5	Adicionando método à classe <code>MainActivity</code>	20
4.6	Exemplo de import de uma classe <code>Android</code>	20
4.7	Adicionando uma <code>Intent</code>	20
4.8	Obtendo o conteúdo da caixa de texto e enviando para outra <i>activity</i>	20
4.9	Constante como chave para um extra	21
4.10	Obtendo a <i>string</i> passada como extra do <code>Intent</code>	23
4.11	Método <code>onCreate()</code> recebendo um <i>Intent</i> e mostrando a mensagem	23
5.1	Exemplo de <i>Launcher activity</i>	28
5.2	<code>LinearLayout</code> no arquivo de <i>layout</i>	33
5.3	Código de uma <code>ListView</code>	33
5.4	<code>string-array</code> populada com elementos	33
5.5	Código de uma <i>activity</i> com lista clicável	34
5.6	Código do arquivo <code>item.xml</code>	36
5.7	Código da lista customizada	37
5.8	Código XML de uma Lista expansível	38
5.9	Layout <code>list_item_parent.xml</code>	39
5.10	Layout <code>list_item_child.xml</code>	39
5.11	Classe <code>Parent</code>	40
5.12	Classe <code>CustomAdapter</code>	42
5.13	Construindo a lista expansível na <i>activity</i>	43
5.14	Layout do <code>GridView</code>	45
5.15	Classe <code>ImageAdapter</code>	46
5.16	<i>activity</i> com grade	47

5.17	Layout <code>full_image.xml</code>	48
5.18	Classe <code>FullImageActivity</code>	49
5.19	Código da <i>activity</i> após as modificações	49
5.20	Classe <code>BasicFragment</code>	51
5.21	Layout da <i>activity</i> com um fragmento	52
5.22	Layout da <i>activity</i> com o <code>FrameLayout</code>	52
5.23	<i>activity</i> com adição dinâmica de fragmento	53
5.24	Layout da <i>activity</i> <code>TabHostLayout</code>	54
5.25	Layout do fragmento da aba.	55
5.26	Classe <code>Tab1Fragment</code>	55
5.27	Classe <code>TabInfo</code>	56
5.28	Primeira parte da classe <code>TabLayoutActivity</code>	56
5.29	Classe <code>TabFactory</code>	57
5.30	Método <code>initialiseTabHost()</code>	57
5.31	Método <code>onTabChanged()</code>	58
5.32	Método <code>onSaveInstanceState()</code>	59
5.33	layout do <code>ViewPager</code>	60
5.34	Classe <code>PagerAdapter</code>	61
5.35	<i>Activity</i> com <code>PagerAdapter</code>	62
5.36	Layout das abas com adição do <code>ViewPager</code>	63
5.37	Método <code>initialiseViewPager()</code>	63
5.38	Método <code>onTabChanged()</code> alterado	64
5.39	Métodos da interface <code>ViewPager.OnPageChangeListener</code>	64
5.40	Menu padrão dos exemplos	65
5.41	Método padrão <code>onCreateOptionsMenu()</code>	66
5.42	Método <code>OnOptionsItemSelected()</code>	66
5.43	Adicionando novo item na <i>ActionBar</i>	67
5.44	Configurando <i>ActionBar</i> no método <code>onCreate()</code>	67
5.45	Criando a caixa de busca na <i>ActionBar</i>	68
6.1	Atribuindo permissão de acesso à <i>Internet</i> no <i>Manifest</i>	70
6.2	Classe <code>RequestTask</code>	71
6.3	Usando <code>RequestTask</code> na <i>activity</i>	72
6.4	Modificando o método para requisições <i>POST</i>	73
6.5	Classe <code>JSONParser</code>	74
6.6	Criando JSON	74
6.7	Fazendo uma chamada telefônica	75
6.8	Permissão para fazer chamadas telefônicas	75
6.9	Permissão para enviar mensagens SMS	76
6.10	Método <code>sendSMS()</code>	76

6.11 Chamando método <code>sendSMS()</code>	77
7.1 Utilizando <code>SharedPreferences</code> para salvar dados primitivos	79
7.2 Passos iniciais do <i>listener</i> , criando um alerta.	80
7.3 Salvando um arquivo e mostrando um <code>Toast</code>	81
7.4 Fechando o alerta ao clicar em <i>close</i>	82
7.5 Criando um alerta com os arquivos salvos	82
7.6 Criando um alerta com os arquivos salvos	83
7.7 Verificando se o armazenamento externo está disponível	84
7.8 Classe <code>CarOpenHelper</code> do <code>SQLite</code>	86
7.9 Usando o <code>CarOpenHelper</code> na <i>activity</i>	87
7.10 Método <code>openCarList()</code>	88
7.11 Método <code>fetchCarList()</code>	89
8.1 Requisitando permissão para usar a câmera	90
8.2 Requisitando permissão para gravar no armazenamento externo	90
8.3 Requisitando permissão para gravar áudio	91
8.4 Classe <code>CameraAccess</code>	92
8.5 Classe <code>CameraPreview</code>	93
8.6 <i>Layout</i> da <i>Activity</i> que irá conter a visualização	94
8.7 Configurando a orientação da <i>activity</i> no <i>Manifest</i>	95
8.8 Primeira parte da classe <code>CameraActivity</code>	95
8.9 Criando um <code>Callback</code> para imagens <code>JPEG</code>	96
8.10 Método <code>getOutputMediaFile()</code>	97
8.11 Método <code>getOutputMediaFile()</code>	98
8.12 Melhorando a qualidade das fotos tiradas	98
8.13 Método <code>prepareForRecording()</code>	99
8.14 Método <code>releaseMediaRecorder()</code>	100
8.15 Liberando a câmera no método <code>onPause()</code>	100
8.16 Configurando o <i>listener</i> do botão de gravar vídeo	101
8.17 Chamando a <i>activity</i> de câmera com <code>Intent</code>	102
8.18 Método <code>getOutputMediaFileUri()</code>	102
8.19 Método <code>onActivityResult()</code>	103
8.20 Criando um <code>Intent</code> para vídeo	104
9.1 Método <code>prepareRecording()</code> para gravações de áudio	105
9.2 Método <code>releaseRecorder()</code>	106
9.3 Configurando o botão de gravar áudio	106
9.4 Método <code>startPlaying()</code>	107
9.5 Método <code>stopPlaying()</code>	107
9.6 Configurando o botão de tocar áudio	108
9.7 Variável <code>testFilename</code>	108

10.1	Permissões para obter localização	109
10.2	Criando um <code>LocationListener</code>	110
10.3	Configurando o <code>LocationManager</code>	111
10.4	Configurando a <i>API Key</i> no <i>Manifest</i>	113
10.5	Adicionando o uso do OpenGL no <i>Manifest</i>	114
10.6	Adicionando o mapa como um fragmento no XML	114
10.7	<i>Activity</i> com <i>Google Maps</i>	115
10.8	Método <code>setUpMapIfNeeded()</code>	116
10.9	Método <code>onLocationChanged()</code> modificado	116
11.1	Enviando um texto simples através de um <code>Intent</code>	117
11.2	Chamando <code>createChooser()</code>	118
11.3	Botões para compartilhar texto e imagem	119
11.4	Configurando os <code>intent-filter</code> no <i>Manifest</i>	120
11.5	Obtendo os dados do <code>Intent</code> e mostrando ao usuário	120
12.1	Permissão para acessar os contatos	122
12.2	<i>Activity</i> que irá conter a lista de contatos	123
12.3	Variáveis para o adaptador da lista	123
12.4	Variáveis para o <code>Cursor</code> do conjunto resultante da busca	124
12.5	Variáveis de controle	124
12.6	Método <code>onActivityCreated()</code>	125
12.7	Método <code>onCreateLoader()</code>	125
12.8	Método <code>onItemClick()</code>	126
12.9	Interfaces das consultas dos contatos	127
12.10	Classe <code>ContactDetailsActivity</code>	127
12.11	Método <code>onCreate()</code> de <code>ContactDetailsActivity</code>	128
12.12	Método <code>setContact()</code>	128
12.13	Método <code>onCreateLoader()</code>	129
12.14	Método <code>onLoadFinished()</code>	130
13.1	Classe <code>AccelActivity</code>	132
13.2	Método <code>onCreate()</code> de <code>AccelActivity</code>	133
13.3	Método <code>onSensorChanged()</code>	133
13.4	Métodos <code>onResume()</code> e <code>onPause()</code>	134
14.1	Classe <code>DeviceListActivity</code>	135
14.2	Primeira parte do método <code>onCreate()</code>	136
14.3	Método <code>onActivityResult()</code>	137
14.4	Segunda parte do método <code>onCreate()</code>	138
14.5	<code>BroadcastReceiver</code> que captura dispositivos <i>Bluetooth</i>	138
14.6	Registrando e removendo o <code>BroadcastRegister</code>	139
14.7	Classe <code>AcceptThread</code>	140

14.8	Classe <code>ConnectThread</code>	141
14.9	Método <code>manageConnection()</code>	141
14.10	Classe <code>ConnectedThread</code>	142
14.11	Método <code>writeMessage()</code>	143
14.12	Método <code>showMessage()</code>	143
14.13	<code>Handler</code> e <code>Runnable</code>	144
14.14	Implementação do método <code>onItemClick()</code>	144
14.15	Terceira parte do método <code>onCreate()</code>	144
15.1	<code>strings.xml</code> padrão	146
15.2	<code>strings.xml</code> em português	146

Introdução

O Android hoje está em centenas de milhões de dispositivos móveis ao redor do mundo, e vem crescendo. É uma plataforma para desenvolvimento em dispositivos móveis como *smartphones*, *tablets* e outros.

Construído em uma colaboração *open-source* com a comunidade de Linux, o Android se tornou a plataforma móvel mais utilizada e que mais cresce no mundo. Sua abertura o tornou o favorito de consumidores e desenvolvedores, levando a um rápido crescimento no número de aplicativos e jogos. Está disponível em centenas de dispositivos diferentes e de fabricantes diferentes em versões diferentes.

Atualmente¹ existem 5 principais versões do Android, são elas da mais atual para mais antiga:

- *KitKat* versão 4.4 que melhorou o gerenciamento de memória e reduziu o consumo, trazendo uma proteção agressiva da memória do sistema contra apps usando grandes quantidades de RAM além de introduzir uma nova interface.²
- *Jelly Bean* versão 4.1 à 4.3 que trouxe otimizações de performance, uma nova interface

¹Data em que foi escrito: 04/2014

²*KitKat*: <http://developer.android.com/about/versions/kitkat.html>

do sistema e outros ³

- *Ice Cream Sandwich* versão 4.0 trouxe uma interface refinada e unificada para *smartphones* e *tablets* além de facilidade com multitasking e outros ⁴
- *Honeycomb* versão 3.0 desenvolvida exclusivamente para *tablets* ⁵
- *Gingerbread* versão 2.3 introduziu refinamentos da interface, mais performance e tornou o sistema mais intuitivo ⁶

O Google coletou os dados referentes a distribuição das versões do Android:

Versão	Codínome	API	Distribuição
2.2	Froyo	8	1.1%
2.3.3 - 2.3.7	Gingerbread	10	17.8%
3.2	Honeycomb	13	0.1%
4.0.3 - 4.0.4	Ice Cream Sandwich	15	14.3%
4.1.x	Jelly Bean	16	34.4%
4.2.x	Jelly Bean	17	18.1%
4.3	Jelly Bean	18	8.9%
4.4	KitKat	19	5.3%

Tabela 1.1: Tabela com as distribuições das versões do Android, coletada durante um período de 7 dias acabando em 1 de abril de 2014. Como esses dados são adquiridos através do novo app da play store, o qual dá apenas suporte para o Android 2.2 e superior, versões mais antigas não estão inclusas. Todavia, em Agosto de 2013, versões mais antigas que o Android 2.2 somavam um total de 1% dos dispositivos que logaram nos servidores da google

³*Jelly Bean*: <http://developer.android.com/about/versions/jelly-bean.html>

⁴*Ice Cream Sandwich*: <http://developer.android.com/about/versions/android-4.0-highlights.html>

⁵*Honeycomb*: <http://developer.android.com/about/versions/android-3.0-highlights.html>

⁶*Gingerbread*: <http://developer.android.com/about/versions/android-2.3-highlights.html>

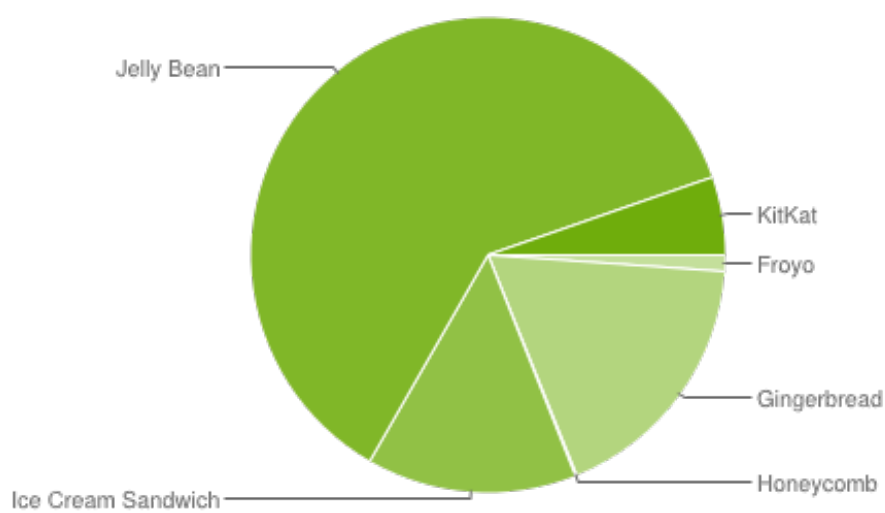


Figura 1.1: Distribuição das versões do Android

Esse material irá cobrir alguns tópicos no desenvolvimento de aplicativos para android, tais como:

- Configuração do ambiente de desenvolvimento: Como configurar o ambiente para começar a desenvolver aplicativos, os primeiros passos para criar seu primeiro aplicativo de maneira simples;
- Elementos da interface: Como projetar seu aplicativo para usar as principais interfaces. Listas, Listas compostas, Grades, Abas, Menus são as interfaces mais usadas nos diversos aplicativos no mercado; e
- Elementos de hardware: Como projetar seu aplicativo para usar as APIs de hardware: Bluetooth, GPS, SMS, Chamadas.

Para esse material, algumas convenções serão seguidas:

- Os códigos estarão sempre com a sintaxe colorida para facilitar a leitura;
- URLs das referências estarão nas notas de rodapé; e
- Dicas estarão envoltas por uma caixa para facilitar a visualização

Configuração do Ambiente

A instalação e configuração do ambiente de desenvolvimento para Android é simples, o Google fornece um pacote chamado ADT (*Android Development Tools*) que contém o ambiente Eclipse com o *plugin* do Android, algumas ferramentas para instalação dos aplicativos nos *smartphones*, o gerenciador do SDK e as imagens para o emulador do Android. Essas ferramentas são suficientes para o desenvolvimento na plataforma. O pacote ADT pode ser encontrado em: [Android SDK](http://developer.android.com/sdk/)¹.

Basta fazer o download do pacote e extrair que tudo já está pré-configurado para iniciar o desenvolvimento, portanto não há muito o que configurar.

Caso opte por utilizar uma instalação já existente do ambiente Eclipse, você pode instalar o *plugin* do Android automaticamente através da ferramenta de instalação de *plugins* do ambiente. Após a instalação será necessário abrir o *SDK Manager* e instalar:

- *Android SDK Tools*;
- *Android SDK Platform-Tools*; e
- Para cada API que você irá utilizar, instalar o *SDK Platform* e opcionalmente o *Documentation for Android SDK* e o *Samples for SDK*.

¹<http://developer.android.com/sdk/>

Linguagem do Android

3.1 Linguagem

A linguagem usada para programar na plataforma Android é Java. Então antes de engajar no aprendizado Android é altamente recomendável estudar material Java e principalmente o paradigma de orientação a objetos.

O Android tem algumas particularidades na organização e configuração que é feita através de arquivos XML específicos do Android. Alguns arquivos XML servem para configurar o aplicativo, layout de cada tela e outros dão suporte a strings para facilitar o suporte a múltiplos idiomas. Felizmente o conjunto Eclipse com ADT já cuida disso automaticamente e possui uma série de facilidades alcançadas por meio de interfaces gráficas para os programadores. Por esse motivo, para qualquer iniciante nessa área é recomendável a utilização do ambiente Eclipse.

A criação de layouts dos aplicativos pode ser feita inteiramente através da interface gráfica disponível no ambiente, no estilo *drag and drop*.

3.2 Entendendo a estrutura de uma aplicação Android

Uma aplicação Android consiste de uma ou mais *activities*. Uma *activity* é uma tela com *views* que interagem com o usuário. Como o Android segue o padrão MVC (*Model-View-Control*) as *activities* são os *controllers* e as *views*, *views*. As *activities* são classes do Java, o *layout* e outros recursos são definidos em arquivos XML.

Dentre os diversos arquivos XML existentes na configuração de um aplicativo Android o mais importante é o `AndroidManifest.xml`¹ pois é nele que se exprimem as configurações gerais do aplicativo. Nesse texto não iremos adentrar muito nos detalhes das configurações, mas apenas deixar claro que é nesse arquivo que se colocam as versões do Android que seu aplicativo será compatível com, as permissões para usar os recursos do aparelho como Internet, GPS, Bluetooth, etc.

A pasta `src/` contém o pacote com as classes do seu aplicativo isto é, o código fonte do seu aplicativo. Tanto *activities* como classes de suporte devem estar dentro do pacote.

Dentro da pasta `res/` de recursos, encontram-se outros arquivos, referentes à disposição do layout, valores de strings e imagens que sua aplicação irá utilizar. A pasta `layout/` junto com as pastas `drawable/` servem para dispor o layout. Cada *drawable* comporta imagens para um tamanho diferente de tela, enquanto que a pasta de *layout* contém a disposição geral do layout. São nesses arquivos que se colocam os itens (*views*) que irão nas telas, como botões, caixas de texto, caixas de seleção, etc.

Na pasta `values/` o mais importante é o arquivo `strings.xml` que contém os valores das strings do aplicativo. Sempre que você quiser referenciar alguma string, a mesma deverá estar expressa nesse arquivo. Fica fácil dessa forma fazer o aplicativo suportar múltiplos idiomas, pois basta traduzir esse único arquivo para alterar todos os textos do aplicativo.

A pasta `menu/` contém os *layouts* do menus do aplicativo, esses são aqueles que podem ser acessados através da *Action Bar*² ou através dos botões físicos do aparelho.

¹Documentação do `AndroidManifest`: <http://developer.android.com/guide/topics/manifest/manifest-intro.html>

²*ActionBar*: <http://developer.android.com/design/patterns/actionbar.html>

3.3 Alguns arquivos importantes:

/res/layout/activity_main.xml:

O arquivo *activity_main.xml* localizado na pasta *layout* define a interface gráfica da tela. Por padrão, ao criar o projeto esse arquivo contém uma tag `<TextView>` para exibir um simples texto na tela. Essa tag define o atributo *android:text="string/hello"*, que utiliza uma mensagem identificada pela chave *hello* localizada no arquivo *strings.xml*.

/res/values/strings.xml:

O arquivo *strings.xml* localizado na pasta *values* contém as mensagens da aplicação para organizar os textos em um único arquivo centralizado, o que é uma boa prática de programação. Desta forma podemos facilmente traduzir este arquivo para diversos idiomas e tornar nossa aplicação bastante internacionalizáveis. Por padrão, esse arquivo contém o nome da aplicação que digitamos ao criar o projeto e a mensagem que aparece na tela principal definida pelo arquivo *main.xml*. O nome da aplicação é definido pela chave *app_name*, e o texto que aparece na tela é definido pela chave *hello*. O padrão para acessar essas mensagens é `@string/nomeDaChave`. Se for necessário mais de um idioma no aplicativo, basta criar uma pasta */res/values/values-(codigo do idioma)* e traduzir o arquivo *strings.xml* desta pasta.

R.java:

A classe *R* tem constantes para facilitar acesso aos recursos do projeto, como por exemplo, um arquivo XML que define uma tela ou uma imagem localizada na pasta *drawable*. Sempre que um recurso é adicionado no projeto, como por exemplo, uma nova imagem, essa classe é gerada automaticamente pelo Eclipse para conter uma constante para o novo recurso criado.

NUNCA ALTERE A CLASSE R MANUALMENTE.

MainActivity.java:

Esta é a classe principal do projeto e representa a tela inicial da aplicação. Observe que essa classe é filha de *android.app.Activity*. A classe *android.app.Activity* representa uma tela da aplicação e é responsável por controlar o estado e os eventos da tela. Assim, para cada tela da aplicação você criará uma classe-filha de *Activity*. O método *onCreate(bundle)* precisa ser implementado obrigatoriamente e é chamado de forma automática pelo Android quando a tela é criada. No entanto, a classe *android.app.Activity* não sabe desenhar nada na tela e para

isso precisa da ajuda da classe *android.view.View* que, por sua vez, se encarrega de desenhar os componentes visuais, como campos de texto, botões e imagens. Para isso, existem diversas subclasses especializadas de *android.view.View*.

AndroidManifest.xml:

O arquivo *AndroidManifest.xml* é o arquivo principal do projeto e centraliza as configurações da aplicação. Note que existe, no arquivo xml a tag *uses-sdk* a qual é utilizada para informar o level mínimo da API exigido pela aplicação. Por exemplo, uma aplicação pode ser compatível com a API level 8 mas otimizado para a 19, isto é, o Android 4.4: `<uses-sdk android:minSdkVersion="8" android:targetSdkVersion="19"/>`

Resumindo:

- `AndroidManifest.xml`: Configurações gerais do aplicativo;
- `src/`: Classes do aplicativo; e
- `res/`: Recursos do aplicativo tais que:
 - `strings/`: Todos os textos da sua aplicação, suporte a múltiplos idiomas;
 - `layout/`: Todos os *layouts* de suas telas (*activities*);
 - `drawable/`: Todas as imagens, separados por tamanho de tela; e
 - `menu/`: *layout* dos menus do aplicativo.

Criando seu primeiro aplicativo

Para exemplificar a criação de um aplicativo, seguiremos o exemplo dado pelo próprio manual do Google sobre o Android (Ver original¹). Trata-se de aplicativo simples do tipo "Hello World".

Iniciaremos criando um novo projeto no Eclipse acessando o menu: *File -> New -> Android Application Project*.

Na janela que apareceu você deve colocar o nome do aplicativo, do projeto e do pacote. O nome do pacote deve seguir a convenção do Java².

- *Minimum Required SDK*: É a versão mínima do sistema operacional Android que sua aplicação irá suportar, o mais comum é a versão 8 do SDK que se refere ao Android 2.2. Alguns tipos de layouts mais complexos não são suportados em versões mais antigas;
- *Target SDK*: É a versão principal do Android para qual seu aplicativo está sendo desenvolvido;
- *Compile With*: Versão do Android com qual seu aplicativo será compilado; e
- *Theme*: Cores do layout.

¹Original em: <http://developer.android.com/training/basics/firstapp/creating-project.html>

²Convenção sobre nome dos pacotes: <http://docs.oracle.com/javase/tutorial/java/package/namingpkgs.html>

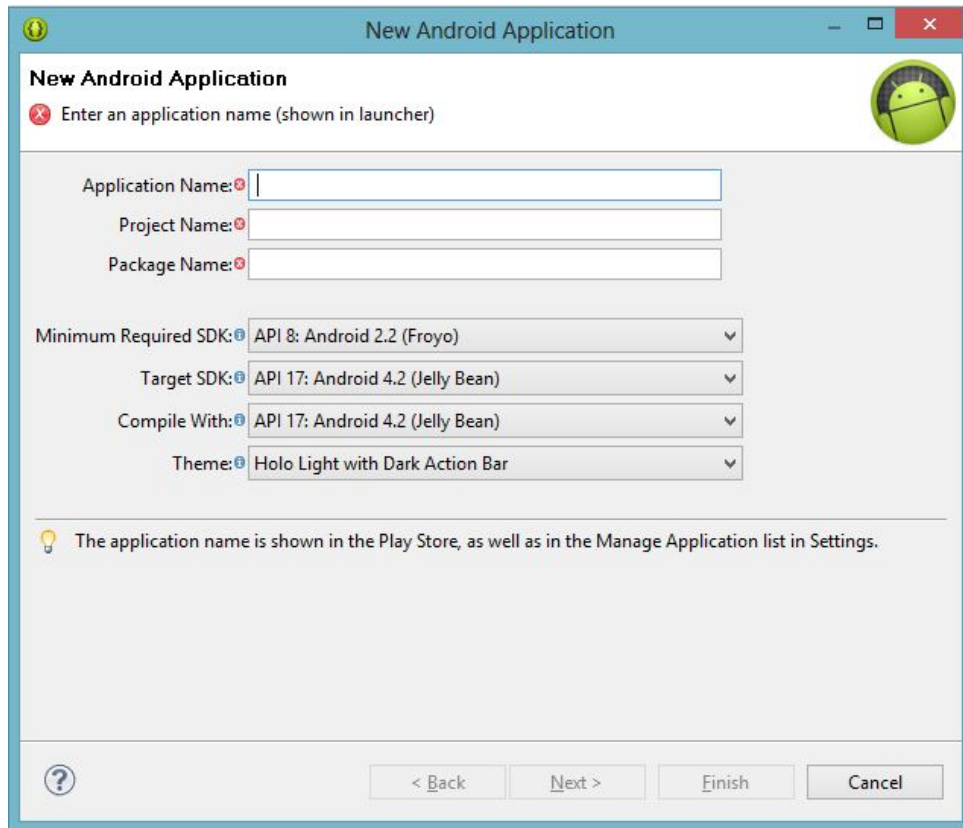


Figura 4.1: Primeira janela de criação de novo aplicativo

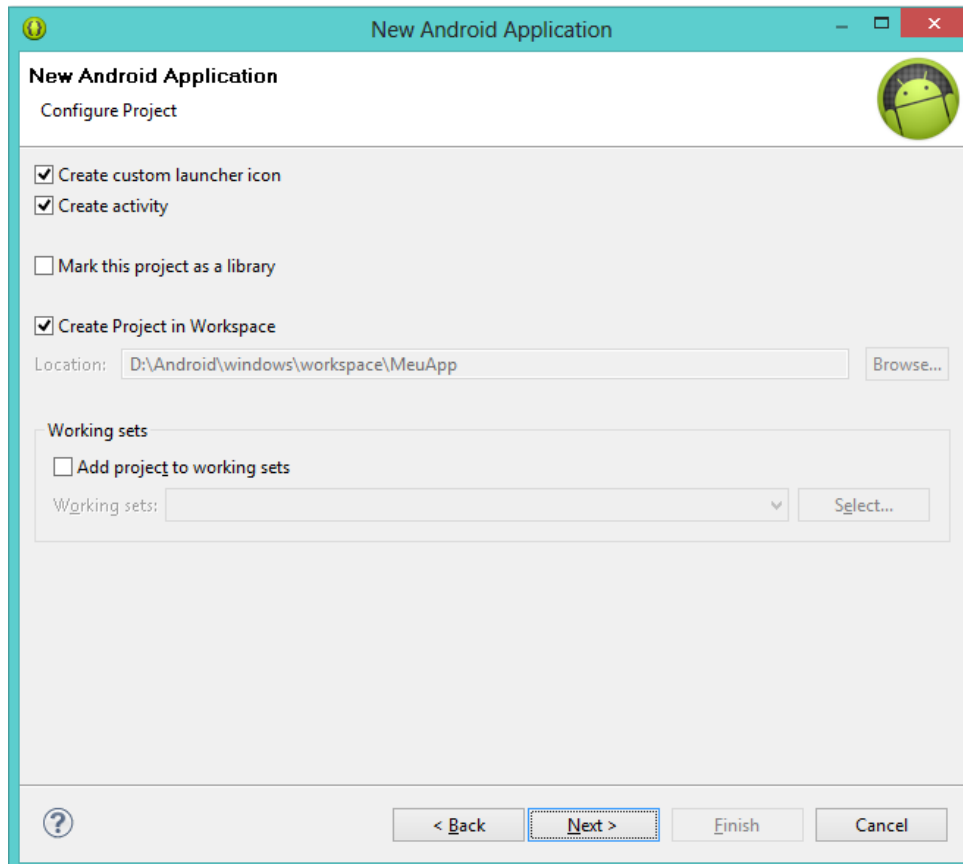


Figura 4.2: Segunda janela de criação de novo aplicativo

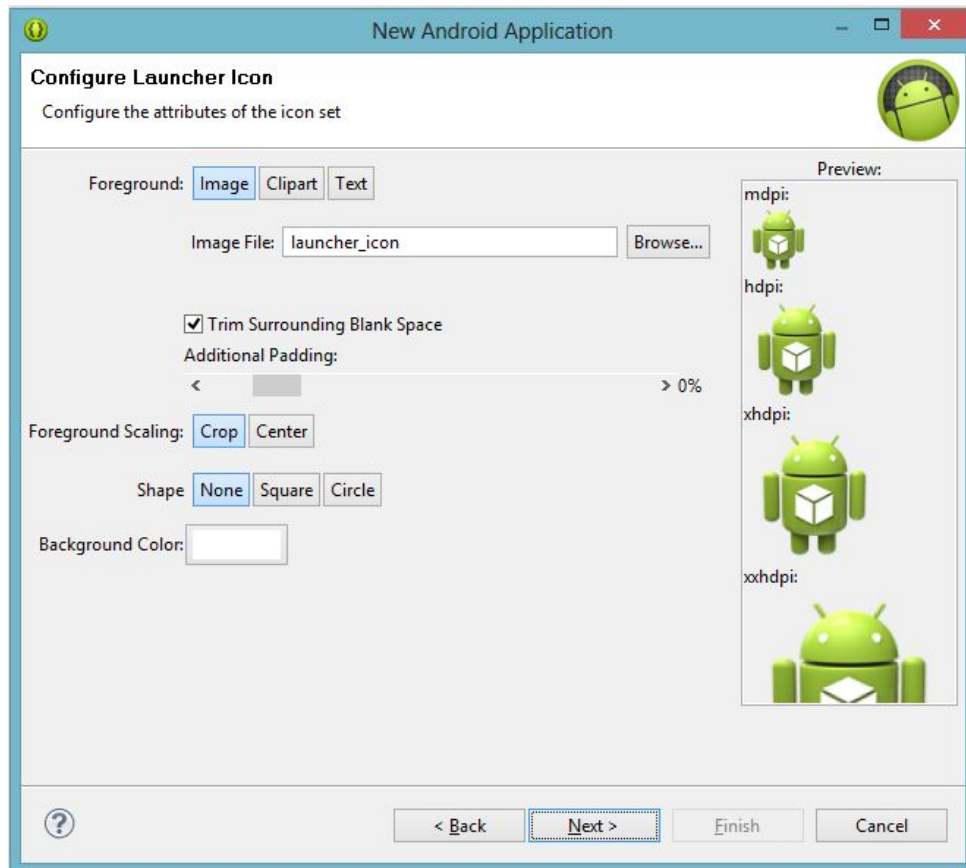


Figura 4.3: Terceira janela de criação de novo aplicativo

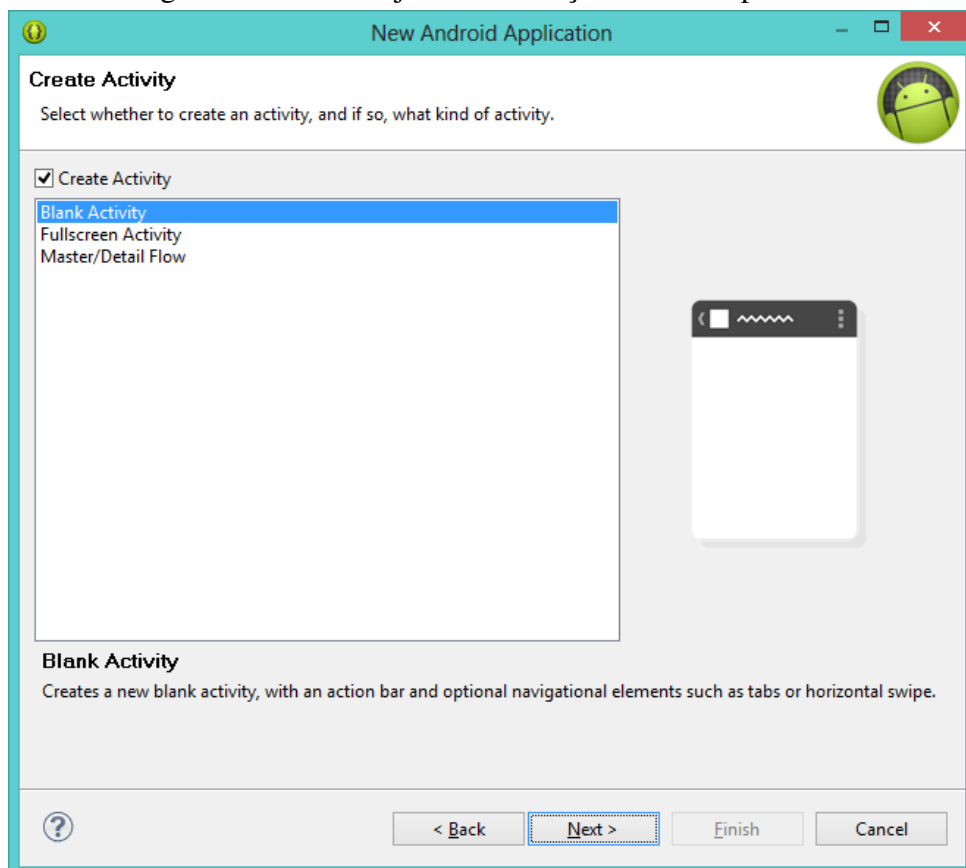


Figura 4.4: Quarta janela de criação de novo aplicativo

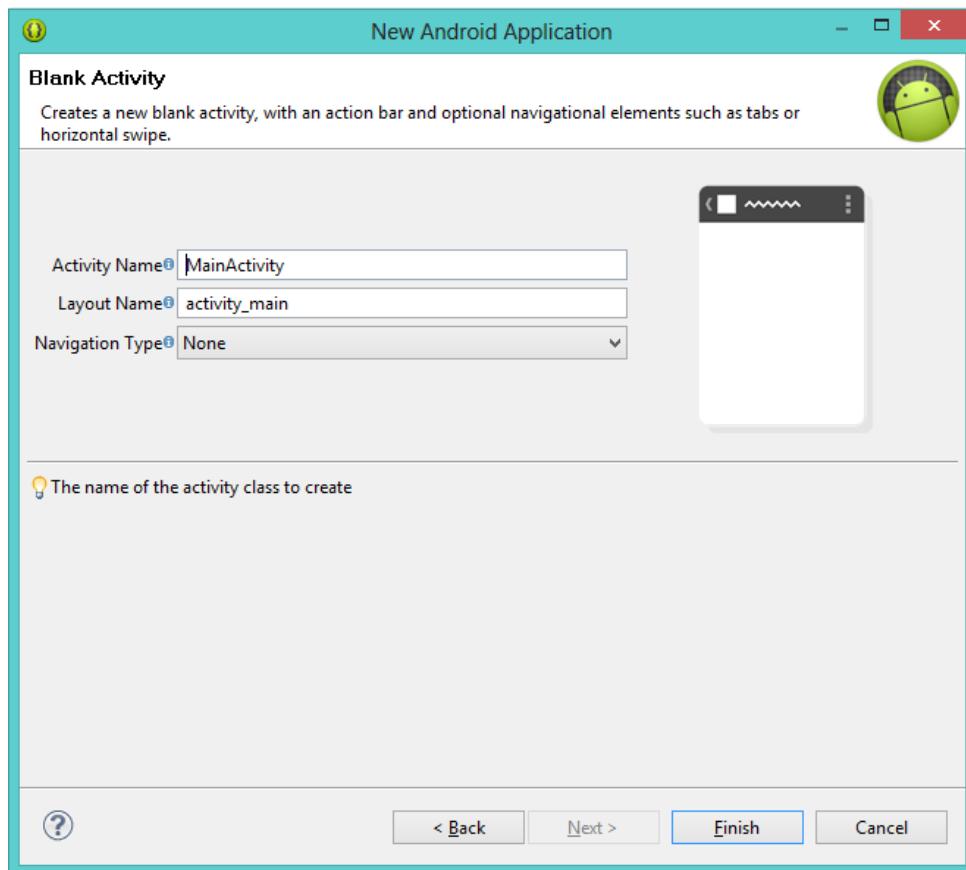


Figura 4.5: Quinta janela de criação de novo aplicativo

Observe na Figura 4.1 a janela de criação de uma nova aplicação Android. Em *Application Name* você deve colocar o nome do aplicativo, em *Project Name*, o nome do projeto e em *Package Name* o nome do pacote. Para esse exemplo utilizaremos como *Minimum Required SDK* a versão API 8, já que nesse exemplo não usaremos nenhum layout que não é suportado em versões mais antigas. Em *Target SDK* e *Compile With* optaremos pela versão mais nova, a API 17. Por final o *Theme* eu optei pelo *Holo Light with Dark Action Bar* que é um tema com fundo branco e barra superior preta, um dos padrões do Android.

Dica: Para obter o máximo de compatibilidade sempre procure utilizar *layouts* compatíveis com versões antigas, observe na figura 1.1 que versões antigas ainda tem uma fatia considerável do mercado.

A figura 4.2 mostra a segunda janela da configuração inicial do seu aplicativo. Você pode escolher um ícone personalizado se marcar a caixa *Create custom launcher icon* o que te levará para a janela da figura 4.3. Se marcar *Create Activity* o assistente de criação te levará para a janela da figura 4.4 onde poderá escolher qual *activity* vai ser criada para seu aplicativo. Em todos os exemplos escolheremos a opção *Blank Activity*. Como nosso projeto não é uma biblioteca não marcaremos *Mark this project as a library*. Se marcar *Create Project in Workspace* o assistente irá salvar o projeto na pasta que foi configurada para o *Workspace*, caso contrário ele irá pedir para escolher outro caminho. Como não trabalharemos com *Working Sets* do Eclipse, a opção *Add project to working sets* permanece desmarcada.

Finalmente a figura 4.5 mostra a janela para nomear a *activity* inicial, nesse exemplo mantive *MainActivity*. O nome do *layout* dessa *activity* mantive como *activity_main* que é o padrão. Na caixa *Navigation Type* existem algumas opções de *layout* pré-definidas pelo Android. São elas:

- *None*: O *layout* vem apenas com uma *Action Bar*³
- *Fixed Tabs + Swipe*: O *layout* vem com algumas abas e com gesto de arrastar entre as abas (*activities*) pré-programados.⁴
- *Scrollable Tabs + Swipe*: O *layout* vem com algumas abas e com gesto de arrastar entre as abas pré-programados, porém nesse o estilo das abas é diferente, em vez de abas fixas,

³Documentação da *ActionBar*: <http://developer.android.com/guide/topics/ui/actionbar.html>

⁴*Tabs*: <http://developer.android.com/design/building-blocks/tabs.html>

são abas que movem para dar espaço a outras.

- *Dropdown*: O *layout* vem com a troca de *activities* através de um menu na *Action Bar*.

Você pode configurar a versão do SDK manualmente modificando os valores no *manifest*.

Como mostrado no exemplo abaixo:

```
1 <uses-sdk
2     android:minSdkVersion="8 "
3     android:targetSdkVersion="17" />
```

Algoritmo 4.1: Exemplo de configuração de versão do SDK no arquivo `AndroidManifest.xml`

A tag `uses-sdk` serve apenas para o compilador saber quais versões do Android você pretende que seu aplicativo suporte. Dessa forma quando seu aplicativo for lançado na loja *Google Play* o aplicativo só será visível para aqueles usuários que possuem a versão mínima do Android indicada no atributo.

Primeiro vamos criar um *layout* para o aplicativo usando o construtor de interfaces presente no ambiente, primeiro abra o arquivo `res/layout/activity_main.xml`, segundo o *manifest*, é essa *activity* que será aberta quando o aplicativo for iniciado, isso é configurado através do *intent-filter*⁵.

Selecione o "Hello world" e o remova da sua *activity*.

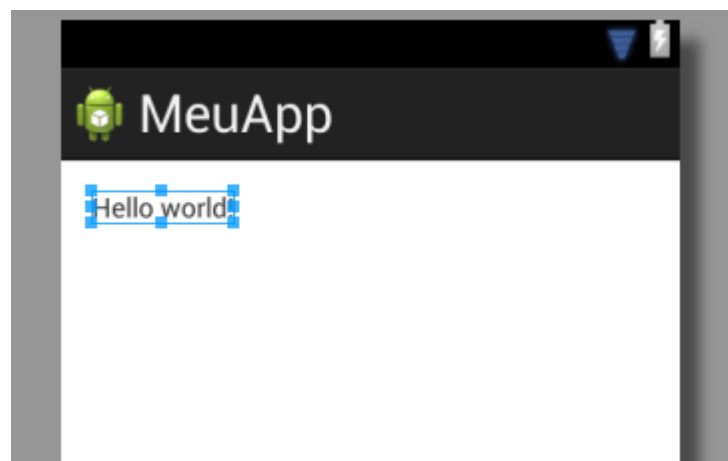
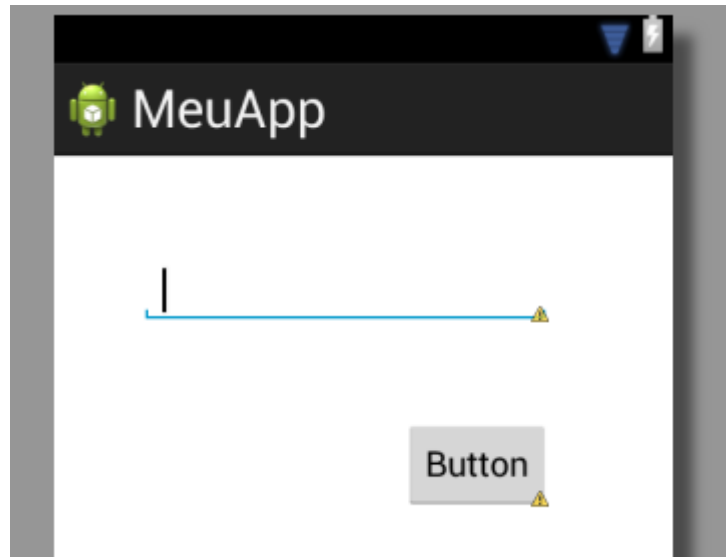


Figura 4.6: Selecionando o Hello world

A tela deverá ficar parecida com a da figura 4.4. Agora arraste um *Text Field* -> *Plain Text* e um *Form Widgets* -> *Button* para sua *activity*.

⁵Mais informações na seção ??

Figura 4.7: *activity* com os elementos colocados na tela

Ao clicar duas vezes no elemento no modo visual, você será levado ao marcador desse elemento no XML correspondente da *activity*. Clique duas vezes na caixa de texto, o seguinte código será exibido:

```
1 <EditText
2   android:id="@+id/nameField"
3   android:layout_width="wrap_content"
4   android:layout_height="wrap_content"
5   android:layout_alignParentLeft="true"
6   android:layout_alignParentTop="true"
7   android:layout_marginLeft="28dp"
8   android:layout_marginTop="35dp"
9   android:ems="10"
10  android:hint="@string/name">
11  <requestFocus />
12 </EditText>
```

Algoritmo 4.2: Código da caixa de texto no arquivo `activity_main.xml`

Primeiro, na linha 2 modifique o *id* do *Text Field* para um nome mais intuitivo, nesse exemplo chamaremos apenas de *nameField*. O Android definiu que todo novo atributo *id* deve ser precedido de `@+id/`. O símbolo `@` diz para o compilador que estamos acessando os recursos do Android, esses recursos são compilados na classe `R` automaticamente. O símbolo `+` diz para o compilador que estamos criando um novo recurso. Por fim, *id* diz que estamos especificando um novo identificador para esse recurso e só então damos o nome a esse identificador.

Dica: Existem vários tipos de recursos, porém é importante salientar os diferentes tipos de *id*. Quando referimos aos recursos podemos usar `@android:id/` para acessar recursos que já estão definidos no sistema Android. Usamos `@id/` para acessar recursos que já foram definidos no seu projeto. Para criar um novo recurso, usamos `@+id/`.

Os outros atributos são para definir o tamanho, alinhamento e margem da caixa de texto. O valor `wrap_content` dos atributos `layout_width` e `layout_height` (linhas 3 e 4) força a *view* a mudar de tamanho automaticamente para abrigar seu conteúdo. Os atributos `layout_alignParentLeft` e `layout_alignParentTop` servem (linhas 5 e 6) para alinhar essa *view* com a *view* pai dela, dessa forma ficará alinhado com a borda esquerda e com a borda superior do pai. Os atributos `layout_marginLeft` e `layout_marginTop` (linhas 7 e 8) deslocam o elemento colocando uma margem entre a borda e a *view*, esses valores estarão diferentes pois são computados automaticamente quando a *view* é colocada através do construtor de interfaces. Note que isso só acontecerá caso esteja usando `RelativeLayout`⁶ que é o nosso caso. Por último o atributo `ems` (linha 9) configura o tamanho da fonte através da unidade de medida `Em`.

Depois adicione uma *hint* para essa caixa de texto, uma *hint* é algo que vai estar escrito na caixa de texto quando ela estiver vazia, indicando que tipo de texto você pretende que seja escrito nessa caixa de texto, Neste exemplo (linha 10) a *hint* é uma referência a *string* chamada *name* que iremos definir depois.

Depois modifique o código do botão que está no mesmo arquivo, troque o *id* do botão (linha 2), também edite o atributo `text` (linha 8) para fazer uma referência a uma *string* definida no arquivo de *strings* que iremos chamar de *send_button*. Por último adicione um atributo `onClick` (linha 9) que define o método que será chamado quando esse botão for pressionado.

```

1 <Button
2     android:id="@+id/sendButton"
3     android:layout_width="wrap_content"
4     android:layout_height="wrap_content"
5     android:layout_alignRight="@id/nameField"
6     android:layout_below="@id/nameField"
7     android:layout_marginTop="48dp"
8     android:text="@string/send_button"
9     android:onClick="sendMessage" />

```

Algoritmo 4.3: Código do botão

Agora iremos definir as *strings* usadas anteriormente no arquivo `res/values/strings.xml`. Abra ele e o modifique para que fique como mostrado no Algoritmo 4.4.

```

1 <resources>
2     <string name="app_name">MeuApp</string>
3     <string name="action_settings">Settings</string>
4     <string name="send_button">Enviar</string>
5     <string name="name">Nome</string>
6 </resources>

```

Algoritmo 4.4: Arquivo de strings com as duas strings adicionadas

Após terminar abra a classe `MainActivity.java` localizada na pasta `src/com.example.meuapp` do seu projeto e adicione um novo método que chamei de *sendMessage*, ele será responsável

⁶Mais informações na seção 5.3.2

por obter o conteúdo da caixa de texto e enviar para uma nova *activity* que irá mostrar esse conteúdo.

```
1 public void sendMessage(View view) {  
2     // Fazer alguma coisa em resposta ao clique do botao  
3 }
```

Algoritmo 4.5: Adicionando método à classe MainActivity

Dica: Isso vai requer você importe a classe View, você pode apertar Ctrl+Shit+O no Eclipse para importar classes que estejam faltando

```
1 import android.view.View;
```

Algoritmo 4.6: Exemplo de import de uma classe Android

Primeiro, crie um novo Intent⁷, um Intent é um objeto que provê uma facilidade para realizar uma ligação entre códigos de diferentes aplicações. O uso mais signficante é a inicialização de novas *activities*.

```
1 public void sendMessage(View view) {  
2     // Fazer alguma coisa em resposta ao clique do botao  
3     Intent intent = new Intent(this, DisplayMessageActivity.class);  
4 }
```

Algoritmo 4.7: Adicionando uma Intent

Agora iremos obter o texto que está escrito na caixa para fazer algo com ele, no caso iremos enviar para outra *activity* que irá mostrar esse texto. Como é feito no algoritmo 4.7.

```
1 public void sendMessage(View view) {  
2     Intent intent = new Intent(this, DisplayMessageActivity.class);  
3     EditText textBox = (EditText) findViewById(R.id.nameField);  
4     String message = textBox.getText().toString();  
5     intent.putExtra(EXTRA_MESSAGE, message);  
6     startActivity(intent);  
7 }
```

Algoritmo 4.8: Obtendo o conteúdo da caixa de texto e enviando para outra activity

⁷Documentação Intent: <http://developer.android.com/reference/android/content/Intent.html>

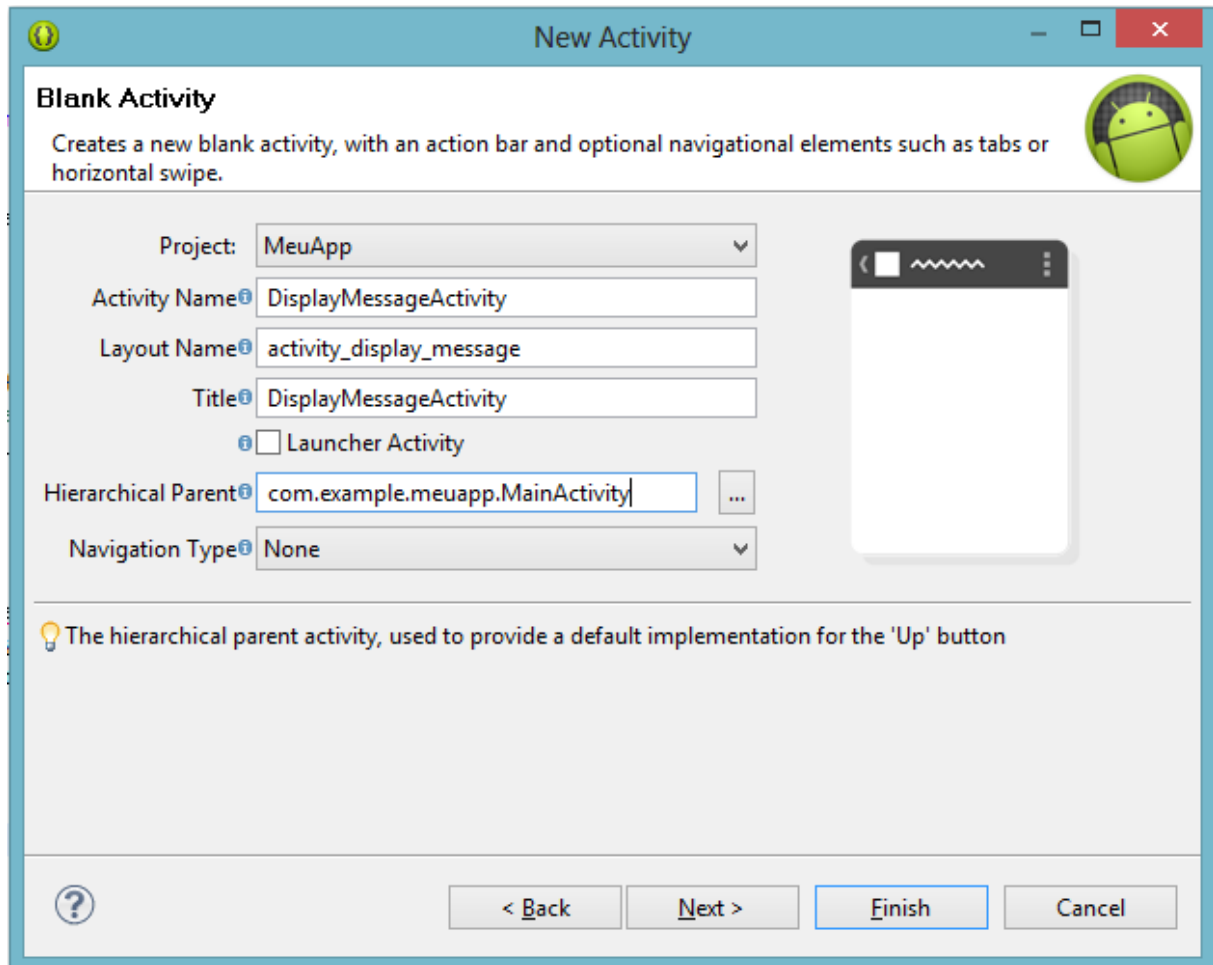
O código na linha 3 está obtendo a referência da caixa de texto usando o método `findViewById()` passando o *id* da caixa de texto como parâmetro, esse *id* é obtido acessando uma variável estática da classe `R` (observe que esse é o mesmo *id* que você colocou no arquivo `xml` do layout da *activity*). Em seguida usando o método `getText()` da caixa de texto, obtém-se a *string* que foi escrita pelo usuário.

Por fim, essa *string* é colocada no `Intent` com o método `putExtra()`, uma `Intent` pode carregar consigo uma coleção de vários tipos de dados como pares chave-valor chamados *extras*, esse método toma a chave como primeiro parâmetro e o valor no segundo parâmetro. Para que a próxima *activity* consiga coletar esse valor, você deve definir uma chave para seu *extra* usando uma constante pública. Para isso adicione a definição de `EXTRA_MESSAGE` no topo da sua classe `MainActivity`.

```
1 public class MainActivity extends Activity {  
2     public final static String EXTRA_MESSAGE  
3     = "com.example.meuapp.MESSAGE";  
4     ...  
5 }
```

Algoritmo 4.9: Constante como chave para um extra

Agora você deve criar uma nova *activity*, para isso vá em *File -> New -> Other -> Android Activity* e selecione *Blank Activity*. Preencha a próxima janela como na figura 4.8, depois clique *Finish*.

Figura 4.8: Criando uma nova *activity*

Observe a figura 4.8. Em *Project* você vai especificar o projeto em que a nova *activity* será adicionado. Em *Activity Name* especifique o nome da sua nova *activity*. Em *Layout Name* defina o nome do arquivo XML que contém o *layout* da nova *activity*. A opção *Title* define o título da *activity*, isso pode ser modificado posteriormente no arquivo de *strings* pois o título será definido ali após a criação da *activity*. A opção *Launcher Activity* ficará desmarcada pois essa *activity* não será usada para inicializar o aplicativo. Em *Hierarchical Parent* você vai definir o pai da nova *activity*, isso é usado para o Android implementar corretamente para qual *activity* o botão de voltar irá voltar. Por último *Navigation Type* deixe como *None* pois só queremos o *design* padrão. Clique em *Finish* para criar a nova *activity*.

Dica: O Eclipse adiciona automaticamente *activities* criadas por esse método no *Manifest*. Observe no *Manifest* como é feito caso você precise adicionar manualmente.

Abra a nova classe que foi criada junto com a *activity*. A classe já vem com alguns métodos implementados, alguns não serão necessários para esse aplicativo e serão explicados em outras seções, mas mantenha-os na classe. Todas as classes que são subclasses de *Activity* precisam implementar o método `onCreate()`⁸ que define o procedimento a ser executado quando a *activity* é criada.

⁸[http://developer.android.com/reference/android/app/Activity.html#onCreate\(android.os.Bundle\)](http://developer.android.com/reference/android/app/Activity.html#onCreate(android.os.Bundle))

Agora, precisamos extrair os dados enviados a essa *activity* através do *intent*, você pode obter a referência do *intent* que começou a *activity* chamando o método `getIntent()`⁹.

```
1 Intent intent = getIntent();
2 String mensagem = intent.getStringExtra(MainActivity.EXTRA_MESSAGE);
```

Algoritmo 4.10: Obtendo a *string* passada como extra do Intent

Após obter a referência do *intent* que iniciou a *activity*, queremos coletar os extras que foram passado junto com ele. Criamos uma *string* que irá armazenar a mensagem que veio junto do *intent* e chamamos o método `getStringExtra()` passando como parâmetro a chave desse extra, que definimos na classe `MainActivity`. Agora para mostrar a mensagem na tela, você precisa criar um `TextView`¹⁰, essa *view* serve para mostrar texto.

```
1 @Override
2 protected void onCreate(Bundle savedInstanceState) {
3     super.onCreate(savedInstanceState);
4
5     // Show the Up button in the action bar.
6     setupActionBar();
7
8     //Obtem o conteudo da Intent
9     Intent intent = getIntent();
10    String mensagem = intent.getStringExtra(MainActivity.EXTRA_MESSAGE);
11
12    //Cria o TextView
13    TextView textView = new TextView(this);
14    textView.setTextSize(40);
15    textView.setText("Hello " + mensagem);
16
17    //Estabelece o text view como o layout da atividade
18    setContentView(textView);
19 }
```

Algoritmo 4.11: Método `onCreate()` recebendo um *Intent* e mostrando a mensagem

As linhas 3, 5 e 6 foram colocadas automaticamente na criação da *activity*, a linha 3 faz uma chamada ao método da superclasse, a linha 6 é um método que inicializa a *Action Bar*, que nesse aplicativo é a barra superior com o nome da *activity* e um menu de opções que já vem pré-programado. Deixamos isso como está.

O algoritmo 4.10 foi colocado nas linhas 9 e 10 para obter a referência ao *Intent*. Nas linhas 12-15 criamos um novo `TextView`, configuramos o tamanho da fonte e atribuímos o texto que será mostrado na tela a *view*, respectivamente.

Agora que o aplicativo está pronto, é necessário testar, caso tenha um smartphone Android você pode conectá-lo no seu computador e rodar diretamente, senão você deverá rodar em um

⁹[http://developer.android.com/reference/android/app/Activity.html#getIntent\(\)](http://developer.android.com/reference/android/app/Activity.html#getIntent())

¹⁰<http://developer.android.com/reference/android/widget/TextView.html>

emulador. Lembrando que para ambos os casos é necessária a instalação do SDK primeiro, acesse *Android SDK Manager* e faça o download do SDK desejado.

Para rodar diretamente no smartphone:

1. Conecte seu smartphone no computador através do cabo USB. Se estiver desenvolvendo no Windows será preciso instalar os drivers USB do seu dispositivo. Se precisar de ajuda para instalar os drivers acesse: [OEM USB](http://developer.android.com/tools/extras/oem-usb.html)¹¹
2. Ative o modo *USB Debugging* no dispositivo
 - Para Android 3.2 ou mais antigos, a opção deve estar em *Configurações -> Aplicativos -> Desenvolvimento*
 - Para Android 4.0 e 4.1, a opção está em *Configurações -> Opções do desenvolvedor*
 - Para Android 4.2 e mais novos, a opção está escondida por padrão, para mostrar a opção você deve entrar em *Sobre o telefone* e clicar em *Número da versão* 7 vezes, ao retornar para tela anterior deverá aparecer *Opções do desenvolvedor*

Dica: Caso ocorra o erro *Launch error: adb rejected command: device not found*. Verifique se o aparelho está conectado e se os drivers estão instalados corretamente. Na área de notificações do aparelho deve ter uma notificação escrita: *Android debugging enabled*.

¹¹<http://developer.android.com/tools/extras/oem-usb.html>

Para rodar no emulador:

1. Abra o *SDK Manager* através do Eclipse em: *Window -> Android SDK Manager*
2. Verifique se, para Android 4.2.2 (API 17) ou outro desejado os seguintes pacotes estejam instalados
 - *SDK Platform* e;
 - *ARM EABI v7a System Image* ou;
 - *Intel x86 Atom System Image*
3. Verifique também se na aba *Tools*, os pacotes *Android SDK Tools* e *Android SDK Platform-tools* estão instalados
4. Agora é necessário criar um AVD (Android Virtual Device¹²). No Eclipse acesse o menu *Window -> Android Virtual Device Manager*
5. No AVD Manager clique em *New*
6. Complete as informações do AVD, especificando um aparelho, nome, plataforma, espaço de armazenamento, quantidade de memória RAM. Em *Device* haverá opções pré-configuradas de aparelhos do google, os *Nexus*, e opções genéricas de acordo com tamanho de tela. Em *Target* você deverá escolher a versão do sistema Android que deseja. Em alguns casos você poderá decidir pela CPU caso deseje ARM ou Intel Atom x86. A quantidade de RAM no Windows fica limitada a 768MB, mais que isso pode acarretar em erros no sistema.
7. Clique *Create AVD*
8. Ainda na janela *Android Virtual Device Manager* selecione o novo AVD e clique *Start*
9. Quando o emulador terminar de carregar, destrave a tela do emulador, usando o mouse.

Agora para rodar o aplicativo basta clicar em *Run* na barra de tarefas do Eclipse e selecionar *Android Application* na janela *Run as*. O Eclipse irá instalar o APK e abrir o aplicativo automaticamente, no dispositivo ou no emulador. As figuras 4.9, 4.10 e 4.11 mostram a execução do aplicativo.

¹²<http://developer.android.com/tools/devices/index.html>

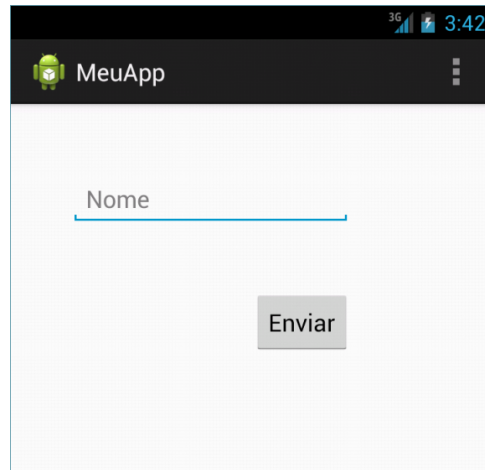


Figura 4.9: Primeira tela do primeiro aplicativo

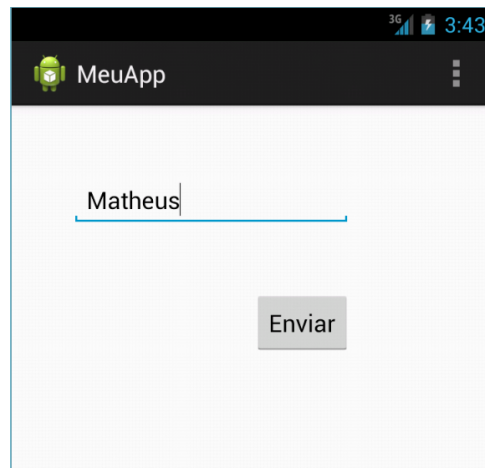


Figura 4.10: Primeira tela após escrever texto na caixa de texto

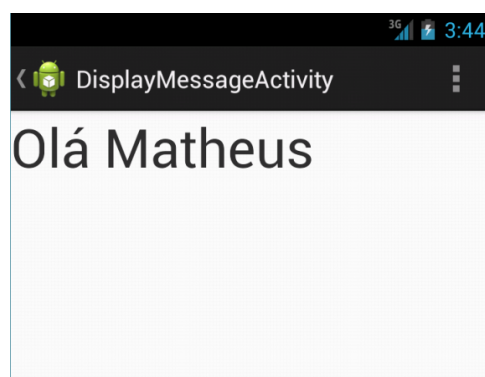


Figura 4.11: Segunda tela mostrando a mensagem enviada

Design

5.1 Activity

Enquanto um usuário navega pelas variadas telas de um aplicativo, sai dele e volta depois, as instâncias de uma *activity* transitam dentre diferentes estados em seu ciclo de vida. Quando um aplicativo é iniciado, uma *activity* inicial é criada o sistema invoca métodos específicos que correspondem a criação dessa *activity*. Durante todo o ciclo de vida vários métodos são chamados, e todos eles correspondem a diferentes estágios desse ciclo de vida.

Observe na imagem abaixo os métodos correspondentes a cada estado da vida de uma *activity*, quando ela é criada o método `onCreate()` é o responsável pela configuração inicial. O sistema ao criar uma nova instância de uma *activity*, cada método muda o estado da *activity* um degrau pra cima na pirâmide.

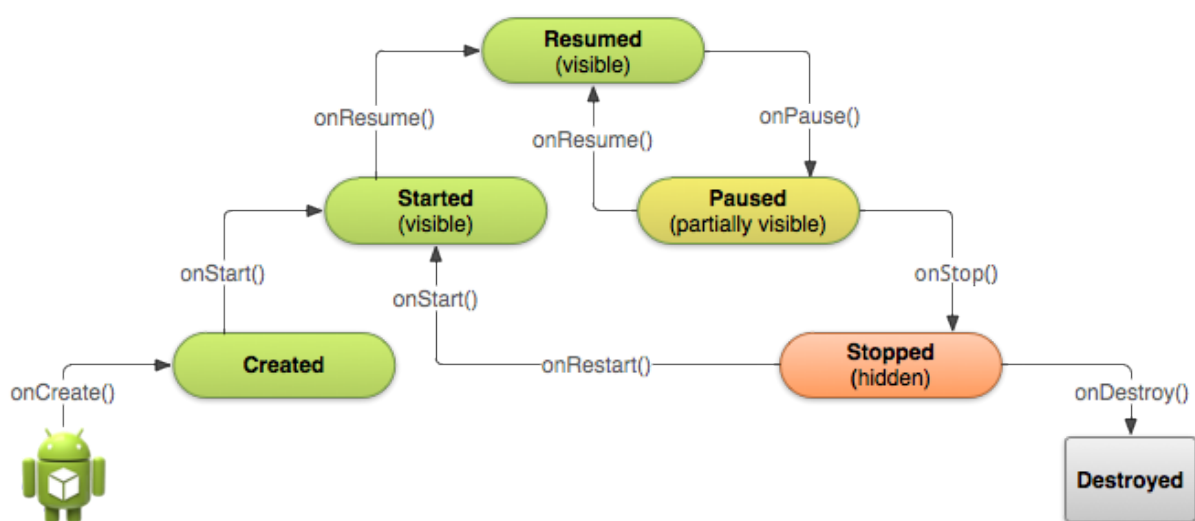


Figura 5.1: Ciclo de vida de uma *activity*

Assim que o usuário começa a sair da *activity*, o sistema invoca outros métodos que movem o estado para níveis mais baixos da pirâmide para começar a desmontar a *activity*. Em alguns

casos a *activity* irá apenas ir até certo ponto e esperar (por exemplo quando o usuário troca para outro aplicativo) tal que ela possa voltar de onde parou caso o usuário volte.

Não são todos métodos que precisam ser implementados pois isso irá depender da complexidade do seu aplicativo. É importante salientar porém que, implementar esses métodos irá garantir que seu aplicativo se comporte de maneira correta, por exemplo você deve garantir que:

- Seu aplicativo não falhe quando o usuário receber uma chamada telefônica ou quando o usuário troca de aplicativo;
- Seu aplicativo não consuma recursos do sistema enquanto não estiver sendo usado;
- Seu aplicativo não perca o progresso do usuário; e
- Seu aplicativo não falhe ou perca o progresso do usuário quando a tela rotaciona entre retrato e paisagem.

Apenas três dentre os estados são estáticos, isto é, a *activity* pode ficar nesse estado por um longo período de tempo:

Retomado (*Resumed*)

Nesse estado a *activity* está em primeiro plano e o usuário pode interagir com ela.

Pausado (*Paused*)

Nesse estado a *activity* está parcialmente obscurecida por outra *activity* - a outra *activity* que está em primeiro plano é semi-transparente ou não ocupa todo espaço da tela. A *activity* quando pausada não consegue interagir com o usuário e não executa nenhum código.

Parado (*Stopped*)

Nesse estado a *activity* está completamente oculto e não está visível para o usuário, está em plano de fundo. Quando está parada, uma instância de uma *activity* e toda informação de seu estado tais como variáveis são mantidos, porém a *activity* não executa nenhum código.

5.2 Especifique a *activity* que inicia seu aplicativo

Quando um usuário abre um aplicativo, o sistema chama o método `onCreate()` da *activity* que foi declarada como sendo a iniciadora do aplicativo. Você pode definir qual *activity* que vai iniciar seu aplicativo no arquivo `AndroidManifest.xml` que está no diretório raiz do seu projeto.

A *activity* que inicia seu aplicativo deve ser declarada no manifesto com um `<intent-filter>`¹ que inclui a `<action> MAIN` e a `<category> LAUNCHER`. Por exemplo:

```
1 <activity android:name=".MainActivity" android:label="@string/app_name">
2   <intent-filter>
3     <action android:name="android.intent.action.MAIN" />
4     <category android:name="android.intent.category.LAUNCHER" />
5   </intent-filter>
6 </activity>
```

Algoritmo 5.1: Exemplo de *Launcher activity*

¹ Documentação `<intent-filter>`: <http://developer.android.com/guide/topics/manifest/intent-filter-element.html>

Dica: Quando você cria um projeto Android no Eclipse, por padrão é incluída uma classe *activity* que está declarada no manifesto com esse filtro.

5.3 Tipos de *Layout*

Uma *Activity* contém *Views* e *ViewGroups*. Uma *view* é um elemento que têm presença na tela do dispositivo tais como botões, textos, imagens e etc. Um *ViewGroup* por sua vez é um elemento agrupador de *views* que provê um *layout* na qual você pode ajustar a ordem e aparição das *views*.

5.3.1 *LinearLayout*

O *LinearLayout* arranja *views* em uma única coluna ou uma única linha, desse modo as *views* podem ser arranjadas verticalmente ou horizontalmente. Como mostrado na figura 5.2:

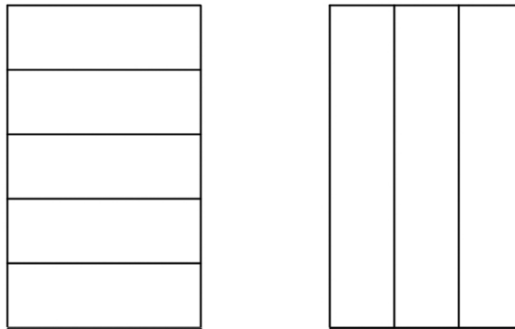


Figura 5.2: *LinearLayout* vertical (à esquerda) e horizontal (à direita)

ViewGroups também podem ser agrupados entre si para a criação de layouts mais complexos, por exemplo é possível agrupar um *LinearLayout* horizontal dentro de um vertical dessa forma é possível colocar *views* lado a lado em uma camadas do *LinearLayout* vertical, representada na figura 5.3.

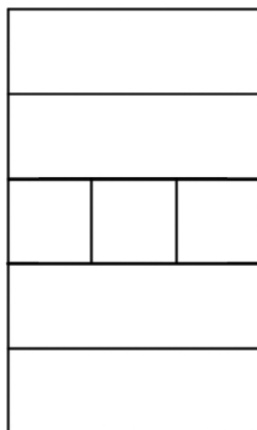


Figura 5.3: *LinearLayout* composto

5.3.2 RelativeLayout

O `RelativeLayout` permite especificar como as *views* são posicionadas uma em relação a outra. Cada *view* embutida no interior de um `RelativeLayout` tem atributos que permitem o seu alinhamento com outras *views*. Esses atributos podem ser encontrados na [documentação](#)²

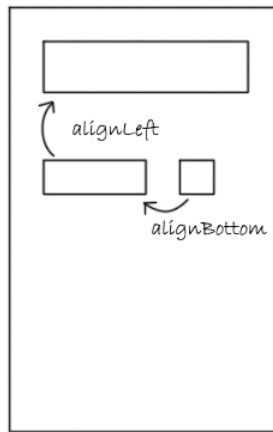


Figura 5.4: Exemplo de `RelativeLayout`

Novamente cabe comentar que é possível aninhar diferentes *ViewGroups* para formar um *layout* com maior complexidade.

5.3.3 FrameLayout

O `FrameLayout` é o mais simples e eficiente tipo de *layout*, pode ser usado apenas para mostrar uma *view* ou *views* que se sobrepõem. Geralmente é usado como um recipiente para os `Fragments`³.

Uma *view* definida em um `FrameLayout` sempre será colocado no canto superior esquerdo da tela do dispositivo ou do *ViewGroup* a que pertence o `FrameLayout`. Se mais de uma *view* foi definida elas serão empilhadas uma em cima da outra. Isso significa que a primeira *view* adicionada ao `FrameLayout` será mostrada na base da pilha, e a última adicionada será mostrada no topo.

Você pode fazer com que as *views* não sobreponham as outras usando o atributo `layout_gravity`⁴, dessa forma uma *view* pode ficar posicionada na borda inferior e outra na borda superior e não ficarem sobrepostas.

É possível posicionar as *views* dentro de um `FrameLayout` usando parâmetros diferentes no `layout_gravity`, no exemplo da figura 5.5 existe um `FrameLayout` com 3 elementos e cada um com parâmetros diferentes. É possível combinar os parâmetros utilizando a barra reta '|'.

²<http://developer.android.com/reference/android/widget/RelativeLayout.LayoutParams.html>

³Mais informações na seção 5.8

⁴<http://developer.android.com/reference/android/widget/FrameLayout.LayoutParams.html>

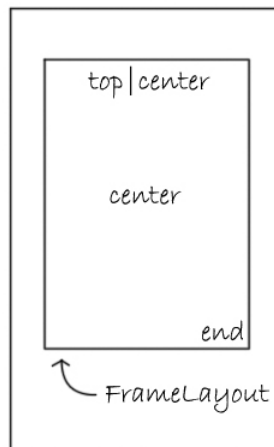


Figura 5.5: `FrameLayout` com exemplo de posicionamento usando `layout_gravity`

5.3.4 `TableLayout`

`TableLayouts` podem ser usadas para apresentar dados tabulados ou alinhar conteúdo como tabelas HTML em uma página web. Um `TableLayout` é composto de `TableRows`, uma para cada linha da tabela. Os conteúdos das `TableRows` são as *views* que vão em cada célula da tabela. Cada linha terá zero ou mais células e cada célula pode conter uma *view*.

O aspecto da `TableLayout` vai depender de alguns fatores. Primeiro, o número de colunas da tabela inteira vai depender do número de colunas da linha que contém mais colunas. Segundo, a largura de cada coluna é definida como a largura do conteúdo mais largo da coluna. Você pode combinar colunas para formar uma célula maior, mas não pode combinar linhas. Leia mais na [documentação](#)⁵

Embora `TableLayouts` possam ser usados para projetar interfaces, geralmente não é a melhor opção já que são derivadas de `LinearLayouts`. Se você tem dados que já estão em formato de tabela, como planilhas, então pode ser uma boa opção.

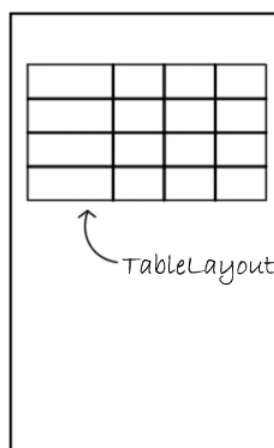


Figura 5.6: Exemplo de `TableLayout`

⁵<http://developer.android.com/reference/android/widget/TableLayout.html>

5.4 Listas (ListView)

⁶ Listas são uma das formas mais simples e poderosas de se mostrar informações ao usuário de forma objetiva. A `ListView` é capaz de apresentar uma lista rolável de itens.



Figura 5.7: Esquema de uma lista

Um item individual da lista pode ser selecionado, essa seleção pode acionar uma outra tela com detalhes do item.

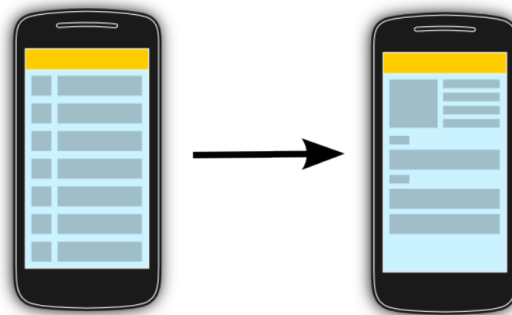


Figura 5.8: Detalhes de um elemento da lista

5.4.1 Adaptadores

Adaptadores são usados para providenciar dados a *views*. O adaptador também define como item da *view* será mostrada. Para `ListsViews` o adaptador define como cada linha será mostrada.

Um adaptador deve estender a classe base `BaseAdapter`. O Android já tem alguns adaptadores padrão, os mais importantes são o `ArrayAdapter` e o `CursorAdapter`.

O `ArrayAdapter` é usado para manipular dados em *arrays* ou listas (`java.util.List`). Já o `SimpleCursorAdapter` consegue manipular dados em banco de dados.

⁶Documentação `ListView`: <http://developer.android.com/reference/android/widget/ListView.html>

5.4.2 Construção

A construção desse tipo de design é simples. No arquivo de *layout* da *activity* use o `LinearLayout` para conter a `ListView`.

```
1 <LinearLayout
2   xmlns:android="http://schemas.android.com/apk/res/android"
3     android:layout_width="fill_parent"
4     android:layout_height="fill_parent"
5     android:orientation="vertical" >
6 </LinearLayout>
```

Algoritmo 5.2: `LinearLayout` no arquivo de *layout*

Se estiver usando o construtor de interface gráfica, pode arrastar uma `ListView` para dentro do *layout*. Caso contrário pode construir manualmente no arquivo XML do *layout* da *activity*.

Você deve colocar o `LinearLayout` como raiz do arquivo XML, o elemento raiz sempre deve conter o atributo `xmlns:android` como mostrado na linha 2 do algoritmo 5.2, não entraremos em detalhes sobre os outros atributos.

Adicione uma `ListView`, escreva o código abaixo dentro do `LinearLayout`.

```
1 <ListView
2   android:id="@+id/listView1"
3   android:layout_width="match_parent"
4   android:layout_height="wrap_content" >
5 </ListView>
```

Algoritmo 5.3: Código de uma `ListView`

Você precisa popular a lista, para isso você pode criar um `string-array` no arquivo `strings.xml` com os elementos que deseja colocar na lista. Nesse exemplo do algoritmo 5.4 foi criada uma lista com nome `listString` e 4 itens que serão mostrados em forma de lista pela `ListView`.

```
1 <string-array name="listString">
2   <item>Menu 1</item>
3   <item>Menu 2</item>
4   <item>Menu 3</item>
5   <item>Menu 4</item>
6 </string-array>
```

Algoritmo 5.4: `string-array` populada com elementos

Finalmente, você deve escrever o código que irá preencher a lista com as *strings*. Como é feito no algoritmo 5.5 abaixo.

```
1 public class MainActivity extends Activity {
2     private ListView lv;
3
4     @Override
5     protected void onCreate(Bundle savedInstanceState) {
6         super.onCreate(savedInstanceState);
7         setContentView(R.layout.activity_main);
8
9         //Obtem o array de strings para popular a lista
10        String listStr[] = getResources().getStringArray(R.array.listString);
11
12        //Obtem a lista
13        ListView lv = (ListView) findViewById(R.id.listView1);
14
15        //Adaptador das strings para a lista
16        lv.setAdapter(new ArrayAdapter<String>
17            (this, android.R.layout.simple_list_item_1, listStr));
18
19        /* Acao para quando clica num elemento da lista
20         * precisa criar um listener e programa-lo para
21         * realizar uma acao. */
22        lv.setOnItemClickListener(new OnItemClickListener() {
23
24            @Override
25            public void onItemClick(AdapterView<?> parent,
26                View view, int position, long id) {
27                //Quando clicado, mostra um Toast
28                Toast.makeText(getApplicationContext(),
29                    ((TextView) view).getText(), Toast.LENGTH_SHORT).show();
30            }
31        });
32    }
33    ...
```

Algoritmo 5.5: Código de uma *activity* com lista clicável

Primeiro, na linha 2, foi criada uma variável do tipo `ListView` para guardar um ponteiro para a *view* já definida no *layout*.

No método `onCreate()` você precisa criar e inicializar a lista na sua *activity*. Na linha 10 obtemos as *strings* do *string-array* e o guardamos na variável `listStr`. Usamos o método `getResources()` para poder adquirir o ponteiro para os recursos do aplicativo. Na linha 13 conseguimos o ponteiro pra lista e o guardamos na variável criada.

Usamos o adaptador ao chamar o método `ListView.setAdapter()` nas linhas 16-17 e passamos como parâmetro a criação de um novo adaptador do tipo `ArrayAdapter`. Para o construtor⁷ desse adaptador está sendo passado o contexto atual da *activity*, um *layout* pré-definido do sistema, o `simple_list_item_1`, e os dados na forma de *array*.

Na linha 22 usamos o método `ListView.setOnItemClickListener` para configurar uma ação a ser executada quando um item da lista for clicado. Neste exemplo é criado um

⁷<http://developer.android.com/reference/android/widget/ArrayAdapter.html>

Toast, o Toast mostra uma mensagem em uma caixa de texto na parte inferior da tela por um curto período de tempo, nesse caso irá mostrar o mesmo texto do item da lista que foi clicado. Uma das aplicações mais comuns é fazer com que ao se clicar em um item da lista, uma nova *activity* seja aberta com detalhes do item.

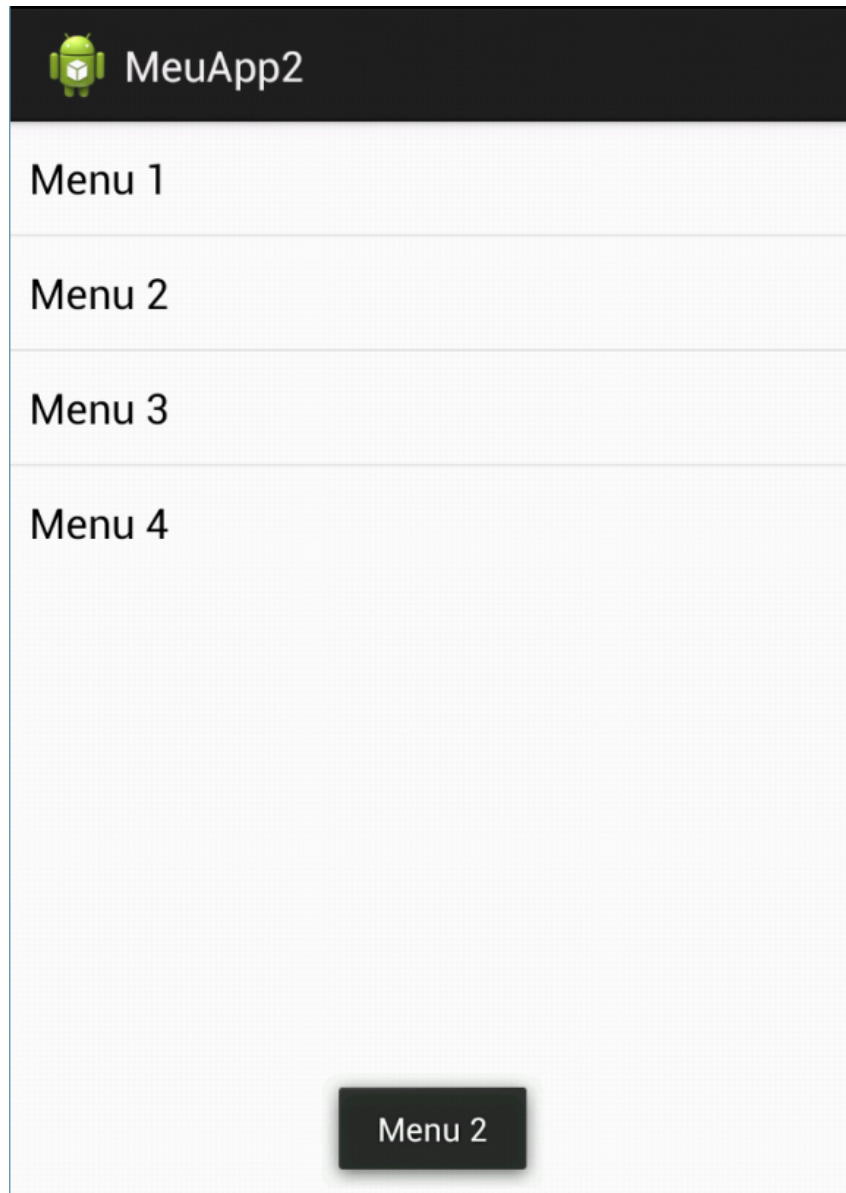


Figura 5.9: Lista simples

A figura 5.9 mostra como ficou o exemplo ao ser executado em um *smartphone*, o item "Menu 2" foi clicado e um Toast foi mostrado no momento do clique.

5.5 Listas Compostas

É possível compor um item da lista colocando mais elementos além de um texto. Para isso você precisa criar um novo arquivo XML que irá definir a customização de cada linha da *ListView*, nesse exemplo iremos definir um arquivo chamado `item.xml`, mostrado abaixo.

```
1 <LinearLayout
2   xmlns:android="http://schemas.android.com/apk/res/android"
3   android:layout_width="wrap_content"
4   android:layout_height="wrap_content"
5   android:orientation="horizontal" >
6
7   <ImageView
8     android:id="@+id/userIcon"
9     android:layout_width="wrap_content"
10    android:layout_height="wrap_content"
11    android:layout_margin="8dp" >
12  </ImageView>
13
14  <LinearLayout
15    android:layout_width="fill_parent"
16    android:layout_height="wrap_content"
17    android:layout_marginBottom="5dp"
18    android:layout_marginTop="5dp"
19    android:orientation="vertical"
20    android:paddingLeft="0px"
21    android:paddingRight="5dp" >
22
23    <TextView
24      android:id="@+id/username"
25      android:layout_width="wrap_content"
26      android:layout_height="wrap_content"
27      android:layout_alignParentLeft="true"
28      android:textColor="#FFF38585"
29      android:textSize="15sp" >
30    </TextView>
31
32    <TextView
33      android:id="@+id/usertext"
34      android:layout_width="wrap_content"
35      android:layout_height="wrap_content"
36      android:layout_marginTop="4dp"
37      android:textColor="#FF444444"
38      android:textSize="13sp" >
39    </TextView>
40
41  </LinearLayout>
42 </LinearLayout>
```

Algoritmo 5.6: Código do arquivo `item.xml`

O algoritmo 5.6 mostra como você pode fazer a customização de um item da lista. Nesse exemplo há uma pequena imagem à esquerda e dois textos de cores e tamanhos diferentes. Para isso primeiro criamos um `LinearLayout` que irá conter uma `ImageView` para mostrar a imagem e outro `LinearLayout` para colocar os dois textos. As cores do textos são configuradas com o atributo `textColor` e usa o padrão HTML de cores.

Agora você precisa usar um adaptador para mostrar esse layout customizado em cada linha da lista, usaremos a classe `SimpleAdapter`⁸. Essa classe faz a adaptação de um `ArrayList` de `Maps` para um *layout* definido.

```
1 public class MainActivity extends Activity {
2     private ListView lv;
3
4     @Override
5     protected void onCreate(Bundle savedInstanceState) {
6         super.onCreate(savedInstanceState);
7         setContentView(R.layout.activity_main);
8
9         //Obtem a lista
10        ListView lv = (ListView) findViewById(R.id.listView1);
11
12        //Cria uma lista de maps(key->value) dos views de cada item do ListView
13        List<Map> list = new ArrayList<Map>();
14        Map map = new HashMap();
15        map.put("userIcon", R.drawable.miku);
16        map.put("userName", "Hatsune Miku");
17        map.put("userText", "Texto exemplo para o adaptador");
18        list.add(map);
19        map = new HashMap();
20        map.put("userIcon", R.drawable.luka);
21        map.put("userName", "Megurine Luka");
22        map.put("userText", "Texto exemplo para o adaptador");
23        list.add(map);
24
25        //Cria um adaptador pro layout customizado
26        SimpleAdapter adapter = new SimpleAdapter(this,
27            (List<? extends Map<String, ?>>) list, R.layout.item,
28            new String[] {"userIcon", "userName", "userText"},
29            new int[] {R.id.userIcon, R.id.username, R.id.usertext});
30
31        lv.setAdapter(adapter);
32    }
```

Algoritmo 5.7: Código da lista customizada

Observando o algoritmo 5.7. Na linha 13 criamos um `ArrayList` de `Maps`. Na linha 14 e 19 criamos um `HashMap` onde a chave é uma *string* que identifica o conteúdo, essas chaves serão *userIcon*, *userName* e *userText* respectivamente. Em *userIcon* colocamos uma imagem, essa imagem deve ser colocada nas subpastas da pasta *drawable* e é acessada através da classe *R*. Em *userName* colocamos um nome de usuário, por exemplo. Em *userText* poderia ser colocada uma descrição, ou uma frase customizada do usuário mas nesse exemplo foi colocado uma sentença qualquer. Nas linhas 18 e 23 adicionamos o `Map` criado no `ArrayList`.

Criamos o `SimpleAdapter` nas linhas 26-29. Para o construtor passamos o `ArrayList` de `Maps` que contém os dados, passamos também o *layout* que definimos anteriormente

⁸<http://developer.android.com/reference/android/widget/SimpleAdapter.html>

`R.layout.item`. Passamos um *array* de *strings* que contém as chaves que serão usadas para obter os dados e por último um *array* de inteiros que contém os *ids* das *views* em que os conteúdos dos Maps serão colocados.

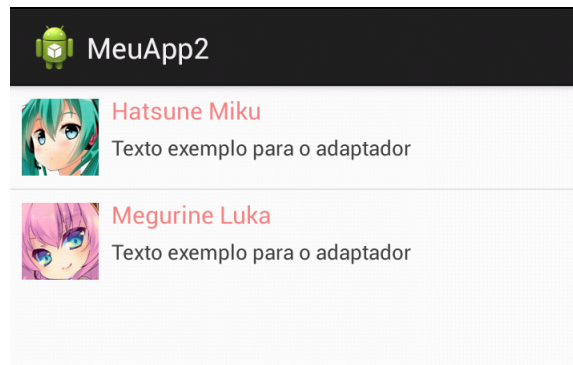


Figura 5.10: Lista Composta

A figura 5.10 mostra como ficou o exemplo acima ao ser executado em um *smartphone*.

5.6 Listas expansíveis (`ExpandableListView`)

Listas expansíveis são úteis para agrupar conjuntos de itens semelhantes, funcionam da mesma maneira que as listas comuns e podem ser customizadas. Comece colocando sua lista no *layout* da *activity* desejada.

```

1 <LinearLayout
2   xmlns:android="http://schemas.android.com/apk/res/android"
3   android:layout_width="fill_parent"
4   android:layout_height="fill_parent"
5   android:orientation="vertical" >
6
7     <ExpandableListView
8       android:id="@+id/expandableList"
9       android:layout_width="match_parent"
10      android:layout_height="wrap_content"
11      android:transcriptMode="alwaysScroll"
12      android:listSelector="@android:color/holo_green_light">
13     </ExpandableListView>
14
15 </LinearLayout>

```

Algoritmo 5.8: Código XML de uma Lista expansível

Dica: O atributo `transcriptMode="alwaysScroll"` vai fazer com que a lista sempre role até o final quando você expande ou contrai um grupo. O atributo `listSelector` colore o item da lista quando este é clicado.

Agora crie 2 novos arquivos XML, um chamado `list_item_parent.xml` e o outro chamado `list_item_child.xml` dentro da pasta `res/layout`.

```
1 <LinearLayout
2   xmlns:android="http://schemas.android.com/apk/res/android"
3   android:id="@+id/list_item"
4   android:orientation="horizontal"
5   android:layout_width="fill_parent"
6   android:layout_height="fill_parent">
7
8   <TextView
9     android:id="@+id/list_item_text_view"
10    android:layout_width="0dp"
11    android:layout_height="wrap_content"
12    android:textSize="20sp"
13    android:padding="10dp"
14    android:layout_weight="1"
15    android:layout_marginLeft="35dp" />
16
17 </LinearLayout>
```

Algoritmo 5.9: Layout `list_item_parent.xml`

Nesses dois *layouts* teremos apenas uma `TextView` para abrigar um texto.

```
1 <LinearLayout
2   xmlns:android="http://schemas.android.com/apk/res/android"
3   android:id="@+id/list_item_child"
4   android:orientation="vertical"
5   android:layout_width="fill_parent"
6   android:layout_height="fill_parent"
7   android:gravity="center_vertical">
8
9   <TextView
10     android:layout_width="wrap_content"
11     android:layout_height="wrap_content"
12     android:id="@+id/list_item_text_child"
13     android:textSize="20sp"
14     android:padding="10dp"
15     android:layout_marginLeft="5dp" />
16
17 </LinearLayout>
```

Algoritmo 5.10: Layout `list_item_child.xml`

Em seguida precisamos criar uma classe que irá abrigar os dados dos elementos pai, elementos estes que serão expandidos quando clicados. Nesse exemplo criamos uma classe *Parent*, como mostrado no algoritmo 5.11

```
1 public class Parent {
2     private String mTitle;
3     private ArrayList<String> mArrayChildren;
4
5     public String getTitle() {
6         return mTitle;
7     }
8
9     public void setTitle(String mTitle) {
10        this.mTitle = mTitle;
11    }
12
13    public ArrayList<String> getArrayChildren() {
14        return mArrayChildren;
15    }
16
17    public void setArrayChildren(ArrayList<String> mArrayChildren) {
18        this.mArrayChildren = mArrayChildren;
19    }
20 }
```

Algoritmo 5.11: Classe Parent

Essa classe contém o texto do item, que será guardado na *string* `mTitle` e um `ArrayList` que irá comportar os sub-itens desse item. Os métodos *get* e *set* são simples.

Em seguida, crie uma nova classe, `CustomAdapter` que será o adaptador da lista expansível para os dados, para esse exemplo estaremos adaptando apenas para o uso de texto. Essa classe deve estender a classe `BaseExpandableListAdapter`.

```
1 public class CustomAdapter extends BaseExpandableListAdapter {
2     private LayoutInflater inflater;
3     private ArrayList<Parent> parent;
4
5     public CustomAdapter(Context context, ArrayList<Parent> parent) {
6         this.parent = parent;
7         inflater = LayoutInflater.from(context);
8     }
9
10    @Override
11    //Obtem o nome de cada item
12    public Object getChild(int groupPosition, int childPosition) {
13        return parent.get(groupPosition).getArrayChildren().
14            get(childPosition);
15    }
16
17    @Override
18    public long getChildId(int groupPosition, int childPosition) {
19        return childPosition;
20    }
```

```
21
22 @Override
23 //Nesse metodo voce seta os textos para ver os filhos na lista
24 public View getChildView(int groupPosition, int childPosition,
25     boolean isLastChild, View view, ViewGroup viewGroup) {
26
27     if(view == null) {
28         view = inflater.inflate(R.layout.list_item_child, viewGroup,
29             false);
30     }
31
32     TextView textView = (TextView)
33         view.findViewById(R.id.list_item_text_child);
34
35     textView.setText(parent.get(groupPosition).getArrayChildren().
36         get(childPosition));
37
38     return view;
39 }
40
41 @Override
42 public int getChildrenCount(int groupPosition) {
43     //retorna o tamanho do array de filhos
44     return parent.get(groupPosition).getArrayChildren().size();
45 }
46
47 @Override
48 //Obtem o titulo de cada pai
49 public Object getGroup(int groupPosition) {
50     return parent.get(groupPosition).getTitle();
51 }
52
53 @Override
54 public int getGroupCount() {
55     return parent.size();
56 }
57
58 @Override
59 public long getGroupId(int groupPosition) {
60     return groupPosition;
61 }
62
63 @Override
64 //Nesse metodo voce seta o texto para ver os pais na lista
65 public View getGroupView(int groupPosition, boolean isExpanded,
66     View view, ViewGroup viewGroup) {
67
68     if(view == null) {
69         //Carrega o layout do parent na view
70         view = inflater.inflate(R.layout.list_item_parent, viewGroup,
71             false);
```

```

72     }
73
74     //Obtem o textView
75     TextView textView = (TextView)
76         view.findViewById(R.id.list_item_text_view);
77
78     textView.setText (getGroup (groupPosition) .toString());
79
80     return view;
81 }
82
83 @Override
84 public boolean hasStableIds() {
85     return true;
86 }
87
88 @Override
89 public boolean isChildSelectable(int groupPosition,
90     int childPosition) {
91     return true;
92 }
93 }

```

Algoritmo 5.12: Classe CustomAdapter

Primeiro precisamos de um `LayoutInflater`⁹ que irá instanciar o *layout* XML nas *views* correspondentes, e um *array* da classe `Parents` que criamos anteriormente, esses serão os itens principais da lista.

Na linha 7 no construtor da classe, usamos o método `LayoutInflater.from()` para obter o *inflator* do contexto da *activity*.

Ao estender a classe `BaseExpandableListAdapter` temos que programar alguns métodos. O método `getChild()` deve adquirir o ponteiro para um subitem de um item na lista. O método `getChildId()` deve obter o *id* de um subitem, porém nesse exemplo não temos nada configurado então usamos a própria posição desse subitem como *id* e retornamos `childPosition`.

O método `getChildView` na linha 24 vai atribuir o *layout* dos subitens na linha 28. Na linha 32 obtemos o `TextView` desse subitem e com o método `TextView.setText()` atribuímos seu respectivo texto. Esse texto está guardado no *array* chamado `mArrayChildren` da classe `Parent`, então a fim de obter esse texto devemos obter o `Parent` correto. Quando você clica em um item da lista, o Android guarda qual item você clicou no parâmetro `groupPosition`. Em seguida se obtém o texto de cada subitem pelo parâmetro `childPosition`.

Outro método importante é o `getGroupView`, funciona da mesma maneira que `getChildView` mas configurando os *views* dos itens pai em vez dos subitens.

Para finalizar, você deve construir os objetos na classe da *activity*, nesse exemplo para popular a lista eu coloquei no arquivo de `strings` alguns fabricantes e modelos de carros, você pode obtê-los no repositório do projeto.

⁹<http://developer.android.com/reference/android/view/LayoutInflater.html>

```
1 public class MainActivity extends Activity {
2     private ExpandableListView mExpandableList;
3
4     @Override
5     protected void onCreate(Bundle savedInstanceState) {
6         super.onCreate(savedInstanceState);
7         setContentView(R.layout.activity_main);
8
9         mExpandableList = (ExpandableListView)
10             findViewById(R.id.listaExpandivel);
11
12         ArrayList<Parent> arrayParents = new ArrayList<Parent>();
13         ArrayList<String> arrayChildren;
14
15         //Array de fabricantes no arquivo de strings
16         String parentsNames[] = getResources().
17             getStringArray(R.array.Fabricantes);
18
19         for(int i = 0; i < parentsNames.length; i++){
20             /*Para cada pai "i" criar um novo objeto
21             Parent para setar o nome e os filhos */
22             Parent parent = new Parent();
23             parent.setTitle(parentsNames[i]);
24
25             arrayChildren = new ArrayList<String>();
26             /* Obtem os carros daquele fabricante
27             * primeiro obtendo o resource id (passando o nome do fabricante)
28             * depois usando esse resource id para obter o array de strings
29             */
30             int resId = getResources().
31                 getIdentifier(parentsNames[i], "array", getPackageName());
32             String childrenNames[] = getResources().getStringArray(resId);
33
34             for(int j = 0; j < childrenNames.length; j++){
35                 arrayChildren.add(childrenNames[j]);
36             }
37
38             parent.setmArrayChildren(arrayChildren);
39             arrayParents.add(parent);
40         }
41
42         mExpandableList.setAdapter(
43             new CustomAdapter(MainActivity.this, arrayParents));
44     }
45     ...
46 }
```

Algoritmo 5.13: Construindo a lista expansível na *activity*

O algoritmo 5.13 é a construção da lista expansível na *activity*, na linha 9-10 obtemos a *view* usando `findViewById()`. A linha 30-31 são um pouco mais complicadas, primeiro é preciso obter o *id* do *string-array* que é subitem do item atual no laço de repetição. Para isso usamos `getResources().getIdentifier()` para obter o *id* do subitem a partir do nome do item pai. Em seguida podemos acessar o *string-array* normalmente como é feito na linha 31.

Na linha 41 usamos o `CustomAdapter` que criamos anteriormente.

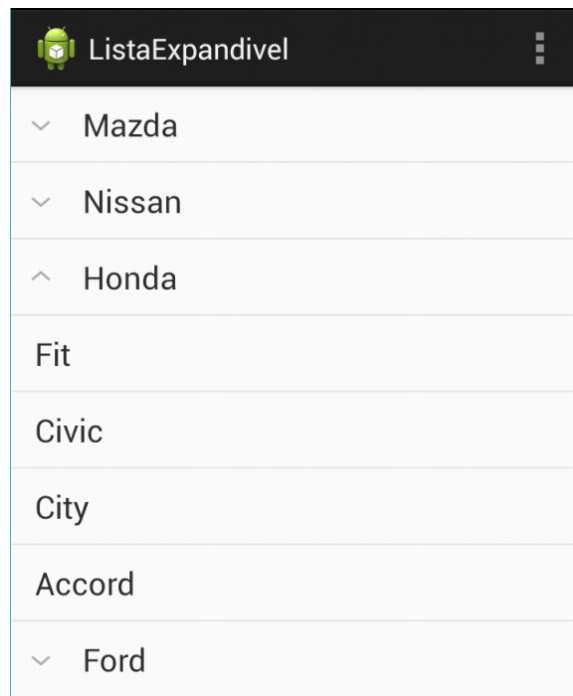


Figura 5.11: Exemplo de lista expansível rodando em um *smartphone*

5.7 Grades (GridView) e imagens ImageView

Grades são úteis para mostrar imagens e fotos como uma galeria, ou permitir a seleção de categorias semelhante a uma lista. A idéia é ter elementos lado a lado para mostrar ou para selecionar e mostrar mais detalhes. Basicamente funciona como uma grade bi-dimensional que pode ser arrastada para os lados ou de cima pra baixo.

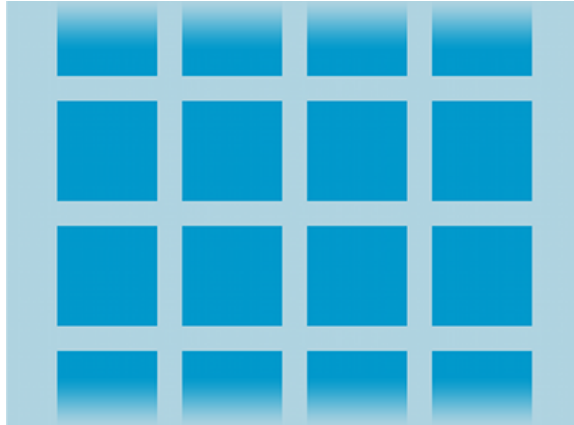


Figura 5.12: Esquema de um GridView

Comece colocando um GridView¹⁰ no *layout* de sua *activity*.

```
1 <LinearLayout
2   xmlns:android="http://schemas.android.com/apk/res/android"
3   android:layout_width="fill_parent"
4   android:layout_height="fill_parent"
5   android:orientation="vertical" >
6
7   <GridView
8       android:id="@+id/gridview"
9       android:layout_width="fill_parent"
10      android:layout_height="fill_parent"
11      android:numColumns="auto_fit"
12      android:columnWidth="90dp"
13      android:horizontalSpacing="10dp"
14      android:verticalSpacing="10dp"
15      android:gravity="center"
16      android:stretchMode="columnWidth" >
17 </GridView>
18
19 </LinearLayout>
```

Algoritmo 5.14: Layout do GridView

Vamos criar uma nova classe que será o adaptador de imagens para o GridView, chame-mos a classe de ImageAdapter, ela é mostrada no algoritmo 5.15.

¹⁰<http://developer.android.com/reference/android/widget/GridView.html>

```
1 public class ImageAdapter extends BaseAdapter {
2     private Context mContext;
3
4     //Mantendo todos os ids num array
5     public Integer[] thumbIds = {
6         R.drawable.sample_0, R.drawable.sample_1,
7         R.drawable.sample_2, R.drawable.sample_3,
8         R.drawable.sample_4, R.drawable.sample_5,
9         R.drawable.sample_6, R.drawable.sample_7
10    };
11
12    //Construtor
13    public ImageAdapter(Context c) {
14        mContext = c;
15    }
16
17    @Override
18    //Retorna o tamanho do array
19    public int getCount() {
20        return thumbIds.length;
21    }
22
23    @Override
24    //Retorna um elemento do array
25    public Object getItem(int position) {
26        return thumbIds[position];
27    }
28
29    @Override
30    //Nao sera usado
31    public long getItemId(int position) {
32        return 0;
33    }
34
35    @Override
36    public View getView(int position, View convertView,
37        ViewGroup parent) {
38        ImageView imageView = new ImageView(mContext);
39        imageView.setImageResource(thumbIds[position]);
40        imageView.setLayoutParams(new GridView.LayoutParams(200, 200));
41        imageView.setScaleType(ImageView.ScaleType.CENTER_CROP);
42        return imageView;
43    }
44 }
```

Algoritmo 5.15: Classe ImageAdapter

A classe ImageAdapter deve ser subclasse da classe BaseAdapter¹¹. Temos, como

¹¹<http://developer.android.com/reference/android/widget/BaseAdapter.html>

variável pública, um *array* das imagens que queremos colocar na grade. Como todo *id* de um recurso da classe *R* é um inteiro, criamos um *array* de inteiros. Note que estamos considerando que todas as imagens já foram devidamente colocadas na pasta *drawable*.

O método mais importante é o método `getView()`. Nele criamos uma nova *ImageView*¹² para abrigar a imagem que queremos colocar na grade. Em seguida configuramos alguns parâmetros desse *ImageView*, o método `ImageView.setImageResource()` é responsável por estabelecer um *drawable* como conteúdo do *ImageView*. Já o método `View.setLayoutParams()` configura os parâmetros de *layout* associados com essa *view*, note que para esse método passamos parâmetros de *layout* de uma *GridView*, que por sua vez recebe (200, 200) como largura e altura de um elemento da grade.

`ImageView.setScaleType` controla como a imagem deve ser redimensionada para condizer com o tamanho do *ImageView*, `ImageView.ScaleType`¹³ são as formas disponíveis para escalar a imagem.

Agora basta criar a grade em sua *activity*.

```
1 public class MainActivity extends Activity {
2
3     @Override
4     public void onCreate(Bundle savedInstanceState) {
5         super.onCreate(savedInstanceState);
6         setContentView(R.layout.activity_main);
7
8         GridView gridView = (GridView) findViewById(R.id.gridview);
9
10        // Instance of ImageAdapter Class
11        gridView.setAdapter(new ImageAdapter(this));
12    }
13    ...
14 }
```

Algoritmo 5.16: *activity* com grade

Exemplo acima rodando em um *smartphone* na figura 5.13:

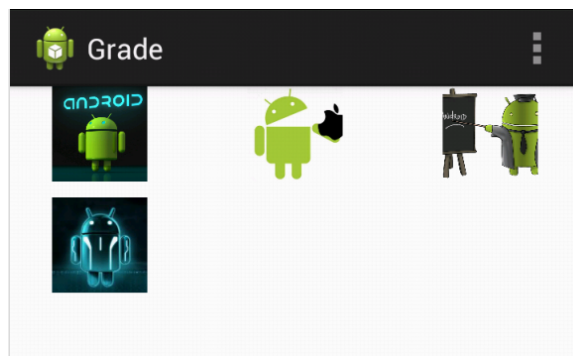


Figura 5.13: Demonstração de um *GridView*

¹²<http://developer.android.com/reference/android/widget/ImageView.html>

¹³<http://developer.android.com/reference/android/widget/ImageView.ScaleType.html>

Para complementar, você pode fazer com que a imagem abra em tela cheia quando clicada na view, para isso é necessário que você passe o *id* do recurso do GridView para uma nova *activity* que irá mostrar a imagem em tela cheia. Para isso precisamos criar um novo *layout* XML, a qual chamaremos de `full_image.xml`, nele teremos apenas uma *ImageView* e um *TextView* que será uma pequena legenda da imagem.

```
1 <RelativeLayout
2   xmlns:android="http://schemas.android.com/apk/res/android"
3   android:id="@+id/relativeLayout"
4   android:layout_width="fill_parent"
5   android:layout_height="fill_parent" >
6
7   <ImageView
8     android:id="@+id/full_image_view"
9     android:layout_width="fill_parent"
10    android:layout_height="fill_parent" />
11
12   <TextView
13     android:id="@+id/myImageViewText"
14     android:layout_width="fill_parent"
15     android:layout_height="40dp"
16     android:gravity="center"
17     android:background="#55555555"
18     android:textSize="16sp"
19     android:textColor="#FFFFFF" />
20
21 </RelativeLayout>
```

Algoritmo 5.17: Layout `full_image.xml`

Note que para o *TextView* usaremos o atributo `layout_width` como `fill_parent` e `layout_height` como `40dp`, dessa forma criamos um pequeno retângulo de altura fixa mas de forma que a largura preencha a tela completamente. O atributo `background` com o valor `#55555555` faz com que a cor do retângulo seja cinza com transparência, já que o parâmetro *alfa* também tem valor `0x55`. Também deixamos o texto com cor branca com o atributo `textColor`.

Em seguida, crie uma nova classe chamada `FullImageActivity`, essa é a *activity* que vai mostrar a imagem em tela cheia. A construção da classe é simples, você deve apenas obter o *id* da imagem passado como *extra* através do *intent* e então obter essa imagem da classe `ImageAdapter`.

```

1 public class FullImageActivity extends Activity {
2     public void onCreate(Bundle savedInstanceState) {
3         super.onCreate(savedInstanceState);
4         setContentView(R.layout.full_image);
5
6         //Obtem os dados do intent
7         Intent intent = getIntent();
8
9         //Seleciona o id da imagem
10        int id = intent.getExtras().getInt("id");
11        ImageAdapter imageAdapter = new ImageAdapter(this);
12
13        //Configura o ImageView para mostrar a imagem correspondente
14        ImageView imageView = (ImageView) findViewById(R.id.full_image_view);
15        imageView.setImageResource(imageAdapter.thumbIds[id]);
16
17        //Configura o TextView para mostrar uma descricao da imagem
18        TextView textView = (TextView) findViewById(R.id.myImageViewText);
19        textView.setText("Image id: " + id);
20    }
21 }

```

Algoritmo 5.18: Classe FullImageActivity

```

1 @Override
2 public void onCreate(Bundle savedInstanceState) {
3     super.onCreate(savedInstanceState);
4     setContentView(R.layout.activity_main);
5     GridView gridView = (GridView) findViewById(R.id.gridview);
6     gridView.setAdapter(new ImageAdapter(this));
7
8     //Cria um listener para o evento de clique em um elemento da grade
9     gridView.setOnItemClickListener(new OnItemClickListener() {
10
11         @Override
12         public void onItemClick(AdapterView<?> parent, View view,
13             int pos, long id) {
14             //Envia o id da imagem para o FullImageActivity
15             Intent intent = new Intent(getApplicationContext(),
16                 FullImageActivity.class);
17             intent.putExtra("id", pos);
18             startActivity(intent);
19         }
20     });
21 }

```

Algoritmo 5.19: Código da *activity* após as modificações

No algoritmo 5.19, configuramos um `View.setOnItemClickListener()` de forma que quando uma imagem da grade for clicada, um `Intent` seja enviado a uma nova *activity* que por sua vez ficará encarregada de mostrar a imagem em tela cheia. Quando um item é clicado conseguimos obter a posição dele na grade com o parâmetro `pos` da função `onItemClick()`, essa posição é equivalente ao *id* da imagem no *array* criado na classe `ImageAdapter`. A `FullImageActivity` por sua vez recebe esse `Intent` que possui o *id* da imagem que deve ser mostrada e configura o `ImageView` de acordo.

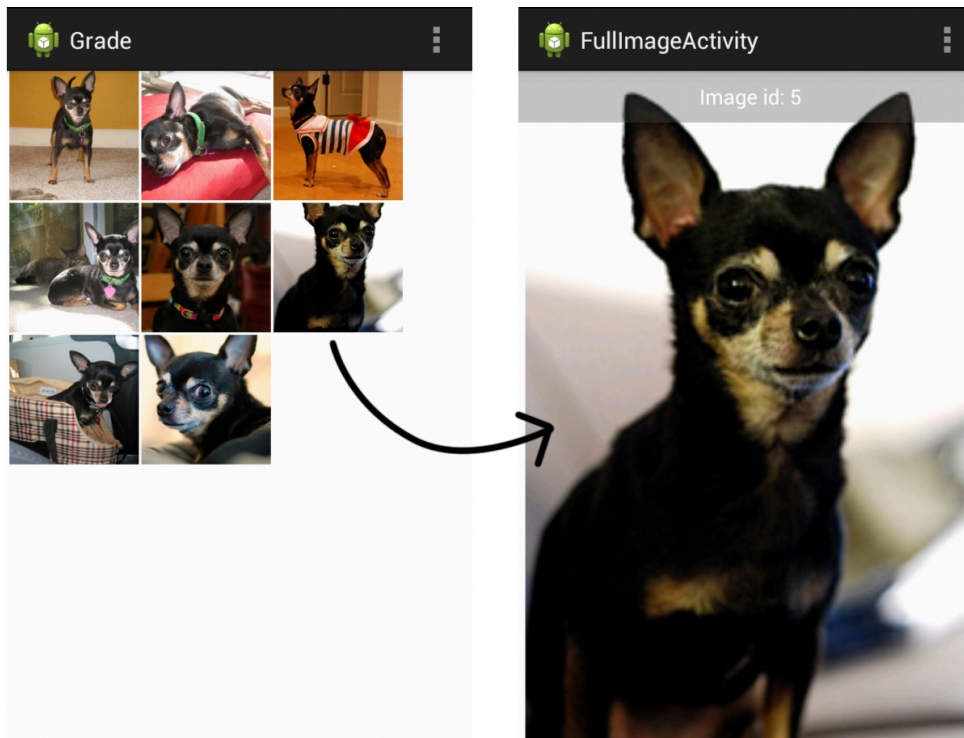


Figura 5.14: Exemplo `GridView` com imagem em tela cheia

5.8 Fragmentos

Fragmentos são a solução do Android para criar interfaces de usuário modulares, eles vivem dentro das *activity* e uma *activity* pode conter vários fragmentos. Assim como as *activity* os fragmentos possuem um ciclo de vida.

Dentre as vantagens de um fragmento estão:

- Modularidade e reuso de código
- Habilidade de construir interfaces com múltiplos painéis
- Facilidade de construir aplicativos para celulares e tablets

O primeiro conceito a ser coberto é como construir um fragmento, comece definindo o *layout* do fragmento.

Um *layout* bem simples, apenas com um botão para efeito de demonstração. Agora crie uma classe `BasicFragment`

```
1 public class BasicFragment extends Fragment {
2
3     @Override
4     public View onCreateView(LayoutInflater inflater,
5         ViewGroup container, Bundle savedInstanceState){
6
7         //Obtem o layout do fragmento em uma view
8         View view = inflater.inflate(R.layout.fragment, container, false);
9
10        //Obtem o botao da view
11        Button button = (Button) view.findViewById(R.id.fragment_button);
12
13        //Um listener simples para o botao
14        button.setOnClickListener(new OnClickListener() {
15
16            @Override
17            public void onClick(View v) {
18                Activity activity = getActivity();
19
20                if(activity != null){
21                    Toast.makeText(activity,
22                        "A toast to a fragment", Toast.LENGTH_SHORT).show();
23                }
24            }
25        });
26        return view;
27    }
28 }
```

Algoritmo 5.20: Classe BasicFragment

Caso você esteja desenvolvendo para API menores que 11 (HoneyComb 3.0) você vai precisar usar a API de retrocompatibilidade que o Google providenciou para essas APIs, você precisa importar a classe de suporte:

```
import android.support.v4.app.Fragment;
```

Agora para incluir o fragmento na *activity* existem duas opções. A primeira é incluir o fragmento no XML da *activity* como você faria com qualquer view.

```

1 <LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
2     android:layout_width="fill_parent"
3     android:layout_height="fill_parent"
4     android:orientation="vertical" >
5
6     <fragment
7         android:id="@+id/fragment_content"
8         android:name="com.example.fragmento.BasicFragment"
9         android:layout_width="fill_parent"
10        android:layout_height="fill_parent" >
11 </fragment>
12 </LinearLayout>

```

Algoritmo 5.21: Layout da *activity* com um fragmento

Você pode usar o `<fragment>` quantas vezes quiser para incluir múltiplos fragmentos. Note que você precisa usar um nome qualificado em `android:name`, veja mais na documentação oficial: [activity-element](#)¹⁴

Novamente, caso esteja desenvolvendo para APIs menores que 11, você vai precisar fazer a *activity* estender a classe `FragmentActivity` e importar a classe de suporte:

```

import android.support.v4.app.FragmentActivity;

public class MainActivity extends FragmentActivity

```

Simplesmente configurando a *activity* para usar o fragmento vai fazer com que o fragmento seja adicionado e renderizado na tela, entretanto você deve querer ter mais controle de quando e como seus fragmentos serão adicionados durante o curso do seu aplicativo. Para isso existe uma maneira alternativa de adicionar o fragmento em tempo de execução. A fim de adicionar o fragmento em tempo de execução você precisa fazer uma mudança no layout da *activity*:

```

1 <LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
2     android:layout_width="fill_parent"
3     android:layout_height="fill_parent"
4     android:orientation="vertical" >
5
6     <FrameLayout
7         android:id="@+id/fragment_content"
8         android:layout_width="fill_parent"
9         android:layout_height="fill_parent" />
10
11 </LinearLayout>

```

Algoritmo 5.22: Layout da *activity* com o `FrameLayout`

E uma mudança na *activity* que vai mostrar o fragmento:

¹⁴<http://developer.android.com/guide/topics/manifest/activity-element.html#nm>

```
1 public class MainActivity extends FragmentActivity {
2
3     @Override
4     protected void onCreate(Bundle savedInstanceState) {
5         super.onCreate(savedInstanceState);
6         setContentView(R.layout.activity_main);
7
8         //Como estamos usando o pacote de suporte
9         //Precisamos usar o Manager desse pacote
10        FragmentManager fm = getSupportFragmentManager();
11        //Voce pode obter um fragmento da mesma forma que obtem
12        //qualquer outra view usando o FragmentManager
13        Fragment fragment = fm.findFragmentById(R.id.fragment_content);
14
15        if(fragment == null){
16            //Comeca uma transacao de fragmentos
17            FragmentTransaction ft = fm.beginTransaction();
18            //Adiciona o fragmento
19            ft.add(R.id.fragment_content, new BasicFragment());
20            //"Committa" a transacao
21            ft.commit();
22        }
23    }
24    ...
```

Algoritmo 5.23: *activity* com adição dinâmica de fragmento

E dessa forma obtemos o mesmo resultado, porém com a adição dinâmica do fragmento, você pode experimentar e fazer com que o botão remova um fragmento e coloca outro diferente no lugar.

5.9 Abas (*Tabs*)

Existem diversas maneiras de criar uma interface com abas no Android, uma delas é usando as interfaces `TabHost` e `TabWidget`, outra é imitando o comportamento usando apenas `Fragments`.

5.9.1 Usando `TabHost` e `TabWidget`

Abas usando essas interfaces são suportadas por todas as versões do Android. Vamos criar uma interface com abas seguindo esse esquema:

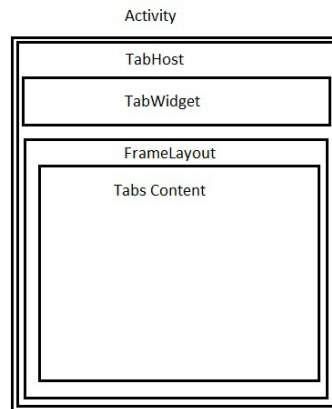


Figura 5.15: Esquema da interface com abas

Primeiro precisamos criar uma *activity* que servirá como recipiente para as abas e seu conteúdo. O `TabWidget`¹⁵ é o controle de seleção das abas. Todo conteúdo das abas ficará contido dentro do `FrameLayout`, é nele que as respectivas *activities* serão mostradas. O `TabHost`¹⁶ por sua vez serve como um recipiente para o `TabWidget` e o `FrameLayout`.

Crie uma nova *activity*, a chamaremos de `TabLayoutActivity`. No XML que define o *layout* da *activity*, insira o `TabHost`, o `TabWidget` e `FrameLayout` como mostrado no algoritmo abaixo.

```

1 <TabHost
2   xmlns:android="http://schemas.android.com/apk/res/android"
3   android:id="@android:id/tabhost"
4   android:layout_width="fill_parent"
5   android:layout_height="fill_parent">
6
7   <LinearLayout
8     android:orientation="vertical"
9     android:layout_width="fill_parent"
10    android:layout_height="fill_parent" >
11
12     <TabWidget
13       android:id="@android:id/tabs"
14       android:layout_width="fill_parent"
15       android:layout_height="wrap_content" />
16
17     <FrameLayout
18       android:id="@android:id/tabcontent"
19       android:layout_width="fill_parent"
20       android:layout_height="fill_parent" />
21
22   </LinearLayout>
23 </TabHost>
  
```

Algoritmo 5.24: Layout da *activity* `TabHostLayout`

¹⁵<http://developer.android.com/reference/android/widget/TabWidget.html>

¹⁶<http://developer.android.com/reference/android/widget/TabHost.html>

Agora precisamos definir o *layout* dos fragmentos, isto é, o *layout* de cada aba. Para simplificar o exemplo, as abas só terão um fundo colorido, de cores diferentes. Para isso usa-se o atributo `background`.

```

1 <LinearLayout
2   xmlns:android="http://schemas.android.com/apk/res/android"
3   android:layout_width="fill_parent"
4   android:layout_height="fill_parent"
5   android:background="#FF0000" />

```

Algoritmo 5.25: *Layout* do fragmento da aba.

Precisamos definir as classes de cada fragmento de aba. Cada classe deverá estender a classe `Fragment` e inflar seu *layout* correspondente. Depois cada fragmento será instanciado pela nossa *activity* principal, `TabLayoutActivity` usando o *fragment manager*. No algoritmo 5.26 está definido a classe `Tab1Fragment` (as classes `Tab2Fragment` e `Tab3Fragment`) são exatamente iguais, exceto que elas inflam seus respectivos *layouts*).

Dica: Você deve importar a classe `android.support.v4.app.Fragment` para suportar versões mais antigas do Android!

```

1 public class Tab1Fragment extends Fragment {
2   public View onCreateView(LayoutInflater inflater,
3     ViewGroup container, Bundle savedInstanceState){
4
5     if(container == null){
6       return null;
7     }
8
9     return (LinearLayout) inflater.
10       inflate(R.layout.tab_fragment1, container, false);
11   }
12 }

```

Algoritmo 5.26: Classe `Tab1Fragment`

Na classe `TabLayoutActivity`, note que estamos estendendo a classe `FragmentActivity` para poder usufruir das funcionalidades dos fragmentos. É necessário configurar o método `onCreate()`, esse é o ponto de início da nossa *activity*. O primeiro passo é inflar o *layout* com abas definido no algoritmo 5.24. O segundo passo é inicializar as abas, para isso invocamos o método `TabHost.setup()`, adicionar as abas e suas informações em um mapa e determinar a primeira aba como ativa.

Primeiro criaremos uma classe que servirá de suporte para guardar as informações relevantes sobre as nossas abas.

```
1 public class TabInfo {
2     private String tag;
3     private Class klass;
4     private Bundle args;
5     private Fragment fragment;
6
7     TabInfo(String tag, Class klass, Bundle args){
8         this.tag = tag;
9         this.klass = klass;
10        this.args = args;
11    }
12 }
```

Algoritmo 5.27: Classe TabInfo

Em seguida, começaremos a escrever nossa classe TabLayoutActivity.

```
1 public class TabLayoutActivity extends Activity
2     implements TabHost.OnTabChangeListener {
3     private TabHost mTabHost;
4     private HashMap<String, TabInfo> mapTabInfo =
5         new HashMap<String, TabInfo>();
6     private TabInfo mLastTab = null;
7
8     @Override
9     protected void onCreate(Bundle savedInstanceState) {
10        super.onCreate(savedInstanceState);
11        //Estabelece o layout da activity
12        setContentView(R.layout.activity_tab_layout);
13
14        //Metodo para inicializar as abas
15        initialiseTabHost(savedInstanceState);
16        if(savedInstanceState != null){
17            //Determina a aba que esta selecionada
18            mTabHost.setCurrentTabByTag
19                (savedInstanceState.getString("tab"));
20        }
21    }
```

Algoritmo 5.28: Primeira parte da classe TabLayoutActivity

Antes de criar o método `initialiseTabHost()` precisamos criar outra classe suporte, essa classe é necessária para criar o conteúdo de uma aba sob demanda. Crie uma classe que chamaremos de `TabFactory` e ela deve implementar a interface `TabContentFactory`¹⁷.

¹⁷<http://developer.android.com/reference/android/widget/TabHost.TabContentFactory.html>

```

1 public class TabFactory implements TabContentFactory{
2     private final Context mContext;
3
4     public TabFactory(Context context){
5         mContext = context;
6     }
7
8     @Override
9     public View createTabContent(String tag) {
10         View v = new View(mContext);
11         v.setMinimumHeight(0);
12         v.setMinimumWidth(0);
13         return v;
14     }
15 }

```

Algoritmo 5.29: Classe TabFactory

O parâmetro *tag* do método `createTabContent()` é que define qual aba foi selecionada. O método retorna uma *view* para mostrar os elementos da aba selecionada.

Na classe `TabLayoutActivity` precisamos criar o método `initialiseTabHost()`. Siga o algoritmo 5.30 abaixo. Note o uso do método `onTabChanged()`. Precisamos implementar esse método em seguida através da interface `TabHost.OnTabChangeListener`.

```

1 private void initialiseTabHost(Bundle args) {
2     mTabHost = (TabHost) findViewById(android.R.id.tabhost);
3     mTabHost.setup();
4     TabInfo tabInfo = null;
5     String tag;
6
7     //Cria Tab1
8     tabSpec = mTabHost.newTabSpec("Tab1");
9     tabSpec.setIndicator("Tab 1");
10    tabSpec.setContent(new TabFactory(this));
11    tag = tabSpec.getTag();
12    tabInfo = new TabInfo("Tab1", Tab1Fragment.class, args);
13    tabInfo.fragment = getSupportFragmentManager().
14        findFragmentByTag(tag);
15    mTabHost.addTab(tabSpec);
16    mapTabInfo.put(tabInfo.tag, tabInfo);
17    /* Repete para Tab2 e Tab3 */
18
19    //Ajusta primeira aba como default
20    onTabChanged("Tab1");
21    mTabHost.setOnTabChangeListener(this);
22 }

```

Algoritmo 5.30: Método `initialiseTabHost()`

Primeiro obtemos a *view* `TabHost` usando o método `findViewById()`. Observe que estamos pegando um recurso já existente do sistema Android, já que estamos chamando a classe `R` do sistema, e não do nosso aplicativo. Em seguida chamamos o método `setup()`, a documentação diz que é necessário invocar esse método antes de adicionar abas se carregamos o `TabHost` usando `findViewById()`.

Criamos um `TabInfo` e um `TabSpec` para nos auxiliar na adição das abas. Para adicionar as abas, primeiro chamamos o método `TabHost.newTabSpec()` para obtermos um novo `TabSpec` associado a esse `TabHost`, colocamos a *tag* `"Tab1"` nele, como pode ser observado na linha 9. Em seguida determinamos um indicador (que será mostrado ao usuário) a essa aba usando `TabSpec.setIndicator()`, na linha 10. Criamos um novo `TabInfo` e passamos a *tag* criada, a classe com o conteúdo da aba e uma série de argumentos que podem ser passados entre *activities*. Na linha 12 usamos o método `addTab()` criado anteriormente. Na linha 13 adicionamos a *tag* e um ponteiro para o recém-criado `TabInfo` no `HashMap`.

Repetimos o mesmo procedimento para as abas 2 e 3, porém mudando os valores da *tag*, do indicador e da classe. Por último definimos a primeira aba como *default* e determinamos o argumento *this* para o método `setOnTabChangeListener()` pois iremos implementar o método `onTabChanged` em seguida.

```

1 @Override
2 public void onTabChanged(String tag) {
3     TabInfo newTab = mapTabInfo.get(tag);
4
5     if(mLastTab != newTab) {
6         FragmentTransaction ft =
7             getSupportFragmentManager().beginTransaction();
8
9         if(mLastTab != null) {
10             if(mLastTab.fragment != null) {
11                 ft.detach(mLastTab.fragment);
12             }
13         }
14
15         if(newTab != null) {
16             if(newTab.fragment == null) {
17                 newTab.fragment = Fragment.instantiate(this,
18                     newTab.klass.getName(), newTab.args);
19                 ft.add(android.R.id.tabcontent, newTab.fragment, newTab.tag);
20             } else {
21                 ft.attach(newTab.fragment);
22             }
23         }
24         mLastTab = newTab;
25         ft.commit();
26         getSupportFragmentManager().
27             executePendingTransactions();
28     }
29 }

```

Algoritmo 5.31: Método `onTabChanged()`

Primeiro obtemos as informações da aba que queremos do mapa com `mapTabInfo.get(tag)`, usamos a `tag` para obter o objeto que queremos. Em seguida testamos para saber se a aba selecionada é a mesma que a anterior, pois não faria sentido recarregar a mesma aba. Na linha 9 testado para saber se a última aba não é nula, isso deve ser feito para evitar uma falha do aplicativo, testamos também se o fragmento é nulo para então usar `detach()` para retirar esse fragmento do *layout*.

Fazemos o mesmo com a nova aba, caso o fragmento seja nulo isso quer dizer que ele não foi instanciado ainda, isto é, é a primeira vez que o usuário seleciona essa aba nesse ciclo de vida do aplicativo. Caso isso ocorra, então usamos `instantiate()` para instanciar esse novo fragmento e `add` para adiciona-lo ao *layout*. Caso ele já tenha sido instanciado, então apenas usamos `attach()` para coloca-lo de volta no *layout*.

Por final é preciso salvar a aba que estavamos caso o aplicativo fique em segundo plano. O método `onSaveInstanceState` fica encarregado disso.

```
1 protected void onSaveInstanceState(Bundle outState) {  
2     outState.putString("tab", mTabHost.getCurrentTabTag());  
3     super.onSaveInstanceState(outState);  
4 }
```

Algoritmo 5.32: Método `onSaveInstanceState()`

A figura abaixo mostra o resultado.

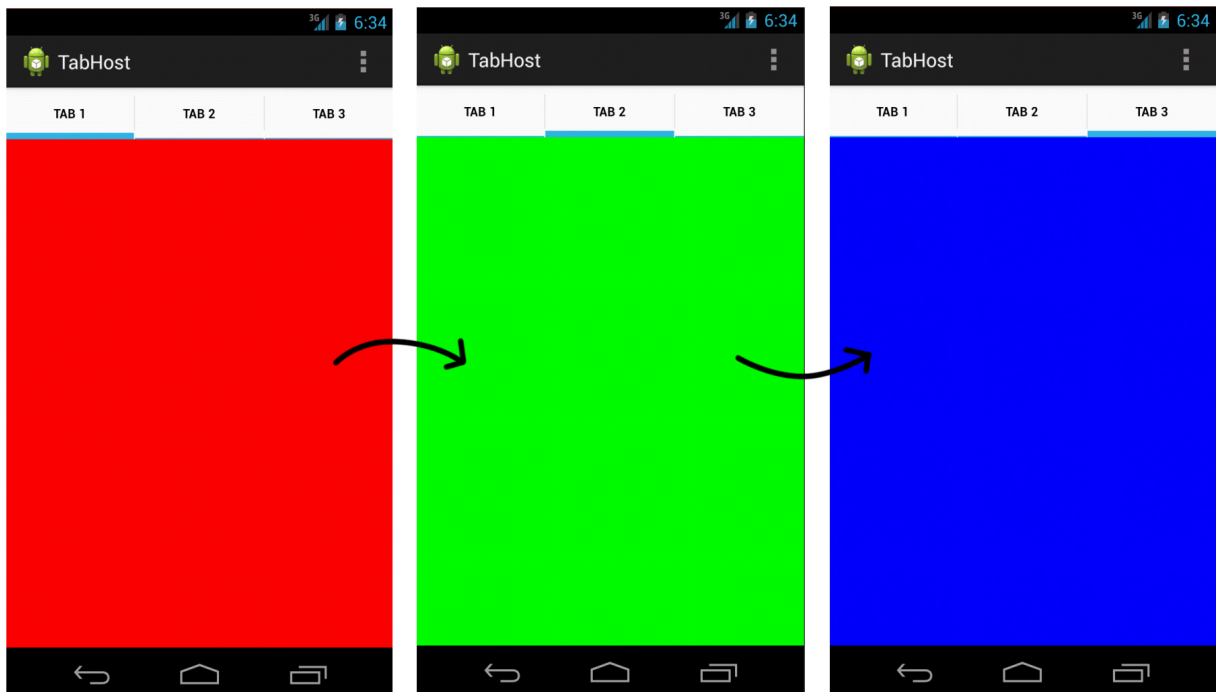


Figura 5.16: Figura mostrando as 3 abas criadas no exemplo

5.10 Trocar de página com gesto de arrastar usando ViewPager

É possível trocar entre fragmentos usando o gesto de arrastar, isto é, arrastando a tela de um lado para o outro acionará a troca entre os fragmentos.

Primeiro iremos definir o *layout* do ViewPager¹⁸. Depois iremos definir o PagerAdapter¹⁹. Por último precisamos definir a *activity* que irá conter o visualizador de páginas.

```
1 <LinearLayout
2   xmlns:android="http://schemas.android.com/apk/res/android"
3   android:layout_width="fill_parent"
4   android:layout_height="fill_parent"
5   android:orientation="vertical" >
6
7   <android.support.v4.view.ViewPager
8       android:id="@+id/viewpager"
9       android:layout_width="fill_parent"
10      android:layout_height="fill_parent" />
11
12 </LinearLayout>
```

Algoritmo 5.33: *layout* do ViewPager

Agora defina uma nova classe chamada PagerAdapter que irá estender a classe FragmentPagerAdapter. Primeiro crie uma lista que irá conter os fragmentos, isto é, as páginas que serão exibidas. Ao estender essa classe precisamos implementar dois métodos: `getItem()` e `getCount()`. O método `getItem()`, na linha 10 do algoritmo 5.34, deve retornar o item que será selecionado pelo parâmetro `position`. O método `getCount()`, na linha 15, deve retornar a quantidade de páginas. Depois crie o construtor como mostrado nas linhas 4-7.

¹⁸<http://developer.android.com/reference/android/support/v4/view/ViewPager.html>

¹⁹<http://developer.android.com/reference/android/support/v4/view/PagerAdapter.html>

```
1 public class PagerAdapter extends FragmentPagerAdapter{
2     private List<Fragment> fragments;
3
4     public PagerAdapter(FragmentManager fm, List<Fragment> fragments) {
5         super (fm);
6         this.fragments = fragments;
7     }
8
9     @Override
10    public Fragment getItem(int position) {
11        return this.fragments.get(position);
12    }
13
14    @Override
15    public int getCount() {
16        return this.fragments.size();
17    }
18 }
```

Algoritmo 5.34: Classe PagerAdapter

Por último devemos construir a *activity* que irá conter o `PagerAdapter` e será responsável por mostrar as páginas. Neste exemplo iremos reutilizar os fragmentos que fizemos na seção anterior quando trabalhamos com abas.

Dica: Você pode importar as classes e os arquivos de *layout*. No *Package Explorer* na IDE Eclipse, clique com o botão direito sobre a pasta que quer importar os arquivos, depois clique em *Import* e selecione *General -> File System*. Agora selecione o caminho da pasta que contém os arquivos no campo *From directory*. Selecione os arquivos que deseja importar e clique em *Finish*.

Note que ao importar classes Java será necessário trocar o pacote a que a classe pertence.

Outra opção é usar o `import` do java para importar as classes sem ter elas no pacote.

Essa *activity*, como mostrada no algoritmo 5.35 abaixo, apenas precisa instanciar os fragmentos (linhas 10,11 e 12), criar e determinar o adaptador, linhas 14, 16 e 17.

```
1 public class ViewPagerLayout extends FragmentActivity {
2     private PagerAdapter mPagerAdapter;
3
4     @Override
5     protected void onCreate(Bundle savedInstanceState) {
6         super.onCreate(savedInstanceState);
7         setContentView(R.layout.activity_viewpager_layout);
8
9         List<Fragment> fragments = new ArrayList<Fragment>();
10        fragments.add(Fragment.instantiate
11            (this, Tab1Fragment.class.getName()));
12        fragments.add(Fragment.instantiate
13            (this, Tab2Fragment.class.getName()));
14        fragments.add(Fragment.instantiate
15            (this, Tab3Fragment.class.getName()));
16
17        mPagerAdapter = new PagerAdapter(getSupportFragmentManager(), fragments);
18
19        ViewPager pager = (ViewPager) findViewById(R.id.viewpager);
20        pager.setAdapter(mPagerAdapter);
21    }
22    ...
23 }
```

Algoritmo 5.35: *Activity* com *PagerAdapter*

5.11 Abas com gesto de arrastar

Ao juntar os dois conceitos, o de *layout* com abas e o gesto de arrastar, podemos fazer o controle das abas arrastando a tela. Esse tipo de *design* é comum em muitos aplicativos pela facilidade e rapidez com o que o usuário pode visualizar vários conteúdos. Nesse exemplo iremos reutilizar o código dos exemplos anteriores com algumas modificações. Serão reutilizadas classes: *TabInfo*, *TabFactory*, *PagerAdapter*, *Tab1Fragment*, *Tab2Fragment*, *Tab3Fragment*.

Primeiro modificaremos o *layout* das abas adicionando o *ViewPager* após *FrameLayout*. Note que esse é o mesmo *layout* do algoritmo 5.24, por isso o algoritmo 5.36 não está completo.

```

1 ...
2 <FrameLayout
3   android:id="@android:id/tabcontent"
4   android:layout_width="fill_parent"
5   android:layout_height="fill_parent" >
6
7   <android.support.v4.view.ViewPager
8     android:id="@+id/viewpager"
9     android:layout_width="fill_parent"
10    android:layout_height="0dp"
11    android:layout_weight="1" />
12
13 </FrameLayout>
14 ...

```

Algoritmo 5.36: *Layout* das abas com adição do ViewPager

Agora iremos usar a classe `TabLayoutActivity` do exemplo anterior e fazer algumas alterações. Nesse exemplo irei mudar o nome da classe para `SwipeTabActivity`. A primeira alteração é a adição de algumas variáveis para serem usadas na classe, adicione uma variável `PagerAdapter` e uma `ViewPager`. Depois devemos alterar o método `onCreate()`, adicione uma chamada ao método `initialiseViewPager()`. A implementação é a mesma do método `onCreate()` do exemplo anterior com a adição da chamada ao método `setOnPageChangeListener()`. Devemos também implementar a interface `OnPageChangeListener`. No método `initialiseTabHost()` você precisa remover a linha `onTabChanged("Tab1");`.

```

1 private void initialiseViewPager() {
2   List<Fragment> fragments = new ArrayList<Fragment>();
3   fragments.add(Fragment.instantiate
4     (this, Tab1Fragment.class.getName()));
5   fragments.add(Fragment.instantiate
6     (this, Tab2Fragment.class.getName()));
7   fragments.add(Fragment.instantiate
8     (this, Tab3Fragment.class.getName()));
9
10  mPagerAdapter = new PagerAdapter(getSupportFragmentManager(), fragments);
11
12  mViewPager = (ViewPager) findViewById(R.id.viewpager);
13  mViewPager.setAdapter(mPagerAdapter);
14  mViewPager.setOnPageChangeListener(this);
15 }

```

Algoritmo 5.37: Método `initialiseViewPager()`

Em seguida precisamos alterar o método `onTabChanged()`, não necessitamos mais fazer verificações já que o próprio *design* irá limitar as falhas que poderiam ocorrer. Precisamos apenas determinar para o `ViewPager` o item atual.

```
1 @Override
2 public void onTabChanged(String tag) {
3     int pos = mTabHost.getCurrentTab();
4     mViewPager.setCurrentItem(pos);
5 }
```

Algoritmo 5.38: Método `onTabChanged()` alterado

Por fim, devemos implementar os métodos da interface `ViewPager.OnPageChangeListener`, somente o método `onPageSelected()` será usado, para selecionar a aba correspondente à página atual.

```
1 @Override
2 public void onPageScrollStateChanged(int arg0) {
3     //Nada
4 }
5
6 @Override
7 public void onPageScrolled(int arg0, float arg1, int arg2) {
8     //Nada
9 }
10
11 @Override
12 public void onPageSelected(int position) {
13     mTabHost.setCurrentTab(position);
14 }
```

Algoritmo 5.39: Métodos da interface `ViewPager.OnPageChangeListener`

Dica: Pela dificuldade em acompanhar passo a passo a construção desse *design*, é recomendável obter o código do projeto no repositório. Está sob o nome *SwipeableTabs*.

5.12 ActionBar

A *ActionBar* é aquela barra presente em todos os aplicativos que fizemos de exemplo até agora. Ela pode mostrar o nome da *activity*, ícones, ações que podem ser acionadas, outras *views* ou botões interativos. Também pode ser usada para navegar entre as *activities* do seu aplicativo.

Dispositivos Android mais antigos possuem um botão físico chamado *Option* que abre um menu na parte inferior do aplicativo. A *ActionBar* é melhor que esse menu pois está claramente visível para o usuário, enquanto que o menu antigo era escondido e o usuário pode não reconhecer que as opções estão disponíveis.

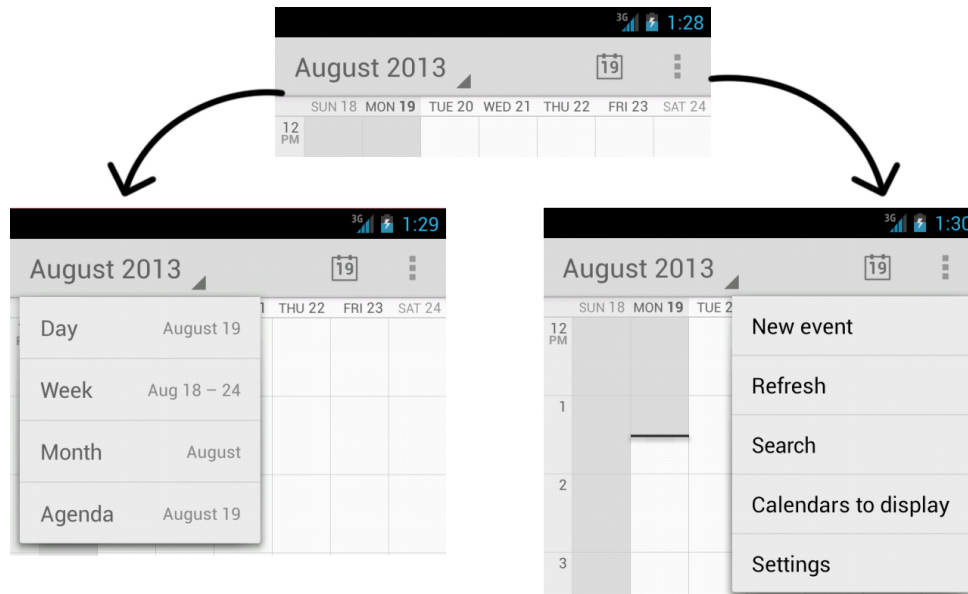


Figura 5.17: Exemplo de *ActionBar* no aplicativo Calendário

A figura 5.18 mostra o uso da *ActionBar* no aplicativo Calendário, padrão dos aparelhos Android mais atuais. É possível observar três principais componentes. O primeiro é um menu *drop-down* que permite ao usuário mudar o modo de visualização do calendário. O segundo é um botão com o dia atual, 19, que ao ser pressionado faz com que o calendário posicione um cursor no dia e hora atuais. O terceiro é um outro menu *drop-down* com algumas opções que podem ser interessantes ao usuário.

5.12.1 Implementando a *ActionBar*

A *activity* popula a *ActionBar* em seu método `onCreateOptionsMenu()`. Entradas na *ActionBar* são chamadas de ações (*actions*).

As ações para a *ActionBar* são definidas em arquivos XML posicionados na pasta `menu/`. O algoritmo abaixo mostra o menu padrão dos exemplos que construímos até agora. Ele só contém um item "*Settings*" que está no *dropdown* menu que pode ser acessado através da *ActionBar*, à direita. O fato dele estar escondido se deve ao atributo `showAsAction` estar com valor *never*, ao ser mudado para *always* o acesso ao *Settings* será diretamente através da *ActionBar*. Existe também o valor *ifRoom* que irá mostrar apenas se houver espaço disponível.

```

1 <menu
2   xmlns:android="http://schemas.android.com/apk/res/android" >
3
4   <item
5     android:id="@+id/action_settings"
6     android:orderInCategory="100"
7     android:showAsAction="always"
8     android:title="@string/action_settings"/>
9
10 </menu>

```

Algoritmo 5.40: Menu padrão dos exemplos

```
1 @Override
2 public boolean onCreateOptionsMenu(Menu menu) {
3     getMenuInflater().inflate(R.menu.main, menu);
4     return true;
5 }
```

Algoritmo 5.41: Método padrão `onCreateOptionsMenu()`

Se uma ação é selecionada, o método `onOptionsItemSelected()` é chamado. Ele recebe a ação selecionada como parâmetro `MenuItem`. Baseando-se nessa informação você pode decidir o que fazer. Nesse exemplo iremos abrir uma nova *activity* que seria a tela de configuração do aplicativo.

No seu projeto, crie uma nova *Activity* do tipo *Settings Activity*.

Dica: Para criar uma nova *activity*, clique com o botão direito sob o projeto selecione *New -> Other* (ou pressione *Ctrl+N*). Selecione *Android Activity* e selecione o tipo desejado.

Agora você deve fazer com que o método `OnCreateOptionsMenu()` abra essa nova *activity*.

```
1 public boolean onOptionsItemSelected(MenuItem item) {
2     if(item.getItemId() == R.id.action_settings) {
3         startActivity(new Intent(this, SettingsActivity.class));
4     }
5     return true;
6 }
```

Algoritmo 5.42: Método `OnOptionsItemSelected()`

Para melhorar nossa *ActionBar* vamos adicionar um campo para pesquisa. Você pode adicionar *views* em sua *ActionBar*. Para isso você deve usar o método `setCustomView()` da classe *ActionBar* e passar uma *view* como parâmetro. Você também precisa ativar a exibição de *views* com o método `setDisplayOptions()` e passar a *flag* `ActionBar.DISPLAY_SHOW_CUSTOM`.

Primeiro vamos adicionar um ícone de busca na nossa *ActionBar*, voltando ao arquivo do *layout* da *ActionBar*, adicione um novo item acima do primeiro como mostrado no algoritmo abaixo.

```

1 <menu xmlns:android="http://schemas.android.com/apk/res/android" >
2
3     <item
4         android:id="@+id/action_search"
5         android:orderInCategory="100"
6         android:showAsAction="always"
7         android:title="Search"
8         android:icon="@android:drawable/ic_menu_search" />
9
10    <item
11        android:id="@+id/action_settings"
12        android:showAsAction="never"
13        android:title="@string/action_settings"/>
14
15 </menu>

```

Algoritmo 5.43: Adicionando novo item na *ActionBar*

Observe o atributo `android:icon`, estamos obtendo um *drawable* que já existe no sistema Android, e se chama `ic_menu_search`. Esse é o ícone da busca, a lupa. Depois vamos adicionar uma ação a ser executada quando esse ícone for clicado. Iremos criar nesse exemplo, uma caixa de texto de forma programática, isto é, em vez de defini-la no XML iremos criá-la com código na *Activity*.

No método `onCreate()` você precisa configurar a *ActionBar* para mostrar *views*. Use o método `getActionBar()` para obter uma referência da *ActionBar* e `setDisplayOptions()` para configurá-la. Depois, no método `onOptionsItemSelected()` você vai programar a ação do novo botão. Inicialmente cria-se uma nova *view* do tipo `EditText`, colocamos algumas configurações e a adicionamos na *ActionBar*. Por não termos uma busca devidamente implementada, vamos mostrar um `Toast` com o conteúdo da busca, a fim de demonstração.

```

1 @Override
2 protected void onCreate(Bundle savedInstanceState) {
3     super.onCreate(savedInstanceState);
4     setContentView(R.layout.activity_main);
5
6     getActionBar().setDisplayOptions
7         (ActionBar.DISPLAY_SHOW_CUSTOM |
8         ActionBar.DISPLAY_SHOW_HOME);
9 }

```

Algoritmo 5.44: Configurando *ActionBar* no método `onCreate()`

No algoritmo 5.45, abaixo, criamos um `EditText` e o configuramos caso o botão com o ícone de busca seja pressionado. Na linha 5-6 criamos um `LayoutParams` que configura o tamanho da *view*. Na linha 8 o método `EditText.setImeOptions()` é responsável por configurar o teclado para uma busca, junto com a linha 9 que diz para o `EditText` que a

entrada será texto, essa combinação faz com que o teclado mostre o ícone da lupa no lugar da tecla *Enter*. A linha 10 configura a cor do texto para branca. Adicionamos a *view* na *ActionBar* com o método `ActionBar.setCustomView()` e passamos como parâmetro a *view* criada e os parâmetros de *layout* criados. O método `EditText.requestFocus()` faz com que o foco seja dado à nova caixa de texto, para que possamos editá-la. Precisamos ainda fazer com que o teclado abra para que possamos editar a caixa de texto, é isso que as linhas 13-14 e 15 estão fazendo. O método `getSystemService()` obtém a referência de um serviço do Android, e nesse caso estamos pedindo pelo serviço de método de entrada, o teclado. O método `InputMethodManager.showSoftInput()` abre o *Soft Input*, ou seja, o teclado virtual para edição da *view*.

Finalmente fazemos com que ao botão de busca no teclado ser clicado, um *Toast* mostre para o usuário o conteúdo da caixa de texto. É isso que a interface `onEditorActionListener` faz com o método `onEditorAction()`.

A intenção desse exemplo é mostrar como você pode adicionar novas *views* na *ActionBar* de forma dinâmica.

```

1 public boolean onOptionsItemSelected(MenuItem item) {
2     if(item.getItemId() == R.id.action_settings) {
3         startActivity(new Intent(this, SettingsActivity.class));
4     } else if(item.getItemId() == R.id.action_search) {
5         LayoutParams lp = new LayoutParams
6             (LayoutParams.MATCH_PARENT, LayoutParams.MATCH_PARENT);
7         EditText search = new EditText(this);
8         search.setImeOptions(EditorInfo.IME_ACTION_SEARCH);
9         search.setTextColor(Color.WHITE);
10        search.setInputType(InputType.TYPE_CLASS_TEXT);
11        getActionBar().setCustomView(search, lp);
12        search.requestFocus();
13        InputMethodManager imm = (InputMethodManager)
14            getSystemService(Context.INPUT_METHOD_SERVICE);
15        imm.showSoftInput(search, InputMethodManager.SHOW_IMPLICIT);
16        search.setOnEditorActionListener(new OnEditorActionListener() {
17
18            @Override
19            public boolean onEditorAction
20                (TextView v, int actionId, KeyEvent event) {
21                Toast.makeText
22                    (MainActivity.this, v.getText(), Toast.LENGTH_SHORT).show();
23                return true;
24            }
25        });
26    }
27    return true;
28 }

```

Algoritmo 5.45: Criando a caixa de busca na *ActionBar*

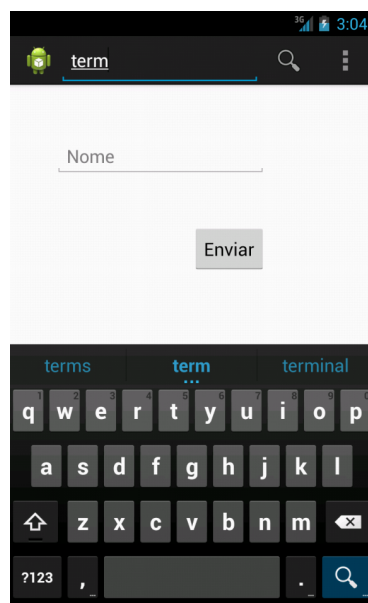


Figura 5.18: Exemplo de busca na *ActionBar*

Comunicação

Agora iremos explorar as funcionalidades de comunicação dos aparelhos Android como acesso a *Internet* no caso de *Tables* e *Smartphones* e envio de SMS e chamadas de voz com os *Smartphones*.

6.1 Internet

Antes de iniciar sua aplicação que faz acesso à *Internet*, devemos dar permissão ao aplicativo através do arquivo de *Manifest*. Logo antes da tag `uses-sdk` você deve adicionar a tag `uses-permission`, como mostrado abaixo:

```
<uses-permission android:name="android.permission.INTERNET"/>
```

Algoritmo 6.1: Atribuindo permissão de acesso à *Internet* no *Manifest*

Nesta seção, veremos como se faz requisições HTTP para obter páginas HTML, saídas de um *script server-side* como PHP ou ASP.NET e *parsing* de respostas JSON ou XML.

6.1.1 HTTP GET

Para fazer uma requisição HTTP GET usaremos as classes da biblioteca *Apache*, tais como *HttpClient*, *HttpGet* e *HttpResponse*. Primeiros crie uma classe que iremos chamar de `RequestTask` e ela deve estender a classe `AsyncTask`¹, ela vai permitir que façamos a requisição na *thread* da interface do usuário sem precisar criar e manipular uma *thread* diferente.

¹<http://developer.android.com/reference/android/os/AsyncTask.html>

```
1 public class RequestTask extends AsyncTask<URI, Integer, String>{
2
3     @Override
4     protected String doInBackground(URI... uri) {
5         HttpClient httpClient = new DefaultHttpClient();
6         HttpResponse response;
7         String responseString = null;
8
9         try {
10             response = httpClient.execute(new HttpGet(uri[0]));
11             StatusLine statusLine = response.getStatusLine();
12
13             if(statusLine.getStatusCode() == HttpStatus.SC_OK){
14                 ByteArrayOutputStream out = new ByteArrayOutputStream();
15                 response.getEntity().writeTo(out);
16                 out.close();
17                 responseString = out.toString();
18             } else {
19                 response.getEntity().getContent().close();
20                 throw new IOException(statusLine.getReasonPhrase());
21             }
22         } catch (ClientProtocolException e) {
23             e.printStackTrace();
24         } catch (IOException e) {
25             e.printStackTrace();
26         }
27         return responseString;
28     }
29 }
```

Algoritmo 6.2: Classe RequestTask

O método `doInBackground()` é o responsável por realizar a operação sem o usuário perceber. Existem outros dois métodos da classe `AsyncTask` que podem ser utilizados: `onProgressUpdate()`, caso queira mostrar o progresso de um *download* para o usuário e `onPostExecute()` para executar alguma ação após o termino do *download*.

Observe o algoritmo 6.2, na linha 5-7 criamos um novo `HttpClient`, `HttpResponse` e uma *string* para armazenar a resposta do servidor. Depois, envolto por um bloco `try-catch` temos uma chamada ao método `HttpClient.execute()` e é passado para ele um novo `HttpGet` com o parâmetro `uri[0]`, `uri` é um dos parâmetros do método e contém uma `URI`² que identifica o destino da requisição.

Na linha 11 foi criado um novo `StatusLine`³ para guardar a resposta da requisição HTTP que obtemos com o método `HttpResponse.getStatusLine()`. Depois comparamos o código do *status* com `HttpStatus.SC_OK` (equivalente ao código 200), o *status* OK significa que a requisição e a resposta ocorreram como esperado e temos a resposta correta vindo do servidor.

²<http://developer.android.com/reference/java/net/URI.html>

³<http://developer.android.com/reference/org/apache/http/StatusLine.html>

Em seguida criamos um novo `ByteArrayOutputStream` que é responsável por guardar a resposta do servidor na chamada ao método `HttpResponse.getEntity().writeTo()`. Por último convertemos o *byte stream* em uma string usando o método `toString()` e a retornamos.

Agora precisamos de uma *Activity* para mostrar a resposta, nesse exemplo obteremos o código HTML da página do DC UFSCar: <http://www.dc.ufscar.br>. Mostraremos o código HTML como uma página *Web* usando o `WebView`.

```
1 @Override
2 protected void onCreate(Bundle savedInstanceState) {
3     super.onCreate(savedInstanceState);
4     setContentView(R.layout.activity_main);
5
6     String uri = "http://www.dc.ufscar.br";
7     AsyncTask<String,String,String> task;
8     String response = null;
9
10    try {
11        task = new RequestTask(this).execute(uri);
12        response = task.get();
13    } catch (Exception e) {
14        e.printStackTrace();
15    }
16
17    if(response != null) {
18        WebView webview = new WebView(this);
19        setContentView(webview);
20
21        webview.loadData(response, "text/html; charset=UTF-8", null);
22    }
23 }
```

Algoritmo 6.3: Usando `RequestTask` na *activity*

Primeiro criamos uma nova variável `task` do tipo `AsyncTask` nos moldes da classe que criamos, dentro de um bloco `try-catch` criamos uma nova URI e atribuímos a ela o endereço da página do DC UFSCar. Em seguida iniciamos a variável `task` criando uma nova `RequestTask` e chamamos o método `execute()`, que ao ser chamado irá executar o método `doInBackground()`.

Por fim, criamos uma nova `WebView`⁴ e carregamos o conteúdo da *string* `response` nela usando o método `WebView.loadData()`. A página é mostrada sem estilo pois não estamos puxando as folhas de estilo junto, somente o código HTML.

Dica: Usamos a `WebView` para mostrar o código HTML como página apenas para demonstração. Caso queira carregar uma página corretamente deve fornecer a *URL* da página diretamente para a `WebView`.

⁴<http://developer.android.com/reference/android/webkit/WebView.html>

6.1.2 HTTP POST

Diferentemente do *GET*, onde os parâmetros para o servidor vão codificados na *URL*, no *POST* os parâmetros vão codificados no final do cabeçalho HTTP. Para enviar uma requisição do tipo *POST*, usaremos a classe `HttpPost`. O algoritmo abaixo é reaproveitado da classe `RequestTask` e do método `doInBackground()`, onde mudaremos apenas algumas linhas. Agora estaremos enviando a requisição para essa página: <http://httpbin.org/post>, ela só aceita requisições do tipo *POST*, se tentar com *GET* irá receber uma mensagem de erro.

```
1 try {  
2     HttpPost post = new HttpPost(uri[0]);  
3     List<NameValuePair> nvp = new ArrayList<NameValuePair>();  
4     nvp.add(new BasicNameValuePair("testing", "Post"));  
5     nvp.add(new BasicNameValuePair("user", "You"));  
6  
7     post.setEntity(new UrlEncodedFormEntity(nvp));  
8     response = httpClient.execute(post);  
9     ...  
10 ...  
11 }
```

Algoritmo 6.4: Modificando o método para requisições *POST*

Criamos um novo `HttpPost`⁵ passando a URI como parâmetro. Criamos uma lista de tuplas chave-valor que representa a variável e seu valor no cabeçalho HTTP e adicionamos dois valores. Em seguida chamamos o método `HttpPost.setEntity()` e passamos um objeto do tipo `UrlEncodedFormEntity`⁶ que recebe a lista como parâmetro, esse objeto irá codificar a lista em variáveis aceitas pelo padrão de uma requisição *POST*.

Por final, executamos a requisição como feito anteriormente. O resto do código é igual.

Note que se colocar a resposta dessa requisição em uma `WebView` ou `TextView` irá observar que está codificado no formato JSON⁷.

6.1.3 Decodificando JSON

Vamos obter dados do objeto JSON retornado pelo exemplo anterior. Se você observar a *string* na tela, irá perceber os dados que foram enviado via *POST* dentro de um objeto JSON chamado *form*. Queremos obter esses dados, para isso precisamos criar uma classe que será responsável por obter especificamente esses dados do *form*.

Dica: Em JSON, { representa um objeto JSON e [representa um *array* dentro de um objeto JSON

Crie uma classe `JSONParser`, como mostrado abaixo.

⁵<http://developer.android.com/reference/org/apache/http/client/methods/HttpPost.html>

⁶<http://developer.android.com/reference/org/apache/http/client/entity/UrlEncodedFormEntity.html>

⁷O que é JSON: <http://www.json.org/>

```
1 public class JSONParser {
2
3     public String getFormData(String jsonstr) {
4         String formData = null;
5
6         try {
7             JSONObject jsonObj = new JSONObject(jsonstr);
8             JSONObject form = jsonObj.getJSONObject("form");
9             formData = form.getString("testing");
10            formData += "\n" + form.getString("user");
11
12        } catch (JSONException e) {
13            e.printStackTrace();
14        }
15
16        return formData;
17    }
18 }
```

Algoritmo 6.5: Classe JSONParser

Para obter dados do *form*, criamos um método `getFormData()`. Primeiro obtemos o objeto JSON dado pela *string* retornada pela requisição *POST* do exemplo anterior ao criar um novo `JSONObject` e passando a *string* como parâmetro. Em seguida queremos obter outro objeto JSON, aquele cujo nome é *form*, para isso chamamos o método `JSONObject.getJSONObject()` passando o nome do objeto junto. Após obter o objeto desejado, usamos o método `JSONObject.getString()` e passamos o nome do valor para obter o valor. Então ao passar "*testing*" e "*user*" esperamos como retorno *POST* e *You*, respectivamente.

6.1.4 Codificando JSON

Podemos também codificar objetos JSON, e é simples. Basta criar um `JSONObject` ou `JSONArray` e usar o método `toString()`. Por exemplo:

```
1 public void writeJSON() {
2     JSONObject object = new JSONObject();
3     try {
4         object.put("name", "Matheus");
5         object.put("age", new Integer(22));
6         object.put("university", "UFSCar");
7     } catch (JSONException e) {
8         e.printStackTrace();
9     }
10 }
```

Algoritmo 6.6: Criando JSON

6.2 Telefone

É possível fazer uma chamada telefônica, porém você terá que usar o discador padrão do Android uma vez que ele não provê uma API pública para fazer chamadas diretamente pelo seu aplicativo. A única forma é através do intermediador `ACTION_CALL`. O Exemplo abaixo mostra como fazer uma chamada usando esse intermediador.

```
1 @Override
2 protected void onCreate(Bundle savedInstanceState) {
3     super.onCreate(savedInstanceState);
4     setContentView(R.layout.activity_main);
5
6     Button call = (Button) findViewById(R.id.button1);
7     call.setOnClickListener(new OnClickListener() {
8
9         @Override
10        public void onClick(View v) {
11            EditText tv = (EditText) findViewById(R.id.editNumber);
12            String phone = tv.getText().toString();
13
14            Intent callIntent = new Intent(Intent.ACTION_CALL);
15            callIntent.setData(Uri.parse("tel:" + phone));
16            startActivity(callIntent);
17        }
18    });
19 }
```

Algoritmo 6.7: Fazendo uma chamada telefônica

Para esse exemplo, foi criado uma caixa de texto e um botão, o número de telefone é colocado na caixa de texto e o botão *Call* inicia a *activity* responsável por realizar a chamada. Observe que é necessário fazer um *parse* do número de telefone para uma *URI* com formato `tel:0123456789`. Cria-se uma *Intent* para o `ACTION_CALL`, coloca-se os dados na *Intent* e chama o método `startActivity()` para fazer a chamada telefônica. É preciso dar permissão ao aplicativo para fazer chamadas adicionando essa linha ao *Manifest*.

```
1 <uses-permission android:name="android.permission.CALL_PHONE"/>
```

Algoritmo 6.8: Permissão para fazer chamadas telefônicas

Usando a classe `TelephonyManager`⁸, no entanto, o Android nos dá a possibilidade obter dados do *Sim Card* e também um *listener* para saber o status atual do telefone, se ele está fazendo uma ligação ou não.

Você pode usar os métodos `getDeviceId()` para obter o número IMEI do aparelho, `getSimSerialNumber()` para obter o número de série do *Sim Card*, `isNetworkRoaming()` para saber se está em modo *roaming*, etc.

⁸<http://developer.android.com/reference/android/telephony/TelephonyManager.html>

6.3 Short Message Service (SMS)

O Android provê uma API pública para mandar mensagens SMS, através da classe `SMSManager`⁹. Assim como fazer ligações, enviar SMS também precisa configurar a permissão no *Manifest*.

```
1 <uses-permission android:name="android.permission.SEND_SMS"/>
```

Algoritmo 6.9: Permissão para enviar mensagens SMS

Agora vamos criar um aplicativo simples que chama a classe `SMSManager` para enviar uma mensagem para um número de telefone. Teremos duas `EditText` *views* para abrigar a mensagem e o número, e um botão para enviar. Inicialmente crie um método `sendSMS()`.

```
1 private void sendSMS(String message, String phone){
2     try{
3         SmsManager smsMan = SmsManager.getDefault();
4         smsMan.sendTextMessage(phone, null, message, null, null);
5     } catch (IllegalArgumentException e) {
6         e.printStackTrace();
7         Toast.makeText(this, "Error sending message", Toast.LENGTH_SHORT)
8             .show();
9     }
10 }
```

Algoritmo 6.10: Método `sendSMS()`

Dentro de um bloco `try-catch` crie um novo `SMSManager` como mostrado na linha 3 do algoritmo 6.11, use o método `SmsManager.sendTextMessage()` para enviar uma mensagem. O primeiro parâmetro é o número de telefone para qual a mensagem será enviada, o segundo é o telefone de origem, colocando `null` o valor será o número do próprio aparelho ou serviço, o terceiro parâmetro é o texto da mensagem.

Completando, com um algoritmo simples para chamar o método quando o botão for pressionado.

⁹<http://developer.android.com/reference/android/telephony/SmsManager.html>

```
1 @Override
2 protected void onCreate(Bundle savedInstanceState) {
3     super.onCreate(savedInstanceState);
4     setContentView(R.layout.activity_main);
5
6     Button btnSend = (Button) findViewById(R.id.buttonSend);
7     btnSend.setOnClickListener(new OnClickListener() {
8
9         @Override
10        public void onClick(View v) {
11            EditText editMsg = (EditText) findViewById(R.id.editMessage);
12            String message = editMsg.getText().toString();
13            EditText editPhone = (EditText) findViewById(R.id.editPhone);
14            String phone = editPhone.getText().toString();
15
16            sendSMS(message, phone);
17
18            //Clean text fields and show toast
19            editMsg.setText("");
20            editPhone.setText("");
21            Toast.makeText
22            (getApplicationContext(), "Message sent", Toast.LENGTH_SHORT)
23              .show();
24        }
25    });
26 }
```

Algoritmo 6.11: Chamando método sendSMS()

Armazenamento

O Android provê algumas opções para que você possa salvar dados persistentes de sua aplicação. A escolha da opção de armazenamento vai depender das necessidades da sua aplicação, isto é, se os dados vão ser visíveis somente pela aplicação ou se o usuário terá acesso a eles, quanto de espaço será necessário, que tipo de dados você pretende salvar. As opções são:

- *Shared Preferences*: Usada principalmente para salvar preferências do usuário, esse método guarda primitivas em duplas chave-valor;
- Armazenamento Interno: Salva dados privados na memória do dispositivo;
- Armazenamento Externo: Salva dados públicos na memória externa compartilhada;
- Banco de Dados *SQLite*: Salva dados estruturados em um banco de dados privado.
- Conexão com a rede: Salva dados na *web* em seu próprio servidor na rede.

7.1 *Shared Preferences*:

A classe `SharedPreferences`¹ fornece um framework que te permite salvar e recuperar dados primitivos persistentes no formato chave-valor. Você pode salvar qualquer tipo de dado primitivo: *booleans*, *floats*, inteiros, *longs* e *strings*.

Você pode usar essa classe para armazenar dados durante o ciclo de vida de uma *activity*. Isto é, antes da *activity* ser destruída na função `onDestroy()`, os dados podem ser salvos durante a execução de `onStop()` e recuperados na execução de `onCreate()`.

Para usar a `SharedPreferences` na sua aplicação você deve chamar `getSharedPreferences()` se precisar de vários arquivos de preferencias que serão identificados pelo nome ou `getPreferences()` se precisar de apenas um arquivo para sua *activity*, esse não necessita de nome pois será único para a *activity*. Em seguida para escrever valores você deve chamar o método `edit()` para obter um `SharedPreferences.Editor`, e usar os métodos como `putString()` para adicionar novos valores. Por último `commit()` deve ser

¹<http://developer.android.com/reference/android/content/SharedPreferences.html>

chamado para salvar os valores. Para ler os valores, você deve chamar métodos como `getBoolean()` ou `getString()`.

O exemplo abaixo salva os dados de uma caixa de texto, uma barra de progresso e uma chave em uma `SharedPreferences` e depois os lê quando o aplicativo é aberto novamente.

```

1 public class MainActivity extends Activity {
2     private EditText mEditText;
3     private SeekBar mSeekBar;
4     private Switch mSwitch;
5
6     @Override
7     protected void onCreate(Bundle savedInstanceState) {
8         super.onCreate(savedInstanceState);
9         setContentView(R.layout.activity_main);
10
11         mEditText = (EditText) findViewById(R.id.editText1);
12         mSeekBar = (SeekBar) findViewById(R.id.seekBar1);
13         mSwitch = (Switch) findViewById(R.id.switch1);
14
15         SharedPreferences prefs = getPreferences(MODE_PRIVATE);
16         String text = prefs.getString("text", "");
17         int progress = prefs.getInt("seek", 0);
18         boolean checked = prefs.getBoolean("switch", false);
19
20         mEditText.setText(text);
21         mSeekBar.setProgress(progress);
22         mSwitch.setChecked(checked);
23     }
24
25     @Override
26     protected void onStop() {
27         super.onStop();
28
29         SharedPreferences prefs = getPreferences(MODE_PRIVATE);
30         SharedPreferences.Editor editor = prefs.edit();
31
32         editor.putString("text", mEditText.getText().toString());
33         editor.putInt("seek", mSeekBar.getProgress());
34         editor.putBoolean("switch", mSwitch.isChecked());
35         editor.commit();
36     }
37     ...
38 }

```

Algoritmo 7.1: Utilizando `SharedPreferences` para salvar dados primitivos

Observe no algoritmo 7.1 a utilização da classe `SharedPreferences`, no método `onStop()` chamamos o método `getPreferences(MODE_PRIVATE)` para obter uma instância da classe no modo privado, isto é, os dados somente são acessíveis por esta *activity* deste

aplicativo. Na linha 30 obtemos o `SharedPreferences.Editor` ao chamar o método `SharedPreferences.edit()`.

O `Editor` é usado então para adicionar os valores que obtemos das *views*. Na linha 32 usamos `putString()` para guardar o conteúdo da caixa de texto, `putInt()` para guardar a posição da `SeekBar` e `putBoolean()` para guardar o estado da chave. Finalmente invocamos `commit()` para salvar os dados.

No método `onCreate()` usamos os métodos `get` para obter os dados que foram salvos. Esses métodos requerem a passagem de um valor *default* para quando não há dados previamente salvos. Observe então nas linhas 16-18 que para a caixa de texto usamos uma *string* vazia, para a `SeekBar` o valor 0 e a chave desligada.

7.2 Armazenamento interno

Você pode salvar arquivos diretamente na memória interna do dispositivo. Por padrão os arquivos salvos no armazenamento interno são privados a seu aplicativo e não podem ser acessados por outros aplicativos.

Para criar um novo arquivo você deve invocar o método `openFileOutput()` passando o nome do arquivo e o modo de operação. Esse método irá retornar um `FileOutputStream`. Você poderá escrever nesse arquivo chamando o método `FileOutputStream.write()` e `FileOutputStream.close()` para fechar o arquivo.

Para exemplificar, faremos um aplicativo do tipo bloco de notas. Será composto de dois botões, um para abrir um arquivo e outro para salvar um arquivo, e uma caixa de texto para escrever. Tudo será feito no método `onCreate()`, primeiro usamos `findViewById()` para obter as *views*. O botão de salvar irá abrir um alerta perguntando o nome do arquivo, e o botão de abrir mostrará um alerta permitindo ao usuário escolher dentre os arquivos já salvos anteriormente.

No código abaixo criamos o *listener* do botão de salvar, dentro dele criamos um alerta e precisamos de um *listener* para confirmar e um para cancelar.

```
1 saveBtn.setOnClickListener(new OnClickListener() {
2
3     @Override
4     public void onClick(View v) {
5
6         AlertDialog.Builder builder =
7             new AlertDialog.Builder(MainActivity.this);
8         LayoutInflater inflater = getLayoutInflater();
9         final View fnameEntry =
10             inflater.inflate(R.layout.save_dialog, null);
11         builder.setView(fnameEntry).setTitle("Save as...")
12             .setPositiveButton("Save", new DialogInterface.OnClickListener() {
13
14                 ...
```

Algoritmo 7.2: Passos iniciais do *listener*, criando um alerta.

Com o `AlertDialog.Builder` começamos a construir um novo alerta. Usamos o `LayoutInflater` para carregar o *layout* com uma `EditText`. Chamamos `setPositiveButton()` para criar um *listener* que irá salvar o arquivo quando o botão for clicado.

```

1 ...
2 .setPositiveButton("Save", new DialogInterface.OnClickListener() {
3
4     @Override
5     public void onClick(DialogInterface dialog, int which) {
6         EditText fnameEt = (EditText) fnameEntry.findViewById(R.id.saveas);
7         String fname = fnameEt.getText().toString();
8         if (fname.isEmpty())
9             fname = "untitled";
10        String text = textEt.getText().toString();
11
12        try {
13            FileOutputStream fos = openFileOutput(fname, MODE_PRIVATE);
14            fos.write(text.getBytes());
15            fos.close();
16        } catch (FileNotFoundException e) {
17            e.printStackTrace();
18        } catch (IOException e) {
19            e.printStackTrace();
20        }
21
22        Toast.makeText(getApplicationContext(),
23            fname + " saved", Toast.LENGTH_SHORT).show();
24    }
25 }) .setNegativeButton("Cancel", new DialogInterface.OnClickListener() {
26 ...

```

Algoritmo 7.3: Salvando um arquivo e mostrando um Toast

Vamos analisar o algoritmo 7.3. Na linha 5 queremos a referência ao `EditText` contido no alerta, para isso precisamos usar a variável `fnameEntry` criada ao chamar o método `LayoutInflater.inflate()`. Depois, na linha 7-8 certificamos que o nome do arquivo não será vazio (se for vazio, o arquivo não é salvo), para isso atribuímos o nome *untitled* a ele. Obtemos também o texto escrito na caixa de texto da *activity* na linha 9.

Agora que já temos o nome e os dados do arquivo, precisamos efetivamente salvá-lo no armazenamento interno. Com o método `openFileOutput()` teremos um `FileOutputStream` que servirá para escrever os dados no arquivo. Isso é feito nas linhas 13-14 com o método `write()` e em seguida o método `close()`. Para certificar ao usuário que o arquivo foi salvo, mostramos um `Toast` com a mensagem que o arquivo foi salvo.

Se o usuário clicar em *cancel* no alerta, precisamos fechá-lo, basta invocar `dialog.close()` no *listener*.

```
1 ...
2 }).setNegativeButton("Cancel", new DialogInterface.OnClickListener() {
3
4     @Override
5     public void onClick(DialogInterface dialog, int which) {
6         dialog.cancel();
7     }
8 });
9 ...
```

Algoritmo 7.4: Fechando o alerta ao clicar em *close*

Depois chame `builder.create().show()` no listener do botão *Save* para mostrar o alerta.

Para o botão *Open* iremos mostrar um alerta com os nomes dos arquivos salvos anteriormente, quando um desses for clicado o arquivo se abrirá preenchendo a caixa de texto.

```
1 openBtn.setOnClickListener(new OnClickListener() {
2
3     @Override
4     public void onClick(View v) {
5
6         AlertDialog.Builder builder =
7             new AlertDialog.Builder(MainActivity.this);
8
9         builder.setTitle("Choose a File")
10            .setItems(fileList(), new DialogInterface.OnClickListener() {
11
12                ...
13            });
14
15         builder.create().show();
16     }
17 });
```

Algoritmo 7.5: Criando um alerta com os arquivos salvos

Dessa vez usamos o método `builder.setItems()` para construir uma lista no alerta. O método `fileList()` retorna um array de nomes de arquivo que foram armazenados internamente no aplicativo.

```

1 ...
2 .setItems(fileList(), new DialogInterface.OnClickListener() {
3
4     @Override
5     public void onClick(DialogInterface dialog, int which) {
6
7         try {
8             File f = getFilesDir().listFiles()[which];
9             BufferedReader in = new BufferedReader(new FileReader(f));
10            StringBuilder text = new StringBuilder();
11
12            try{
13                String line = null;
14                while ((line = in.readLine()) != null) {
15                    text.append(line);
16                    text.append(System.getProperty("line.separator"));
17                }
18            } finally {
19                in.close();
20            }
21
22            textEt.setText(text);
23            in.close();
24        } catch (FileNotFoundException e) {
25            e.printStackTrace();
26        } catch (IOException e) {
27            e.printStackTrace();
28        }
29    }
30 }};
31 ...

```

Algoritmo 7.6: Criando um alerta com os arquivos salvos

Na linha 7 do algoritmo 7.6 queremos obter a referência do arquivo selecionado. Chamando o método `getFilesDir()` nos é fornecido o diretório contendo os arquivos, com `listFiles()` temos um *array* de `Files` e o selecionamos com a variável `which`, que é a posição clicada na lista do alerta. Para ler os dados desse arquivo, usamos a classe `BufferedReader` e guardamos as linhas sendo lidas em um objeto do tipo `StringBuilder`. Nas linhas 13-16 lemos cada linha do arquivo em um loop *while* e usamos o método `append()` duas vezes, uma para adicionar a linha lida na *string* e outra para adicionar o separador de linha (comumente `\n`). Ao fim invocamos `close()` para fechar o `BufferedReader`. E colocamos o texto lido de volta na caixa de texto.

Dica: Você também pode optar por salvar seus arquivos no *cache*, isto é, na pasta *cache* da sua aplicação. Esses arquivos são apagados automaticamente quando o sistema necessita de memória. Para isso use o método `getCacheDir()` para obter o caminho do diretório de *cache*.

7.3 Armazenamento Externo

Todo dispositivo Android suporta um armazenamento externo que é compartilhado e que pode ser usado para salvar arquivos. Esse armazenamento pode ser tanto um cartão SD que esta conectado ao aparelho como a própria memória interna dele. Os arquivos salvos no armazenamento externo podem ser lidos por todos outros aplicativos e modificados quando o usuário conecta ao computador para transferir os arquivos via USB.

Se um dispositivo usa uma partição da memória interna do aparelho como armazenamento externo e ainda oferecer um cartão SD, então a partição do cartão SD não estará disponível para armazenamento, sua aplicação não conseguirá acessar o cartão SD.

Por isso, o armazenamento externo pode ficar indisponível caso o usuário monte o armazenamento externo no computador ou remova o cartão SD. Os arquivos são abertos e podem ser abertos, modificados ou removidos pelo usuário ou por outras aplicações.

Para poder usar o armazenamento externo, primeiro você deve verificar sua disponibilidade usando o método `getExternalStorageState()`. A mídia pode estar conectada a um computador, faltando ou em estado de apenas leitura. Você pode verificar a disponibilidade da mídia como mostrado no exemplo abaixo, retirado da documentação oficial².

```
1 boolean mExternalStorageAvailable = false;
2 boolean mExternalStorageWriteable = false;
3 String state = Environment.getExternalStorageState();
4
5 if (Environment.MEDIA_MOUNTED.equals(state)) {
6     mExternalStorageAvailable = mExternalStorageWriteable = true;
7 } else if (Environment.MEDIA_MOUNTED_READ_ONLY.equals(state)) {
8     mExternalStorageAvailable = true;
9     mExternalStorageWriteable = false;
10 } else {
11     mExternalStorageAvailable = mExternalStorageWriteable = false;
12 }
```

Algoritmo 7.7: Verificando se o armazenamento externo está disponível

O algoritmo 7.7 verifica se o armazenamento externo está disponível. Temos duas variáveis de controle que guardarão o estado da mídia, `mExternalStorageAvailable` e `mExternalStorageWriteable`. A primeira nos diz se o armazenamento externo está disponível e a segunda se é possível escrever nele. Na variável `state` é guardado o estado que é obtido através do método `getExternalStorageState()`. Uma comparação é feita com o resultado obtido, se for equivalente à `Environment.MEDIA_MOUNTED` então sabemos que a mídia está montada e é possível escrever nela, então setamos os estados das variáveis de controle para `true`. Caso o resultado seja equivalente à `Environment.MEDIA_MOUNTED_READ_ONLY` então sabemos que a mídia está montada porém está somente com permissão de leitura, nesse caso setamos apenas a variável de disponibilidade como `true` e mantemos a outra como `false`. Se o resultado não for esses dois estados, mas sim algum outro estado possível, então setamos as duas variáveis para `false`.

Após verificar a disponibilidade, você deve usar o método `getExternalFilesDir()` para obter uma referência ao caminho do diretório onde você deve salvar seus arquivos. Esse

²<http://developer.android.com/guide/topics/data/data-storage.html#filesExternal>

método recebe um parâmetro que especifica o tipo de subdiretório que você quer usar, tais como `DIRECTORY_MUSIC` e `DIRECTORY_RINGTONES`, ou passe `null` para obter a raiz dos diretórios da sua aplicação. O método irá criar o diretório apropriado se necessário. Ao especificar o tipo, você se assegura que o Android irá categorizar seus arquivos de forma apropriada. Se o usuário desinstalar seu aplicativo, todos os dados serão deletados.

Se você quer salvar arquivos que não são específicos da sua aplicação e que não devem ser deletados quando desinstalado, salve-os em um dos diretórios públicos que estão no raiz da mídia de armazenamento externo. São eles os diretórios `Music/`, `Pictures/`, etc. Use o método `getExternalStoragePublicDirectory()` passando o tipo de diretório desejado, da mesma forma que anteriormente.

O sistema classifica os arquivos encontrados nessas pastas na forma:

- `Music/` - Músicas do usuário;
- `Podcasts/` - *podcasts*;
- `Ringtones/` - Toques;
- `Alarms/` - Toques do despertador;
- `Notifications/` - Toques das notificações;
- `Pictures/` - Fotos (exceto aquelas tiradas com a câmera);
- `Movies/` - Filmes (exceto aquelas gravadas com a câmera); e
- `Download/` - Arquivos baixados.

7.4 Banco de dados

O Android fornece suporte ao bancos de dados `SQLite`. Qualquer banco que você criar será acessível apenas pela sua aplicação e não fora dela.

A maneira recomendada de criar um banco de dados é criando uma subclasse da classe `SQLiteOpenHelper` e sobrescrever o método `onCreate()`. Para ler e escrever no banco, chame os métodos `getReadableDatabase()` e `getWritableDatabase()`. Ambas retornam um objeto do tipo `SQLiteDatabase` que fornece métodos para operar sobre o banco.

Usando o método `SQLiteDatabase.query()` podemos realizar uma consulta, o método aceita vários parâmetros, tais como: tabela, seleção, colunas, agrupamento e etc. Para consultas mais complexas você pode usar a classe `SQLiteQueryBuilder` que contém vários métodos para construir consultas.

Toda consulta vai retornar um `Cursor` que aponta para as tuplas do resultado da consulta. Você deverá usá-lo para navegar através dos resultados.

Para exemplificar, vamos fazer um mecanismo simples de busca. O banco será populado com fabricantes de carros e alguns de seus modelos. Para o *design* desse exemplo, optei por ter dois *Spinners* (equivalente ao *Combo Box*) e um botão. Os *Spinners* serão populados com os dados do Banco de Dados de forma que o primeiro contenha todos os fabricantes de carros e o segundo todos os anos de fabricação de seus modelos, esses dados serão obtidos através de consultas ao banco.

Para começar, devemos criar o BD e isso é feito com uma classe que iremos criar que irá ser subclasse de `SQLiteOpenHelper`³. Chamaremos de `CarOpenHelper`. Essa classe tem algumas funções básicas, como criar e popular o BD, configurar a ação a ser tomada quando o BD é atualizado ou desatualizado.

³<http://developer.android.com/reference/android/database/sqlite/SQLiteOpenHelper.html>

O código 7.8 abaixo demonstra a criação do BD usado no exemplo:

```

1 public class CarOpenHelper extends SQLiteOpenHelper {
2     private static final int DATABASE_VERSION = 1;
3     private static final String DATABASE_NAME = "MyCars";
4     private static final String DATABASE_TABLE_NAME = "Cars";
5     private static final String DATABASE_TABLE_CREATE =
6         "CREATE TABLE " + DATABASE_TABLE_NAME + " (" +
7         "SN INT PRIMARY KEY, " +
8         "Manufacturer TEXT, " +
9         "Model TEXT, " +
10        "Year INT);";
11
12    public CarOpenHelper(Context context) {
13        super(context, DATABASE_NAME, null, DATABASE_VERSION);
14    }
15
16    @Override
17    public void onCreate(SQLiteDatabase db) {
18        db.execSQL(DATABASE_TABLE_CREATE);
19
20        db.execSQL( "INSERT INTO " + DATABASE_TABLE_NAME +
21            " SELECT '1' AS SN, 'Mazda' AS Manufacturer, " +
22            " 'MX-5' AS Model, '1991' AS Year" +
23            " UNION SELECT '2', 'Mazda', 'RX-8', '2001'" +
24            " UNION SELECT '3', 'Mazda', 'Speed3', '2007'" +
25            " UNION SELECT '4', 'Subaru', 'Impreza', '2010'" +
26            " UNION SELECT '5', 'Fiat', '500', '2012'" +
27            " UNION SELECT '6', 'Ford', 'Focus', '2008'" +
28            " UNION SELECT '7', 'Fiat', 'Punto', '2012'" +
29            " UNION SELECT '8', 'Ford', 'Fiesta', '2006'" +
30            " UNION SELECT '9', 'Honda', 'Civic', '2013'" +
31            " UNION SELECT '10', 'Honda', 'Fit', '2010'" +
32            " UNION SELECT '11', 'Toyota', 'Corolla', '2010'" +
33            " UNION SELECT '12', 'Chevrolet', 'Celta', '2009'" +
34            " UNION SELECT '13', 'Chevrolet', 'Cruze', '2012'");
35
36    }
37 }

```

Algoritmo 7.8: Classe CarOpenHelper do SQLite

Primeiro criamos algumas *Strings* para ajudar na criação do banco. Em seguida, criamos o construtor da classe, os argumentos passados são: o contexto da *activity* que instanciou a classe, o nome do banco, um *CursorFactory* caso esteja usando (não é o caso), e a versão do banco que começa em 1 e deve incrementar a medida que você o atualiza.

O método `onCreate()` do banco é responsável pela criação do banco e de o popular. Fazemos isso executando consultas SQL diretamente ao banco com o método `execSQL()`. No código 7.8 você pode observar a chamada de `execSQL()` duas vezes. Não se confunda

com a segunda consulta, que adiciona as tuplas ao banco, é só uma maneira de inserir várias tuplas em uma única consulta.

Agora na nossa *activity* precisamos realizar as consultas para popular os *Spinners*. No método `onCreate()` faça:

```

1 @Override
2 protected void onCreate(Bundle savedInstanceState) {
3     ...
4
5     mDatabase = new CarOpenHelper(this);
6     String[] mColumns = {"Manufacturer", "Year"};
7
8     SQLiteDatabase mSQLite = mDatabase.getReadableDatabase();
9     Cursor cursor =
10         mSQLite.query("Cars", mColumns, null, null, null, null, null);
11
12     Set<String> set1 = new HashSet<String>();
13     Set<String> set2 = new HashSet<String>();
14     while(cursor.moveToNext()) {
15         set1.add(cursor.getString(0));
16         set2.add(cursor.getString(1));
17     }
18     List<String> list1 = new ArrayList<String>(set1);
19     List<String> list2 = new ArrayList<String>(set2);
20
21     ArrayAdapter<String> manuAdapter =
22         new ArrayAdapter<String>
23         (this, android.R.layout.simple_spinner_item, list1);
24     manuAdapter.
25         setDropDownViewResource
26         (android.R.layout.simple_spinner_dropdown_item);
27     mSpManufacturer.setAdapter(manuAdapter);
28
29     ArrayAdapter<String> yearAdapter =
30         new ArrayAdapter<String>
31         (this, android.R.layout.simple_spinner_item, list2);
32     manuAdapter.
33         setDropDownViewResource
34         (android.R.layout.simple_spinner_dropdown_item);
35     mSpYear.setAdapter(yearAdapter);

```

Algoritmo 7.9: Usando o `CarOpenHelper` na *activity*

Estou "pulando" a parte de obter as referências dos *spinners* e botões e partindo para o que interessa no código 7.9. Na linha 5 instanciamos o `CarOpenHelper` na variável `mDatabase`. Na linha 6 criamos um *array* que contém as colunas que iremos fazer consulta, isto é necessário para o método `query()` na linha 10.

Ao chamar `mDatabase.getReadableDatabase()` obtemos um objeto do tipo `SQ-`

LiteDatabase⁴ mas que só permite a leitura. Já o método `query()` retorna um `Cursor` que é usado para iterar sobre os resultados. Na linha 10 observe os parâmetros que usamos para fazer a consulta: A tabela que vamos fazer a consulta, um *array* de *strings* com as colunas que queremos obter, uma cláusula `WHERE` (observe que passar **null** aqui implica em retornar todas as tuplas da coluna), argumentos da seleção (para aqueles que conhecem *prepared statements* onde "?" é substituído pelos valores, uma cláusula `GROUP BY` (**null** significa não agrupar), uma cláusula `HAVING`, uma cláusula `ORDER BY` e uma cláusula `LIMIT`.

Em seguida criamos dois `Set` para armazenar os resultados e omitir os repetidos. Usamos o `Cursor` para iterar sobre o resultado e adicioná-los aos conjuntos. O `set1` contém os fabricantes e o `set2` contém os anos de fabricação. Usamos `Cursor.getString()` para obter o elemento do resultado que está naquele índice, nesse caso o índice 0 é a coluna de fabricantes e o índice 1 é a coluna de anos de fabricação. Com esses dois conjuntos em mão podemos criar duas `ArrayList` para serem usadas com os `ArrayAdapter` que irão popular os *Spinners*.

Depois é necessário criar dois *listeners* para os elementos dos *Spinners* que guarda numa variável o fabricante e o ano de fabricação selecionados (aqui irei omitir código, mas pode ser obtido no repositório). E um *listener* para o botão que irá chamar o método `openCarList()`.

```
1 public void openCarList() {
2     Intent intent = new Intent(this, CarDetailActivity.class);
3     List<String> list = fetchCarList();
4     String[] carList = list.toArray(new String[list.size()]);
5     intent.putExtra(CARLIST, carList);
6     startActivity(intent);
7 }
```

Algoritmo 7.10: Método `openCarList()`

Esse método é responsável por obter a lista de carros da seleção feita nos *Spinners* e mandar essa lista para outra *activity*. O método `fetchCarList()` é que irá fazer a chamada ao BD, da mesma forma que foi feito anteriormente, mas dessa vez estamos obtendo a coluna *Model* do banco com `WHERE` sendo os parâmetros obtidos.

⁴<http://developer.android.com/reference/android/database/sqlite/SQLiteDatabase.html>

```
1 public List<String> fetchCarList() {
2     SQLiteDatabase mSQLite = mDatabase.getReadableDatabase();
3     String[] column = {"Model"};
4     Cursor cursor =
5         mSQLite.query("Cars", column,
6             "Year='"+selYear+"' AND Manufacturer='"+selManufacturer + "'",
7             null, null, null, null, null);
8
9     List<String> list1 = new ArrayList<String>();
10    while(cursor.moveToNext()) {
11        list1.add(cursor.getString(0));
12    }
13
14    return list1;
15 }
```

Algoritmo 7.11: Método `fetchCarList()`

O resultado de `fetchCarList()` é enviado para outra *activity* que irá mostrar os modelos em forma de uma lista.

Nesse capítulo pudemos aprender apenas o básico de acesso ao banco de dados SQLite, é um assunto que tem muito mais o que aprender e recomendo olhar a documentação para futuras consultas.

Câmera

O Android fornece um *framework* para acesso às diferentes câmeras e funcionalidades dessas câmeras disponíveis no dispositivo. É possível tirar fotos e gravar vídeos nos aplicativos. Há duas formas de acessar a câmera, a primeira sendo pela API e a segunda pelo `Intent` da câmera. Nesse capítulo iremos discutir brevemente sobre como usar a API e um exemplo de chamar o aplicativo padrão da câmera através de um `Intent`.

8.1 Usando a API

Para usar a API da câmera, primeiro devemos requisitar a permissão para usar a câmera no *Manifest*.

```
1 <uses-permission android:name="android.permission.CAMERA" />
2
3 <uses-feature android:name="android.hardware.camera" />
```

Algoritmo 8.1: Requisitando permissão para usar a câmera

Você pode ainda requisitar outras funcionalidades como *flash*, foco automático, usar a câmera da frente ao requisitar outras *features* usando `<uses-feature>`. A lista das funcionalidades está presente na [documentação](#)¹.

Se você for guardar as fotos no armazenamento externo, precisa requisitar a permissão para escrita:

```
1 <uses-permission android:name=
2   "android.permission.WRITE_EXTERNAL_STORAGE" />
```

Algoritmo 8.2: Requisitando permissão para gravar no armazenamento externo

¹<http://developer.android.com/guide/topics/manifest/uses-feature-element.html#hw-features>

Se for gravar áudio quando estiver gravando vídeo, precisa requisitar a permissão para gravar áudio:

```
1 <uses-permission android:name="android.permission.RECORD_AUDIO" />
```

Algoritmo 8.3: Requisitando permissão para gravar áudio

Segundo a [documentação](#)² existem alguns passos gerais que são seguidos para criar um aplicação usando a API da câmera. São eles:

- **Detectar e acessar a câmera:** Código que verifica a existência de uma câmera e requisita o acesso.
- **Criar uma classe de pré-visualização:** Criar uma classe que estende `SurfaceView` e implementa `SurfaceHolder` para mostrar na tela o que a câmera está vendo.
- **Construir o *layout* da pré-visualização:** Com a classe pronta, crie uma *view* que irá incorporar a classe de pré-visualização e mostrar os controles da interface.
- **Configurar os *listeners* para a captura:** Criar *listeners* para os botões da interface para gravar a imagem.
- **Capturar e salvar os arquivos:** Criar o código que irá capturar a imagem e salvar no armazenamento.
- **Liberar a câmera:** Depois de usar a câmera, a aplicação deve liberar o uso para outras aplicações.

8.1.1 Acessando a câmera

Primeiro vamos criar uma classe `CameraAccess` que irá requisitar o acesso à câmera. A classe ficará responsável apenas por verificar se o dispositivo tem câmera e por instanciar a câmera para o uso.

²<http://developer.android.com/guide/topics/media/camera.html>

```
1 public class CameraAccess {
2     Context c;
3     Camera cam;
4
5     public CameraAccess(Context c) {
6         this.c = c;
7         cam = null;
8     }
9
10    private boolean checkCameraHardware() {
11        if(c.getPackageManager().
12            hasSystemFeature(PackageManager.FEATURE_CAMERA))
13            return true;
14        return false;
15    }
16
17    public Camera getCameraInstance() {
18        Camera cam = null;
19        if(checkCameraHardware()) {
20            try{
21                cam = Camera.open();
22            } catch (Exception e) {
23                e.printStackTrace();
24            }
25        } else {
26            return null;
27        }
28
29        return cam;
30    }
31
32    public void releaseCamera() {
33        cam.release();
34    }
35 }
```

Algoritmo 8.4: Classe CameraAccess

8.1.2 Pré-visualização

Em seguida iremos criar a classe da visualização da câmera. Essa classe estende `SurfaceView` e implementa a interface `SurfaceHolder.Callback`, iremos chamar essa classe de `CameraPreview`. No código 8.5 abaixo começamos a implementar essa classe. Precisamos de duas variáveis `SurfaceHolder` e `Camera`. O construtor recebe o contexto e a câmera previamente instanciada como parâmetro. Obtemos o `SurfaceHolder` usando o método `getHolder()`, adicionamos a classe como *callback*. Por último, na linha 11 configuramos o tipo do `SurfaceHolder` como `SURFACE_TYPE_PUSH_BUFFERS` isso já é deprecado mas é necessário para acessar a câmera em versões do Android mais antigas que 3.0.

```

1 public class CameraPreview extends SurfaceView implements Callback {
2     private SurfaceHolder mHolder;
3     private Camera mCamera;
4
5     public CameraPreview(Context c, Camera cam) {
6         super(c);
7         mCamera = cam;
8
9         mHolder = getHolder();
10        mHolder.addCallback(this);
11        mHolder.setType(SurfaceHolder.SURFACE_TYPE_PUSH_BUFFERS);
12    }
13
14    @Override
15    public void surfaceCreated(SurfaceHolder holder) {
16        try{
17            mCamera.setPreviewDisplay(holder);
18            mCamera.startPreview();
19        } catch (IOException e) {
20            Log.d("CAM", "Error setting camera preview: " + e.getMessage());
21        }
22    }
23
24    @Override
25    public void surfaceChanged(SurfaceHolder holder, int format, int width,
26        int height) {
27        if(mHolder.getSurface() == null){
28            return;
29        }
30
31        try {
32            mCamera.stopPreview();
33        } catch (Exception e) {
34            // Irrelevante
35        }
36
37        try{
38            mCamera.setPreviewDisplay(mHolder);
39            mCamera.startPreview();
40        } catch (Exception e) {
41            Log.d("CAM", "Error setting camera preview: " + e.getMessage());
42        }
43    }
44
45    @Override
46    public void surfaceDestroyed(SurfaceHolder holder) {
47        // Nada, tomar conta de liberar a camera na activity
48    }
49 }

```

Continuando a analisar o algoritmo 8.5. Precisamos sobrecarregar os métodos da interface. Esses são: `surfaceCreated()`, `surfaceChanged()` e `surfaceDestroyed()`. O primeiro, `surfaceCreated()` é responsável por instanciar a pré-visualização. Para isso chame o método `setPreviewDisplay()` passando o `holder` como parâmetro para a câmera saber onde a visualização será desenhada. Depois é só chamar `startPreview()` para iniciar a visualização.

O segundo, `surfaceChanged()` serve para parar e recomeçar a visualização quando o usuário rotaciona a tela, por exemplo. Para isso, é necessário fechar a visualização e depois iniciar novamente. Na linha 27 testamos para saber se o `mHolder` não tem nada desenhado, se for o caso então apenas retorne do método pois não há nada para ser fechado. Caso contrário então precisamos parar a visualização com `stopPreview()` e depois iniciar novamente da mesma forma como é feito no método `surfaceCreated()`.

O terceiro, `surfaceDestroyed()` é irrelevante nesse caso pois estaremos tomando conta de liberar a câmera na *activity* e não nessa classe.

8.1.3 Layout e Activity da visualização

A classe da visualização precisa ser instanciada por uma *activity* que vai efetivamente mostrar o conteúdo. Para o *layout* da *activity* iremos criar um `FrameLayout` que irá conter a visualização e um botão pra capturar a imagem.

```

1 <?xml version="1.0" encoding="utf-8"?>
2 <LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
3     android:orientation="horizontal"
4     android:layout_width="fill_parent"
5     android:layout_height="fill_parent" >
6
7     <FrameLayout
8         android:id="@+id/camera_preview"
9         android:layout_width="fill_parent"
10        android:layout_height="fill_parent"
11        android:layout_weight="1" />
12
13    <Button
14        android:id="@+id/button_capture"
15        android:text="Capture"
16        android:layout_width="wrap_content"
17        android:layout_height="wrap_content"
18        android:layout_gravity="center" />
19 </LinearLayout>

```

Algoritmo 8.6: *Layout* da *Activity* que irá conter a visualização

Você deve também especificar a orientação da *activity* como *landscape* no *Manifest*. Como mostrado na linha 5 do código 8.7. Apesar disso não ser obrigatório, é normal que tiremos fotos no modo paisagem e não retrato, mas isso não é regra.

```
1 ...
2 <activity
3   android:name="com.example.cameral.MainActivity"
4   android:label="@string/app_name"
5   android:screenOrientation="landscape" >
6   ...
```

Algoritmo 8.7: Configurando a orientação da *activity* no *Manifest*

8.1.4 Criando a *activity*

Para começar vamos apenas fazer com que a *activity* mostre a visualização da câmera na tela usando as classes que criamos anteriormente.

```
1 public class CameraActivity extends Activity {
2     private Camera mCam;
3     private CameraPreview mPreview;
4
5     @Override
6     protected void onCreate(Bundle savedInstanceState) {
7         super.onCreate(savedInstanceState);
8         setContentView(R.layout.activity_camera);
9
10        CameraAccess ca = new CameraAccess(this);
11        mCam = ca.getCameraInstance();
12
13        mPreview = new CameraPreview(this, mCam);
14        FrameLayout preview = (FrameLayout) findViewById(R.id.camera_preview);
15        preview.addView(mPreview);
16    }
```

Algoritmo 8.8: Primeira parte da classe *CameraActivity*

Na linha 10 criamos uma instância da classe *CameraAccess* e requisitamos o acesso a câmera usando *getCameraInstance()*. Depois criamos um *CameraPreview* e adicionamos o *preview* ao *FrameLayout*. Nesse ponto, se o aplicativo for executado irá mostrar a imagem da câmera na tela, com o botão ao lado mas que não faz nada ainda. Precisamos em seguida programar para que as imagens fiquem salvas no armazenamento externo.

8.1.5 Capturando e salvando as imagens

Agora você está pronto para capturar e salvar as fotos em sua aplicação. Para isso você deve definir *listeners* para o botão da sua interface que irá responder tirando uma foto.

Dica: Certifique-se de que há permissão para escrita no armazenamento externo para salvar as fotos.

Para capturar uma foto, você deve usar o método `Camera.takePicture()`. Para receber os dados no format JPEG você deve implementar um `Camera.PictureCallback` que recebe os dados da imagem e os escrevem em um arquivo. Observe a implementação da interface `PictureCallback` no código abaixo, que pode ser escrito no método `onCreate()` da *activity*.

```
1 PictureCallback mJPEGCallback = new PictureCallback() {
2
3     @Override
4     public void onPictureTaken(byte[] data, Camera camera) {
5         File picFile = getOutputMediaFile(MEDIA_TYPE_IMAGE);
6         if(picFile == null){
7             Log.d("ERROR",
8                 "Error creating media file, check storage permissions");
9             return;
10        }
11
12        try {
13            FileOutputStream fos = new FileOutputStream(picFile);
14            fos.write(data);
15            fos.close();
16        } catch (FileNotFoundException e) {
17            Log.d("ERROR", "File not found: " + e.getMessage());
18        } catch (IOException e){
19            Log.d("ERROR", "Error accessing file: " + e.getMessage());
20        }
21    }
22 };
```

Algoritmo 8.9: Criando um Callback para imagens JPEG

Para simplificar, todo código que efetivamente cria a imagem em um arquivo ficou no método `getOutputMediaFile()` que pode ser observado abaixo.

```

1 private File getOutputMediaFile(int type){
2     String folder =
3         getResources().getString(R.string.app_name) + "_PICS";
4
5     File mediaStorageDir =
6         new File(Environment.getExternalStoragePublicDirectory(
7             Environment.DIRECTORY_PICTURES), folder);
8
9     if (!mediaStorageDir.exists()){
10         if (!mediaStorageDir.mkdirs()){
11             Log.d("ERROR", "failed to create directory: " + folder);
12             return null;
13         }
14     }
15     String timeStamp =
16         new SimpleDateFormat("yyyyMMdd_HHmmss").format(new Date());
17
18     File mediaFile;
19     if (type == MEDIA_TYPE_IMAGE){
20         mediaFile = new File(mediaStorageDir.getPath() + File.separator +
21             "IMG_" + timeStamp + ".jpg");
22     } else if (type == MEDIA_TYPE_VIDEO) {
23         mediaFile = new File(mediaStorageDir.getPath() + File.separator +
24             "VID_" + timeStamp + ".mp4");
25     } else {
26         return null;
27     }
28
29     return mediaFile;
30 }

```

Algoritmo 8.10: Método `getOutputMediaFile()`

Na linha 2 do método `getOutputMediaFile()` estamos escolhendo o nome da pasta que armazenará as fotos salvas. Para este exemplo, escolhi o nome do aplicativo concatenado com a *string* `"_PICS"`. Como neste exemplo o nome do aplicativo é `"Camera1"`, então a pasta será `"Camera1_PICS"`. Na linha 5 criamos um objeto `File` que contém uma referência a esta pasta que estamos criando, o caminho dela é dado pelo método `Environment.getExternalStoragePublicDirectory()` e em seguida pela variável `Environment.DIRECTORY_PICTURES`. Essa chamada escolhe a pasta `Pictures/` do armazenamento externo e ao ser concatenada com a nossa variável `folder`, temos a pasta que iremos salvar as fotos. Em seguida é feita uma verificação se a pasta já existe, caso não exista ele tenta criá-la com o método `mkdirs()`, que retorna `false` caso não seja possível criar.

Para o nome do arquivo, criamos um *timestamp* que pega a data e hora atual para o nome do arquivo. Isso pode ser observado na linha 16 com a criação de um `SimpleDateFormat` no formato `yyyyMMdd_HHmmss`, ou seja: ano, mês, dia, hora, minuto e segundo, nesta ordem. Na linha 18 criamos novamente um objeto `File` mas que dessa vez irá guardar o caminho do arquivo.

Por fim, mostramos um `Toast` para dar um feedback ao usuário que a foto foi salva e

mostrando a pasta para que ele possa ver.

Agora só falta dar a função do botão de capturar, com o seguinte *listener*:

```
1 mCapture = (Button) findViewById(R.id.button_capture);
2 mCapture.setOnClickListener(new OnClickListener() {
3
4     @Override
5     public void onClick(View v) {
6         mCam.takePicture(null, null, mJpegCallback);
7     }
8 });
```

Algoritmo 8.11: Método `getOutputMediaFile()`

Basta chamar o método `takePicture()`³, que recebe três *callbacks*, são eles: `ShutterCallback`, e dois `PictureCallback`, um para imagens do tipo RAW e outra para imagens do tipo JPEG. Como só criamos um *callback*, o resto será `null`.

Entretanto, as imagens salvas estarão com uma qualidade bem inferior. Para corrigir isso você pode adicionar o seguinte código antes da linha 13 do algoritmo 8.8:

```
1 Parameters params = mCam.getParameters();
2 params.setJpegQuality(100);
3
4 List<Size> sizes = params.getSupportedPictureSizes();
5 Camera.Size size = sizes.get(0);
6 for(int i=0; i<sizes.size(); i++)
7 {
8     if(sizes.get(i).width > size.width)
9         size = sizes.get(i);
10 }
11 params.setPictureSize(size.width, size.height);
12
13 mCam.setParameters(params);
```

Algoritmo 8.12: Melhorando a qualidade das fotos tiradas

Você consegue configurar os parâmetros da câmera, neste caso usaremos para melhorar a qualidade da foto. Para isso comece chamando `getParameters()` e guarde essa referência num objeto `Parameters`. Com `setJpegQuality()` conseguimos configurar a qualidade da compressão JPEG entre 0 e 100, sendo 100 o melhor. Em seguida iteraremos sob uma lista de resoluções suportadas e escolhemos a maior resolução. Dessa forma a foto terá a maior resolução e com a melhor qualidade de compressão. Para finalizar, chame `setParameters()` e os parâmetros serão gravados.

³<http://developer.android.com/intl/es/reference/android/hardware/Camera.html>

8.2 Gravando vídeos

Agora você já tem um aplicativo que tira fotos, mas também quer gravar vídeos. Para isso, algumas modificações serão necessárias, assim como algumas adições. Para capturar vídeos, é necessário um controle cauteloso do objeto `Camera` em coordenação com a classe `MediaRecorder`⁴. Diferente de tirar fotos, gravar vídeos requer chamadas à métodos em uma ordem particular. Você deve seguir essa ordem para preparar a aplicação e capturar o vídeo.

Os três primeiros passos são: 1) Abrir a câmera, 2) Criar uma visualização e 3) Visualizar a câmera, já foram feitos na seção anterior. Para começar a gravar o vídeo são necessários mais alguns passos:

8.2.1 Preparar a gravação

Preparar a gravação significa configurar a classe `MediaRecorder`, todos os passos da configuração devem ser feitos em uma **ordem específica**. Para preparar a gravação criaremos um método `prepareForRecording()`.

```
1 private boolean prepareForRecording() {
2     recorder = new MediaRecorder();
3     mCam.unlock();
4     recorder.setCamera(mCam);
5
6     recorder.setAudioSource(MediaRecorder.AudioSource.CAMCORDER);
7     recorder.setVideoSource(MediaRecorder.VideoSource.CAMERA);
8
9     CamcorderProfile profile =
10         CamcorderProfile.get(CamcorderProfile.QUALITY_HIGH);
11     recorder.setProfile(profile);
12
13     recorder.setOutputFile(getOutputMediaFile(MEDIA_TYPE_VIDEO).toString());
14     recorder.setPreviewDisplay(mPreview.getHolder().getSurface());
15
16     try{
17         recorder.prepare();
18     } catch (IllegalStateException e){
19         releaseMediaRecorder();
20         return false;
21     } catch (IOException e) {
22         releaseMediaRecorder();
23         return false;
24     }
25
26     return true;
27 }
```

Algoritmo 8.13: Método `prepareForRecording()`

Temos uma variável `recorder` da classe `MediaRecorder` e o instanciamos. O Primeiro passo é destravar a câmera com `unlock()` e atribuir a câmera ao `recorder` com `setCa-`

⁴<http://developer.android.com/intl/es/reference/android/media/MediaRecorder.html>

mera(). O segundo passo é configurar a fonte de áudio e vídeo com `setAudioSource()` e `setVideoSource()`, respectivamente, como é feito nas linhas 6 e 7. O terceiro passo é configurar um *profile* com o `setProfile()`, você pode obter um *profile* usando o método `CamcorderProfile.get()`. O quarto passo é configurar o arquivo de saída com o método `setOutputFile()`, aqui usaremos o método `getOutputMediaFile()` que fizemos antes. O quinto passo é configurar a visualização do recorder com `setPreviewDisplay()`, como é feito na linha 14 e passamos como parâmetro o método `getSurface()` da nossa classe `CameraPreview`. Por último devemos tentar chamar o método `prepare()`, mas caso dê errado precisamos liberar a câmera de volta, para isso usaremos o método `releaseMediaRecorder()`, que iremos criar abaixo.

```

1 private void releaseMediaRecorder() {
2     if(recorder != null) {
3         recorder.reset();
4         recorder.release();
5         recorder = null;
6         mCam.lock();
7     }
8 }

```

Algoritmo 8.14: Método `releaseMediaRecorder()`

O código de `releaseMediaRecorder()` é simples, são os passos necessários para liberar a câmera do recorder e dá-la de volta à *activity*. O método `reset()` limpa as configurações do recorder.

É importante também liberar a câmera para outras aplicações caso o usuário troque de aplicação, quando isso acontece o método `onPause()` é chamado no ciclo de vida da *activity*.

```

1 @Override
2 protected void onPause() {
3     super.onPause();
4     releaseMediaRecorder();
5     if(mCam != null) {
6         mCam.release();
7         mCam = null;
8     }
9 }

```

Algoritmo 8.15: Liberando a câmera no método `onPause()`

Por último, configurar um *listener* para o botão de gravar vídeo, como mostrado no algoritmo 8.16 abaixo.


```
1 mRecord = (Button) findViewById(R.id.button_record);
2 mRecord.setOnClickListener(new OnClickListener() {
3
4     @Override
5     public void onClick(View v) {
6         if(isRecording){
7             recorder.stop();
8             releaseMediaRecorder();
9             ((Button) v).setText("Record");
10            isRecording = false;
11        } else {
12            if(prepareForRecording()) {
13                recorder.start();
14                ((Button) v).setText("Stop");
15                isRecording = true;
16            } else {
17                releaseMediaRecorder();
18            }
19        }
20    }
21 });
```

Algoritmo 8.16: Configurando o *listener* do botão de gravar vídeo

Com uma variável de controle `isRecording` guardamos se a câmera está gravando ou não. Caso esteja gravando e o usuário clique no botão chamamos `stop()`, liberamos o `recorder` com `releaseMediaRecorder()`, mudamos o texto do botão, além de marcar `isRecording` como falso, pois paramos de gravar. Caso contrário, isto é, o usuário quer começar a gravar então chamamos `prepareForRecording()` que criamos anteriormente, se o resultado for positivo significa que todas as preparações deram certo, podemos chamar `start()` e colocar "Stop" no botão onde antes era "Record", além de marcar `isRecording` como verdadeiro. Se a preparação falhar, então apenas chamamos `releaseMediaRecorder()` para cancelar qualquer preparação que tenha sido feita.

Dica: Porque a complexidade de configurar a gravação de vídeo é alta, é recomendável baixar o projeto `Camera1` do repositório e estudar o código completo.

8.3 Usando um Intent

A maneira mais fácil de tirar uma simples foto na verdade é chamando um `Intent` que se encarregará de abrir a *activity* da câmera e retornar a foto tirada.

8.3.1 Intent de capturar foto

O `Intent` da câmera pode receber opcionalmente um objeto `Uri` que especifica o caminho e o nome do arquivo onde você gostaria que as fotos fossem salvas. Apesar de ser opcional, esse parâmetro é recomendável. Caso não seja especificado, a aplicação da câmera irá salvar a foto no local padrão com um nome padrão que pode ser visto no campo `Intent.getData()` que é retornado.

O código abaixo mostra como executar um Intent para capturar fotos.

```

1 private static final int CAPTURE_IMAGE_ACTIVITY_REQUEST_CODE = 100;
2 private static final int MEDIA_TYPE_IMAGE = 1;
3 private static final int MEDIA_TYPE_VIDEO = 2;
4 private Uri fileUri;
5
6 @Override
7 public void onCreate(Bundle savedInstanceState) {
8     ...
9
10    Intent intent = new Intent(MediaStore.ACTION_IMAGE_CAPTURE);
11
12    fileUri = getOutputMediaFileUri(MEDIA_TYPE_VIDEO);
13    intent.putExtra(MediaStore.EXTRA_OUTPUT, fileUri);
14
15    startActivityForResult(intent, CAPTURE_IMAGE_ACTIVITY_REQUEST_CODE);
16 }

```

Algoritmo 8.17: Chamando a *activity* de câmera com Intent

Primeiro cria-se um novo Intent com parâmetro `MediaStore.ACTION_IMAGE_CAPTURE` que configura uma *activity* com câmera. Depois usamos o método `getOutputMediaFileUri()` que apenas faz uma chamada ao método `getOutputMediaFile()` (algoritmo 8.10) que usamos na seção anterior porém retornando uma `Uri`, que é o caminho do arquivo. Colocamos como extra no Intent o caminho retornado em `fileUri`, nesse caso `MediaStore.EXTRA_OUTPUT` é apenas uma convenção para ficar claro que o valor passado no extra é para ser usado na *activity* de câmera para salvar o arquivo. Por fim chamamos `startActivityForResult()`, que recebe o Intent e um código que indica uma *activity* que captura imagens.

```

1 private Uri getOutputMediaFileUri(int type) {
2     return Uri.fromFile(getOutputMediaFile(type));
3 }

```

Algoritmo 8.18: Método `getOutputMediaFileUri()`

8.3.2 Recebendo resultado do Intent

Quando você chama `startActivityForResult()` significa que você espera uma resposta da *activity* que chamou. Essa resposta será obtida no método `onActivityResult()` que você precisa escrever para manipular esses dados da resposta. O algoritmo 8.19 exemplifica esse método, que trata a resposta da câmera, isto é, se foi possível capturar a foto ou gravar o vídeo.

```
1 private static final int CAPTURE_IMAGE_ACTIVITY_REQUEST_CODE = 100;
2 private static final int CAPTURE_VIDEO_ACTIVITY_REQUEST_CODE = 200;
3
4 @Override
5 protected void onActivityResult
6     (int requestCode, int resultCode, Intent data) {
7
8     if (requestCode == CAPTURE_IMAGE_ACTIVITY_REQUEST_CODE) {
9         if (resultCode == RESULT_OK) {
10             Toast.makeText(this, "Image saved to:\n" +
11                 uri.toString(), Toast.LENGTH_LONG).show();
12         } else if (resultCode == RESULT_CANCELED) {
13             // Opcional
14         } else {
15             Toast.makeText(this, "Error capturing image...",
16                 Toast.LENGTH_LONG).show();
17         }
18     }
19
20     if (requestCode == CAPTURE_VIDEO_ACTIVITY_REQUEST_CODE) {
21         if (resultCode == RESULT_OK) {
22             Toast.makeText(this, "Video saved to:\n" +
23                 uri.toString(), Toast.LENGTH_LONG).show();
24         } else if (resultCode == RESULT_CANCELED) {
25             // Opcional
26         } else {
27             Toast.makeText(this, "Error recording video...",
28                 Toast.LENGTH_LONG).show();
29         }
30     }
31 }
```

Algoritmo 8.19: Método `onActivityResult()`

A lógica está apenas em verificar o retorno da *activity* e mostrar uma mensagem ao usuário, o informando do caminho em que foi salvo a imagem ou vídeo.

Dica: Quando você passa uma `Uri` para a *activity* de câmera, o retorno dela na variável `data` será sempre `null`. Você pode, entretanto, não especificar uma `Uri` e usar o campo `data` para criar um `Bitmap`.

8.3.3 Intent de capturar vídeo

Para requisitar uma *activity* que grava vídeo, o processo é semelhante embora você possa passar um extra adicional ao `Intent` que informa a qualidade do vídeo.

```
1 @Override
2 public void onCreate(Bundle savedInstanceState) {
3     ...
4
5     Intent intent = new Intent(MediaStore.ACTION_VIDEO_CAPTURE);
6
7     fileUri = getOutputMediaFileUri(MEDIA_TYPE_VIDEO);
8     intent.putExtra(MediaStore.EXTRA_OUTPUT, fileUri);
9
10    intent.putExtra(MediaStore.EXTRA_VIDEO_QUALITY, 1);
11
12    startActivityForResult(intent, CAPTURE_VIDEO_ACTIVITY_REQUEST_CODE);
13 }
```

Algoritmo 8.20: Criando um Intent para vídeo

Isso conclui o capítulo de câmera, passamos por todas as fases da criação de um aplicativo que usa a câmera como principal elemento. Também vimos como é simples utilizar a câmera como uma *activity* secundária cuja função é apenas capturar uma imagem. Sempre que existirem dúvidas, faça o *download* dos projetos do repositório para ver o código completo em funcionamento.

Áudio

9.1 Gravando e tocando áudio

Gravar áudio é uma tarefa muito semelhante a gravar vídeos no Android. Usaremos a mesma classe: `MediaRecorder`, para configurar e gravar áudio do microfone do aparelho. Para tocar áudio veremos uma nova classe chamada `MediaPlayer`.

Para este exemplo criaremos um aplicativo simples com 2 botões, um para gravar e um para tocar a gravação. A fim de simplificar, toda gravação será feita em apenas um arquivo, logo sempre que gravar algo novo estará sobrescrevendo a gravação antiga. Entretanto, você pode modificar o método `getOutputMediaFile()` (algoritmo 8.10) que criamos no capítulo anterior para gravar em diferentes arquivos.

Iniciaremos criando uma função que chamaremos de `prepareRecording()`, similar aquela que fizemos para no capítulo de vídeo.

```
1 private boolean prepareRecording() {  
2     recorder = new MediaRecorder();  
3     recorder.setAudioSource(MediaRecorder.AudioSource.MIC);  
4     recorder.setOutputFormat(MediaRecorder.OutputFormat.THREE_GPP);  
5     recorder.setOutputFile(testFilename);  
6     recorder.setAudioEncoder(MediaRecorder.AudioEncoder.AAC);  
7  
8     try{  
9         recorder.prepare();  
10    } catch (IOException e) {  
11        Log.e("AudioRecorder", "prepareRecording() failed");  
12        return false;  
13    }  
14    return true;  
15 }
```

Algoritmo 9.1: Método `prepareRecording()` para gravações de áudio

Assim como fizemos no capítulo anterior, a preparação precisa seguir passos determinados. Primeiro precisamos configurar a fonte com `setAudioSource()`, nesse caso passamos o MIC, o microfone do aparelho. O segundo passo é configurar o formato da saída com `setOutputFormat()`, nesse exemplo optei pelo formato 3gp. O terceiro passo é determinar o arquivo de saída com `setOutputFile()` e para esse método passamos uma *string* que chamei de `testFilename`, o valor de `testFilename` neste exemplo é o caminho do armazenamento externo mais o nome do arquivo. Depois, `setAudioEncoder()` para selecionar a codificação do áudio. Por último chame o método `prepare()` para consolidar as configurações.

Também faça uma função `releaseRecorder()` para liberar o gravador quando terminar a gravação.

```
1 private void releaseRecorder() {  
2     recorder.release();  
3     recorder = null;  
4 }
```

Algoritmo 9.2: Método `releaseRecorder()`

Em seguida faça o *listener* do botão de gravar da mesma forma do botão de gravar vídeo do capítulo anterior.

```
1 mRecordButton.setOnClickListener(new OnClickListener() {  
2  
3     @Override  
4     public void onClick(View v) {  
5         if(isRecording){  
6             recorder.stop();  
7             releaseRecorder();  
8             mRecordButton.setText("Record");  
9             isRecording = false;  
10        } else {  
11            if(prepareRecording()){  
12                recorder.start();  
13                mRecordButton.setText("Stop");  
14                isRecording = true;  
15            } else {  
16                releaseRecorder();  
17            }  
18        }  
19    }  
20 });
```

Algoritmo 9.3: Configurando o botão de gravar áudio

São necessárias apenas verificações simples para saber se está gravando ou não e se foi possível fazer a preparação do gravador ou não. Se está gravando e o usuário pressiona o

botão então chamamos `stop()` para parar a gravação, trocamos o texto do botão e atribuímos `isRecording` como falso, pois estamos parando de gravar. Caso contrário, ou seja, queremos gravar, então chamamos `start()` para iniciar a gravação, então mudamos o texto e atribuímos verdadeiro para `isRecording`. Se a preparação falhar chamamos `releaseRecorder()` para liberar o microfone.

Para o tocador, iremos usar a classe `MediaPlayer`, iniciaremos criando um método `startPlaying()` responsável por instanciar um `MediaPlayer` e configurá-lo.

```
1 private void startPlaying() {
2     player = new MediaPlayer();
3
4     player.setOnCompletionListener(new OnCompletionListener() {
5
6         @Override
7         public void onCompletion(MediaPlayer mp) {
8             mPlayButton.setText("Play");
9             stopPlaying();
10            isPlaying = false;
11        }
12    });
13
14    try{
15        player.setDataSource(testFilename);
16        player.prepare();
17        player.start();
18    } catch (IOException e) {
19        Log.e("AudioRecorder", "file not found");
20    }
21 }
```

Algoritmo 9.4: Método `startPlaying()`

Na linha 4 você pode observar o uso de um *listener* para o `MediaPlayer` chamado `OnCompletionListener`, podemos usar essa interface para saber quando o áudio termina de tocar, nesse caso o estamos usando para poder configurar o botão. Dentro do bloco `try-catch` usamos `setDataSource()` para configurar o arquivo de áudio a ser tocado. Depois chamamos `prepare()` e finalmente `start()` para começar a tocar.

Para complementar, a função `stopPlaying()` também é útil para quando quisermos parar de tocar o áudio.

```
1 private void stopPlaying() {
2     player.stop();
3     player.release();
4     player = null;
5 }
```

Algoritmo 9.5: Método `stopPlaying()`

Por fim, basta configurar o *listener* do botão de tocar áudio, como mostrado no algoritmo abaixo:

```
1 mPlayButton.setOnClickListener(new OnClickListener() {
2
3     @Override
4     public void onClick(View v) {
5         if(isPlaying){
6             stopPlaying();
7             isPlaying = false;
8             mPlayButton.setText("Play");
9         } else {
10            startPlaying();
11            isPlaying = true;
12            mPlayButton.setText("Stop");
13        }
14    }
15 });
```

Algoritmo 9.6: Configurando o botão de tocar áudio

A lógica desse método é praticamente o mesmo do botão de gravar áudio.

Para testar, a variável `testFilename` é dada por:

```
1 testFilename =
2     Environment.getExternalStorageDirectory().getAbsolutePath()
3     + "/audiorecordtest.3gp";
```

Algoritmo 9.7: Variável `testFilename`

Dica: Não se esqueça das permissões no *Manifest*!

Com isso terminamos nosso aplicativo que não só faz uma simples gravação de áudio, mas que também nos permite escutá-la usando a classe `MediaPlayer`.

Localização e Mapas

O Google já tem uma extensa documentação que nos ensina como acessar a API do Google *Maps* no seu aplicativo. Porém, muitas vezes ela pode ser confusa e você sente que faltam informações um pouco mais claras. Nesse capítulo iremos abordar tanto o acesso a localização do dispositivo usando `LocationManager` e colocar essas informações no Google *Maps*.

Já existe uma nova API que utiliza o Google *Play Services*, a documentação oficial pode ser vista [aqui](#)¹

10.1 Acessando a localização

Antes de iniciar, você precisa definir se quer obter a localização usando GPS (*Global Positioning System*) ou usando a Internet. Caso opte por usar o GPS, deve-se atentar ao fato que o GPS demora muito para inicializar e obter as coordenadas. Enquanto que usando a Internet isso é praticamente instantâneo, porém a localização não é tão precisa. Ou seja, se quer rapidez use a Internet como seu *Location Provider*, se quer precisão use o GPS.

Sabendo disso então você deve colocar as permissões no *Manifest*, mostradas abaixo.

```
1 <uses-permission android:name="android.permission.INTERNET" />
2 <uses-permission
3   android:name="android.permission.ACCESS_FINE_LOCATION" />
4 <uses-permission
5   android:name="android.permission.ACCESS_COARSE_LOCATION" />
```

Algoritmo 10.1: Permissões para obter localização

A permissão `INTERNET` só será necessária caso não queira obter a localização usando o GPS.

O próximo passo é criar um `LocationListener` para acessar a localização.

¹<https://developer.android.com/google/play-services/location.html>

```
1 public class MyLocationListener implements LocationListener {
2     private double c_lat;
3     private double c_long;
4
5     @Override
6     public void onLocationChanged(Location location) {
7         c_lat = location.getLatitude();
8         c_long = location.getLongitude();
9     }
10
11    @Override
12    public void onProviderDisabled(String provider) {
13        Toast.makeText(getApplicationContext(),
14            provider + " provider disabled", Toast.LENGTH_SHORT).show();
15    }
16
17    @Override
18    public void onProviderEnabled(String provider) {
19        Toast.makeText(getApplicationContext(),
20            provider + " provider enabled", Toast.LENGTH_SHORT).show();
21    }
22
23    @Override
24    public void onStatusChanged
25        (String provider, int status, Bundle extras) {
26    }
27
28    ...
29 }
```

Algoritmo 10.2: Criando um LocationListener

O método mais importante deste *listener* é `onLocationChanged()`, toda vez que as coordenadas de sua posição forem alteradas, ele irá chamar este método. Para esse exemplo eu apenas atualizei o valor de duas variáveis, que podem ser utilizadas para atualizar a posição em um mapa, por exemplo. Já os métodos `onProviderDisabled()` e `onProviderEnabled()` realizam uma ação quando o provedor de localização é desativado ou ativado, isso irá depender muito de sua aplicação. Mas para um exemplo simples estou apenas mostrando um Toast com a mensagem de que ele foi desativado ou ativado. O método `onStatusChanged()` é chamado quando o estado do provedor muda, por exemplo se ele se torna disponível depois de um período de indisponibilidade.

O próximo passo é definir o `LocationListener` que criamos para um `LocationManager`. Que deve ser feito no `onCreate()`.

```
1 LocationManager mlocManager =  
2     (LocationManager) getSystemService(Context.LOCATION_SERVICE);  
3 MyLocationListener mlocListener = new MyLocationListener();  
4 mlocManager.requestLocationUpdates  
5     (LocationManager.GPS_PROVIDER, 1000, 0, mlocListener);
```

Algoritmo 10.3: Configurando o LocationManager

A referência ao `LocationManager` deve ser obtida com o método `getSystemService()`. Criamos uma nova instância da classe `MyLocationListener` que criamos anteriormente. Depois deve-se registrar o *listener* para receber atualizações usando `requestLocationUpdates()`. Os parâmetros são: o provedor que será usado, o intervalo mínimo entre atualizações (em milissegundos), a distância mínima entre atualizações (em metros) e o *listener*. Nesse exemplo, estamos usando o GPS, caso queira usar a Internet, deve-se usar `LocationManager.NETWORK_PROVIDER`.

10.2 Google Maps

Configurar o *Google Maps* para uso é trabalhoso. Você deve registrar sua aplicação para obter uma chave que dá permissão ao seu aplicativo para usar o *Maps*. Nessa seção faremos o passo a passo de como obter a chave e colocar um mapa na sua *activity*.

Antes de iniciar, quero deixar claro que o *Google Maps* não funciona no emulador Android. Isso é devido a utilização do *Google Play Services* que não está disponível nos emuladores (até essa data). Existe sim a possibilidade de instalar o *Play Services* nos emuladores, mas isso não está previsto pelo Google e pode não funcionar corretamente.

10.2.1 Obtendo a chave de certificação *debug*

Para poder testar o *Maps* nos aplicativos que está desenvolvendo você deve obter uma chave de certificação *debug*. Na verdade toda aplicação ao ser lançada na *Play Store* precisa estar assinada digitalmente usando uma chave *release* que pode ser obtida da mesma maneira.²

Caso esteja usando o *Eclipse*, a chave de certificação *debug* pode ser obtida no menu *Window > Preferences > Android > Build*, sob o campo *SHA-1 fingerprint*.

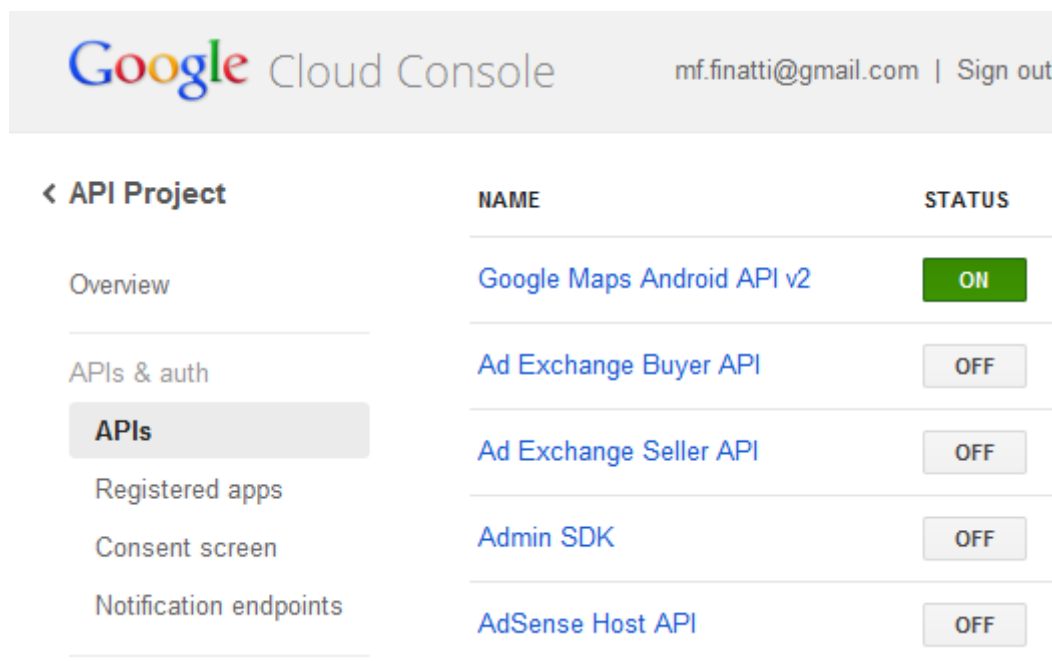
Você também pode adquiri-la pela linha de comando, usando o software *keytool* que vem junto do JDK. Você irá precisar apontar para o arquivo `debug.keystore` que está na pasta `.android` do seu sistema. Basta executar o *keytool* com o seguinte comando.

```
keytool -list -v -keystore ".android/debug.keystore" -alias androiddebugkey -storepass android -keypass android
```

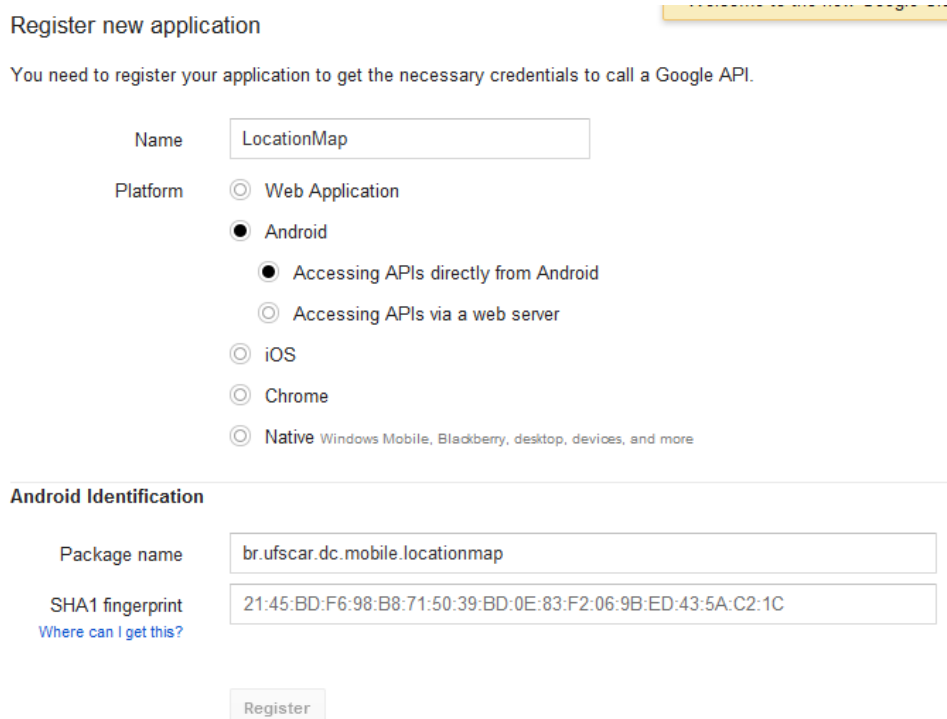
10.2.2 Obtendo a *API Key*

Com a chave de certificação *debug* em mãos, você deve acessar o *Google API Console* e criar um novo projeto. Após criar o projeto, acesse *API* no menu à esquerda e marque *Google Maps Android API v2* com *status ON*. Como mostrado na figura abaixo:

²<http://developer.android.com/intl/es/tools/publishing/app-signing.html>

Figura 10.1: Ativando *Maps API* no *Google API Console*

Em seguida, é necessário registrar uma nova aplicação. Para isso, acesse *Registered apps* no menu à esquerda e clique em *REGISTER APP*. Preencha os campos de acordo e clique em *Register*.



Register new application

You need to register your application to get the necessary credentials to call a Google API.

Name: LocationMap

Platform: ☒ Web Application ☒ Android

☒ Accessing APIs directly from Android ☐ Accessing APIs via a web server

☐ iOS ☐ Chrome ☐ Native Windows Mobile, Blackberry, desktop, devices, and more

Android Identification

Package name: br.ufscar.dc.mobile.locationmap

SHA1 fingerprint: 21:45:BD:F6:98:B8:71:50:39:BD:0E:83:F2:06:9B:ED:43:5A:C2:1C

[Where can I get this?](#)

Register

Figura 10.2: Registrando um *app* no *Google API Console*

Com isso, você terá acesso a sua *API Key* que será colocada no *Manifest* para habilitar o uso do *Google Maps* no dispositivo.



LocationMap

Android Application

Use the controls below to set up your application's authorization credentials. What you select depends on the type of data your application needs to access.

▶ **OAuth 2.0 Client ID**
Access user data via a consent screen

▼ **Android Key**
Use this for accessing data not associated with an account from Android

API KEY



ANDROID APPS
70:AB:3C:8A:32:7F:77:E7:0B:CD:9A:E1:54:8B:7B:9B:54:C7:85:8B;br.ufscar.dc.mobile.locationmap + -

ACTIVATED ON
Nov 19, 2013 1:26 PM

ACTIVATED BY
mf.finatti@gmail.com - **you**

Figura 10.3: Passo final para obter a *API Key*

Agora, com o código em mãos, você deve abrir seu *Manifest* e adicionar a seguinte linha dentro da *tag* <application>. Onde *value* é a *key* adquirida.

```

1 <meta-data
2   android:name="com.google.android.maps.v2.API_KEY"
3   android:value="AaaaBbB1cCcCCDDd22eee-ffFFFGgG3HHhh4i5" />

```

Algoritmo 10.4: Configurando a *API Key* no *Manifest*

10.2.3 Configurando o Google *Play Services*

Primeiro, você deve fazer *download* do Google *Play Services* no *SDK Manager*. Após o término do *download* é necessário importar a biblioteca no Eclipse e referenciá-la em seu projeto.

A pasta com a biblioteca do Google *Play Services* se encontrará na pasta `sdk/extras/google/google_play_services/libproject/` e se chama `google-play-services-lib`. É necessário importar essa pasta como projeto existente no Eclipse, não se esqueça de marcar a opção para copiar o projeto para o seu *workspace*.

Depois de importar, você deve abrir as propriedades do seu projeto e selecionar *Android* no menu à esquerda. Na parte inferior onde está escrito *Library* clique em *Add...* e selecione o `google-play-services-lib`. Se aparecer um símbolo verde de "correto", então deu certo.

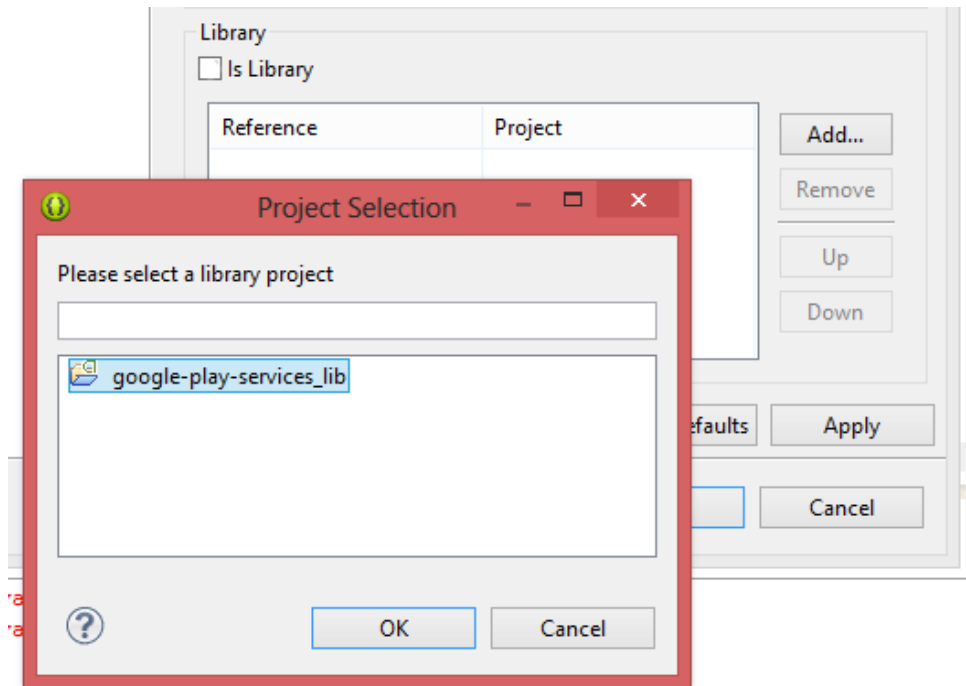


Figura 10.4: Adicionando a Google *Play Services* como biblioteca do projeto

10.2.4 Adicionando um mapa em sua *activity*

O Google *Maps* no Android usa OpenGL ES versão 2 para renderizar o mapa, logo é necessário adicionar o uso do OpenGL no *Manifest*.

```

1 <uses-feature
2   android:glEsVersion="0x00020000"
3   android:required="true" />

```

Algoritmo 10.5: Adicionando o uso do OpenGL no *Manifest*

Agora que já obtemos a localização, podemos usá-la para marcar a posição no mapa. Existem diversas maneiras de adicionar um mapa em uma *activity*. Nesse exemplo utilizaremos um *FrameLayout* para colocar o mapa dentro. Uma outra forma é simplesmente declarando um fragmento no XML do *layout* como mostrado abaixo:

```

1 <fragment
2   android:id="@+id/map"
3   android:layout_width="match_parent"
4   android:layout_height="match_parent"
5   android:name="com.google.android.gms.maps.MapFragment" />

```

Algoritmo 10.6: Adicionando o mapa como um fragmento no XML

O *layout* da *activity* consiste de dois *TextView* para mostrar a latitude e a longitude do dispositivo no momento, além disso tem um *FrameLayout* que irá conter o fragmento do mapa.

```
1 public class MainActivity extends FragmentActivity {
2     private LatLng mLocation;
3     private TextView latTv;
4     private TextView longTv;
5     private FrameLayout mapFrame;
6
7     private static final String MAP_FRAGMENT_TAG = "map";
8     private GoogleMap mMap;
9     private SupportMapFragment mMapFragment;
10    private Marker marker;
11
12
13    @Override
14    protected void onCreate(Bundle savedInstanceState) {
15        super.onCreate(savedInstanceState);
16        setContentView(R.layout.activity_main);
17
18        LocationManager mlocManager =
19            (LocationManager) getSystemService(Context.LOCATION_SERVICE);
20        MyLocationListener mlocListener = new MyLocationListener();
21        mlocManager.requestLocationUpdates
22            (LocationManager.GPS_PROVIDER, 0, 0, mlocListener);
23
24        latTv = (TextView) findViewById(R.id.latitude);
25        longTv = (TextView) findViewById(R.id.longitude);
26
27        mMapFragment = (SupportMapFragment) getSupportFragmentManager()
28            .findFragmentByTag(MAP_FRAGMENT_TAG);
29
30
31        if (mMapFragment == null) {
32            mMapFragment = SupportMapFragment.newInstance();
33
34            FragmentTransaction fragmentTransaction =
35                getSupportFragmentManager().beginTransaction();
36            fragmentTransaction.add
37                (R.id.mapFrame, mMapFragment, MAP_FRAGMENT_TAG);
38            fragmentTransaction.commit();
39        }
40
41        setUpMapIfNeeded();
42    }
```

Algoritmo 10.7: Activity com Google Maps

O primeiro detalhe a ser notado é que a *Activity* está estendendo *FragmentActivity* a fim de poder usar o *SupportFragment* para compatibilidade com versões anteriores do Android. As linhas 18 a 22 já são conhecidas, do exemplo anterior.

Na linha 27 estamos usando `findFragmentByTag()` para podermos obter o fragmento do mapa, caso ele não esteja declarado no XML, será criada uma nova instância na linha 32. Na

linha 34 a 38 estamos adicionando o fragmento do mapa ao `FrameLayout` do *layout* usando `FragmentManager` e por fim chamando uma função `setUpMapIfNeeded()` para obtermos a instância do mapa (e não do seu fragmento).

```

1 private void setUpMapIfNeeded() {
2     if (mMap == null) {
3         mMap = mMapFragment.getMap();
4
5         if (mMap != null)
6             marker = mMap.addMarker
7                 (new MarkerOptions().position(new LatLng(0, 0)));
8     }
9 }

```

Algoritmo 10.8: Método `setUpMapIfNeeded()`

Apenas obtemos a instância do mapa de seu fragmento e criando um marcador na posição (0,0). Importante salientar que o principal motivo de se fazer esse método é para poder chamá-lo no método `onResume()` da *activity* a fim de atualizar o mapa caso o aplicativo tenha sido parado por algum motivo. Agora iremos modificar o método `onLocationChanged()` do `MyLocationListener` para atualizar a posição no mapa e adicionar um marcador na posição correta.

```

1 @Override
2 public void onLocationChanged(Location location) {
3     c_lat = location.getLatitude();
4     c_long = location.getLongitude();
5     mLocation = new LatLng(c_lat, c_long);
6
7     latTv.setText("Lat: " + c_lat);
8     longTv.setText("Long: " + c_long);
9
10    marker.remove();
11    marker = mMap.addMarker
12        (new MarkerOptions().position(mLocation).title("You are here"));
13    mMap.moveCamera
14        (CameraUpdateFactory.newLatLngZoom(mLocation, 14.0f));
15 }

```

Algoritmo 10.9: Método `onLocationChanged()` modificado

Primeiro removemos o marcador anterior, depois adicionamos um novo marcador na posição obtida pelo *listener*. Na linha 13 chamamos o método `moveCamera()` para colocar a câmera na posição do marcador.

Para mais exemplos de mapas, você pode importar o projeto *maps* da pasta *samples* onde você obteve o *libproject*.

Compartilhamento

Já aprendemos a enviar dados à outras *activities* usando a classe `Intent`. Agora veremos como faz para enviar esses dados e dar ao usuário a opção de escolher qual aplicativo já instalado no dispositivo ele quer usar para receber esses dados. Isso pode ser visto como compartilhamento. Imagine que você tirou uma foto e quer mostrar ao seus amigos pelo *Facebook*. Na verdade está enviando uma foto à *activity* do Facebook que recebe esse tipo de dado e assim ela tomará o controle.

Quando você construir um `Intent` você deve especificar a ação que você espera que ele acione. O Android já define vários tipos de ações e entre eles está o `ACTION_SEND` que indica que o `intent` está enviando dados de uma *activity* para outra.

Se você quer enviar apenas um texto, por exemplo, basta seguir o pequeno exemplo abaixo:

```
1 Intent sendIntent = new Intent();
2 sendIntent.setAction(Intent.ACTION_SEND);
3 sendIntent.putExtra(Intent.EXTRA_TEXT, "This is my text to send.");
4 sendIntent.setType("text/plain");
5 startActivity(sendIntent);
```

Algoritmo 11.1: Enviando um texto simples através de um `Intent`

Contanto que exista um aplicativo instalado no aparelho com um filtro que aceite `ACTION_SEND` e o tipo MIME `"text/plain"` o Android irá executá-lo, caso exista mais de um então um alerta aparecerá perguntando qual aplicativo o usuário deseja escolher para receber o `Intent`.

Dica: Se chamar `Intent.createChooser()` então o Android sempre irá mostrar o alerta. Você pode configurar o texto a ser mostrado no parâmetro.

```
startActivity(Intent.createChooser(sendIntent, "Share text..."));
```

Algoritmo 11.2: Chamando `createChooser()`

Você pode usar alguns *extras* padrões nos `Intent`, tais como: `EXTRA_EMAIL` para o texto ser o corpo do e-mail, `EXTRA_SUBJECT` para o texto ser o assunto do e-mail. Porém o aplicativo que estiver recebendo o `Intent` deve estar preparado para esse tipo de *extra*, se não nada vai acontecer. Existe ainda a opção de criar *extras* personalizados, mas também só irá funcionar caso o aplicativo esteja projetado para esse tipo de *extra*. Nesse caso é comum criar um *extra* personalizado caso você tenha feito um conjunto de aplicativos que podem trocar informações através de `Intents` e que irão usar esse *extra*.

Além de texto é possível enviar dados binários ou até mesmo mais de um dado ao enviar uma lista. Para enviar uma foto que você tirou com a câmera, por exemplo, você deve usar o *extra* chamado `EXTRA_STREAM` e configurar o tipo do `Intent` para `"image/jpeg"`.

Uma forma mais completa de enviar imagens ou outros tipos de dados é usando um `ContentProvider`¹ que irá criar uma interface para a prover os dados a outras aplicações.

Outra parte importante do compartilhamento é fazer com que sua aplicação consiga receber dados de `Intents`. O primeiro passo para isso é configurar os `intent-filter` no *Manifest*. Quando você define um `intent-filter` você está dizendo para o Android que uma determinada *activity* é capaz de receber um determinado de `Intent`. Uma *activity* pode ter múltiplos filtros, cada um relacionado a um determinado tipo de dado.

Para exemplificar o conceito, iremos fazer um aplicativo que escreve um texto e tira uma foto e um aplicativo que recebe um texto ou uma imagem e mostra o que foi recebido. O aplicativo usa conceitos já conhecidos então iremos nos limitar a explicar somente a parte do compartilhamento. Entretanto, caso haja dúvidas é possível obter ambos aplicativos do repositório.

No primeiro aplicativo, é interessante salientar apenas os botões que adicionei para compartilhar. Um tem a função de compartilhar o texto e outro a imagem. Você pode, por exemplo compartilhar um e depois o outro no aplicativo do *Gmail* que ambos irão ser adicionados ao corpo do e-mail.

Observe no código abaixo, como ficaram os botões de compartilhamento.

¹<http://developer.android.com/intl/es/reference/android/content/ContentProvider.html>

```
1 final Intent shareIntent = new Intent();
2 shareIntent.setAction(Intent.ACTION_SEND);
3
4 shareText.setOnClickListener(new OnClickListener() {
5
6     @Override
7     public void onClick(View v) {
8         String text = textField.getText().toString();
9         shareIntent.setType("text/plain");
10        shareIntent.putExtra(Intent.EXTRA_TEXT, text);
11        startActivity(Intent.createChooser(shareIntent, "Send to..."));
12    }
13 });
14
15 sharePic.setOnClickListener(new OnClickListener() {
16
17     @Override
18     public void onClick(View v) {
19         shareIntent.setType("image/png");
20         shareIntent.putExtra(Intent.EXTRA_STREAM, picUri);
21         startActivity(Intent.createChooser(shareIntent, "Send to..."));
22     }
23 });
```

Algoritmo 11.3: Botões para compartilhar texto e imagem

A parte mais importante é o método `setType`, ele recebe um *MIME Type*², como é o caso de `text/plain` e `image/png`. Outros *MIME Types* poderiam ser usados como `image/*` caso não soubéssemos o formato da imagem ou até mesmo `/*/*` que representa qualquer tipo de dado. O problema disso é que todos aplicativos irão aparecer na lista e alguns podem não receber o que você está tentando enviar! Portanto, seja cauteloso.

Já o segundo aplicativo irá receber tanto um texto como uma foto e mostrar ao usuário, apenas para exemplificar o recebimento de `Intents`. O primeiro passo é criar os `intent-filter` para receber os *MIME Type* corretos.

²MIME: <http://en.wikipedia.org/wiki/MIME>

```

1 <activity
2   android:name="br.ufscar.dc.mobile.receivecontent.MainActivity"
3   android:label="@string/app_name" >
4   <intent-filter>
5     <action android:name="android.intent.action.MAIN" />
6     <category android:name="android.intent.category.LAUNCHER" />
7   </intent-filter>
8   <intent-filter>
9     <action android:name="android.intent.action.SEND"/>
10    <category android:name="android.intent.category.DEFAULT"/>
11    <data android:mimeType="image/*"/>
12  </intent-filter>
13  <intent-filter>
14    <action android:name="android.intent.action.SEND"/>
15    <category android:name="android.intent.category.DEFAULT"/>
16    <data android:mimeType="text/plain"/>
17  </intent-filter>
18 </activity>

```

Algoritmo 11.4: Configurando os intent-filter no *Manifest*

É necessário criar um intent-filter para cada tipo de dado que será recebido. Como nesse caso só existe uma *activity*, ela é tanto uma *activity Launcher*, ou seja, aquela que abre o aplicativo. Mas também recebe dois *MIME Types*, são eles `text/plain` e `image/*`. Em seguida, precisamos receber os dados do Intent e mostrá-los para o usuário, como é feito no código abaixo:

```

1 Intent intent = getIntent();
2 String action = intent.getAction();
3 String type = intent.getType();
4
5 if(Intent.ACTION_SEND.equals(action) && type != null){
6   if("text/plain".equals(type)){
7     String sharedText = intent.getStringExtra(Intent.EXTRA_TEXT);
8     if(sharedText != null)
9       sharedTextView.setText(sharedText);
10  } else if (type.startsWith("image/")) {
11    Uri imageUri = (Uri) intent.getParcelableExtra(Intent.EXTRA_STREAM);
12    if(imageUri != null){
13      picView.setImageBitmap
14      (decodeSampledBitmapFromResource(
15        getResources(), imageUri.getPath(), 200, 200));
16    }
17  }
18 }

```

Algoritmo 11.5: Obtendo os dados do Intent e mostrando ao usuário

Lembre-se que nesse pedaço de código, estamos apenas preocupados em obter os dados do

`Intent` e colocá-los nas suas *views* e portanto não é o código completo da *activity*. Basicamente o que fazemos é comparar as *strings* recebidas no `Intent` para saber o que está sendo recebido. Ao verificar o *action*, sabemos que o `Intent` veio através de compartilhamento e ao analisar o *type*, sabemos o *MIME type* e podemos construir a *activity* de acordo.

Dica: Modifique a *activity* para salvar a imagem ou o texto no caso do usuário voltar para compartilhar outra coisa. Você pode salvar os dados no `Bundle`.

Esse capítulo tentou apenas para dar uma introdução ao compartilhamento de dados entre aplicativos. Outras coisas interessantes podem ser feitas, por exemplo adicionar o botão de *Easy Share* como descrito [nessa página](http://developer.android.com/intl/es/training/sharing/shareaction.html)³ da documentação. Além disso, existe também a classe `ContentProvider` que fornece uma interface para compartilhamento.

³<http://developer.android.com/intl/es/training/sharing/shareaction.html>

Agenda e Contatos

12.1 Usando o *Contacts Provider*

O Android contém um repositório central onde estão armazenadas as informações dos contatos do usuário, incluindo os dados de redes sociais. Esse repositório se chama *Contacts Provider*¹.

Seguindo as lições da documentação oficial, iremos criar uma pequena aplicação que obtém todos seus contatos e popula uma lista com eles. Mais detalhes do contato poderão ser obtidos ao se clicar no contato. O primeiro passo, sempre obrigatório para se acessar o repositório de contatos, é adicionar permissão no *Manifest*. Demonstrado no trecho abaixo:

```
<uses-permission android:name="android.permission.READ_CONTACTS" />
```

Algoritmo 12.1: Permissão para acessar os contatos

Em seguida precisamos de uma *activity* com uma lista para conter os contatos. Nesse exemplo criamos um XML de *layout* que só contenha uma `ListView` e em seguida um outro XML para definir o *layout* de um item da lista, nesse caso será somente uma `TextView`. Você poderá obter mais detalhes no código do projeto disponível no repositório.

Dica: Você pode incrementar essa lista adicionando outras informações do contato na lista, como a foto de cada um.

Após ter definido o *layout* precisamos mostrar a lista de contatos, comece definindo a *activity* que irá conter a lista. A lista deverá ser populada utilizando a classe `ContactsContract`² que já nos providencia algumas constantes e métodos para acessar a lista de contatos.

¹<http://developer.android.com/guide/topics/providers/contacts-provider.html>

²<http://developer.android.com/reference/android/provider/ContactsContract.html>

```

1 public class ContactsActivity extends FragmentActivity implements
2   LoaderCallbacks<Cursor>, OnItemClickListener {

```

Algoritmo 12.2: Activity que irá conter a lista de contatos

Como pode ser observado no código 12.2 estamos estendendo a classe `FragmentActivity`³ que nos permite usar o `Loader` em Android mais antigos. Além disso estamos implementando as interfaces `LoaderManager.LoaderCallbacks`⁴ que permite o carregamento de dados de forma assíncrona. Estamos implementando também a interface `OnItemClickListener`⁵ para que possamos carregar mais informações do contato quando este é clicado na lista.

Precisamos de algumas variáveis para auxiliar na seleção dos contatos e na criação do adaptador da lista. A variável `FROM_COLUMNS`, abaixo, são os dados que serão mostrados e que nesse caso é apenas o nome do contato. Observe que estamos fazendo uma pequena comparação da versão do SDK pois a partir da versão 3.0 do SDK, a classe `Contacts` foi alterada. Já a variável `TO_IDS` serve para o adaptador saber onde colocar a informação, passamos para ele o *id* do `TextView` que colocamos no XML. Como essa lição se baseia na lição da documentação oficial, estamos usando um dos *ids* padrões do Android.

```

1 @SuppressWarnings("InlinedApi")
2 private final static String[] FROM_COLUMNS =
3     {Build.VERSION.SDK_INT >= Build.VERSION_CODES.HONEYCOMB ?
4       Contacts.DISPLAY_NAME_PRIMARY : Contacts.DISPLAY_NAME };
5 private final static int[] TO_IDS = {android.R.id.text1};

```

Algoritmo 12.3: Variáveis para o adaptador da lista

As próximas variáveis são aquelas da seleção dos contatos. Nesse exemplo vamos obter todos os contatos e portanto deixar a variável `mSearchString` vazia, mas caso queira obter contatos específicos, basta fazer algo que preencha essa variável. Temos a variável `SELECTION` que funciona como a parte *WHERE* de uma consulta SQL, ela já está preparada para a busca por algum nome, caso queira. A variável `PROJECTION` é uma constante que define as colunas que você quer retornar das consultas, nesse caso queremos 3: o *id* do contato, sua chave de busca (*lookup key*) e o seu nome. Por fim, as variáveis `CONTACT_ID_INDEX` e `LOOKUP_KEY_INDEX` funcionam como ponteiros para o `Cursor` que irá se mover nos resultados da pesquisa, basta definir o valor da variável como sendo a posição da coluna na projeção. Como temos a coluna *id* sendo a primeira e a *lookup* sendo a segunda, colocamos os valores 0 e 1, respectivamente. Isso é necessário para se obter dados de uma coluna individual do `Cursor`.

³<http://developer.android.com/reference/android/support/v4/app/FragmentActivity.html>

⁴<http://developer.android.com/reference/android/app/LoaderManager.LoaderCallbacks.html>

⁵<http://developer.android.com/reference/android/widget/AdapterView.OnItemClickListener.html>

```

1 @SuppressWarnings("InlinedApi")
2 private final static String[] PROJECTION = {
3     Contacts._ID,
4     Contacts.LOOKUP_KEY,
5     Build.VERSION.SDK_INT >= Build.VERSION_CODES.HONEYCOMB ?
6         Contacts.DISPLAY_NAME_PRIMARY : Contacts.DISPLAY_NAME,
7 };
8 private static final int CONTACT_ID_INDEX = 0;
9 private static final int LOOKUP_KEY_INDEX = 1;
10
11 @SuppressWarnings("InlinedApi")
12 private static final String SELECTION =
13     Build.VERSION.SDK_INT >= Build.VERSION_CODES.HONEYCOMB ?
14     Contacts.DISPLAY_NAME_PRIMARY + " LIKE ?" :
15     Contacts.DISPLAY_NAME + " LIKE ?";
16
17 private String mSearchString = "";
18 private String[] mSelectionArgs = { mSearchString };

```

Algoritmo 12.4: Variáveis para o Cursor do conjunto resultante da busca

As últimas variáveis de que precisamos são estas, para armazenar os resultados da consulta e guardar referências. Iremos usar a Uri do contato para buscar mais informações posteriormente.

```

1 private ListView mContactList;
2 private long mContactId;
3 private String mContactKey;
4 private Uri mContactUri;
5 private SimpleCursorAdapter mCursorAdapter;

```

Algoritmo 12.5: Variáveis de controle

Depois, precisamos implementar o método `onCreate()`. Nesse método, iremos configurar o adaptador da lista que será um `SimpleCursorAdapter`⁶. Esse adaptador liga os resultados da busca com a `ListView`.

⁶<http://developer.android.com/reference/android/widget/SimpleCursorAdapter.html>

```

1 protected void onCreate(Bundle savedInstanceState) {
2     super.onCreate(savedInstanceState);
3     setContentView(R.layout.activity_main);
4
5     mContactList = (ListView) findViewById(R.id.contactList);
6     mCursorAdapter = new SimpleCursorAdapter(
7         this,
8         R.layout.list_item,
9         null,
10        FROM_COLUMNS,
11        TO_IDS, 0);
12    mContactList.setAdapter(mCursorAdapter);
13    mContactList.setOnItemClickListener(this);
14
15    getSupportLoaderManager().initLoader(0, null, this);
16 }

```

Algoritmo 12.6: Método onActivityCreated()

Na linha 4 do algoritmo 12.6, estamos criando o `SimpleCursorAdapter`, como já discutido na lição de listas, é necessário passar para o adaptador o *layout* interno da lista assim como os mapas com os dados e os *ids* das *views* em que esses dados serão inseridos. São esses parâmetros as variáveis `FROM_COLUMNS` e `TO_IDS`. Na linha 10 determinamos o adaptador para a lista e na linha 11 determinamos que o listener da lista será a própria classe. Isso é possível pois estamos implementando a interface `OnItemClickListener`.

Como estamos usando o `CursorLoader`⁷ para buscar os dados, nós precisamos inicializar a *thread* que ficará no plano de fundo que irá controlar a busca de forma assíncrona.

O próximo passo é implementar o método `onCreateLoader()`, que irá realizar a consulta, ao retornar um `CursorLoader` que recebeu como parâmetros da busca.

```

1 @Override
2 public Loader<Cursor> onCreateLoader(int arg0, Bundle arg1) {
3     mSelectionArgs[0] = "%" + mSearchString + "%";
4     return new CursorLoader(
5         getActivity(),
6         Contacts.CONTENT_URI,
7         PROJECTION,
8         SELECTION,
9         mSelectionArgs,
10        null);
11 }

```

Algoritmo 12.7: Método onCreateLoader()

Depois, você deve implementar os métodos `onLoadFinished()` e `onLoaderReset()`. O primeiro é chamado quando o *Contacts Provider* retorna os resultados da consulta, usamos

⁷<http://developer.android.com/reference/android/content/CursorLoader.html>

o método `swapCursor()` para trocar o *cursor* do adaptador pelo *cursor* do método `onLoadFinished()`. Já o segundo é chamado quando o *framework* detecta que o *cursor* contém dados desatualizados, nesse caso basta apagar a referência ao *cursor* no adaptador.

Por fim, você deve implementar o método `onItemClick()` para abrir uma nova *activity* ao clicar no contato.

```
1 public void onItemClick(AdapterView<?> arg0, View arg1,
2     int position, long arg3) {
3     Cursor cursor = mCursorAdapter.getCursor();
4     cursor.moveToPosition(position);
5     mContactId = cursor.getLong(CONTACT_ID_INDEX);
6     mContactKey = cursor.getString(LOOKUP_KEY_INDEX);
7
8     mContactUri = Contacts.getLookupUri(mContactId, mContactKey);
9     Intent intent = new Intent(this, ContactDetailsActivity.class);
10    intent.putExtra("URI", mContactUri);
11    startActivity(intent);
12 }
```

Algoritmo 12.8: Método `onItemClick()`

Primeiro nós obtemos a posição na lista através do parâmetro `position`, em seguida colocamos o *cursor* nessa posição e obtemos os dados do contato. Usando o método `getLookupUri()` nós conseguimos a URI do contato para que seja possível acessar suas informações detalhadas.

12.1.1 Detalhes de um contato

Conseguir os detalhes do contato usa o mesmo princípio de conseguir todos os contatos. Devemos fazer algumas consultas, mas com seleções e projeções diferentes. Para obter os telefones de um contato, deverá acessar o campo *Phone* do *ContactsContract*, para obter os e-mails: *Email*, e para acessar os endereços: *StructuredPostal*.

Nesse exemplo, criaremos uma nova *activity* que irá conter os dados do contato escolhido na primeira *activity*. Para facilitar a leitura do código, iremos criar três interfaces para serem usadas, essas interfaces irão conter as projeções e seleções de cada um dos tipos de dado.

```

1 public interface ContactDetailsQuery {
2     final static int QUERY_ID = 1;
3     @SuppressWarnings("InlinedApi")
4     final static String[] PROJECTION = {
5         Build.VERSION.SDK_INT >= Build.VERSION_CODES.HONEYCOMB ?
6             Contacts.DISPLAY_NAME_PRIMARY : Contacts.DISPLAY_NAME,
7     };
8     final static int DISPLAY_NAME = 0;
9 }
10
11 public interface ContactPhoneQuery {
12     final static int QUERY_ID = 2;
13     final static String[] PROJECTION = {
14         Phone.NUMBER
15     };
16     final static String SELECTION =
17         Phone.CONTACT_ID + " = ?";
18
19     final static int NUMBER = 0;
20 }
21
22 public interface ContactAddressQuery {
23     final static int QUERY_ID = 3;
24     final static String[] PROJECTION = {
25         StructuredPostal.FORMATTED_ADDRESS
26     };
27     final static String SELECTION =
28         StructuredPostal.CONTACT_ID + " = ?";
29
30     final static int ADDRESS = 0;
31 }

```

Algoritmo 12.9: Interfaces das consultas dos contatos

Dessa forma, mostrada no algoritmo 12.9, conseguimos organizar melhor o código para as consultas que iremos fazer. Essa *activity*, que decidi chamar `ContactDetailsActivity` também irá estender `FragmentActivity` e implementar `LoaderCallbacks<Cursor>`. Iremos utilizar essas três variáveis, das quais duas serão obtidas do `Intent` que abriu essa *activity*.

```

1 public class ContactDetailsActivity extends FragmentActivity
2 implements LoaderCallbacks<Cursor> {
3     private Uri mContactUri;
4     private long mContactId;
5     private ViewGroup container;

```

Algoritmo 12.10: Classe `ContactDetailsActivity`

Em seguida, construa o método `onCreate()`. A diferença dessa *activity* para a anterior

é a obtenção dos dados do Intent e a referência ao ViewGroup que iremos utilizar mais tarde.

```

1 @Override
2 protected void onCreate(Bundle savedInstanceState) {
3     super.onCreate(savedInstanceState);
4     setContentView(R.layout.activity_contact_details);
5
6     Intent intent = getIntent();
7     Uri u = intent.getParcelableExtra("URI");
8     long i = intent.getLongExtra("ID", 0);
9     setContact(u, i);
10
11     container = (ViewGroup) findViewById(R.id.viewgroup);
12
13     getSupportLoaderManager().initLoader(0, null, this);
14 }

```

Algoritmo 12.11: Método onCreate() de ContactDetailsActivity

O método setContact() é usado para colocar os dados obtidos do Intent nos atributos da classe e chamar as consultas necessárias. Chamamos o método restartLoader() para forçar a criação da *thread* que irá realizar a consulta. Observe que cada uma é responsável por uma consulta, passamos o QUERY_ID de cada uma das interfaces para diferenciá-las.

```

1 public void setContact(Uri contactUri, long contactId){
2     if(Build.VERSION.SDK_INT >= Build.VERSION_CODES.HONEYCOMB){
3         mContactUri = contactUri;
4     } else {
5         mContactUri = Contacts.lookupContact
6             (getContentResolver(), contactUri);
7     }
8     mContactId = contactId;
9
10    getSupportLoaderManager().
11        restartLoader(ContactDetailsQuery.QUERY_ID, null, this);
12    getSupportLoaderManager().
13        restartLoader(ContactAddressQuery.QUERY_ID, null, this);
14    if(hasPhone())
15        getSupportLoaderManager().
16            restartLoader(ContactPhoneQuery.QUERY_ID, null, this);
17 }

```

Algoritmo 12.12: Método setContact()

O método hasPhone(), chamado na linha 14, verifica se o contato tem algum número de telefone cadastrado. Esse método demonstra uma outra forma de se adquirir os dados de um

contato, sem ser em outra *thread*, mas na mesma *thread* da *UI* (*User Interface*)⁸.

Agora iremos implementar o método `onCreateLoader()`, de forma semelhante à última vez, mas agora iremos criar um `CursorLoader` diferente dependendo do *id* que foi passado para o método - no caso é o `QUERY_ID` que foi o parâmetro do método `restartLoader()`, visto no algoritmo 12.12.

```

1 @Override
2 public Loader<Cursor> onCreateLoader(int id, Bundle arg1) {
3     switch (id) {
4         case ContactDetailsQuery.QUERY_ID:
5             return new CursorLoader(this,
6                                     mContactUri,
7                                     ContactDetailsQuery.PROJECTION,
8                                     null, null, null);
9         case ContactAddressQuery.QUERY_ID:
10            String addressArgs[] = {String.valueOf(mContactId)};
11            return new CursorLoader(this,
12                                    StructuredPostal.CONTENT_URI,
13                                    ContactAddressQuery.PROJECTION,
14                                    ContactAddressQuery.SELECTION,
15                                    addressArgs, null);
16        case ContactPhoneQuery.QUERY_ID:
17            String phoneArgs[] = {String.valueOf(mContactId)};
18            return new CursorLoader(this,
19                                    Phone.CONTENT_URI,
20                                    ContactPhoneQuery.PROJECTION,
21                                    ContactPhoneQuery.SELECTION,
22                                    phoneArgs, null);
23        default:
24            break;
25    }
26    return null;
27 }

```

Algoritmo 12.13: Método `onCreateLoader()`

Nesse método, nós estamos fazendo um `switch` para selecionar qual consulta iremos fazer e então usamos as projeções e seleções de cada interface para realizar a consulta. Se atente as diferenças entre cada um: no primeiro caso, nós usamos `mContactUri` no segundo parâmetro do construtor do `CursorLoader` pois nós já temos o "endereço" do nome do contato nesse atributo, nesse caso não é preciso fazer uma seleção.

No segundo e terceiro casos, no segundo parâmetro passamos o "endereço" da tabela dos endereços e telefones dos contatos, mas usamos a seleção para poder selecionar qual contato. A variável `addressArgs` contém o *id* do contato que irá servir como parâmetro da seleção.

⁸Tudo que fizemos até agora (exceto o `AsyncTask` na lição de comunicação) está rodando na chamada *UI thread*, isso significa que o Android está executando seu código na mesma *thread* em que cria a interface do usuário. Usar o `Loader` implica em fazer com que as consultas rodem em uma outra *thread*, liberando a *UI thread* do serviço e impedindo que o aplicativo pareça estar travado enquanto busca informações. O `AsyncTask` é uma outra forma de executar *background threads* de forma assíncrona.

Note que no Android essas informações dos contatos estão guardadas em um banco de dados e o que estamos fazendo na realidade são *SELECT* comuns, mas usando as classes pré-definidas para nos auxiliar. O Android também já junta várias tabelas automaticamente para facilitar as buscas.

Para finalizar, vamos implementar o método `onLoadFinished()`, nesse método é que adquirimos os dados do `cursor` e criamos as *views* para mostrar esses dados.

```
1 public void onLoadFinished(Loader<Cursor> loader, Cursor cursor) {
2     if(mContactUri == null){
3         return;
4     }
5     switch(loader.getId()){
6         case ContactDetailsQuery.QUERY_ID:
7             if(cursor.moveToFirst()){
8                 String contactName =
9                     cursor.getString(ContactDetailsQuery.DISPLAY_NAME);
10                TextView nameView = new TextView(this);
11                nameView.setText(contactName);
12                nameView.setTextSize(30);
13                container.addView(nameView);
14            }
15            break;
16         case ContactPhoneQuery.QUERY_ID:
17             if(cursor.moveToFirst()){
18                 do {
19                     String contactPhone =
20                         cursor.getString(ContactPhoneQuery.NUMBER);
21                     TextView phoneView = new TextView(this);
22                     phoneView.setText(contactPhone);
23                     phoneView.setTextSize(20);
24                     container.addView(phoneView);
25                 } while(cursor.moveToNext());
26             }
27            break;
28         case ContactAddressQuery.QUERY_ID:
29             if(cursor.moveToFirst()){
30                 do {
31                     String address =
32                         cursor.getString(ContactAddressQuery.ADDRESS);
33                     TextView addrView = new TextView(this);
34                     addrView.setText(address);
35                     addrView.setTextSize(15);
36                     container.addView(addrView);
37                 } while(cursor.moveToNext());
38             }
39            break;
40     }
41 }
```

Algoritmo 12.14: Método `onLoadFinished()`

Novamente, o método irá tomar uma ação diferente dependendo de qual consulta foi realizada. Se existe um resultado, então nós criamos uma `TextView` e colocamos o texto adquirido nela. Aqui é que usamos a variável `container`, usamos o método `addView()` para colocar essa *view* dentro dela.

Esse capítulo tentou dar uma introdução à obtenção dos contatos da agenda. É um tema especialmente difícil por haver muito rigor nos detalhes. É altamente recomendável estudar mais a fundo a lição presente na documentação oficial e o exemplo oferecido por eles.

Acelerômetro

Os dispositivos Android contam com um sensor chamado acelerômetro que consegue captar a aceleração do aparelho nos três eixos. É útil para fazer jogos, saber a orientação do aparelho, entre outras utilidades. Para isso o Android contém um *framework* para os sensores, uma das classes é a `SensorManager` que contém vários métodos para acessar, registrar *listeners* e constantes para calibrar os sensores, reportar a precisão do sensor e configurar a taxa de aquisição dos dados. Para saber mais sobre o *framework*, acesse a [documentação oficial](http://developer.android.com/guide/topics/sensors/sensors_overview.html)¹.

Agora, para exemplificar vamos construir uma aplicação bem simples que apenas irá mostrar na tela os valores lidos do acelerômetro. Criaremos apenas uma *activity* com três `TextView` para mostrar os valores lidos do acelerômetro.

```
1 public class AccelActivity extends Activity implements
2   SensorEventListener {
3     private TextView xAxis;
4     private TextView yAxis;
5     private TextView zAxis;
6
7     private SensorManager mSensorMan;
8     private Sensor mSensor;
9
10    private double gravity[] = {1, 1, 1};
11    private double acceleration[] = {1, 1, 1};
12
```

Algoritmo 13.1: Classe AccelActivity

Usaremos os dois *arrays*, `gravity` e `acceleration` para guardar os valores lidos do sensor. Em seguida, no método `onCreate()`, você deverá adquirir uma referência do `SensorManager` usando `getSystemService()`.

¹http://developer.android.com/guide/topics/sensors/sensors_overview.html

```

1 protected void onCreate(Bundle savedInstanceState) {
2     super.onCreate(savedInstanceState);
3     setContentView(R.layout.activity_accel);
4
5     xAxis = (TextView) findViewById(R.id.xAxis);
6     yAxis = (TextView) findViewById(R.id.yAxis);
7     zAxis = (TextView) findViewById(R.id.zAxis);
8
9     mSensorMan =
10         (SensorManager) getSystemService(Context.SENSOR_SERVICE);
11     mSensor = mSensorMan.getDefaultSensor(Sensor.TYPE_ACCELEROMETER);
12 }

```

Algoritmo 13.2: Método onCreate() de AccelActivity

É necessário agora criar o *listener* que irá escutar o sensor e escrever os dados na tela. O método onSensorChanged() é chamado de acordo com o intervalo definido em registerListener() na linha 4 do algoritmo 13.4.

```

1 @Override
2 public void onSensorChanged(SensorEvent event) {
3     if (event.sensor.getType() != Sensor.TYPE_ACCELEROMETER)
4         return;
5
6     final double alpha = 0.8;
7
8     gravity[0] = alpha * gravity[0] + (1 - alpha) * event.values[0];
9     gravity[1] = alpha * gravity[1] + (1 - alpha) * event.values[1];
10    gravity[2] = alpha * gravity[2] + (1 - alpha) * event.values[2];
11
12    acceleration[0] = event.values[0] - gravity[0];
13    acceleration[1] = event.values[1] - gravity[1];
14    acceleration[2] = event.values[2] - gravity[2];
15
16    DecimalFormat df = new DecimalFormat("##.####");
17    xAxis.setText("X Axis: " + df.format(acceleration[0]));
18    yAxis.setText("Y Axis: " + df.format(acceleration[1]));
19    zAxis.setText("Z Axis: " + df.format(acceleration[2]));
20 }

```

Algoritmo 13.3: Método onSensorChanged()

Primeiro verificamos se o sensor que chamou o método é o acelerômetro, só prosseguimos caso seja. Os dados lidos estão armazenados no *array* event. Cada índice indica um eixo: 0 é o eixo X, 1 é o eixo Y e 2 é o eixo Z. Para calcular corretamente a aceleração, precisamos retirar o componente gravitacional da conta. Para isso fazemos um filtro passa-baixo, como indicado na documentação².

²http://developer.android.com/guide/topics/sensors/sensors_motion.html

Por último, deve-se registrar o *listener* quando a *activity* está em primeiro plano e desregistrar quando a *activity* for pausada. Isso é necessário para não ficar consumindo a bateria de forma desnecessária. Logo, será necessário fazer isso nos métodos `onResume()` e `onPause()`.

```
1 protected void onResume() {  
2     super.onResume();  
3     mSensorMan.registerListener  
4         (this, mSensor, SensorManager.SENSOR_DELAY_NORMAL);  
5 }  
6  
7 protected void onPause() {  
8     super.onPause();  
9     mSensorMan.unregisterListener(this);  
10 }
```

Algoritmo 13.4: Métodos `onResume()` e `onPause()`

Quando registrar o *listener*, é importante indicar o intervalo de atualização (terceiro parâmetro), estamos usando `SENSOR_DELAY_NORMAL` pois é o mais lento.

Esse capítulo buscou apenas dar uma introdução bem simples ao uso de sensores, mas mais especificamente ao uso do acelerômetro. Todos os sensores podem ser programados de forma similar.

Bluetooth

O Android providencia uma API para conectar o aparelho em outros dispositivos *Bluetooth*, sejam outros celulares, *tablets*, fones de ouvido e até mesmo HDP (*Health Device Profile*) que é um tipo de conexão *Bluetooth* para aparelhos médicos. Com a API para *Bluetooth* você pode: sondar outros dispositivos *Bluetooth*, buscar por dispositivos já pareados, estabelecer conexões, transferir dados e gerenciar múltiplas conexões.

Para exemplificar iremos construir uma pequena aplicação que envia uma mensagem para um dispositivo pareado. Esse exemplo irá introduzir diversos novos conceitos do sistema Android e do próprio Java, entre eles: `BroadcastReceiver`, `Thread`, `Handler`, `Runnable`, além claro, do *Bluetooth*.

Primeiro nossa aplicação deve recuperar todos os dispositivos *bluetooth* já conhecidos, ou seja, aqueles que já foram pareados com o nosso aparelho. Colocaremos numa `ListView` tanto os dispositivos pareados como aqueles novos que iremos procurar.

Iniciaremos com uma classe que chamei de `DeviceListActivity`, essa *activity* irá mostrar a lista dos dispositivos encontrados, ao se clicar em um desses dispositivos, uma mensagem será enviada através de um *socket*. Somente se o outro dispositivo estiver rodando a aplicação no mesmo momento é que a mensagem poderá ser visualizada.

```
1 public class DeviceListActivity extends Activity
2     implements OnItemClickListener {
3     private ListView deviceList;
4
5     private BluetoothAdapter mBluetoothAdapter;
6     private ArrayAdapter<String> mAdapter;
```

Algoritmo 14.1: Classe `DeviceListActivity`

Usaremos um `ArrayAdapter`¹ para mostrar na lista o nome e o endereço MAC dos dispositivos. Além disso, a variável `BluetoothAdapter`² é usada para iniciar a varredura por

¹<http://developer.android.com/reference/android/widget/ArrayAdapter.html>

²<http://developer.android.com/reference/android/bluetooth/BluetoothAdapter.html>

novos dispositivos, obter a lista de dispositivos pareados instanciar um `BluetoothDevice`³ e criar um `BluetoothServerSocket`⁴ que irá aceitar novas conexões.

Agora já iremos para o método `onCreate()`, nesse método quatro importantes passos deverão ser seguidos. Os três primeiros estão sendo mostrados no código abaixo.

```
1 @Override
2 protected void onCreate(Bundle savedInstanceState) {
3     super.onCreate(savedInstanceState);
4     setContentView(R.layout.activity_device_list);
5
6     /* 1 */
7     mBluetoothAdapter = BluetoothAdapter.getDefaultAdapter();
8     if(mBluetoothAdapter == null) {
9         // Bluetooth nao suportado
10    }
11
12    /* 2 */
13    if(!mBluetoothAdapter.isEnabled()){
14        Intent enableBluetooth = new Intent(
15            BluetoothAdapter.ACTION_REQUEST_ENABLE);
16        startActivityForResult(enableBluetooth, REQUEST_ENABLE_BT);
17    }
18
19    mArrayAdapter = new ArrayAdapter<String>(
20        this, android.R.layout.simple_list_item_1);
21
22    /* 3 */
23    Set<BluetoothDevice> pairedDevices =
24        mBluetoothAdapter.getBondedDevices();
25    if(pairedDevices.size() > 0){
26        for(BluetoothDevice device : pairedDevices){
27            mArrayAdapter.add(device.getName() + "\n"
28                + device.getAddress());
29        }
30    }
31    ... //[1]
32    deviceList.setAdapter(mArrayAdapter);
33    ... //[2]
34 }
```

Algoritmo 14.2: Primeira parte do método `onCreate()`

O primeiro passo, marcado na linha 6, é obter a referência do adaptador *bluetooth* com a chamada ao método `BluetoothAdapter.getDefaultAdapter()`. Caso esse método retorne `null` então o aparelho Android não tem *hardware Bluetooth* e, portanto, a aplicação não irá funcionar.

³<http://developer.android.com/reference/android/bluetooth/BluetoothDevice.html>

⁴<http://developer.android.com/reference/android/bluetooth/BluetoothServerSocket.html>

O segundo passo é verificar se o *Bluetooth* está ativado, essa verificação é feita com uma chamada ao método `isEnabled()` como é feito na linha 13. Se o retorno for falso então nós iremos perguntar ao usuário se ele gostaria de ativar o adaptador *Bluetooth*. Para isso é necessário lançar um `Intent` com *action* igual à `BluetoothAdapter.ACTION_REQUEST_ENABLE`. Ao chamar o método `startActivityForResult()` um alerta irá perguntar ao usuário se ele quer ativar o *Bluetooth*. A variável `REQUEST_ENABLE_BT` pode ser qualquer valor maior que 0.

O terceiro passo é obter todos os dispositivos já pareados, a classe `BluetoothAdapter` mantém uma lista desses dispositivos que pode ser adquirida com a chamada ao método `getBondedDevices()`. Depois, adicionaremos o nome e endereço MAC desses dispositivos pareados à `ListView` usando o `ArrayAdapter`.

O método `onActivityResult()` não precisa de nada especial. Mostrar para o usuário um *feedback* já é suficiente.

```
1 @Override
2 protected void onActivityResult(int requestCode,
3     int resultCode, Intent data) {
4     super.onActivityResult(requestCode, resultCode, data);
5     if(requestCode == RESULT_OK) {
6         Toast.makeText(this, "Bluetooth is on", Toast.LENGTH_LONG)
7             .show();
8     } else if(requestCode == RESULT_CANCELED) {
9         Toast.makeText(this, "Bluetooth is off", Toast.LENGTH_LONG)
10            .show();
11     }
12 }
```

Algoritmo 14.3: Método `onActivityResult()`

Voltando ao método `onCreate()` Observe o comentário `// [1]` na linha 31 do algoritmo 14.2, nesse pedaço do código iremos adicionar um rodapé na `ListView` com um botão que iniciará a varredura por novos dispositivos.

Apesar de não ter apresentado este conceito antes, rodapés e cabeçalhos em `ListsViews` são bem simples, construa um *layout XML* e use o `LayoutInflater` para obter a `View`, depois adicione essa `View` à lista usando `addFooterView()`. Rodapés e cabeçalhos precisam ser adicionados à `ListView` antes de definir o adaptador. Também adicionaremos um *listener* a um botão para que ele inicie a varredura. O método `startDiscovery()` é usado para fazer a varredura.

```

1 deviceList = (ListView) findViewById(R.id.deviceList);
2
3 View v = getLayoutInflater().inflate(R.layout.device_list_footer, null);
4 Button scan = (Button) v.findViewById(R.id.button_scan);
5
6 scan.setOnClickListener(new OnClickListener() {
7
8     @Override
9     public void onClick(View v) {
10         mBluetoothAdapter.startDiscovery();
11     }
12 });
13
14 deviceList.addFooterView(v);

```

Algoritmo 14.4: Segunda parte do método onCreate()

Porém, chamar `startDiscovery()` não é suficiente, é necessário construir um `BroadcastReceiver` para capturar os `Intents` que tenham as informações dos dispositivos *Bluetooth* que foram descobertos.

```

1 private IntentFilter filter;
2
3 private final BroadcastReceiver mReceiver = new BroadcastReceiver() {
4     @Override
5     public void onReceive(Context context, Intent intent) {
6         String action = intent.getAction();
7         if (BluetoothDevice.ACTION_FOUND.equals(action)) {
8             BluetoothDevice device =
9                 intent.getParcelableExtra(BluetoothDevice.EXTRA_DEVICE);
10            mArrayAdapter.add(device.getName() + "\n"
11                             + device.getAddress());
12        }
13    }
14 };

```

Algoritmo 14.5: `BroadcastReceiver` que captura dispositivos *Bluetooth*

O que esse `BroadcastReceiver` está fazendo é capturar os `Intents` e verificar em quais deles o campo *action* equivale a `BluetoothDevice.ACTION_FOUND`. Sendo positivo, é obtido um `BluetoothDevice` que veio no campo *extra* do `Intent`. Então repetimos o procedimento de adicionar esse dispositivo no adaptador da lista. É necessário criar um `IntentFilter`⁵ para ser usado com o `BroadcastReceiver`.

Porém, para o `BroadcastReceiver` começar a funcionar é necessário registrá-lo com o método `registerReceiver()`. É importante também remove-lo com o método `unregisterReceiver()` no método `onDestroy()`.

⁵<http://developer.android.com/reference/android/content/IntentFilter.html>

```
1 @Override
2 protected void onDestroy() {
3     super.onDestroy();
4     unregisterReceiver(mReceiver);
5 }
6
7 @Override
8 protected void onResume() {
9     super.onResume();
10    filter = new IntentFilter(BluetoothDevice.ACTION_FOUND);
11    registerReceiver(mReceiver, filter);
12 }
```

Algoritmo 14.6: Registrando e removendo o BroadcastRegister

Agora, iremos criar três *threads*. A primeira, chamada de `AcceptThread` será responsável por instanciar um `BluetoothSocketServer` e aceitar uma nova conexão vinda de um outro dispositivo. A segunda, `ConnectThread` será usada quando a aplicação funcionar como cliente, essa *thread* irá tentar estabelecer uma conexão com outro dispositivo. Já a terceira, `ConnectedThread` será usada em ambos os casos, para gerenciar a conexão e a entrada e saída de dados. Todas essas *threads* estão escritas como classes *inline* (dentro da classe `DeviceListActivity`).

A primeira *thread*, `AcceptThread`, irá instanciar um `BluetoothServerSocket`⁶ usando o método `listenUsingRfcommWithServiceRecord()`, esse método cria um *socket* seguro em um canal *Bluetooth*, portanto toda informação será criptografada. Os parâmetros são `name` e `uuid`, `name` é o nome do serviço que nesse caso pode ser qualquer coisa. Já o `uuid` é um identificador universal único para ser usado com a aplicação, pode-se obter um número usando geradores aleatórios de UUID disponíveis na *web*. A *thread* irá ficar tentando escutar uma conexão com o método `accept()`. Note que este método é bloqueante e irá retornar um *socket* se a conexão foi aceita ou irá gerar uma exceção. Como não queremos aceitar mais nenhuma conexão, chamamos `close()` que irá fechar o `BluetoothServerSocket` mas não irá fechar o `BluetoothSocket` criado, mantendo a conexão.

Se o *socket* conseguir ser instanciado corretamente, iremos chamar o método `manageConnection()`, responsável por instanciar a *thread* que gerencia a conexão.

⁶<http://developer.android.com/reference/android/bluetooth/BluetoothServerSocket.html>

```

1 public static String MY_UUID = "d17deaaa-94e9-45ae-8ab0-0f73ef29e417";
2 public static String NAME = "myBluetoothChat";
3
4 private class AcceptThread extends Thread {
5     private final BluetoothServerSocket mServerSocket;
6
7     public AcceptThread() {
8         BluetoothServerSocket tmp = null;
9         try {
10             tmp = mBluetoothAdapter.listenUsingRfcommWithServiceRecord(
11                 NAME, UUID.fromString(MY_UUID));
12         } catch (IOException e) { }
13         mServerSocket = tmp;
14     }
15
16     public void run() {
17         BluetoothSocket socket = null;
18         while(true) {
19             try {
20                 socket = mServerSocket.accept();
21             } catch (IOException e) { break; }
22
23             if (socket != null) {
24                 manageConnection(socket);
25                 try {
26                     mServerSocket.close();
27                     break;
28                 } catch (IOException e) { }
29             }
30         }
31     }
32 }

```

Algoritmo 14.7: Classe AcceptThread

O algoritmo 14.8 abaixo mostra a segunda *thread*, essa *thread* é responsável por estabelecer uma conexão com outro dispositivo que esteja rodando a mesma aplicação, por isso é usado o mesmo UUID. O construtor irá receber um `BluetoothDevice`, que foi selecionado da lista, e tentará criar a conexão com o método `createRfcommSocketToServiceRecord()`. A *thread* irá cancelar a varredura e irá tentar conectar ao socket usando `connect()`, que também é uma chamada bloqueante. Se a conexão falhar ou ocorrer um *timeout*, ele irá gerar uma exceção. Por segurança, chamamos `close()` dentro do bloco `catch` para tentar fechar o *socket* e liberar os recursos do aparelho. Se tudo der certo, chamamos o método `manageConnection()` que veremos logo. Por enquanto deixaremos a explicação da linha 29 em espera, voltaremos logo nela.

```

1 private class ConnectThread extends Thread {
2     private final BluetoothSocket mSocket;
3     private final BluetoothDevice mDevice;
4
5     public ConnectThread(BluetoothDevice device) {
6         BluetoothSocket tmp = null;
7         mDevice = device;
8
9         try {
10             tmp = device.createRfcommSocketToServiceRecord(
11                 UUID.fromString(mUUID));
12         } catch (IOException e) {}
13         mSocket = tmp;
14     }
15
16     public void run() {
17         mBluetoothAdapter.cancelDiscovery();
18
19         try {
20             mSocket.connect();
21         } catch (IOException openExc) {
22             try {
23                 mSocket.close();
24             } catch (IOException closeExc) {}
25             return;
26         }
27
28         manageConnection(mSocket);
29         mHandler.post(mShowDialog);
30     }
31 }

```

Algoritmo 14.8: Classe ConnectThread

O método `manageConnection()` apenas irá instanciar a *thread* `ConnectedThread`.

```

1 public void manageConnection(BluetoothSocket socket) {
2     tConnected = new ConnectedThread(socket);
3     tConnected.start();
4 }

```

Algoritmo 14.9: Método `manageConnection()`

Agora veremos a última *thread*, essa responsável pela entrada e saída de dados através do *socket*.

```

1 private class ConnectedThread extends Thread {
2     private final BluetoothSocket mmSocket;
3     private final InputStream mmInput;
4     private final OutputStream mmOutput;
5
6     public ConnectedThread(BluetoothSocket socket) {
7         mmSocket = socket;
8         InputStream tmpIn = null;
9         OutputStream tmpOut = null;
10
11         try {
12             tmpIn = socket.getInputStream();
13             tmpOut = socket.getOutputStream();
14         } catch (IOException e) { }
15
16         mmInput = tmpIn;
17         mmOutput = tmpOut;
18     }
19
20     public void run() {
21         byte[] buffer = new byte[1024];
22         int bytes;
23
24         while(true) {
25             try {
26                 bytes = mmInput.read(buffer);
27                 mMessage = new String(buffer);
28                 mHandler.post(mShowMessage);
29             } catch (IOException e) { break; }
30         }
31     }
32
33     public void write(byte[] bytes) {
34         try {
35             mmOutput.write(bytes);
36         } catch (IOException e) { }
37     }
38
39     public void cancel() {
40         try {
41             mmSocket.close();
42         } catch (IOException e) { }
43     }
44 }

```

Algoritmo 14.10: Classe ConnectedThread

Primeiro obtemos o `InputStream` e `OutputStream` do *socket* usando os métodos `getInputStream()` e `getOutputStream()`, respectivamente. Você pode ler e escrever o fluxo de dados usando `read()` e `write()`, é necessário fazer isso numa *thread* pois essas

chamadas são bloqueantes. O método `read()` irá bloquear enquanto não houver nada para ler e o método `write()` pode bloquear caso o outro lado esteja lendo os *buffers* muito lentamente.

A *thread* irá repetir enquanto não houver exceção e ficará tentando ler do *stream* usando `read()`. Aqui novamente colocaremos a explicação da linha 28 em espera e voltaremos nela posteriormente. Faremos também um método `write()` para essa classe, para embrulhar o método do *socket*, que poderá ser chamado da *activity*.

Agora temos dois métodos, um para receber e mostrar a mensagem e outro para enviar a mensagem. O método `writeMessage()`, abaixo, abre um alerta com um `EditText` em que o usuário pode escrever a mensagem para ser enviada.

```

1 public void writeMessage() {
2     final EditText input = new EditText(this);
3     AlertDialog.Builder dialog = new AlertDialog.Builder(this)
4         .setTitle("Send a message")
5         .setView(input)
6         .setPositiveButton("OK", new DialogInterface.OnClickListener() {
7             @Override
8             public void onClick(DialogInterface dialog, int which) {
9                 String msg = input.getText().toString();
10                try{
11                    tConnected.write(msg.getBytes());
12                } catch (NullPointerException e) {}
13            }
14        });
15    dialog
16    .setNegativeButton("Cancel", new DialogInterface.OnClickListener() {
17        @Override
18        public void onClick(DialogInterface dialog, int which) {
19            dialog.cancel();
20        }
21    }).show();
22 }
23 
```

Algoritmo 14.11: Método `writeMessage()`

O método `showMessage()` mostra a mensagem recebida em um `Toast`.

```

1 public void showMessage(String msg){
2     Toast.makeText(this, msg, Toast.LENGTH_LONG).show();
3 }

```

Algoritmo 14.12: Método `showMessage()`

Com isso, podemos voltar e explicar a linha 29 do algoritmo 14.7 e a linha 28 do algoritmo 14.10. Essas são chamadas para o `Handler` da *UI thread*. Dessa forma, podemos chamar métodos que precisam estar na *UI thread*, mas fora dela. A chamada irá esperar até que a *UI thread* entre para executar.

```

1 private String mMessage;
2 private final Handler mHandler = new Handler();
3 private final Runnable mShowMessage = new Runnable() {
4     public void run() {
5         showMessage(mMessage);
6     }
7 };
8 private final Runnable mShowDialog = new Runnable() {
9     public void run() {
10        writeMessage();
11        tConnected.cancel();
12    }
13 };

```

Algoritmo 14.13: Handler e Runnable

Os Runnable estão encarregados de executar os dois métodos, para mostrar e enviar a mensagem. O Handler irá chamar o Runnable quando estiver na *UI thread*.

Agora é necessário implementar o método `onItemClickListener()` para abrir a conexão com o dispositivo selecionado.

```

1 @Override
2 public void onItemClick(AdapterView<?> arg0, View v, int pos, long id) {
3     String address = mAdapter.getItem(pos).split("\n")[1];
4     BluetoothDevice mDevice = mBluetoothAdapter.getRemoteDevice(address);
5
6     ConnectThread tConnect = new ConnectThread(mDevice);
7     tConnect.start();
8 }

```

Algoritmo 14.14: Implementação do método `onItemClickListener()`

Obtemos a *string* do adaptador e usamos `split` para separar somente o endereço MAC. Criamos uma `ConnectThread` para conectar ao dispositivo *Bluetooth*.

Por último iremos completar o método `onCreate()` (algoritmo 14.2), onde antes tinha `// [2]` complete com uma chamada à `AcceptThread`.

```

1 deviceList.setOnItemClickListener(this);
2
3 AcceptThread tAccept = new AcceptThread();
4 tAccept.start();

```

Algoritmo 14.15: Terceira parte do método `onCreate()`

Com isso, finalizamos a aplicação. É recomendável verificar o projeto completo no repositório caso ainda existam dúvidas. Qualquer tipo de dado em forma de um *array* de *bytes*

pode ser enviado através do canal de comunicação *Bluetooth*. Nesse exemplo estamos enviando apenas uma mensagem mas poderia ser qualquer tipo de arquivo.

Esse capítulo tentou mostrar uma simples aplicação que utiliza o *Bluetooth* no envio de dados. Hoje o *Bluetooth* está presente em diversos dispositivos: *smartphones*, fones de ouvido, caixas de som, aparelhos médicos, entre outros. O Android, por exemplo, utiliza o NFC para abrir uma conexão *Bluetooth* entre dois aparelhos e enviar dados. Isso é chamado *Android Beam*.

Internacionalização

É possível fazer com que sua aplicação dê suporte a diversos idiomas, o Android fornece uma maneira fácil de internacionalizar suas *strings* e *bitmaps*.

Antes, nós usamos o arquivo de recursos `strings.xml` para armazenar qualquer tipo de texto dos nossos exemplos. Dessa forma você consegue manter esse tipo de informação separada do código. Isso permite separar os textos por idioma, quando isso acontece o Android muda automaticamente o idioma do aplicativo de acordo com o idioma do aparelho.

Para adicionar suporte a mais idiomas, crie diretórios `values/` adicionais dentro do diretório `res/` que incluam um hífen e o código ISO do país no final. Por exemplo, `values-es/` para espanhol ou `values-pt/` para português. Dentro de cada uma dessas pastas você deve ter um arquivo `string.xml` com os textos traduzidos. Por exemplo, em `values/string.xml` você deve manter o idioma padrão da aplicação, preferencialmente inglês:

```
1 <?xml version="1.0" encoding="utf-8"?>
2 <resources>
3     <string name="title">My Application</string>
4     <string name="hello_world">Hello World!</string>
5 </resources>
```

Algoritmo 15.1: `strings.xml` padrão

Em `values-pt/strings.xml` você pode escrever os textos em português:

```
1 <?xml version="1.0" encoding="utf-8"?>
2 <resources>
3     <string name="title">Meu aplicativo</string>
4     <string name="hello_world">Oi Mundo!</string>
5 </resources>
```

Algoritmo 15.2: `strings.xml` em português

Outros recursos também podem aproveitar da localização, como é o caso dos `drawables`. Você pode ter imagens traduzidas dentro dessas pastas.

Os códigos do idioma seguem o padrão ISO-639-1¹.

Se você quiser separar por país, por exemplo entre português brasileiro e português de Portugal, você deve adicionar o código do país ao nome da pasta, precedido por um `-r`. Português do Brasil seria `values-pt-rBR/` e de Portugal seria `values-pt-rPT/`.

¹http://en.wikipedia.org/wiki/List_of_ISO_639-1_codes