02

# Copiando e Colando

Vamos aprender a dar comportamento às nossas views. Vimos na aula anterior que a classe R é responsável por fazer a ponte entre o código java (pasta src) e os recursos (pasta res).

É através dela que conseguimos trazer uma view do xml para o código Java de forma a podermos dar comportamento a view. Para tal, iremos usar também o método da Activity findViewById(...) que recebe um id e retorna uma `View`, a seguir um exemplo de utilização:

```
Button botaoDeOk = (Button) findViewById(R.id.botao);
```

Ao utilizar o `findViewById` normalmente estamos recuperando um objeto diferente de uma `View`, e neste caso teremos que fazer um cast do retorno para o objeto que estamos esperando.

Essa classe R é gerada pelo plugin do Eclipse e não deve ter seus valores alterados por nós. Ela será utilizada como referência para acessar os diversos componentes da nossa interface. É como uma grande coleção de identificadores, que a plataforma saberá utilizar para localizar seus valores e propriedades de maneira otimizada.

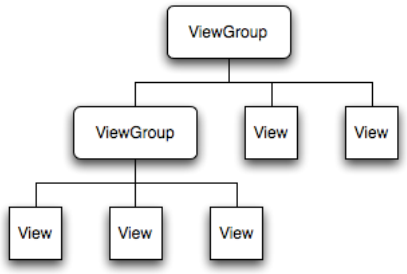
O `R.java` é gerado de acordo com os diversos arquivos utilizados para configuração da sua aplicação. Se houver algum erro nesses diversos arquivos, é capaz de que o plugin não consiga regerá-lo, dessa forma propagando um erro em quase todos os seus arquivos fonte, dada a ausência da classe R.

Devemos então ficar atentos e buscar o erro que está impedido sua geração, e tomar cuidado para não importar a classe `android.R`, que não é a da sua aplicação (no nosso caso `br.com.caelum.olamundo.R`), e sim uma de uso interno do Android.

Como você já sabe, todas as telas do Android são feitas num arquivo XML, que ficam dentro de `res/layout`. Lá configuramos os detalhes de cada componente, que são itens gráficos.

Dentro do XML, cada componente declarado poderá ser manipulado em código Java. Todos esses componentes são filhos de `android.view.View`. Esses Views serão futuramente agrupados dentro de `ViewGroup`. Não coincidentemente, `ViewGroup` é filha de `View`, formando o composite pattern que aparece também no Swing, onde Container é filha de Component.

Um `ViewGroup` pode conter `Views` e inclusive `ViewGroups` conforme imagem a seguir:



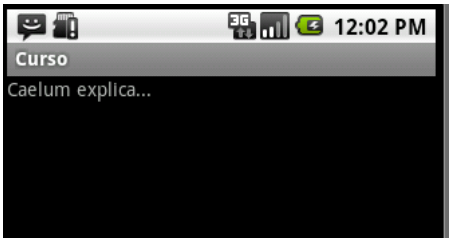

Algumas pessoas vão chamar esses componentes de widget, porém há um outro conceito no android, que é uma pequena aplicação que pode rodar dentro da home do dispositivo móvel, também chamado de widget.

Durante o curso vamos apresentar alguns dos componentes mais usados em aplicações Android. No momento vamos apresentar algumas importantes propriedades que podem ser configuradas nesses componentes.

Precisamos também aprender a vincular comportamento aos componentes da tela, para que possamos interagir com o usuário da aplicação.

No momento vamos focar nos componentes mais básicos:

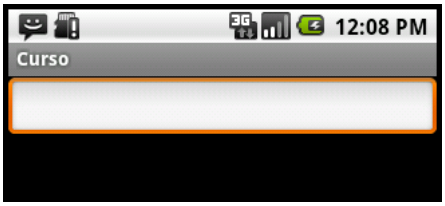
**TextView:** Serve para escrevermos um texto na tela do Android, como um label.




```
<TextView
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:text="Caelum explica..."
/>
```

Cada componente possui diversas propriedades. Aliás, `layout_width` e `layout_height` são obrigatórios para todos eles. Você poderia utilizar uma medida específica, como em pixels, mas não é recomendado. As opções relativas são mais frequentes, como o **wrap_content** e **match_parent** (este último era chamado de **fill_parent** antes da versão 2.2, level 8) aqui utilizados, e veremos mais adiante o significado das principais opções.

**EditText:** Serve como campo de edição para o usuário do sistema.



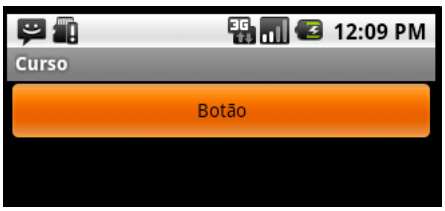

```
<EditText
    android:id="@+id/cidade"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
/>
```

Para ler o valor do campo, deve-se invocar getText e depois toString:

```
EditText cidade = (EditText) findViewById(R.id.cidade);
Log.i("Meu Texto: ", cidade.getText().toString());
```

`Log.i` é a forma de utilizarmos o DDMS, o sistema de gerenciamento da máquina Dalvik. Esse log será guardado em uma localização específica do seu dispositivo, e pode ser facilmente visto através do emulador.

**Button:** Esta view é um botão como na web ou em sistemas desktop.




```
<Button
    android:id="@+id/botaoConfirmar"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:text="Botão"
/>
```

Para usarmos na lógica, devemos buscar o botão e adicionar o listener desejado. Esse listener será disparado quando houver um clique no botão. Para fazer isso, precisamos de uma classe que implemente a interface interna `View.OnClickListener` :

```
class BotaoDeConfirmarListener implements View.OnClickListener {
    public void onClick(View view) {
        // pega os dados da Activity e
        // grava os dados na base SQL
    }
}
```

Dentro de nossa activity, podemos registrar uma instância desse listener para o botão:

```
Button cadastrar = (Button) findViewById(R.id.botaoConfirmar);
cadastrar.setOnClickListener(new BotaoDeConfirmarListener()));
```

A classe `BotaoDeConfirmarListener` irá precisar acessar algumas informações da tela após ser clicada. Mas como a instância de `BotaoDeConfirmarListener` irá pegar os dados dos campos que estão na activity?

Uma hipótese seria ter diversos getters na Activity e passar `this` como argumento para o construtor de `BotaoDeConfirmarListener`.

Ou então já definir no construtor parâmetros de forma a passarmos apenas os itens necessários para que o listener funcione, por exemplo um `TextView`, ou outro campo da tela.

Apesar das duas possibilidades, a forma mais comum é utilizar classes anônimas:

```
Button cadastrar = (Button) findViewById(R.id.botaoConfirmar);

cadastrar.setOnClickListener(new View.OnClickListener() {
    public void onClick(View view) {
        // pega os dados da Activity
        // (agora que temos acesso a todos componentes, fica mais simples)
        // grava os dados na base SQL
    }
});
```

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

Se quiser um alerta que precise de uma confirmação do usuário, uma opção é utilizar o `AlertDialog`.

```
AlertDialog.Builder builder = new AlertDialog.Builder(Aplicacao.this);
builder.setMessage(mensagem);
builder.setNeutralButton("OK", null);

AlertDialog dialog = builder.create();

dialog.setTitle(titulo);
dialog.show();
```

O código acima irá produzir uma mensagem como a seguinte:


Utilizado frequentemente para confirmar operações, o alerta pode ser muito customizado, e em vez de apresentar um texto, pode até mesmo apresentar uma view customizada dentro de seu espaço.

Componentes possuem atributos específicos a seu comportamento, mas muitos deles possuem atributos que são comuns e que aparecem com frequência. Já vimos alguns deles, outros são:

`android:id`: Especifica a identificação da View.

`android:layout_width`: Especifica a largura. Pode ser usado `match_parent` (todo o espaço), `wrap_content` (o suficiente) ou dp (Density-Independent Pixel).

`android:layout_height`: Especifica a altura. Pode ser usado `match_parent` (todo o espaço), `wrap_content` (o suficiente) ou dp (Density-Independent Pixel).

`android:text`: Essa propriedade serve para mostrar o texto que é passado como parâmetro.

`android:textColor`: Essa propriedade serve para definir uma cor para o texto exibido.

`android:background`: Essa propriedade serve para definir uma cor de fundo.

`android:textStyle`: Essa propriedade serve para definir um estilo a fonte (negrito e/ou itálico).

`android:textSize`: Essa propriedade serve para definir o tamanho da fonte. O tamanho da fonte pode ser especificado em várias notações: px(pixels), sp(scaled-pixels), mm(milímetros), in (inches) e etc.

`android:typeface`: Essa propriedade serve para definir uma fonte ao texto (Arial , Times NewRoman, Courier New e etc).

`android:capitalize`: Essa propriedade serve para definir o tipo capitalização das palavras. Por padrão, o valor e "none"(nenhum).

`android:password`: Com essa propriedade você habilita a digitação de senhas.

`android:visibility`: Define se o componente deve ser visível desde a primeira aparição dessa tela.

`android:inputType`: Define o teclado para a inserção de dados. Entrou no lugar de algumas propriedades que foram depreciadas.

## Nome dos componentes

Cada view que receber um id deve especificá-lo seguindo o padrão de nomenclatura de id's **@+id/**.

Quando você identifica o id de uma view, ele é compilado e aparece na classe `R.java` . Por exemplo, se o id for **@+id/botao_confirmacao**, na identificação da sua lógica, apareceria **R.id.botao_confirmacao**.

Outra abordagem à criação de listeners é configurar no xml da view um método para ser invocado na Activity relacionada a ela. Fazemos isso colocando o nome do método a ser executado no atributo `onClick` da tag `Button`:

```
<LinearLayout...>

    <EditText ...>

    <Button
        android:text="Copiar texto"
        android:id="@+id/botao"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:onClick="imprime" />

    <TextView ...>
</LinearLayout>
```

Agora basta implementarmos na Acivity um método chamado imprime que recebe uma `View`.

```
public class OlaMundoActivity extends Activity {
    //... outros métodos

    public void imprime(View view) {
        Log.i("Informação:", "Botao clicado!");
    }
}
```

## Posicionando as Views

Quando definimos um arquivo de layout, jogamos dentro dele todos os componentes que gostaríamos de ter na tela. Uma questão que fica é:

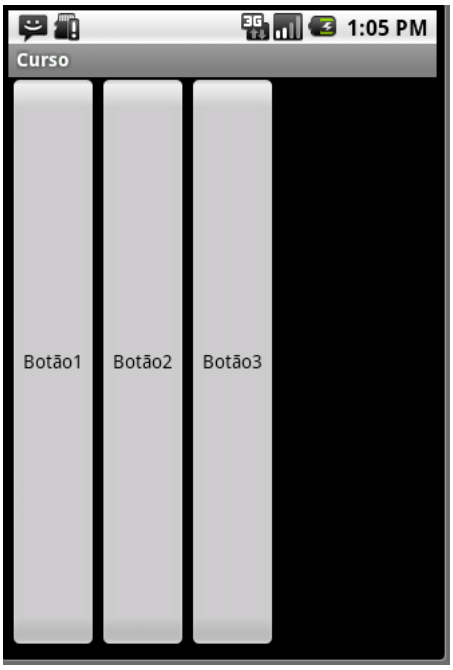
**Como esses componentes estarão dispostos?**

Apesar deles possuírem diretivas em relação a sua altura e largura, quem vem primeiro? Quem estará próximo do outro? Como definir com mais exatidão esse posicionamento?

Assim como o **Swing** (API do Java para aplicações desktop) possui seus `LayoutManagers`, o Android possui classes filhas de `ViewGroup` que agem como os *layout managers*. Eles definirão como seu layout será aplicado aos diversos componentes ( `Views` ) contidos nele.

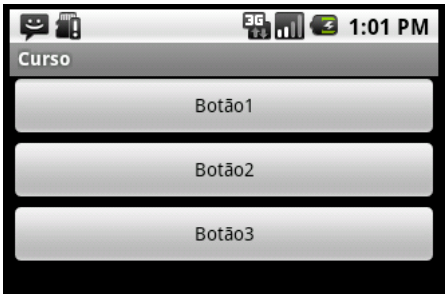
Um dos mais usados é o LinearLayout. Este layout é utilizado para alinhar horizontalmente ou verticalmente o conteúdo da tela do Android.

No modo horizontal podemos ter um resultado estranho:



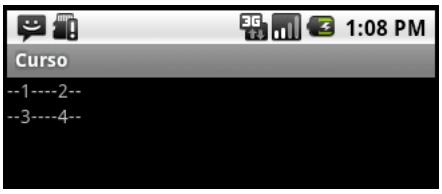
```
<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="horizontal"
    android:layout_width="match_parent"
    android:layout_height="match_parent">
    :
</LinearLayout>
```

O modo vertical é o mais usado:




```
<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="match_parent"
    android:layout_height="match_parent">
    :
</LinearLayout>
```

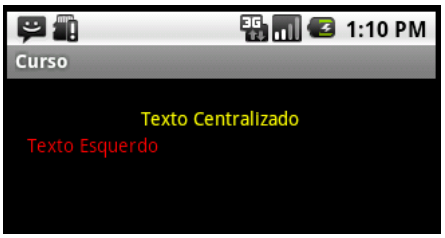
O `TableLayout` é utilizado para dividir a tela em células. Ele monta uma tabela na tela com diversas linhas ( `TableRow` ). Nestas linhas, as colunas são a quantidade de views nelas colocadas.




```
<TableLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent">

    <TableRow>
    :
    </TableRow>
    <TableRow>
    :
    </TableRow>
    <TableRow>
    :
    </TableRow>
</TableLayout>
```

O `AbsolutLayout` é utilizado para definir exatamente as posições na tela. Esse layout foi depreciado, a Google recomenda que não usemos em produção.




```
<?xml version="1.0" encoding="utf-8"?>
<AbsoluteLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
```

```
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        >
        <TextView
            android:text="Texto Centralizado"
            android:textColor="#FFFF00"
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            android:layout_x="100px"
            android:layout_y="20px"
        />
        <TextView
            android:text="Texto Esquerdo"
            android:textColor="#FF0000"
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            android:layout_y="40px"
            android:layout_x="15px"
        />
    </AbsoluteLayout>
```

O `RelativeLayout` tem sido cada vez mais usado para a organização das telas do dispositivo por permitir que os componentes sejam alocados na tela com relação à outros componentes ou à própria tela. Com este layout podemos usar algumas tags para fazer este alinhamento. Segue uma pequena lista:

android:layout_alignTop="@+id/item" - Alinha o topo do componente em relação a outro

android:layout_alignLeft="@+id/item" - Alinha a esquerda do componente em relação a outro

android:layout_toLeftOf="@+id/item" - Alinhado à esquerda do componente

android:layout_toRightOf="@+id/item" - Alinhado à direita do componente

android:layout_below="@+id/item" - Alinha abaixo de outro componente

android:layout_centerHorizontal="true" - Alinhado no centro horizontal

android:layout_centerVertical="true" - Alinhado no centro vertical

android:layout_alignParentTop="true" - Alinhado em cima da tela

android:layout_alignParentBottom="true" - Alinhado em baixo da tela

android:layout_alignParentLeft="true" - Alinhado à esquerda

android:layout_alignParentRight="true" - Alinhado à direita

Mas lembre-se que o modo de visualização *Graphical Layout* vai te ajudar a fazer esta organização. A dica de ouro é colocar id's adequados aos seus componentes.

Além do que vimos, ainda temos:

**FrameLayout -** Praticamente sem layout. As Views ficam umas em cima das outras. As vezes é o que precisamos para a nossa tela, como quando vamos montar um video com legendas.

**GridView -** Serve para você mostrar diversas views numa mesma tela

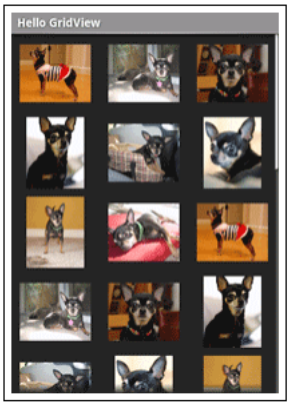
**TabLayout -** Como o nome diz, possibilita a colocação de abas na nossa aplicação

Linear Layout

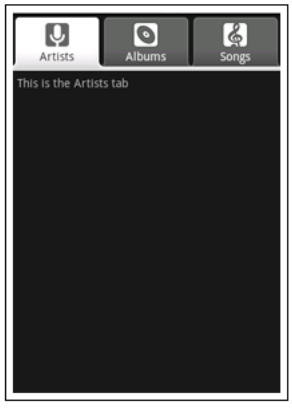

Relative Layout


Table Layout

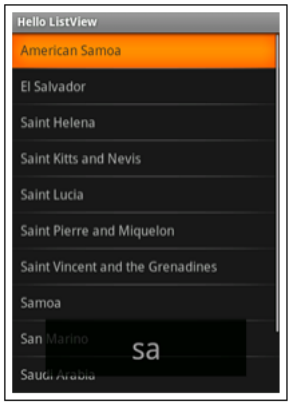

Grid View




Tab Layout




List View




É comum misturarmos diferentes layouts para fazer telas um pouco mais complexas, um único layout nem sempre é o suficiente. Mas cuidado pois muitos aninhamentos podem tornar sua aplicação menos performática.

Um importante recurso de layout no Android é o atributo `layout_gravity`. Através dele podemos influenciar no posicionamento de nossas `Views`. Se criarmos um texto dentro de um `FrameLayout` ele ficará no canto superior esquerdo por padrão.


```
<FrameLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent">

    <TextView
        android:layout_width="wrap_content"
```

```
        android:layout_height="wrap_content"
        android:text="Testando Layouts"
        android:textSize="30sp"
        />

</FrameLayout>
```

Vamos agora customizar o posicionamento do `TextView` . Se quisermos que ele seja um título e apareça centralizado no topo da tela podemos utilizar o `layout_gravity` com o valor `center_horizontal` , conforme código abaixo:


```
<FrameLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent">

    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Testando Layouts"
        android:textSize="30sp"
        android:layout_gravity="center_horizontal"
        />

</FrameLayout>
```

O resultado está representado na imagem abaixo:

Podemos também determinar que nosso TextView deve aparecer no centro vertical, usando o valor center_vertical para nosso layout_gravity.

```
<FrameLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent">

    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Testando Layouts"
        android:textSize="30sp"
        android:layout_gravity="center_vertical"
        />

</FrameLayout>
```

O resultado:


Repare que o `TextView` encontra-se no centro vertical, mas voltou a estar alinhado à esquerda. Se quisermos que ele esteja centralizado tanto vertical quanto horizontalmente podemos utilizar mais de um valor para o `layout_gravity` usando o caractere | (pipe) para concatenar os valores. Isso não é necessário para o caso do centro da tela já que existe o valor center. No exemplo abaixo demonstramos como colocar um `TextView` centralizado na tela e outro no canto inferior direito utilizando o `layout_gravity` .


```
<FrameLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent">

    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Testando Layouts"
        android:textSize="30sp"
        android:layout_gravity="center"
        />

    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Direita inferior"
```

```
        android:textSize="30sp"
        android:layout_gravity="bottom|right"
        />

</FrameLayout>
```

Quando criamos um elemento de nossa View podemos também estar preocupados em como posicionar seu interior. Suponha agora que desejamos colocar um texto na cor preta em um fundo branco. Esse elemento da nossa tela deverá estar dentro de um `FrameLayout` que por sua vez terá um fundo preto. Podemos customizar um `TextView` para que ele tenha essa aparência:

```
<FrameLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent">

    <TextView
        android:layout_width="250dp"
        android:layout_height="250dp"
        android:text="Com fundo"
        android:textSize="30sp"
        android:textColor="#000000"
        android:background="#FFFFFF"
        android:layout_gravity="center"
        />

</FrameLayout>
```

O código anterior vai gerar uma tela conforme a imagem a seguir:


Nosso `TextView` de fundo branco está centralizado na tela, porém no interior do `TextView` o texto encontra-se posicionado na parte superior esquerda. Para centralizarmos o texto no interior do `TextView` podemos usar o atributo `gravity` juntamente com o `layout_gravity` previamente utilizado.


```
<FrameLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent">

    <TextView
        android:layout_width="250dp"
        android:layout_height="250dp"
        android:text="Com fundo"
        android:textSize="30sp"
        android:textColor="#000000"
        android:background="#FFFFFF"
        android:layout_gravity="center"
        android:gravity="center"
        />

</FrameLayout>
```

Quando o assunto é layout, outro artifício do desenvolvedor Android é o uso do `layout_weight` .

Quando existe um espaço sobrando na tela, através da propriedade `layout_weight` podemos orientar o Android a dividir esse espaço aplicando um peso maior ou menor para os elementos da `View` . Ele funciona parecido com percentuais.

Para ilustrar, no código abaixo vamos colocar três `Button's` ocupando toda a largura da tela, porém a altura queremos dividir de tal maneira que o segundo botão tenha o dobro da altura do primeiro e o terceiro botão, por sua vez tenha o triplo da altura do primeiro. Queremos também que o conjunto dos três botões ocupem toda a nossa tela. Nosso objetivo encontra-se ilustrado na imagem abaixo:

Para atingir nosso objetivo vamos utilizar o `layout_weight` para dar peso à divisão do espaço na tela. Nosso primeiro botão terá peso 1, o segundo terá peso 2 e o terceiro terá peso 3, conforme o código abaixo:


```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical" >

    <Button
        android:text="Primeiro"
        android:layout_weight="1"
        android:layout_height="0dp"
        android:layout_width="match_parent"
        />

    <Button
        android:text="Segundo"
        android:layout_weight="2"
        android:layout_height="0dp"
        android:layout_width="match_parent"
        />

    <Button
        android:text="Terceiro"
        android:layout_weight="3"
        android:layout_height="0dp"
        android:layout_width="match_parent"
        />

</LinearLayout>
```

## Lint

Os **warnings** que encontramos nos códigos xml da nossa aplicação são colocados pelo **Lint**, uma ferramenta que verifica possíveis melhorias/erros na construção de layouts como:

. id duplicado

. layout duplicados

. layout não usado

. texto não externalizado no arquivo strings.xml

. tamanhos não externalizados no arquivo de dimens.xml

. API's de versões superiores usadas em versões inferiores

. dentre outros

É uma boa ferramenta para verificar se a construção de seu layout está seguindo as regras do Android.

O **Lint** está disponível somente em versões do plugin ADT superiores a versão 14. Para rodar o Lint, basta clicar no menu `Window -> Run Android Lint` .