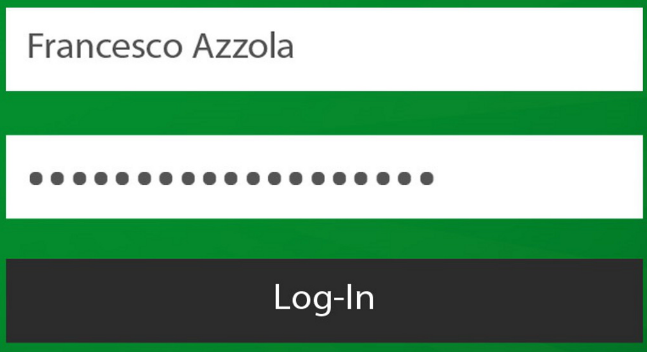
ANDROID UI DESIGN { BASICS }

Kick-start your Android User Interfaces



FRANCESCO AZZOLA

Java Code Geeks
JAVA 2 JAVA DEVELOPERS RESOURCE CENTER

Android UI Design

Contents

Preface

Android is an operating system based on the Linux kernel and designed primarily for touchscreen mobile devices such as smartphones and tablet computers. Android OS has taken over the world of mobile devices and is growing rapidly the last few years.

Android's user interface is based on direct manipulation, using touch inputs that loosely correspond to real-world actions, like swiping, tapping, pinching and reverse pinching to manipulate on-screen objects. The response to user input is designed to be immediate and provides a fluid touch interface.

In this course, you will get a look at the fundamentals of Android UI design. You will understand user input, views and layouts, as well as adapters and fragments.

Furthermore, you will learn how to add multimedia to an app and also leverage themes and styles. Of course, all these are tied up with a sample application, which you will build from scratch.

About the Author

Francesco Azzola is an electronic engineer, having achieved his degree at the University of Perugia (Italy). He has more than 15 years experience in architecting and developing JEE applications. He worked in the telecom area for several years architecting VAS service with JEE technology and developing mobile applications. In particular his experience and activities are focused in:

- Android software development
- JEE enterprise application
- Multichannel application (Web, Mobile, Voice channel)

He designed and implemented a java library for UCP protocol to exchange information with a SMS gateway. He also designed the Vodafone MAM java library to send and receive SMS. He designed and developed a GSM AT commands library to connect to a GSM external model, so that an application using this library can send and receive SMS and monitor the GSM modem status.

Francesco developed a SMS/GSM application in JEE, used by some italian mobile companies to support huge SMS/MMS traffic and to develop value added services. He worked on designing and developing a tool to analyze SIM Card GSM/UMTS in order to represent its file system in XML format. This tool was developed in Java/JEE technology and it can be used to verify sim card compliance to the specifications.

He designed and implemented a JEE enterprise application using all JEE technologies for an insurance company. He developed and implemented a jee library useful to support multichannel application that can be accessed using normal browser, mobile application (Android and so on) and by Voice (Voice recognition and Text to speech).

Francesco has also developed several Android apps some of them are published on Google Play.

He wrote several articles for Java.net regarding JavaFX, JBI (Java Business Integration) and SMS, JavaFX JEE integration using Spring and also some other articles for italian online magazine.

Lately he is working for a company that provides software solutions for public administration and in more detail my activities are focused on document management.

His areas of interest include mobile application development in Android, JEE architecture and OpenSource software that he used for developing some projects described above.

During his career he has received these certifications:

- Sun Certified Enterprise Architect (SCEA)
- Sun Certified Web Component Developer (SCWCD)
- Sun Certified Java Programmer (SCJP)

Chapter 1

Android UI Overview

1.1 Introduction

Android is a widely used OS made for smart phones and tablets. It is an open source project led by Google and it is released under Apache License. This permissive license helped this OS to be widely adopted and allows the manufacturers to freely modify and customize it. As matter of fact, despite Android being designed for smartphones and tablets, it is also used in TVs, cameras and so on. Moreover, Android has a very large community that extend its features and creates apps that cover almost all aspects.

All android applications, called apps, are built on Android UI framework. App interface is the first thing a user sees and interacts with. From the user perspective, this framework keeps the overall experience consistent for every app installed in our smartphone or tablets. At the same time, from the developer perspective, this framework provides some basic blocks that can be used to build complex and consistent user interface (API).

Android UI interface is divided in three different areas:

- **Home screen**
- **All apps**
- **Recent screen**

The **home screen** is the “landing” area when we power our phone on. This interface is highly customizable and themed. Using widgets we can create and personalize our “home” screen. **All apps** is the interface where the app installed are displayed, while **recent screens** are the list of last used apps.

Since its born, Android has changed a lot in terms of its features and its interfaces. The growth of the smartphone power made possible creating ever more appealing apps.

At the beginning, apps in Android did not have a consistent interface and well defined rules so every app had a different approach, navigation structure and buttons position. This caused user confusion and it was one of the most important missing features compared to the iOS.

1.2 Android App Structure and UI patterns

Android apps are very different from each other because they try to address different user needs. There are simple apps with a very simple UI that has just only one view and there are other apps much more complex with a very structured navigation and multiple views.

In general, we can say an Android app is made by a **top-level view** and **detail/level view**.

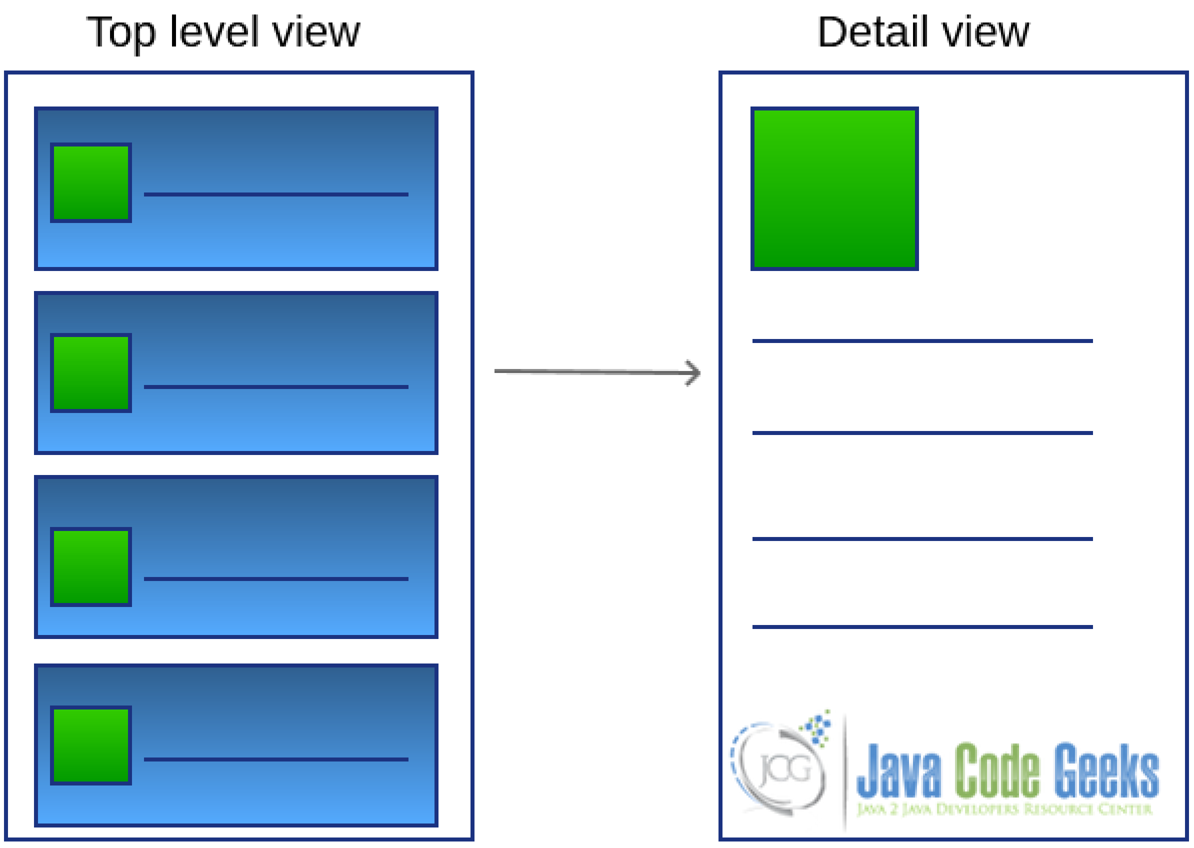


Figure 1.1: screenshot

One of the biggest efforts made by Google was to define a well defined set of rules that helps developers to create appealing user interfaces. At the same time, these rules help users to navigate through every app in the same way. We call this **UI consistency**. Android, moreover, guarantees to the developers the required flexibility to customize the app look and feel and make it unique. These rules are known as UI patterns. Patterns are proven solution approaches to well known problems. Thus, having a well defined UI pattern catalog and knowing when and where apply them, we can create appealing apps that are not only full of interesting features but they are enjoyable by users and easy to use.

1.2.1 Top level view

As said, the top level view is the “landing” area of our app, so we have to reserve to it a special attention, because this is the first thing an user sees of our app. There are some specific patterns that can be applied when designing this view depending on the type of information we want to show:

- **Fixed tabs**
- **Spinner**
- **Navigation drawer**

We have to choose one of them carefully depending on the nature of our app. We can use **fixed tabs** when we want to give to the user an overview of the different views present in our app, so that a user can switch easily between them to show different type of information. A typical example is a app that shows tech news: in this case we could use different tabs to group news like (‘Android’,*iOS*, ‘Games’ and so on).

Spinner is used when we want to move directly to a specific view, this is the case of a calendar app when we can use spinner to go directly to a specific month.

The **navigation drawer** is one of the newest patterns introduced by Google. This is a sliding menu, usually at the left side of the smartphone screen, that can be opened and closed by the user. This pattern can be used when we have a multiple top level view and we want to give to the user a fast access to one of them, or we want to give to the user the freedom to move to one low level view directly. This pattern replaces, somehow, an old pattern called dashboard widely used in the past. This pattern is simple a view where there are some big buttons/icons to access to specific views/app features.

1.2.2 Detail view

The detail view is a low level view where a user can interact with data directly. It is used to show data and edit them. In this kind of view the layout plays an important role to make data well organized and structured. At this level, we can implement an efficient navigation to improve usability of our app. In fact, we can use swipe view pattern so that user can move between different detail views. Depending on the type of component we use to show detail information to the user, we can implement some low level patterns that simplify the user interaction with our app.

1.2.3 Action Bar

The action bar is relatively new in Android and was introduced in Android 3.0 (API level 11). It is a well known pattern that plays an important role. An action bar is a piece of the screen, usually at the top, that is persistent across multiple views. It provides some key functions:

- App branding: icon area
- Title area
- Key action area
- Menu area

1.3 Standard components

How do we build an user interface? Android gives some key components that can be used to create user interface that follows the pattern we talked about before. All the Android user interface are built using these key components:

- **`View`** It is the base class for all visual components (control and widgets). All the controls present in an android app are derived from this class. A `View` is an object that draws something on a smartphone screen and enables an user to interact with it.
- **`Viewgroup`** A `ViewGroup` can contain one or more `Views` and defines how these `Views` are placed in the user interface (these are used along with Android Layout managers.
- **`Fragments`** Introduced from API level 11, this component encapsulates a single piece of UI interface. They are very useful when we have to create and optimize our app user interface for multiple devices or multiple screen size.
- **`Activities`** Usually an Android app consists of several activities that exchange data and information. An `Activity` takes care of creating the user interface.

Moreover, Android provides several standard UI controls, Layout managers and widgets that we can use without much effort and with which we can create apps fast and simply.

Furthermore, we can can extend them and create a custom control with a custom layout and behaviour. Using these four components and following the standard UI patterns we can create amazing apps that are “easy-to-use”. There are some other aspects, anyway, we have to consider when building and coding an app, like themes, styles, images and so on, and those will be covered in the following articles.

As said, Android provides some standard UI components that can be grouped in:

- **Tabs**

- **Spinners**
- **Pickers**
- **Lists**
- **Buttons**
- **Dialogs**
- **Grid lists**
- **TextFields**

The figure below shows some Android custom components:

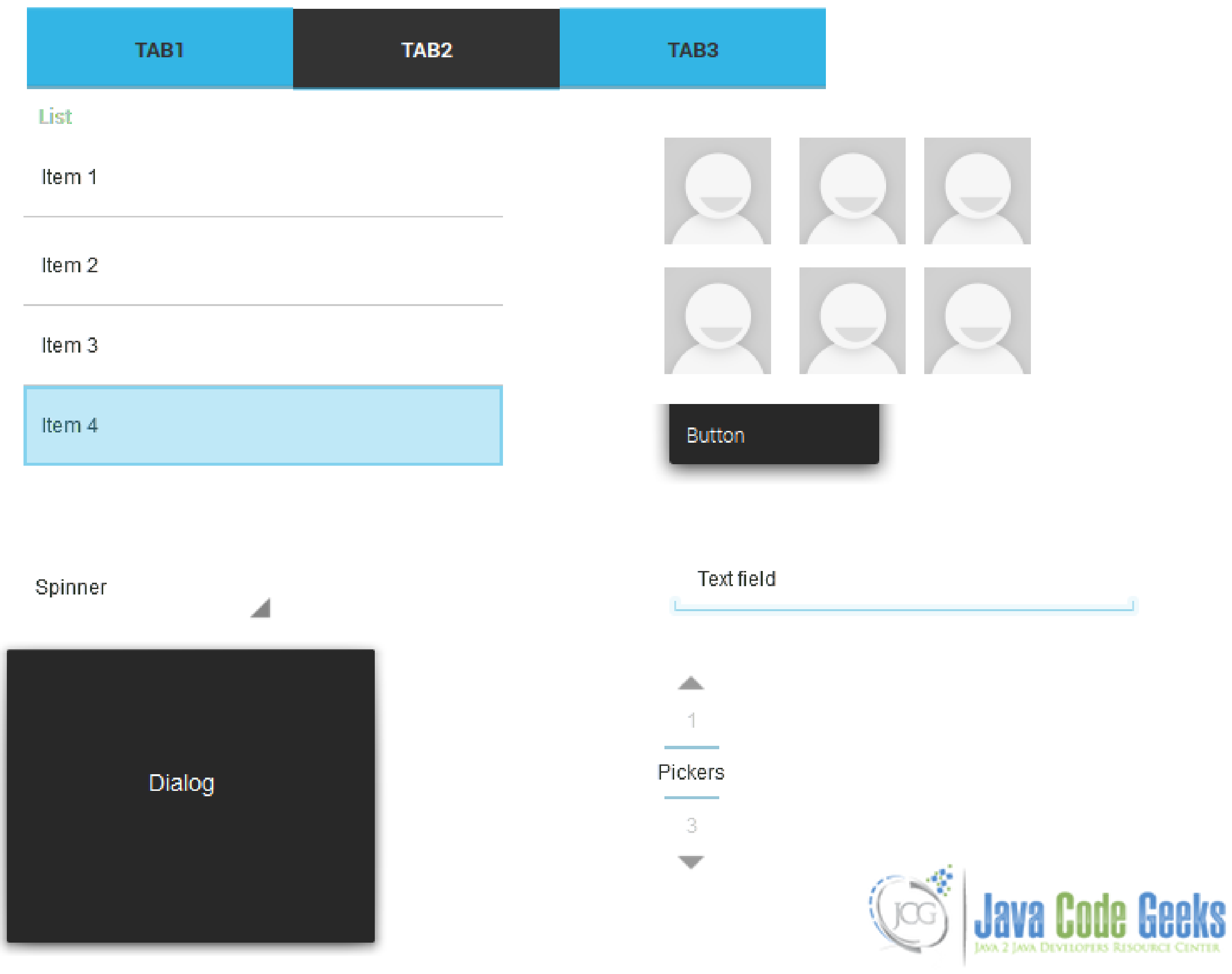


Figure 1.2: screenshot

If we analyze in more detail an Android user interface, we can notice that it has an hierarchical structure where at the root there's a `ViewGroup`. A `ViewGroup` behaves like an invisible container where single views are placed following some rules. We can combine `ViewGroup` with `ViewGroup` to have more control on how views are located. We have to remember that more complex is the user interface more time the system requires to render it. Therefore, for better performance we should create simple UIs. Additionally, a clean interface helps user to have a better experience when using our app.

A typical UI structure is shown below:

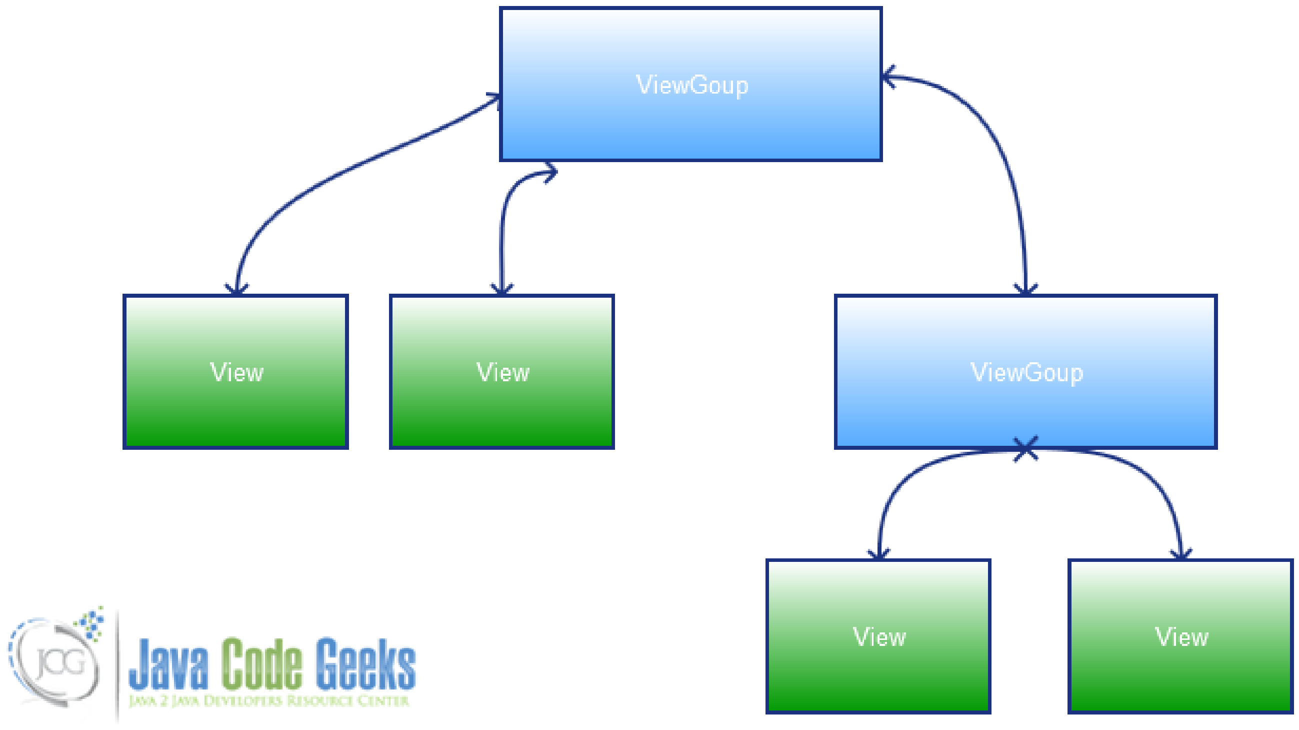
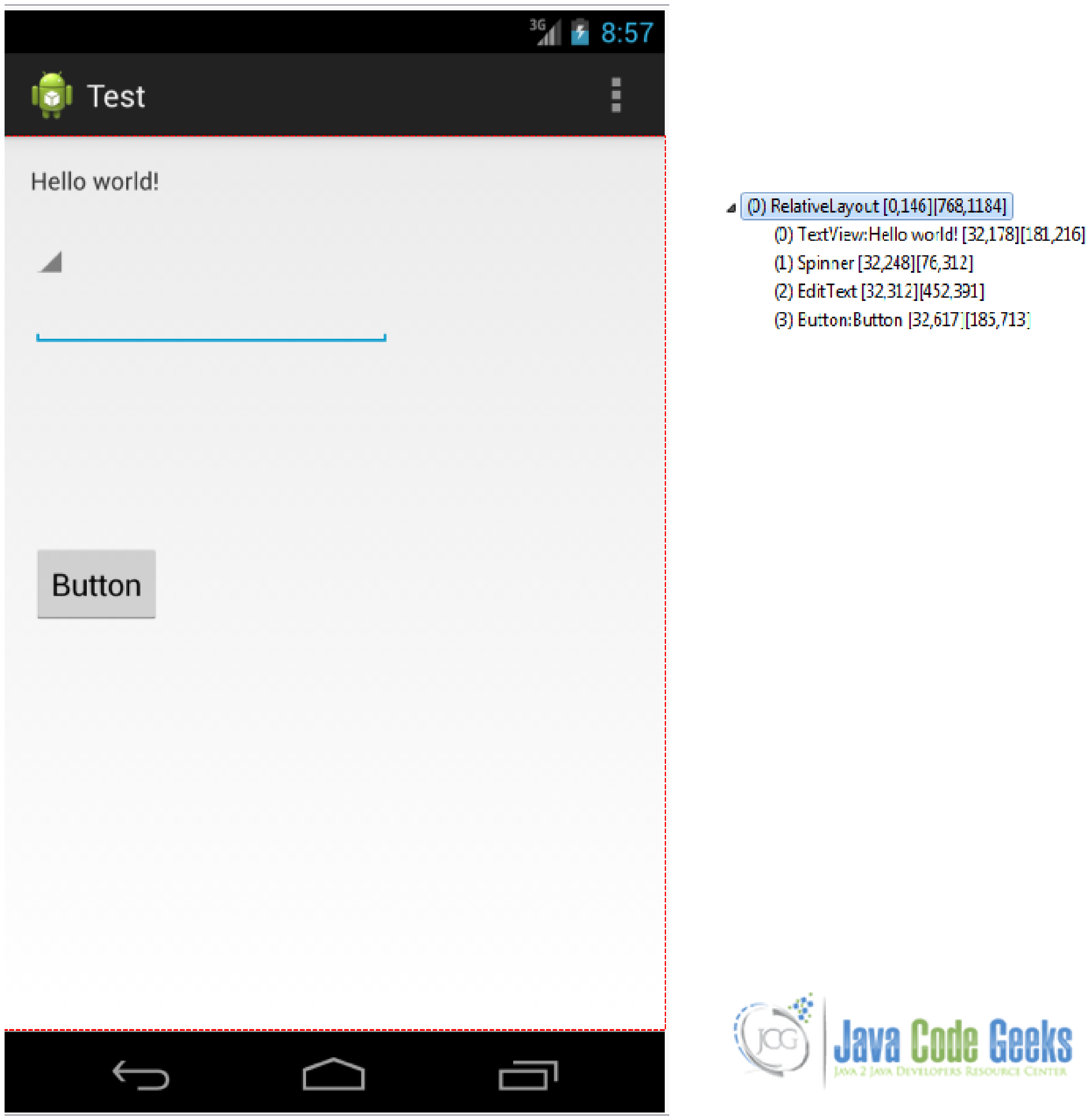


Figure 1.3: screenshot

If we create a simple android app, using our IDE, we can verify the UI structure:



```
(0) RelativeLayout [0,146][768,1184]
    (0) TextView:Hello world! [32,178][181,216]
    (1) Spinner [32,248][76,312]
    (2) EditText [32,312][452,391]
    (3) Button:Button [32,617][185,713]
```

Figure 1.4: screenshot

From the example above, we can notice that at the top of the hierarchy there is a `ViewGroup` (called `RelativeLayout`) and then there are a list of view child (controls and widgets).

When we want to create an UI in Android we have to create some files in XML format. Android is very powerful from this point of view because we can “describe” our UI interface just in XML format. The OS will then convert it in a real code lines when we compile our app and create the apk. We will cover this topic in a later article where we will describe how we can code a real Android UI using layouts and so on.

1.4 Multiple devices support

As we know by now, Android is a widely used system by smartphones and tablets. It is installed on many devices and it is a great opportunity for developers because it is possible to reach a wide audience. On the other side, this large number of devices that use Android is a big challenge for every Android developer. To provide a great user experience, we have to take under account that our app can run on variety of devices with different screen resolutions and physical screen sizes.

We have to consider that our app can run on a smartphone or on a tablet and we have to provide the same usability even if the differences in terms of screen resolution and size are big. Our app has to be so flexible that it can adapt its layout and controls depending on the device where it is installed on. Let's suppose, for example, that we have an app that shows a list of items and when the user clicks on one of them, the app shows the item detail. It is a very common situation; if our app runs on a smartphone we need to have two screens one for the list and one for the details as shown below:

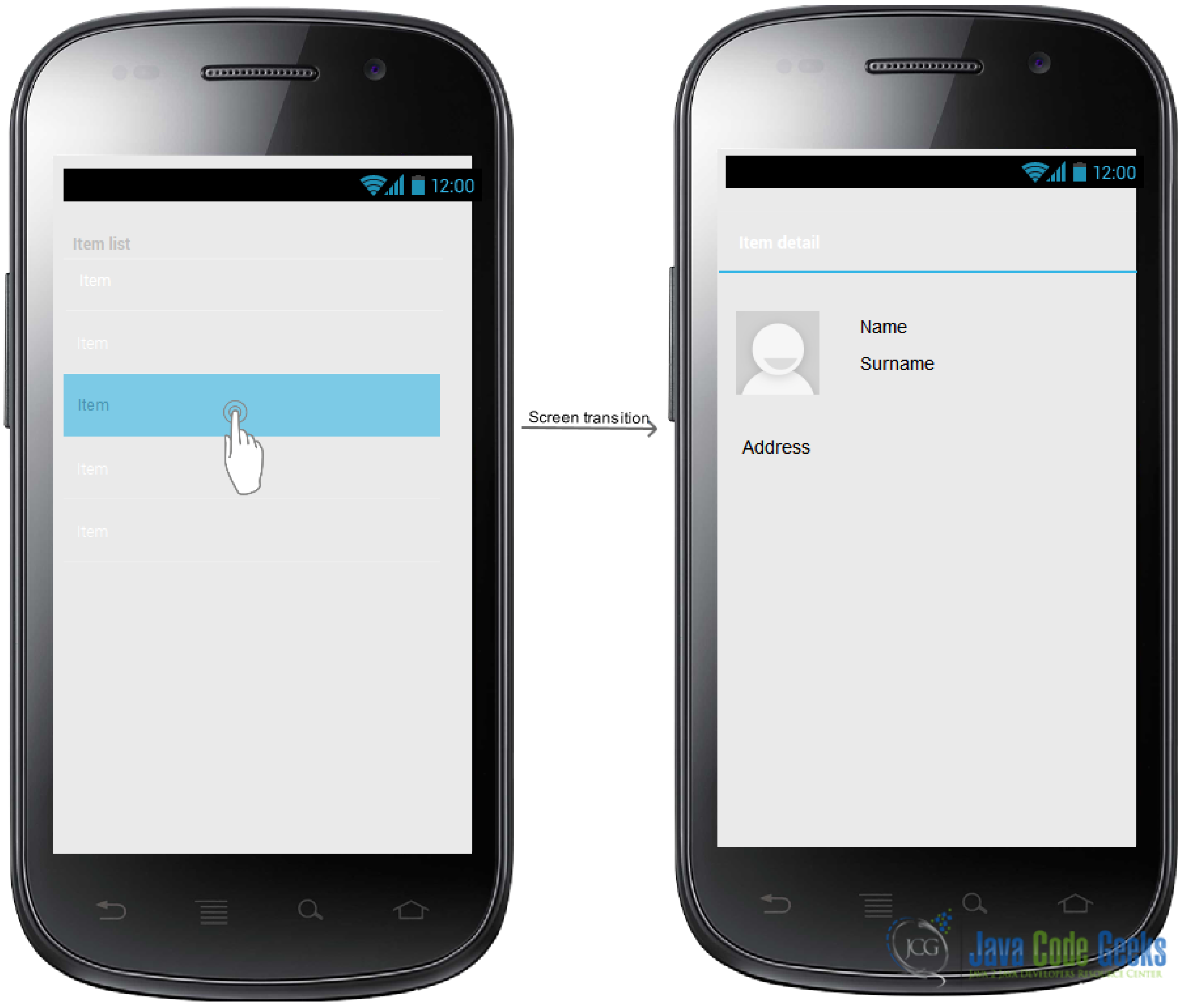


Figure 1.5: screenshot

while if our app runs on a tablet, it has to show the details on the same screen:

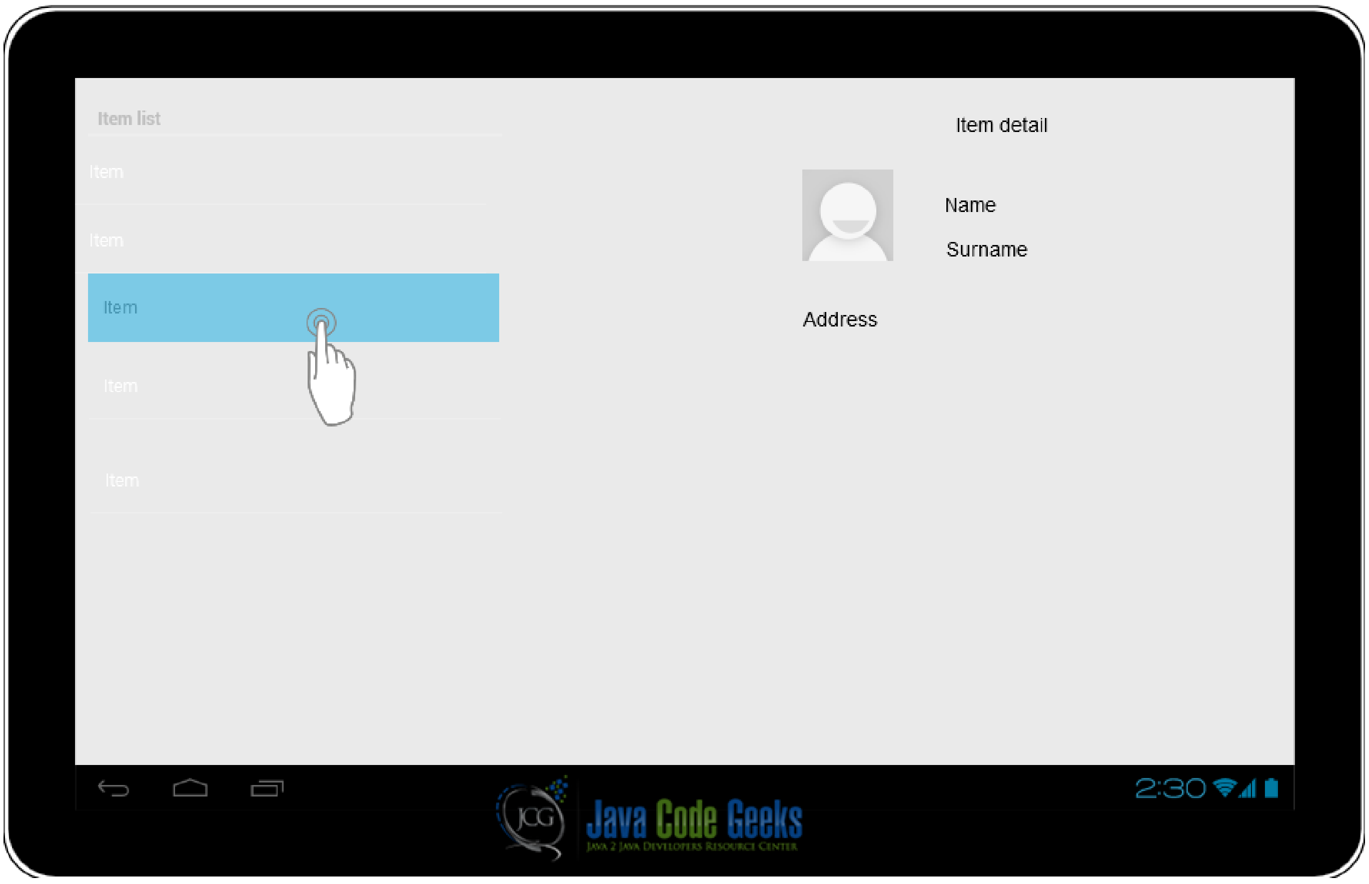


Figure 1.6: screenshot

Even if the system tries its best to scale and resize our app so that it works on different screen sizes, we have to make our best effort to ensure that our app supports several screens. This is a big challenge and we can win it if we follow some guidelines.

There are some key concepts we have to understand before coding our app:

- **Screen size** It is the physical screen or in other words, the real dimension of our device screen.
- **Density** It is the number of pixels in a given area. Usually we consider dot per inch (dpi). This is a measure of the screen quality.
- **Orientation** This is how the screen is oriented. It can be landscape or portrait.
- **Density independent pixel** This is a new pixel unit measure introduced by Android. It is called dp. One dp is equivalent at one pixel at 160dpi screen. We should use dp unit in our measures when creating UI, at the runtime the system takes care to convert it in real pixel.

From the screen size point of view, Android groups the devices in four areas,small, normal, large and extra large (xlarge), depending on the actual screen dimension expressed in inches. From the dpi point of view, on the other hand, we can group devices in: ldpi (low dpi), mdpi (medium dpi), hdpi (high dpi), xhdpi (extra high dpi) and lately xxhdpi. This is important when we use drawables (i.e bitmaps), because we have to create several images according to the different screen resolution.

This categorization is reflected in our IDE (i.e. Eclipse or Android Studio), so if we look under the `res` directory we can find a structure like the one shown below:

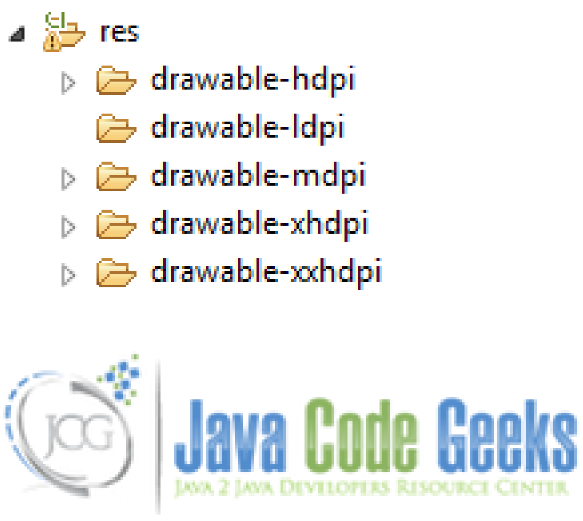


Figure 1.7: screenshot

The best practices we have to follow are:

- Don't use fixed dimensions expressed in pixel, instead we should use dp.
- Provide several layout structures for different screen size, we can do it creating several layout files.
- Provide several bitmap with different resolution for different screen resolution. We can consider using 9-Patches-Drawables.

Moreover, Android provides a resource qualifier mechanism that helps us to have more control on how the system select the resource. Usually we can use:

<resource_name>_<qualifier>

where the **resource_name** can be for example drawable or layout while the **qualifier** can be hdpi, large and so on. Look here if you want to have more information.

Even if we follow all the best practices to support multiple screen in our app, there might be some situations where they are not enough, especially when we want to support smartphones and tablets. In this case, Android has introduced the `Fragment` concept. `Fragments` are available since API 11 and there is a library to maintain the backward compatibility. We will cover these topics in the next articles. Stay tuned!

Chapter 2

Android UI: Understanding Views

2.1 Overview

In our previous chapter, we introduced some basic concepts about Android UI patterns and we saw how we can create consistent UI following those patterns. In this article, we want to explore more about `Views` and how we can use them to build great user interfaces. This article will cover concepts about `Views` and `Adapters`. Developing a user interface in Android can be quite simple since we can describe them using XML files and the system will do the heavy work converting those in real user interface components, placing them according to the screen size and resolution.

The Android UI framework provides some basic UI components, that are called controls or widgets, helping developers while coding and building an app.

2.2 Views

The `View` class is the basic class that all the components extend. A `View` draws something on a piece of screen and it is responsible to handle events while user interacts with it. Even the generic `ViewGroup` class extends `View`. A `ViewGroup` is a special `View` that holds other views and places these views following some rules. We will see that Android provides some specialized views that helps us to handle text, images, user inputs, buttons and so on.

All these views have some key aspects in common:

- **All views have a set of properties:** These properties affect the way the view is rendered. There is a set of properties common to all views, while there are some other properties depending on the type of view.
- **Focus:** The system manages the focus on each view and depending on the user input, we can modify and force the focus on a specific view.
- **Listeners:** All views have listeners which are used to handle events when the user interacts with the view. We can register our app to listen to specific events that occur on a view.
- **Visibility:** We can control if a view is visible or not and we can change the view visibility at runtime too.

A view property is a key value pair that we can use to customize the view behavior. The property values can be:

- a number
- a string
- a reference to a value written somewhere else

The first two options are trivial, the third is the most interesting one because we can create a file (always in XML) that holds a list of values and we can reference it in our property value. This is the best approach to follow especially when we use themes and styles or we want to support multi-language apps.

One of the most important property is **view id**: this is a unique identifier for the view and we can look up a specific view using this id. This is a "static" property meaning we cannot change it once we have defined it.

When we want to set a view property, we have two ways to do it:

- using XML while we define our view
- programmatically

For example, let's suppose we want to define a `TextView` (it writes a text). In XML we have:

```
<TextView
    android:id="@+id/textView1"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="@string/hello_world" />
```

As you can notice, we defined an id called textView1, some other properties that have a string values and another property called android:text that reference a value written somewhere else.

We could do the same thing using just code lines:

```
TextView tv = new TextView(this);
tv.setText("blabla");
```

These two approaches are equivalent; however, the first one in XML is the preferred one. There is a correspondence between XML view property and the Java method: usually for each XML property there is a set method that we can use in our code. You can look here to have more information about this correspondence. In XML, a view property is called attribute.

Two properties that play an important role are `layout_width` and `layout_height`. These two properties define how large and how tall should be the view. We can use two predefined values:

- `MATCH_PARENT`
- `WRAP_CONTENT`

With `MATCH_PARENT` value, we mean that we want our view as big as its parent that holds it, while with `WRAP_CONTENT` we specify that our view must be big enough to hold its content.

There is another option: using a numeric value. In this case, we specify the exact measure of our view. In this case, the best practice suggests using dp unit measure so that our view can be adapted to different screen density.

Talking about view listeners, this is a well-known approach when we want to be notified about some events. A listener in Java is an interface that defines some methods, which our listener class must implement so that it can receive notifications when a specific event occurs.

The Android SDK has several listener types, so different interfaces can be implemented. Two of the most popular are: `View.OnClickListener`, `View.OnTouchListener` and so on.

Android provides several standard components and of course if we cannot find a component that fulfills our needs, we can always implement a custom view. Using a custom view, we can define what our view will draw on the screen and its behaviour. We can even define custom properties that affect the view's behavior. In this case, we have to extend the basic `View` class and override some methods.

2.2.1 TextView component

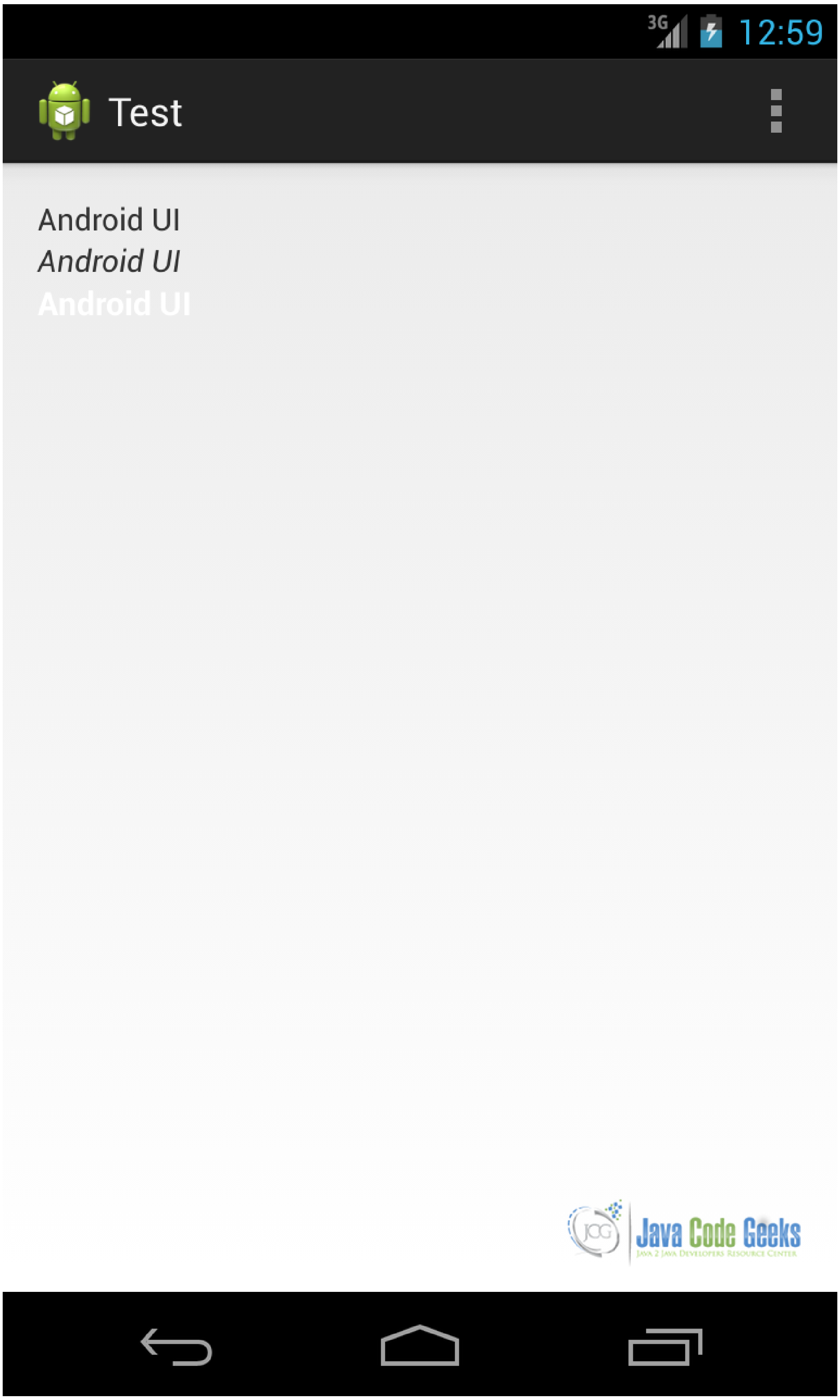
This is one of the simplest components. We use it when we want to show a text on the screen. We have seen it before:

```
<TextView
    android:id="@+id/textView1"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="@string/hello_world" />
```

where `android:text` holds the text we want to show. It has some attributes we can use to specify how we want to show the text:

- `android:fontFamily` - It is the font-family.
- `android:shadowColor` - It is the shadow color.
- `android:shadowDx` - It is the shadow x axis offset.
- `android:shadowDy` - It is the shadow y axis offset.
- `android:textStyle` - It is the style (bold, italic, bolditalic).
- `android:textSize` - It is the text size.

Below are some examples of `TextView`:



Android UI
Android UI
Android UI

Figure 2.1: screenshot

and the corresponding XML is:

```
<TextView
    android:id="@+id/textView1"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="Android UI" />
<TextView
    android:id="@+id/textView2"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="Android UI"
    android:textStyle="italic" />
<TextView
    android:id="@+id/textView2"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="Android UI"
    android:textColor="#FFFFFF"
    android:textSize="15dp"
    android:textStyle="bold" />
```

2.2.2 ImageView component

This component is used to show an image on the screen. The image we want to show can be placed inside our apk or we can load it remotely. In the first case, our image has to be placed under the `res/drawable` directory. Base on what we discussed in our previous chapter, we already know we have to keep in mind that our app can run on multipe devices with different screen density.

To better support different screen densities, we can create different images with different resolutions. The `res/drawable` directory is the default directory where the system looks if it cannot find an image with the right density for the screen. Generally speaking, we have to create at least four different image with different resolutions, in fact we can notice in our IDE that there are at least five directories:

- `drawable-ldpi` (not supported any more)
- `drawable-mdpi` (medium dpi)
- `drawable-hdpi` (high dpi)
- `drawable-xhdpi` (extra-high dpi)
- `drawable-xxhdpi` (x-extra-high dpi)
- `drawable`

An important thing to remember is the following: the images must have the same name. Once we have it, we can use the `ImageView` in this way:

```
<ImageView
    android:id="@+id/img"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:src="@drawable/robot" />
```

where we referenced the image, called robot.png, using `@drawable/robot` (without the extension). The result is shown below:

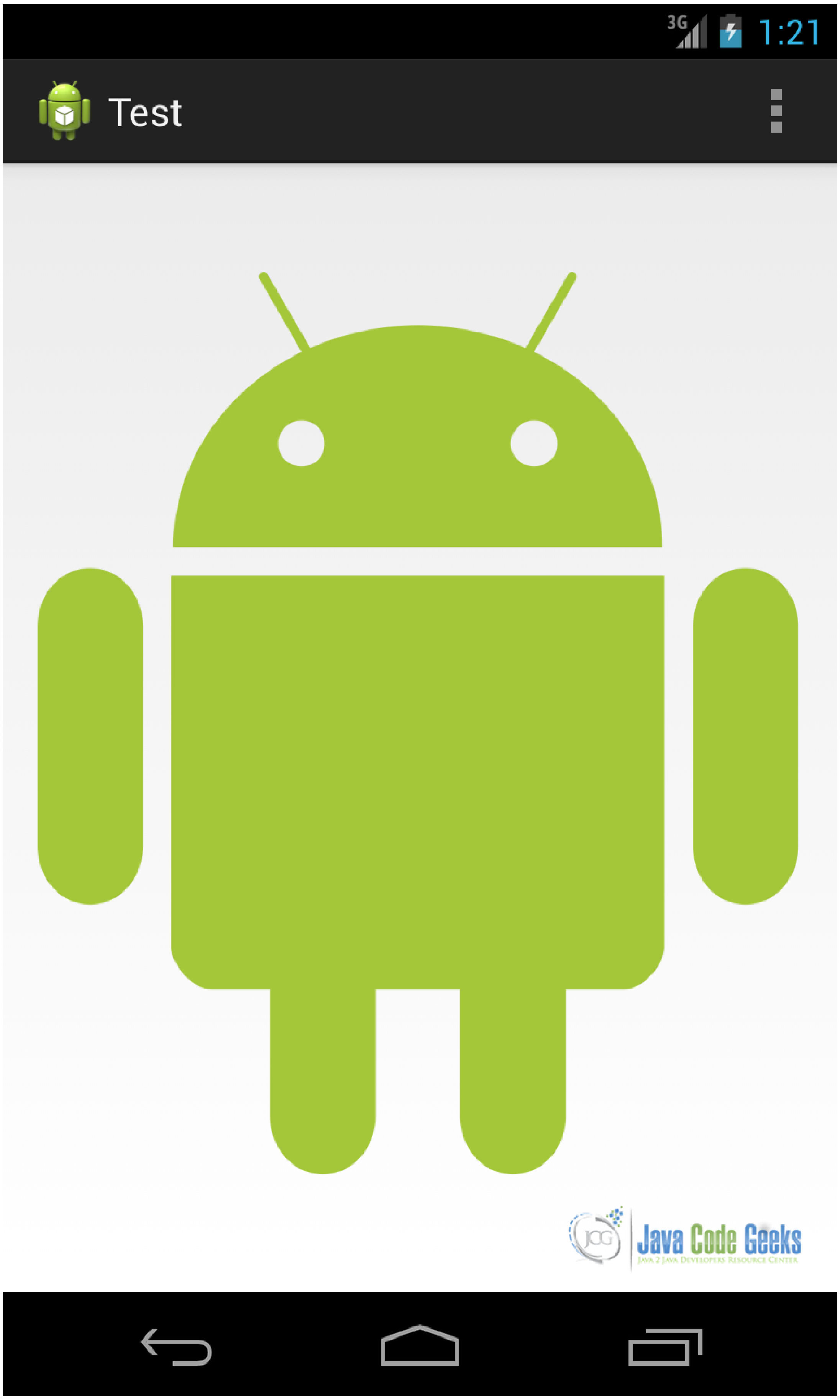


Figure 2.2: screenshot

If the image has to be loaded remotely, there are some considerations to keep in mind while doing it. First, you should consider that loading an image from a remote server is a time consuming operation and you should do it in a separate thread so that you can avoid ANR (Application Not Responding) problem. If you want to keep things simple you could use an `AsyncTask` or you can use the Volley lib. The second thing to remember is that you cannot set the image in an XML file, but you can define only the `ImageView` without referencing an image. In this case, you can set the image using:

```
img.setImageBitmap(your_bitmap)
```

where your_bitmap is the image you have loaded remotely.

2.2.3 Input controls

`Input controls` are components that the user can interact with. Using a component like that, you can, for example, give to the user the chance to insert some values. The Android UI framework provides a wide range of input controls: `text field`, `input field`, `toggle buttons`, `radio buttons`, `buttons`, `checkbox`, `pickers` and so on. Each of them has a specialized class and we can build complex interfaces using these components. There is a list of common components, below, with the class that handles it:

Table 2.1: datasheet

Component	Description	Class
`Button`	This one of the most used components. It can be pressed by a user and when pressed we can launch an action.	Button
`TextFields`	This is an editable text and we give to the user the chance to insert some data. `EditText` is the classic input field while `AutoCompleteTextView` is a component we can use when we have a pre-defined list of result and we want the system to complete some words inserted by user	EditText AutoCompleteTextView
`CheckBox`	This is an on/off component. We can use it when we want user to select one or more choices.	CheckBox
`Radio buttons`	Very similar to the checkbox except for the fact that the user can select only one item.	RadioGroup RadioButton
`Toggle button`	On/off component with a state indicator.	ToggleButton
`Pickers`	Component that helps the user to select one value using up/down buttons or gesture. Usually we use `DatePicker` and `TimePicker`	DatePicker TimePicker

We want to focus our attention on a `Button` component that this is widely used in Android UI. First, we have to define our UI using XML file:

```
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:paddingBottom="@dimen/activity_vertical_margin"
    android:paddingLeft="@dimen/activity_horizontal_margin"
    android:paddingRight="@dimen/activity_horizontal_margin"
    android:paddingTop="@dimen/activity_vertical_margin"
    tools:context=".MainActivity" >

    <Button
        android:id="@+id/Btn"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="Click here!" />

</RelativeLayout>
```

Notice that in the button id we used `@+id` meaning we are adding a new id and the `android:text` attribute is the string we want to show on the button. You could use a reference instead of writing text directly. If we create a simple app using this layout and run it we will have:

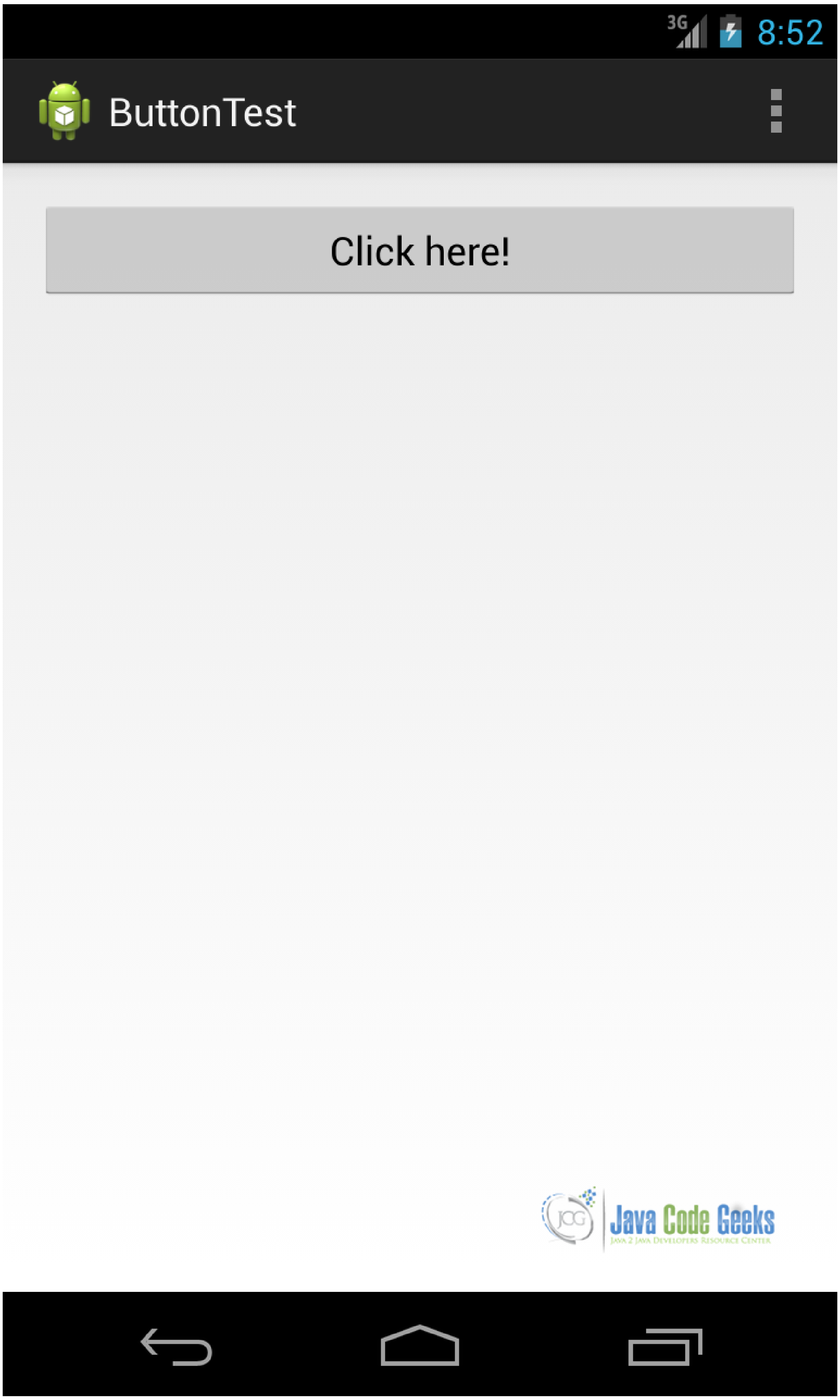


Figure 2.3: screenshot

We will see later how we can catch UI events so that we can handle user clicks. We can consider a bit more complex interface:

```
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:paddingBottom="@dimen/activity_vertical_margin"
    android:paddingLeft="@dimen/activity_horizontal_margin"
    android:paddingRight="@dimen/activity_horizontal_margin"
    android:paddingTop="@dimen/activity_vertical_margin"
    tools:context=".MainActivity" >

    <Button
        android:id="@+id/Btn"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="Click here!" />

    <CheckBox
        android:id="@+id/checkBox1"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_alignLeft="@+id/Btn"
        android:layout_below="@+id/Btn"
        android:layout_marginTop="29dp"
        android:text="CheckBox" />

    <ToggleButton
        android:id="@+id/toggleButton1"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_alignLeft="@+id/checkBox1"
        android:layout_below="@+id/checkBox1"
        android:layout_marginTop="20dp"
        android:text="ToggleButton" />

    <EditText
        android:id="@+id/editText1"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_alignLeft="@+id/toggleButton1"
        android:layout_centerVertical="true"
        android:ems="10"
        android:hint="Your name here" >

        <requestFocus />
    </EditText>

    <Switch
        android:id="@+id/switch1"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_alignBaseline="@+id/toggleButton1"
        android:layout_alignBottom="@+id/toggleButton1"
        android:layout_alignParentRight="true"
        android:text="Switch" />

    <RadioGroup
        android:id="@+id/radioGroup"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_alignLeft="@+id/editText1"
        android:layout_below="@+id/editText1"
        android:layout_marginTop="24dp" >

        <RadioButton
            android:id="@+id/radioButton1"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_alignLeft="@+id/editText1"
            android:layout_below="@+id/editText1"
            android:layout_marginTop="40dp"
            android:text="Option 1" />

        <RadioButton
            android:id="@+id/radioButton2"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_alignLeft="@+id/radioButton1"
            android:layout_below="@+id/radioButton1"
            android:text="Option 2" />
    </RadioGroup>

</RelativeLayout>
```

In this interface, we used almost all the UI components and if we run the app we have:

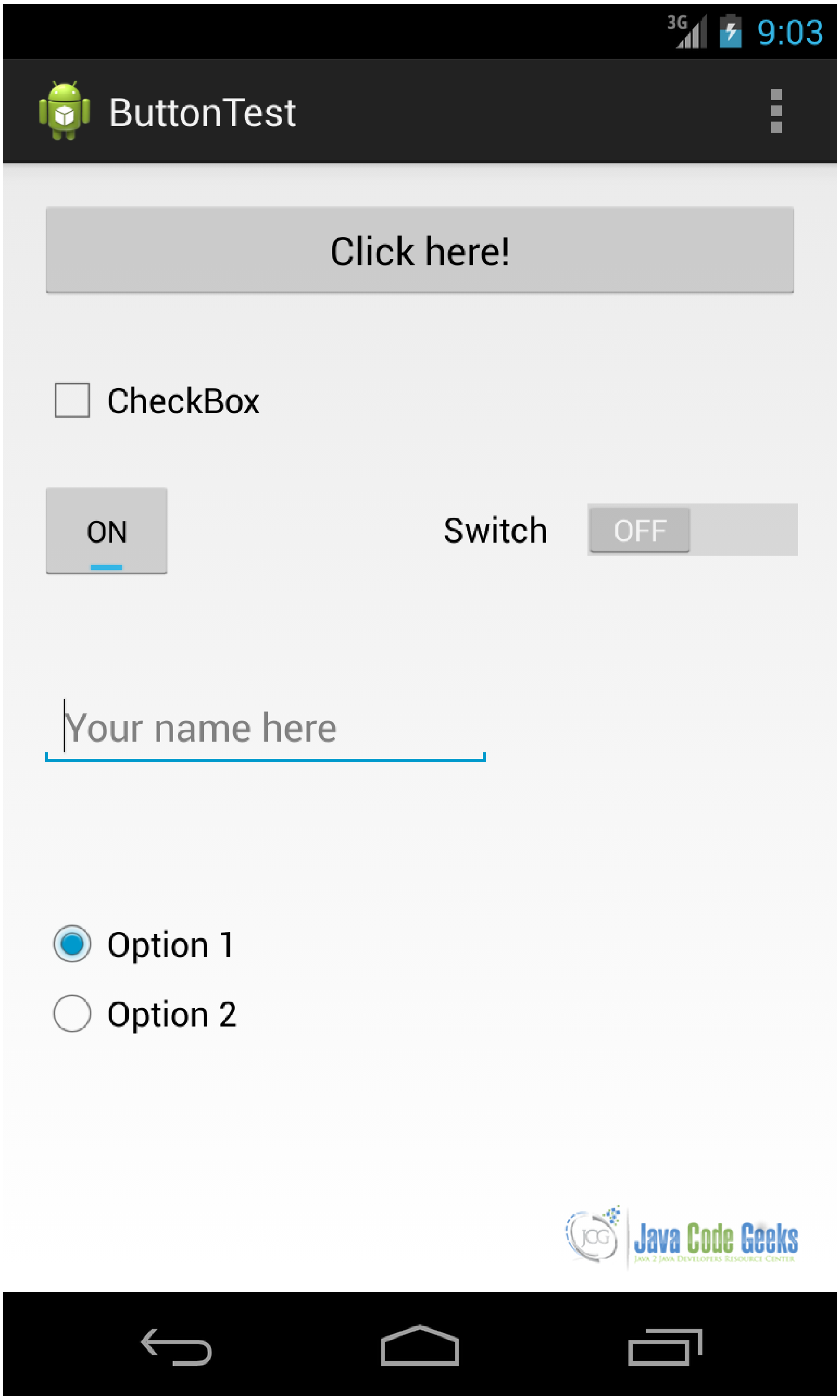


Figure 2.4: screenshot

2.3 UI Events and Listeners

By now, we have created Android user interfaces but we have not considered how to handle events that the user generates when he interacts with our app.

`Listeners` are an important aspect when developing UIs in Android. Generally speaking, when an user interacts with our app interface, the system "creates" some events. Each view that builds our interface is capable of generating events and providing the means to handle them.

From an API point of view, if we look at a `View` class, we can notice there are some public methods; these methods are called by the system when some events occur. They are called `callback` methods, and they are the way we can capture the events that occur while an user interacts with our app. Usually these callback methods starts with `on + event_name`.

Every `View` provides a set of callback methods. Some are common between several views while others are view-specific. Anyway, in order to use this approach, if we want to catch an event, we should extend the `View` and this is clearly not practical. For this reason, every `View` has a set of interfaces that we have to implement in order to be notified about the events that occur. These interfaces provide a set of callback methods. We call these interfaces as event listeners. Therefore, for example, if we want to be notified when a user clicks on a button we have an interface to implement called `View.onClickListener`.

Let us consider the example above when we used a `Button` in our UI. If we want to add a listener, the first thing we have to do is to get a reference to the `Button`:

```
Button b = (Button) findViewById(R.id.btn);
```

In our `Activity` there is a method called `findViewById` that can be used to get the reference to a `View` defined in our user interface. Once we have the reference, we can handle the user click:

```
b.setOnClickListener(new View.OnClickListener() {
        @Override
        public void onClick(View v) {
                // here we handle the event
        }
});
```

As you can see we used a compact form and you can notice the callback method called `onClick`. This method is called by the system when a user clicks on our button, so here we have to handle the event. We could obtain the same result in a different way: we can make our `Activity` implement the `View.OnClickListener` interface and implement `onClick` again. Running the app we have as a result:

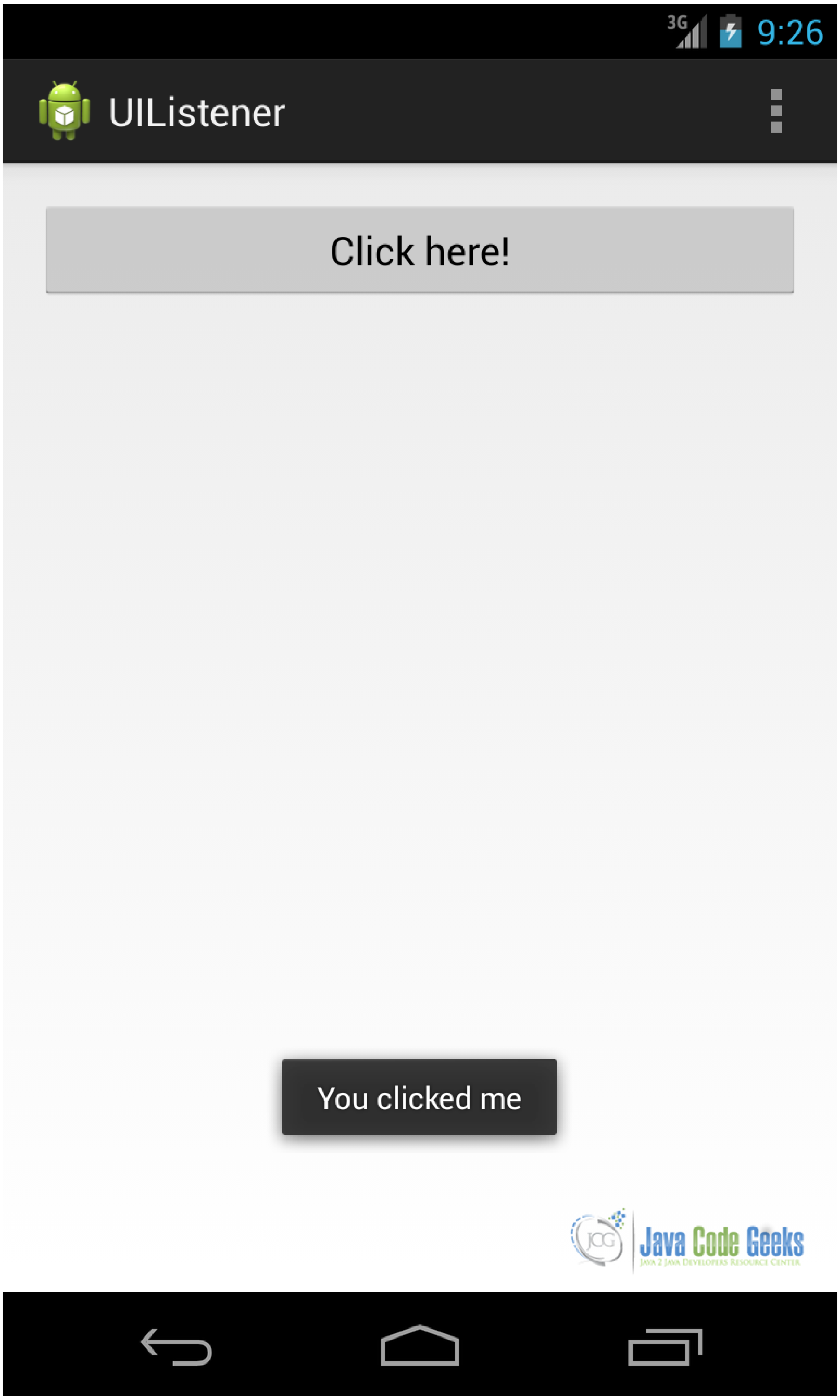


Figure 2.5: screenshot

We can use the same technique if we want to listen to different events.

2.4 UI Development

Previously, we shortly introduced the `Activity` concept. It is time to better explain its function. An Android UI cannot exist if there is not a container that holds it. This container is called `Activity` and it is one of the most important concepts in Android. Every app we will create has at least one `Activity` and usually, more complex apps have several activities that interact each other exchanging data and information using `Intents`. An `Activity` takes care of creating a window where we place our UI.

We can, now, build a complete app that uses the UI components shown above and mix them. Let's suppose we want to create an app that asks the user's name and it has a button to confirm it.

First thing is to create a layout. We assume you already know how to create an Android project. If not, please refer to our "Android Hello World" article. In Android (i.e. using Eclipse as IDE) we have to place the XML files under `res/layout`. If you look carefully, you can notice that there are different sub-directories with different names and extensions. By now, it is enough that we add our files under `res/layout` (the default directory). Other dirs will be used depending on the type of the device, so that we have the chance to optimize our UI according to the screen size and orientation.

```
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:id="@+id/edt"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:paddingBottom="@dimen/activity_vertical_margin"
    android:paddingLeft="@dimen/activity_horizontal_margin"
    android:paddingRight="@dimen/activity_horizontal_margin"
    android:paddingTop="@dimen/activity_vertical_margin"
    tools:context=".MainActivity" >

    <TextView
        android:id="@+id/textView1"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Please insert your username" />

    <EditText
        android:id="@+id/editText1"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_alignLeft="@+id/textView1"
        android:layout_below="@+id/textView1"
        android:ems="10" />

    <Button
        android:id="@+id/btn1"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:layout_alignParentBottom="true"
        android:layout_marginBottom="16dp"
        android:text="Confirm" />

</RelativeLayout>
```

Next we have to create an activity. If we use an IDE, we will notice that it has already created a default class (in Eclipse it is called `MainActivity.java`). Looking its source code, there is a `onCreate` method. Here, the first thing we do is to "attach" our UI to the activity:

```
setContentView(R.layout.activity_main);
```

Next we can handle the click event: when user clicks, we can retrieve the `EditText` value and show a message:

```
// Lookup Button reference
Button b = (Button) findViewById(R.id.btn1);

// Lookup EditText reference
final EditText edt = (EditText) findViewById(R.id.editText1);

b.setOnClickListener(new View.OnClickListener() {

        @Override
        public void onClick(View v) {
                String txt = edt.getText().toString();
                Toast.makeText(MainActivity.this, "Hello " + txt, Toast.LENGTH_LONG).show() ↩
                    ;
        }
});
```

Running the app we obtain:

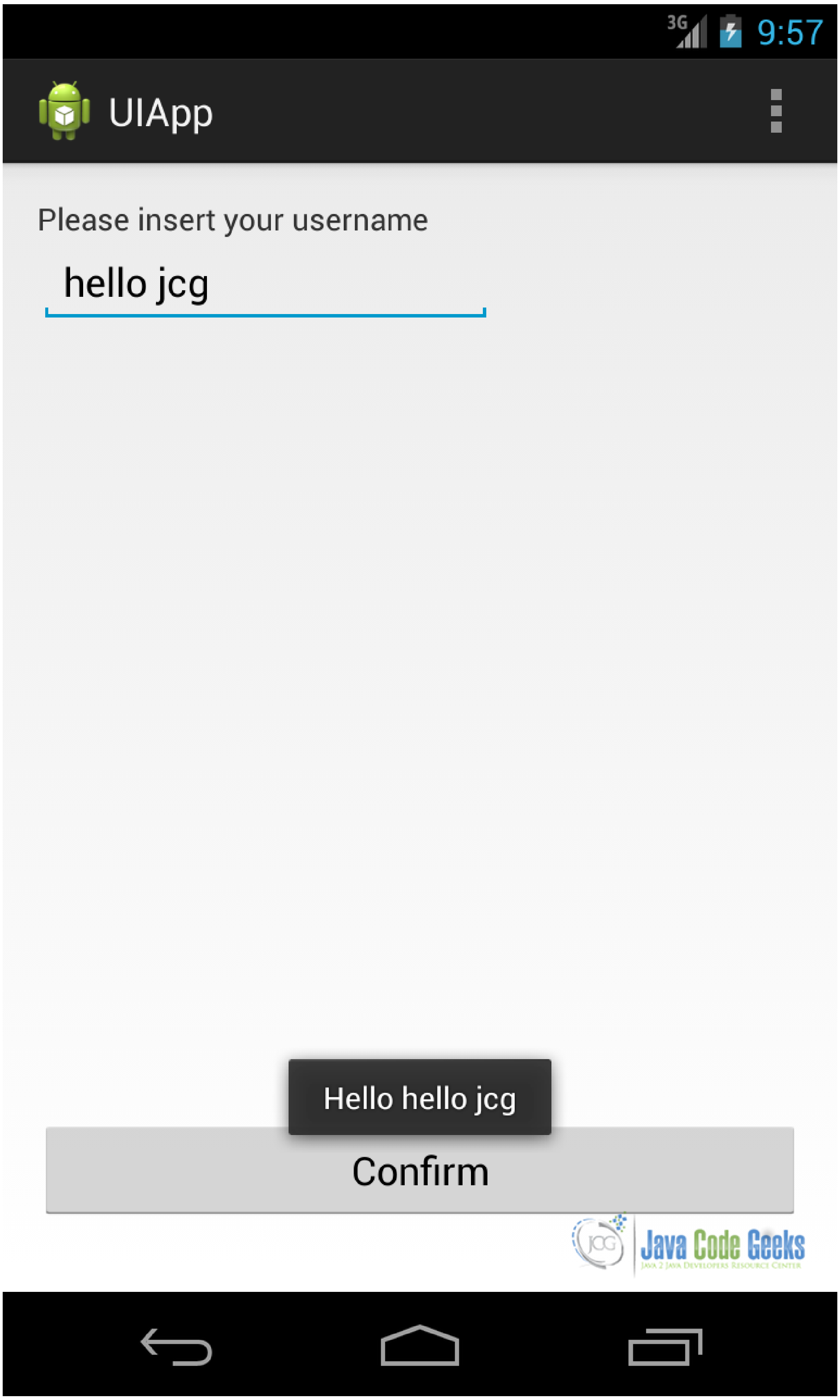


Figure 2.6: screenshot

This is a very simple example but it is useful to have an idea of how we can create a user interface.

2.5 View and Adapters

By now, we talked about UI components that show just only one value. Some components can display multiple values or a collection of data. This type of control is called `List` control. If we want to use these controls in Android, we cannot use a single component, as we did before, but we have to use two components: one that manages how data is displayed and another one that manages the underlying data. The last one is called an `Adapter`. A list control extends `AdapterView` and the most "famous" list controls are:

- ListView
- Spinner
- GridView
- Gallery

A `ListView` component displays its item in a list. This is used very often in Android UI.

`Spinner` is another component that displays only one item, but let the user choice among several items.

`GridView` shows its child in a table form, the table is scrollable and we can set the number of column to show.

`Gallery` shows item in an horizontally scrolling way. This component is deprecated since API level 16. We will focus our attention on the first two component sbecause they are very useful when building user interfaces.

Looking at the API, the `AdapterView` extends `ViewGroup` so that they both extend indirectly `View` class. Extending `ViewGroup` means that a list control holds a list of views as children and at the same time it is a `View` too. The adapter role is to manage the data and to provide the appropriate child view according to the data that has to be displayed. Android provides some general-purpose adapters. Data can be loaded from a database, from resources or we can create it dynamically.

Let us suppose we want to create a user interface where we display a list of items. Let's suppose that items are simply strings. In this case, the first thing we have to do is creating an `ArrayAdapter`. This one of the most used adapters because they are very simple to use. We can use a pre-built layout for our children so that we do not have to work too much. Android provides several pre-built layouts ready to be used.

Our app UI in XML looks like:

```
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:paddingBottom="@dimen/activity_vertical_margin"
    android:paddingLeft="@dimen/activity_horizontal_margin"
    android:paddingRight="@dimen/activity_horizontal_margin"
    android:paddingTop="@dimen/activity_vertical_margin"
    tools:context=".MainActivity" >

    <ListView
        android:id="@+id/list"
        android:layout_width="match_parent"
        android:layout_height="match_parent" />

</RelativeLayout>
```

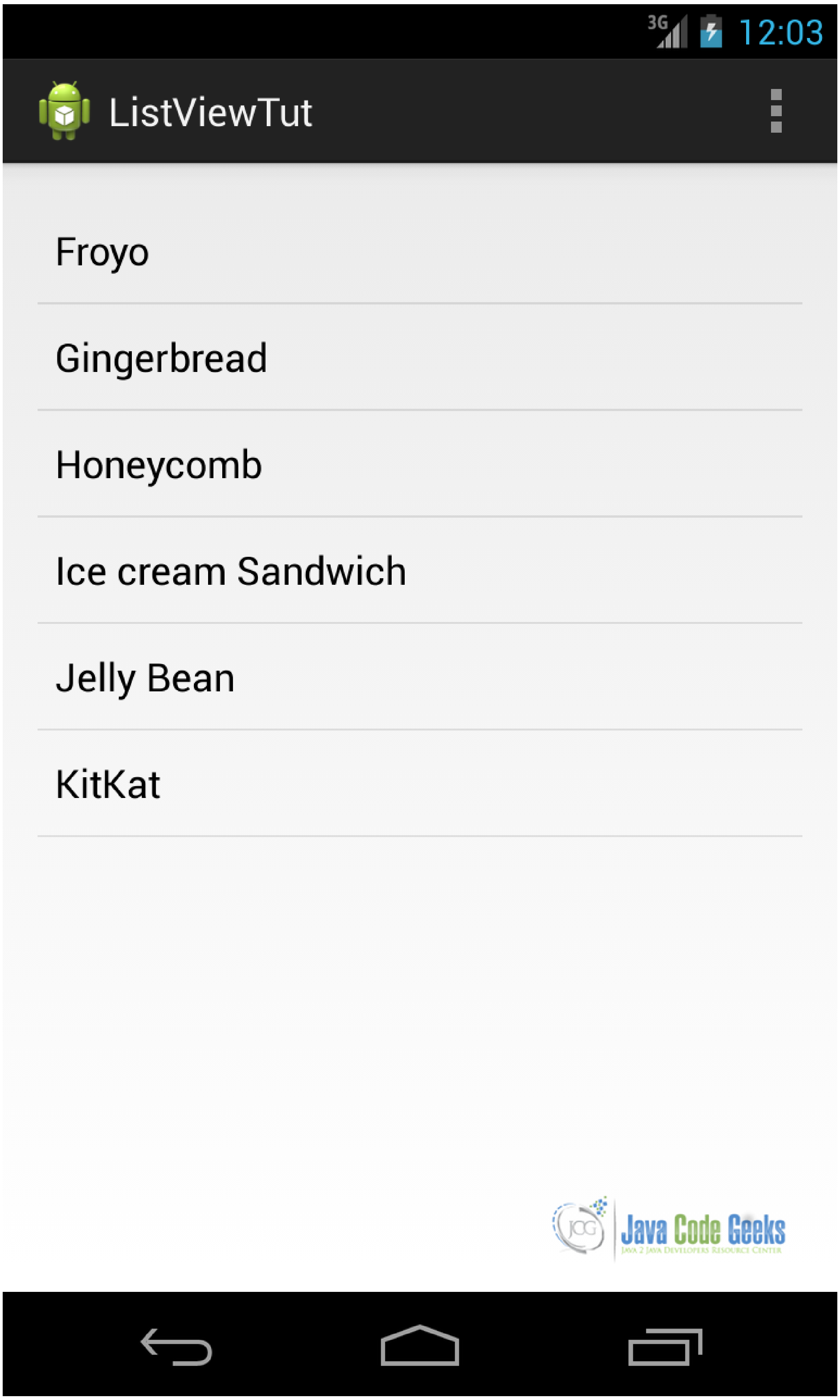
Now we have to create the Adapter so that we can display items:

```
ArrayAdapter<String> aAdpt = new ArrayAdapter<String>(this, android.R.layout. ↩
    simple_list_item_1,
 new String[]{"Froyo", "Gingerbread", "Honeycomb",  "Ice cream Sandwich", "Jelly  Bean", " ↩
     KitKat"});
```

In this way, we created an `ArrayAdapter` using as child layout the `android.R.layout.simple_list_item_1`, a layout provided by Android and we created a list of string as data. Now, we have to add the adapter to the listview:

```
ListView lv = (ListView) findViewById(R.id.list);
lv.setAdapter(aAdpt);
```

Running the app, we have:



Froyo

Gingerbread

Honeycomb

Ice cream Sandwich

Jelly Bean

KitKat

Figure 2.7: screenshot

In the example above, we provided the items as a list of Strings written directly in the source code. We can provide such list in a resource file. We can suppose we have a `Spinner` component where its child are retrieved from a resource file. First, we create our `ArrayAdapter`:

```
ArrayAdapter<CharSequence> aAdpt = ArrayAdapter.createFromResource(this, R.array.days,   ↩
    android.R.layout.simple_spinner_item);
```

where `R.array.days` is our array resource file (containing day names) and `android.R.layout.simple_spinner_item` is the children layout. Then we attach the adapter to the spinner:

```
aAdpt.setDropDownViewResource(android.R.layout.simple_spinner_dropdown_item);
spinner.setAdapter(aAdpt);
```

Running the app, we have:

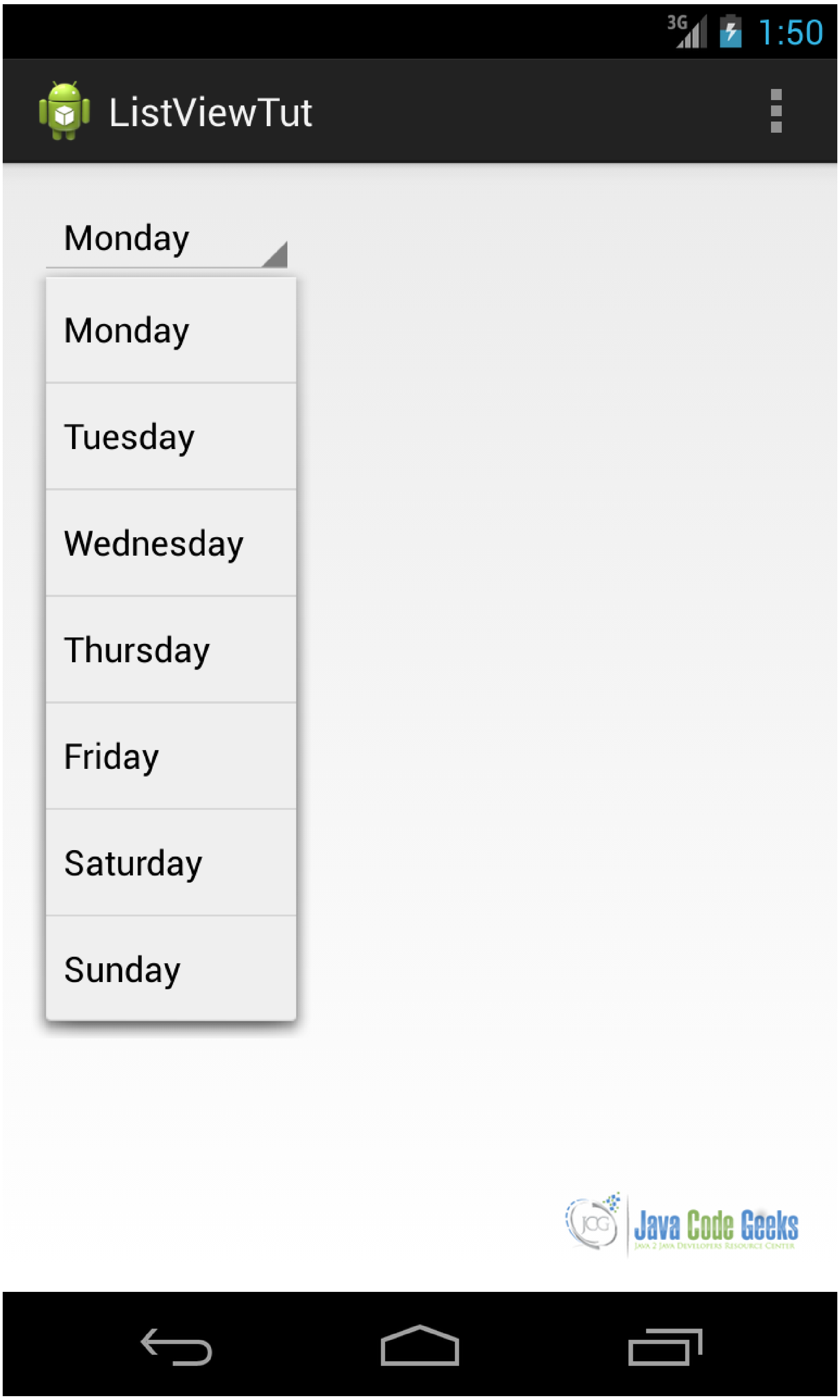


Figure 2.8: screenshot

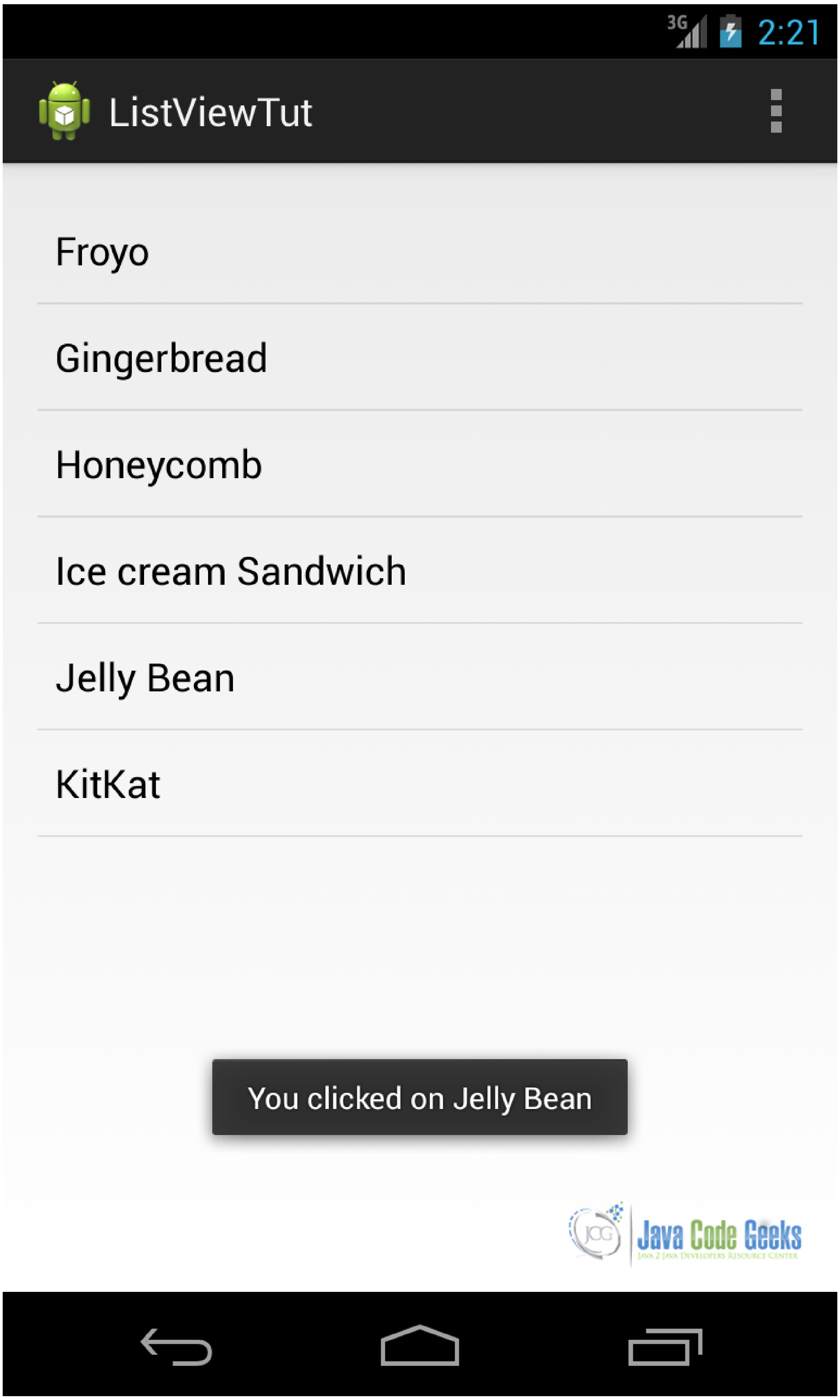
There are some cases where the information to display is stored in a database. In this case, Android provides some specialized adapter that helps us to read data from the DB and populate the corresponding list control. This adapter is called `CursorAdapter` or if we prefer we can use `SimpleCursorAdapter`. For example, we can use this type of adapter when we want to populate a `ListView` with the contacts stored in our smart phone. In this case, we can use `CursorAdapter`.

2.5.1 Handling ListView events

Another important aspect is the interaction between user and list components. We focus our attention on the `ListView`. The same considerations can be applied to other types of list components. Usually, when we have a `ListView` in the app interface, we want to know the item clicked by user. We can use, as we did for other UI components, event listeners. If you recall the previous example about `ListView`, the code to handle the event is the following:

```
lv.setOnItemClickListener(new AdapterView.OnItemClickListener() {
     @Override
     public void onItemClick(AdapterView<?> aView, View view, int position,long id) {
        // We handle item click event
        TextView tv = (TextView) view;
        String text = (String) tv.getText();

        Toast.makeText(MainActivity.this, "You clicked on " + text, Toast.LENGTH_LONG).show ←
            ();

    }
 });
```

Analyzing the code above, we simply set an item click listener and in this case we have to override a method called `onItemCick`. In this method, we receive, as a parameter, the `AdapterView`, the `View` that is clicked, the position inside the `ListView` and its id. We already know that the `View` is a `TextView` because we used an `ArrayAdapter` of strings so we can safely cast the view to the text view. Once we know the `TextView` clicked we can retrieve the text.



You clicked on Jelly Bean

Figure 2.9: screenshot

2.5.2 Custom Adapter and View Holder pattern

There are some cases when the standard adapters are not sufficient and we want to have more control over the child view or we want to add other components to the child view. There are other reasons you should consider when you decide to implement a custom adapter: it could be perhaps that you have some data management requirements that you cannot satisfy using custom adapters or you want to implement a caching policy.

In all these cases, we can implement a custom adapter: you can decide to extend a `BaseAdapter` abstract class or you can extend some specialized class as `ArrayAdapter` for example. The choice depends on the type of information you have to handle. To keep things simple and to focus our attention on the custom adapter, we will suppose we want to create a ListView where child views has two text line.

First, we create our adapter and start implementing some methods:

```java
public class CustomAdapter extends BaseAdapter {

        @Override
        public int getCount() {
                // TODO Auto-generated method stub
                return 0;
        }

        @Override
        public Object getItem(int arg0) {
                // TODO Auto-generated method stub
                return null;
        }

        @Override
        public long getItemId(int arg0) {
                // TODO Auto-generated method stub
                return 0;
        }

        @Override
        public View getView(int arg0, View arg1, ViewGroup arg2) {
                // TODO Auto-generated method stub
                return null;
        }
}
```

We have to implement four methods. Before doing it, we define a class that is our data model. For simplicity, we can assume we have a class called `Item` with two public attributes: name and descr. Now we need a child layout so that we can display data:

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

    <TextView
        android:id="@+id/name"
        android:layout_width="match_parent"
        android:layout_height="match_parent" />

    <TextView
        android:id="@+id/descr"
        android:layout_width="match_parent"
        android:layout_height="match_parent" />

</LinearLayout>
```

In our adapter we need a constructor so that we can pass the data we want to display:

```
public CustomAdapter(Context ctx, List<Item> itemList) {
        this.ctx = ctx;
        this.itemList = itemList;
}
```

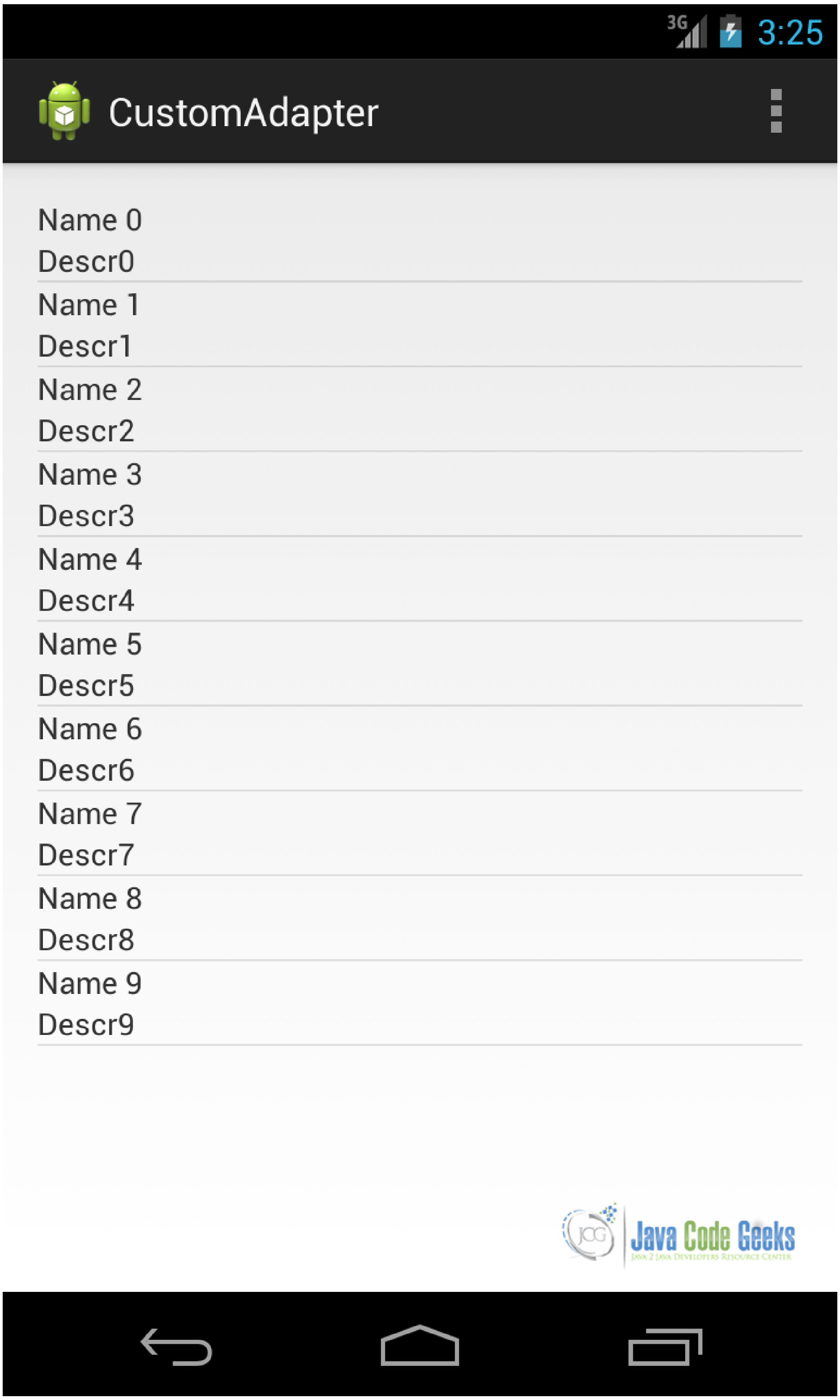
Now it is quite trivial to override method, except `getView`:

```
@Override
public int getCount() {
        return itemList == null ? 0 : itemList.size();
}

@Override
public Object getItem(int pos) {
        return itemList == null ? null : itemList.get(pos);
}

@Override
public long getItemId(int pos) {
        return itemList == null ? 0 : itemList.get(pos).hashCode();
}
```

Notice that in `getItemId` we return a unique value that represents the item. A bit more complex is the `getView` method. This is the heart of the custom adapter and we have to implement it carefully. This method is called everytime the `ListView` has to display child data. In this method, we can create the view manually or inflate an XML file:

```
@Override
public View getView(int position, View convertView, ViewGroup parent) {
    View v = convertView;

     if (v == null) {
        LayoutInflater lInf = (LayoutInflater)
        ctx.getSystemService(Context.LAYOUT_INFLATER_SERVICE);
        v = lInf.inflate(R.layout.item_layout, null);
    }

  TextView nameView = (TextView) v.findViewById(R.id.name);
  TextView descrView = (TextView) v.findViewById(R.id.descr);

  nameView.setText(itemList.get(position).name);
  descrView.setText(itemList.get(position).descr);
  return v;
}
```

Analyzing the code, we can notice that we first check that the view passed as parameter is not null. This is necessary because Android system can reuse a View to not waste resources. Therefore, if the view is not null we can avoid inflating XML layout file. If it is null then we have to inflate the layout file. The next step is looking up the `TextView` inside our layout so that we can set their values. Running the app, we will obtain:



Name 0
Descr0
Name 1
Descr1
Name 2
Descr2
Name 3
Descr3
Name 4
Descr4
Name 5
Descr5
Name 6
Descr6
Name 7
Descr7
Name 8
Descr8
Name 9
Descr9

Figure 2.10: screenshot

Looking at the getView method, we can notice we call findViewById several times. This is an expensive operation and we should avoid calling it when it is not necessary because it can slow down the overall Listview performances and when user scrolls, the ListView it can happen it is not smooth.

Moreover, we use findViewById even if the view is recycled. In this situation we can apply a pattern that reduces the use of this method. This pattern is called View Holder, it requires we create an object to hold the references to the views inside our child layout. Applaying this pattern getView method become:

```
@Override
public View getView(int position, View convertView, ViewGroup parent) {
        View v = convertView;
        TextHolder th = null;

        if (v == null) {
                LayoutInflater lInf = (LayoutInflater)
                ctx.getSystemService(Context.LAYOUT_INFLATER_SERVICE);
                v = lInf.inflate(R.layout.item_layout, null);

                TextView nameView = (TextView) v.findViewById(R.id.name);
                TextView descrView = (TextView) v.findViewById(R.id.descr);

                th = new TextHolder();
                th.nameView = nameView;
                th.descrView = descrView;
                v.setTag(th);
        }
        else
                th = (TextHolder) v.getTag();

        th.nameView.setText(itemList.get(position).name);
        th.descrView.setText(itemList.get(position).descr);

        return v;
}


static class TextHolder {
        TextView nameView;
        TextView descrView;
}
```

2.6 Download the Source Code

This was a lesson of using Android Views. You may download the source code here: AndroidView.zip

Chapter 3

Android UI: Layouts with View Groups and Fragments

3.1 Overview

In the previous chapter, we talked about `Views` and we explained how to build user interfaces using `Views`. We discovered different kinds of `Views`, with different controls. Android provides several common controls and using them we can build appealing user interfaces. We saw some useful patterns we can implement when we create UIs so that we can guarantee consistency to our app interface.

In this article we want to explore how we can organize such views and how these views can be placed on the screen. In other words, we will analyze in detail layout managers, or simply layouts.

3.2 Layout overview

When we create our app interface, we use some special view that acts as container. These special views control how other views are placed on the smartphone/tablet screen. Android provides a collection of Layout Managers and each of them implements a different strategy to hold, manage and place its children. From the API point of view, all the Layout managers derive from the `ViewGroup` class. There are some layouts that place their children horizontally or vertically, and others that implement a different strategy. We will analyze them in details later in this article.

In Android, layouts can be nested so we can use different layouts for different areas of our interface. However, please ve aware that it is not advisable to create too complex layouts, because this can affect the overall app performance. We can declare a layout in two ways:

- **Using XML:** In this case, using an XML file we “describe” how our user interface looks like. We define the elements (views and sub layouts) that appear in the user interface. At the same time, we define their attributes, as we saw in the previous chapter.
- **At runtime:** In this case, we code our layout instantiating our `ViewGroup` and attaching `Views` to it. We can manipulate their properties programmatically setting their attributes.

We can use both approaches in our app. We could, for example, use XML to create the user interface and assign to its `Views` some properties. At run time, we can find (or lookup) this `Views` and `ViewGroup` (layout) and change their properties programmatically. We could for example, have a `View` with red background and at runtime we change it to green color. Android is very powerful and flexible from this point of view.

Using XML, we somehow decouple the presentation from the code that handles its behavior. In XML, the UI description is external to the source code, so theoretically we could change the presentation, meaning just the XML file, without touching our source code. This is the case when, for example, we want to adapt our user interface for multiple screen dimensions. In this case, we define different layouts having the same name but in different directories, and the system chooses the one that best matches the screen dimensions. This is the standard approach taken by Android in order to handle multiple screen sizes. Moreover, we will see later that we can use another technique based on Fragments. If we use XML, it is possible to use draw the layout and debug it easily.

From the API point of `View`, each `ViewGroup` defines a nested class called `LayoutParameter` that holds some parameters that define size and position for each views belonging to the `ViewGroup`. All `ViewGroup` have in common two parameters called width and height (or `layout_width` and `layout_height`) that every `View` must define. These two parameters represent the width and the height of the `View`. We can specify a numeric value or more often we can use two constants:

- `wrap_content`, meaning that the dimension of the view will depend on the actual content
- `fill_parent` (or `match_parent`), meaning that the view has to become as big as its parent holding it

A view in Android is a rectangle, and the view location is expressed as a pair of coordinates left and top. These two values determine the `View` position inside its `ViewGroup`. Another important `View` property inside a Layout is padding, expressed with four values (let, top, right, bottom). Using padding we can move the content of the `View`.

Android provides several standard layout managers:

- **Linear Layout**
- **Table Layout**
- **Relative Layout**
- **Frame Layout**
- **Grid Layout**

3.2.1 LinearLayout

This is the simplest Layout manager. This layout disposes its children vertically or horizontally depending on the orientation parameter. To define this layout in XML, we can use:

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

</LinearLayout>
```

The orientation attribute is the most important attribute because it determines how views are placed. It can assume two values: horizontal or vertical. In the first case, the views are placed horizontally and in the second case they are placed vertically. There are two other parameters that affect the position and the size of the views. They are gravity and weight.

The gravity parameter behaves like the alignment. So if we want to align a view to the left side we can set the gravity left or right, if we want to align it to the right. The corresponding attribute in XML is `layout_gravity`. If we create an app, using the layout shown below:

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

    <TextView
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:gravity="left"
        android:text="Left" />
    <TextView
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:gravity="right"
        android:text="Right" />

    <TextView
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:gravity="center"
        android:text="Center" />

</LinearLayout>
```

and run it, we have:

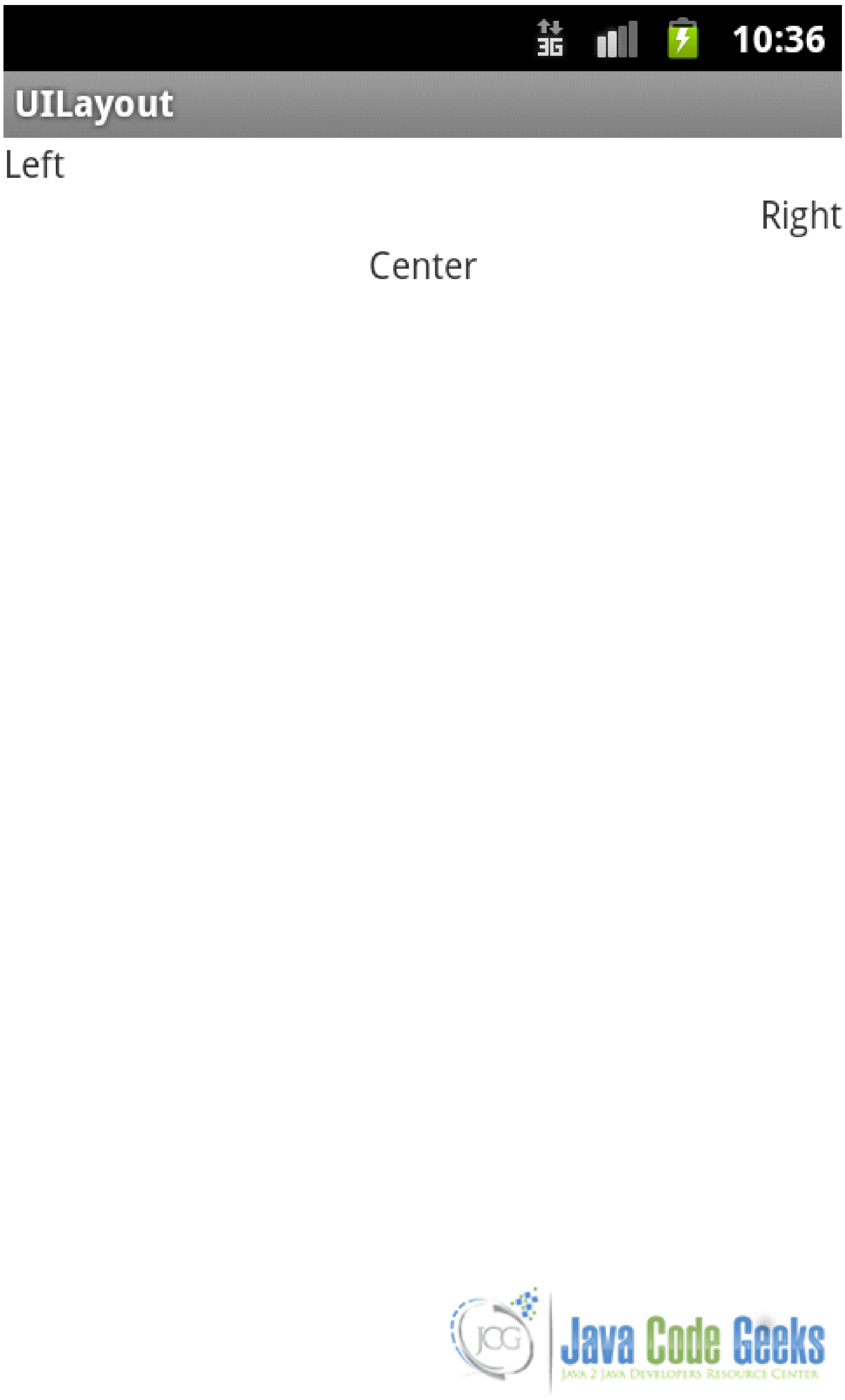


Figure 3.1: screenshot

Another important parameter is weight (or `layout_weight` in XML). Using the weight parameter we can assign an importance value to a `View` respect to the others. The `View` with higher importance value is more important than `Views` with lower values. In other words, `Views` with higher weight value consume more space than other `Views`. For example, see this layout:

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

    <TextView
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:layout_weight="0"
        android:gravity="left"
        android:text="Left" />

    <TextView
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:layout_weight="1"
        android:gravity="right"
        android:text="Right" />

    <TextView
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:layout_weight="0"
        android:gravity="center"
        android:text="Center" />

</LinearLayout>
```

In this case, we used layout_weight and we gave more importance to the `TextView` with right text. Running the app, we have:

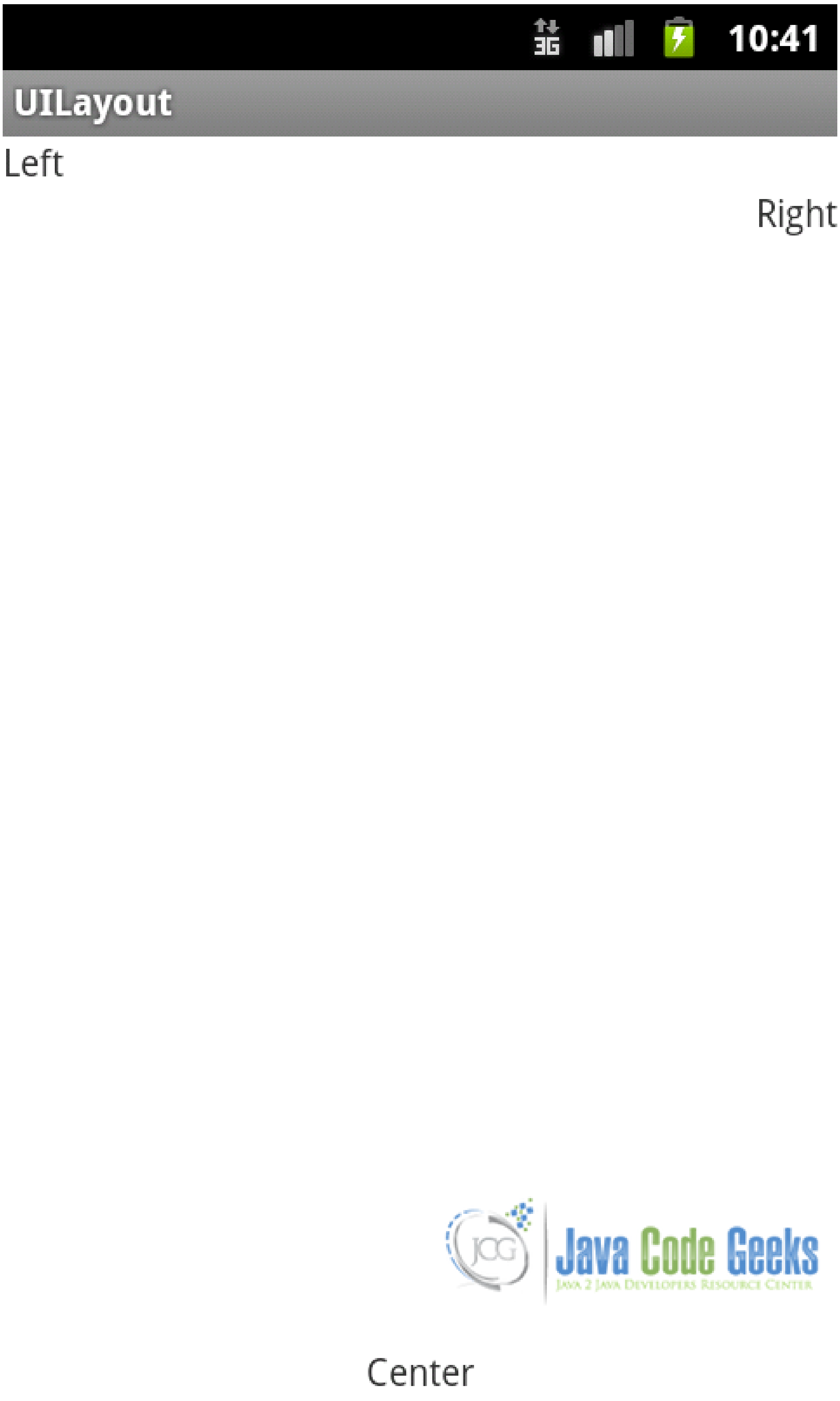


Figure 3.2: screenshot

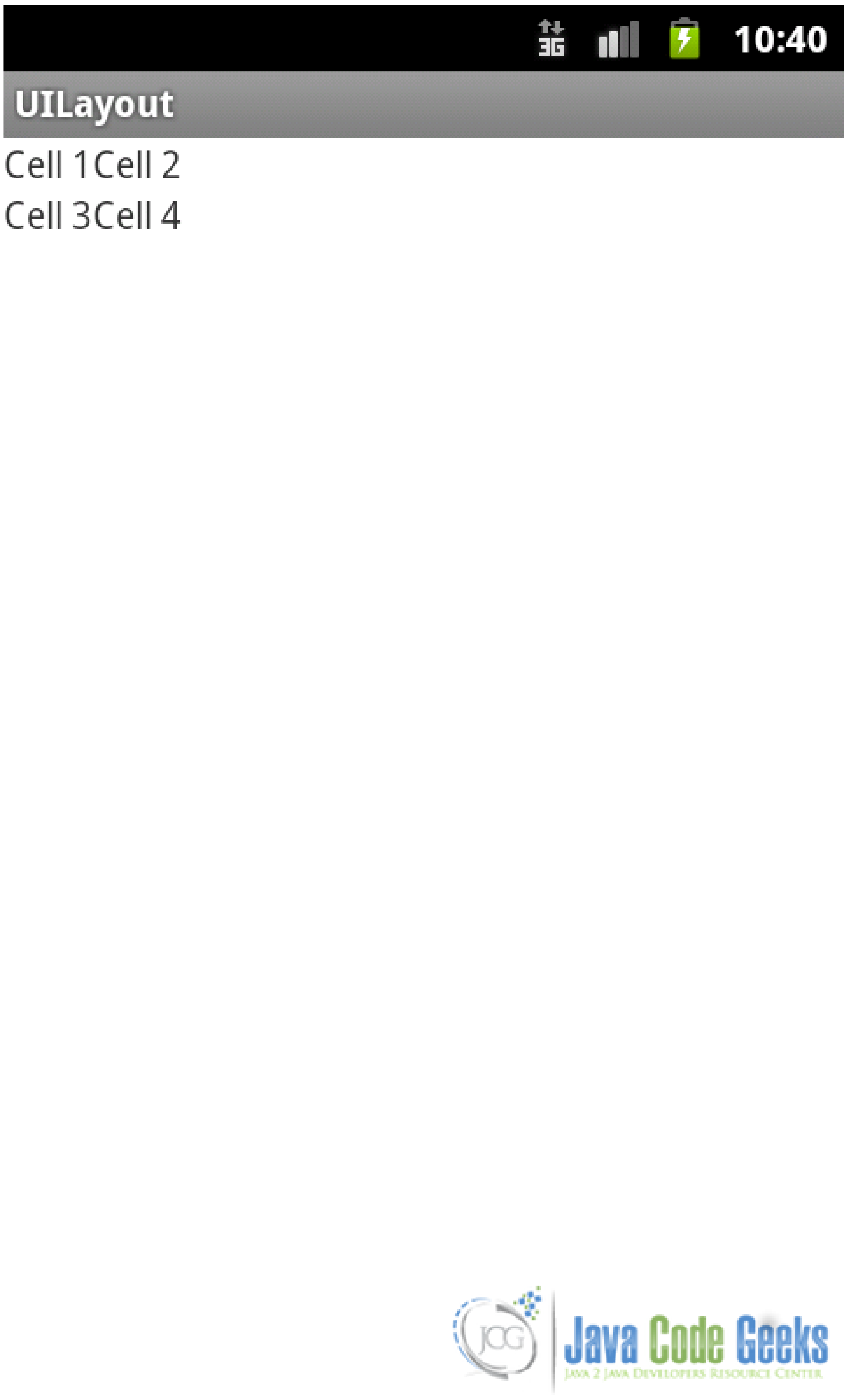
Another important aspect we have to consider is the difference between android:gravity and layout_gravity. Even if they look very similar, they have a different meaning. android:gravity is an attribute used by the View, while layout_gravity is a parameter used by the container.

3.2.2 TableLayout

This is layout manager that disposes its children in a table, grouping them in rows and columns. For example, using the layout shown below, we create two different rows holding two cells.

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

    <TableRow>

        <TextView
            android:layout_width="match_parent"
            android:layout_height="match_parent"
            android:text="Cell 1" >
        </TextView>

        <TextView
            android:layout_width="match_parent"
            android:layout_height="match_parent"
            android:text="Cell 2" >
        </TextView>
    </TableRow>

    <TableRow>

        <TextView
            android:layout_width="match_parent"
            android:layout_height="match_parent"
            android:text="Cell 3" >
        </TextView>

        <TextView
            android:layout_width="match_parent"
            android:layout_height="match_parent"
            android:text="Cell 4" >
        </TextView>
    </TableRow>

</TableLayout>
```

We used as TableLayout child the TableRow, which represents a row inside the table. Running an app with this layout we have:



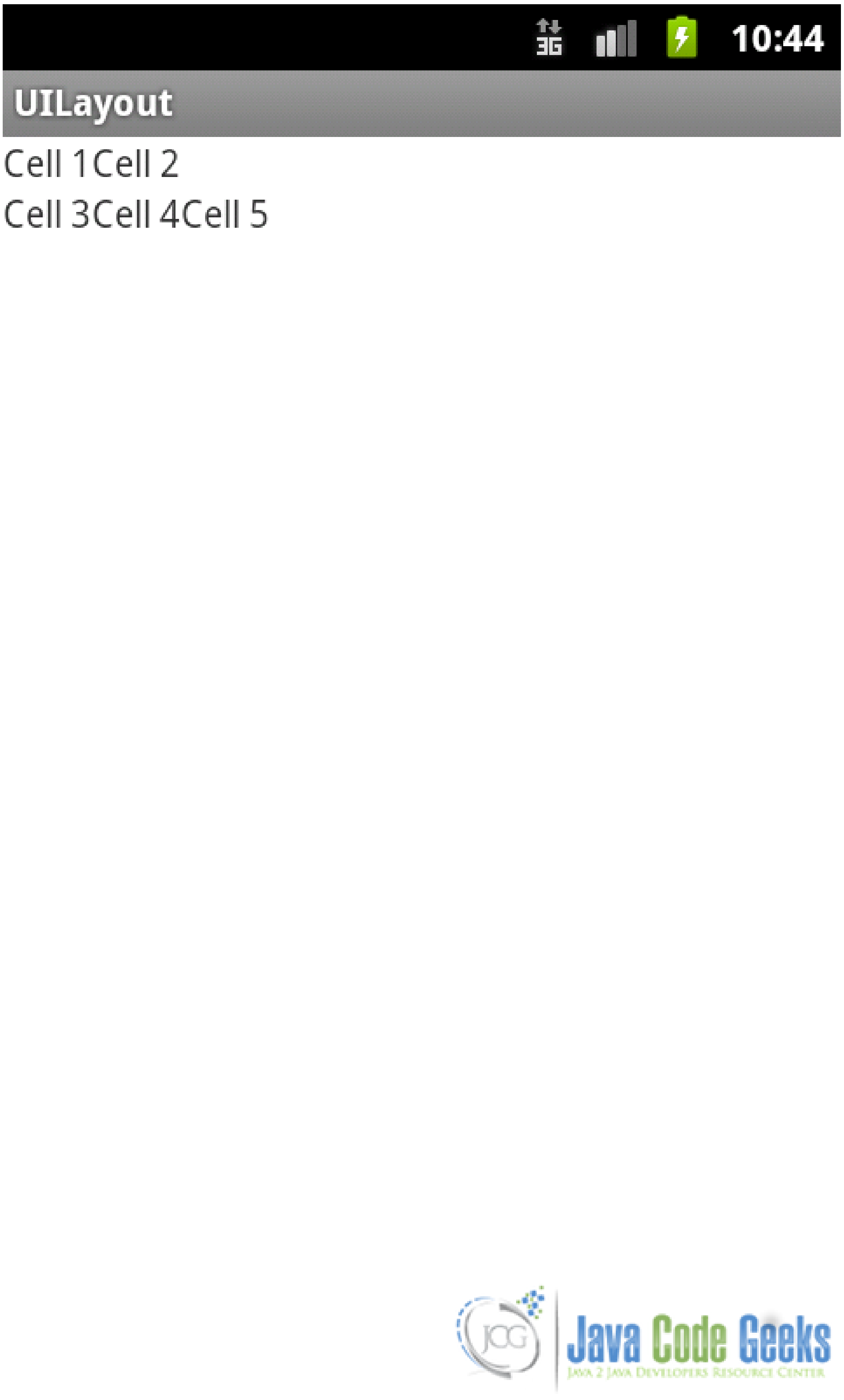
Cell 1Cell 2
Cell 3Cell 4

Figure 3.3: screenshot

We can even use different cell numbers for different rows like the example shown below, where in the second row we have three cells:

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

    <TableRow>

        <TextView
            android:layout_width="match_parent"
            android:layout_height="match_parent"
            android:text="Cell 1" >
        </TextView>

        <TextView
            android:layout_width="match_parent"
            android:layout_height="match_parent"
            android:text="Cell 2" >
        </TextView>
    </TableRow>

    <TableRow>

        <TextView
            android:layout_width="match_parent"
            android:layout_height="match_parent"
            android:text="Cell 3" >
        </TextView>

        <TextView
            android:layout_width="match_parent"
            android:layout_height="match_parent"
            android:text="Cell 4" >
        </TextView>

        <TextView
            android:layout_width="match_parent"
            android:layout_height="match_parent"
            android:text="Cell 5" >
        </TextView>
    </TableRow>

</TableLayout>
```

In this case, the first row has an empty cell. Running the example we have:



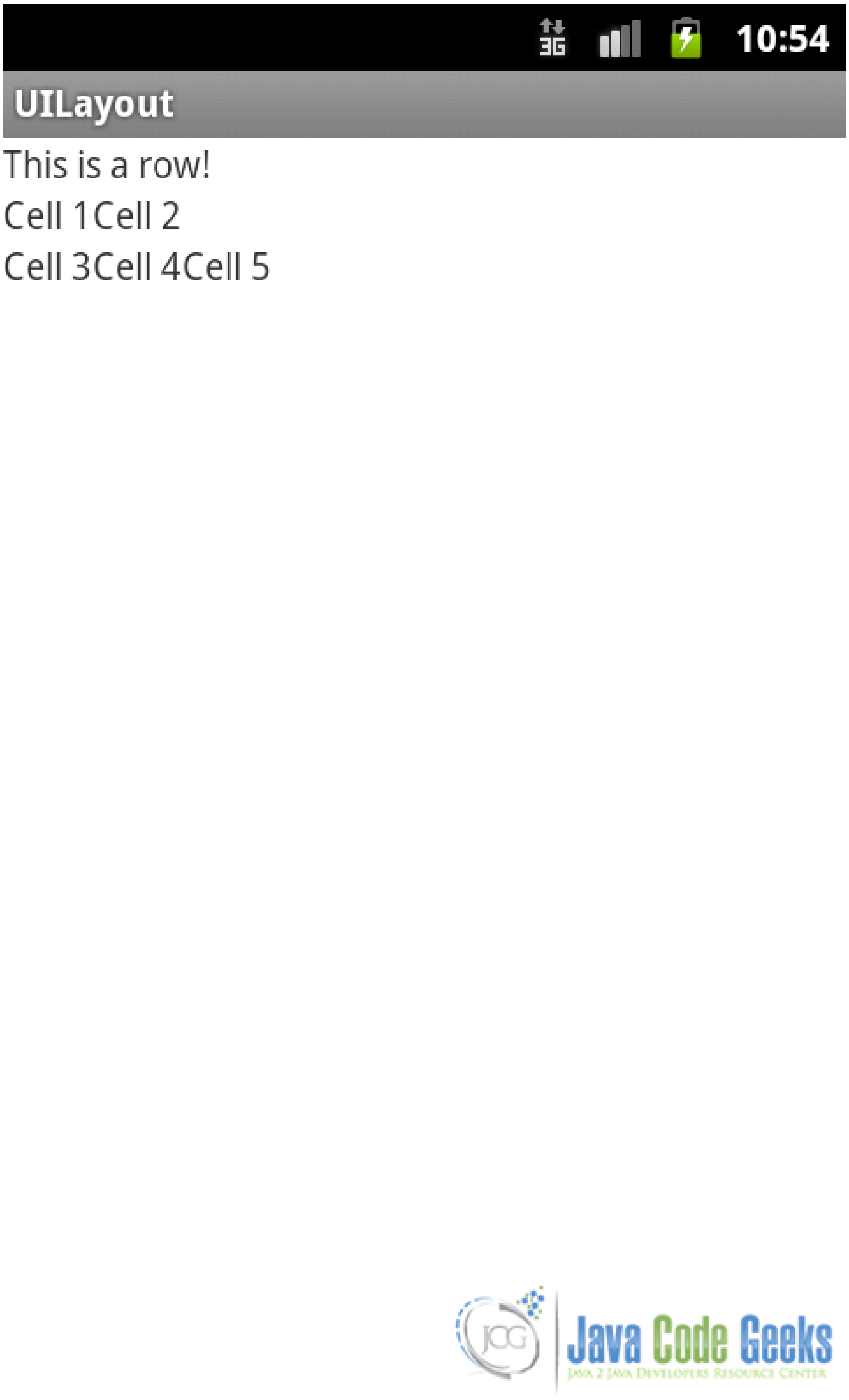
Cell 1Cell 2
Cell 3Cell 4Cell 5

Figure 3.4: screenshot

It is not required to use TableRow, because we can use all the components that extend the View class. In this case, this component will have the same width as the table. For example:

```
<?xml version="1.0" encoding="utf-8"?>
<TableLayout 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="This is a row!" >
    </TextView>

    <TableRow>

        <TextView
            android:layout_width="match_parent"
            android:layout_height="match_parent"
            android:text="Cell 1" >
        </TextView>

        <TextView
            android:layout_width="match_parent"
            android:layout_height="match_parent"
            android:text="Cell 2" >
        </TextView>
    </TableRow>

    <TableRow>

        <TextView
            android:layout_width="match_parent"
            android:layout_height="match_parent"
            android:text="Cell 3" >
        </TextView>

        <TextView
            android:layout_width="match_parent"
            android:layout_height="match_parent"
            android:text="Cell 4" >
        </TextView>

        <TextView
            android:layout_width="match_parent"
            android:layout_height="match_parent"
            android:text="Cell 5" >
        </TextView>
    </TableRow>

</TableLayout>
```

In this case, we did not use TableRow but we used TextView as the first TableLayout child and we specified that the TextView width should be as big as the content. If we run an app with this layout, we have:



```
This is a row!
Cell 1Cell 2
Cell 3Cell 4Cell 5
```

Figure 3.5: screenshot

You can notice that even if we specified that the `TextView` should be as big as the content, it still occupies the entire row.

3.2.3 RelativeLayout

This is the most flexible layout in Android. This layout manager uses a policy where the container places its Views relative to other `Views`. We can implement, using this layout manager, very complex UI structures.

`RelativeLayout` implements some view attributes that we can use to place the `View`. There are attributes that control the position of the `View` respect to other `Views`:

- `layout_toLeftof`: the right edge position of this View is to the left of a specific View
- `layout_toRightof`: the left edge position of this View is to the right of a specific View
- `layout_below`: the top edge of this view is below to a specific View
- `layout_above`: the bottom edge of this view is above to a specific View

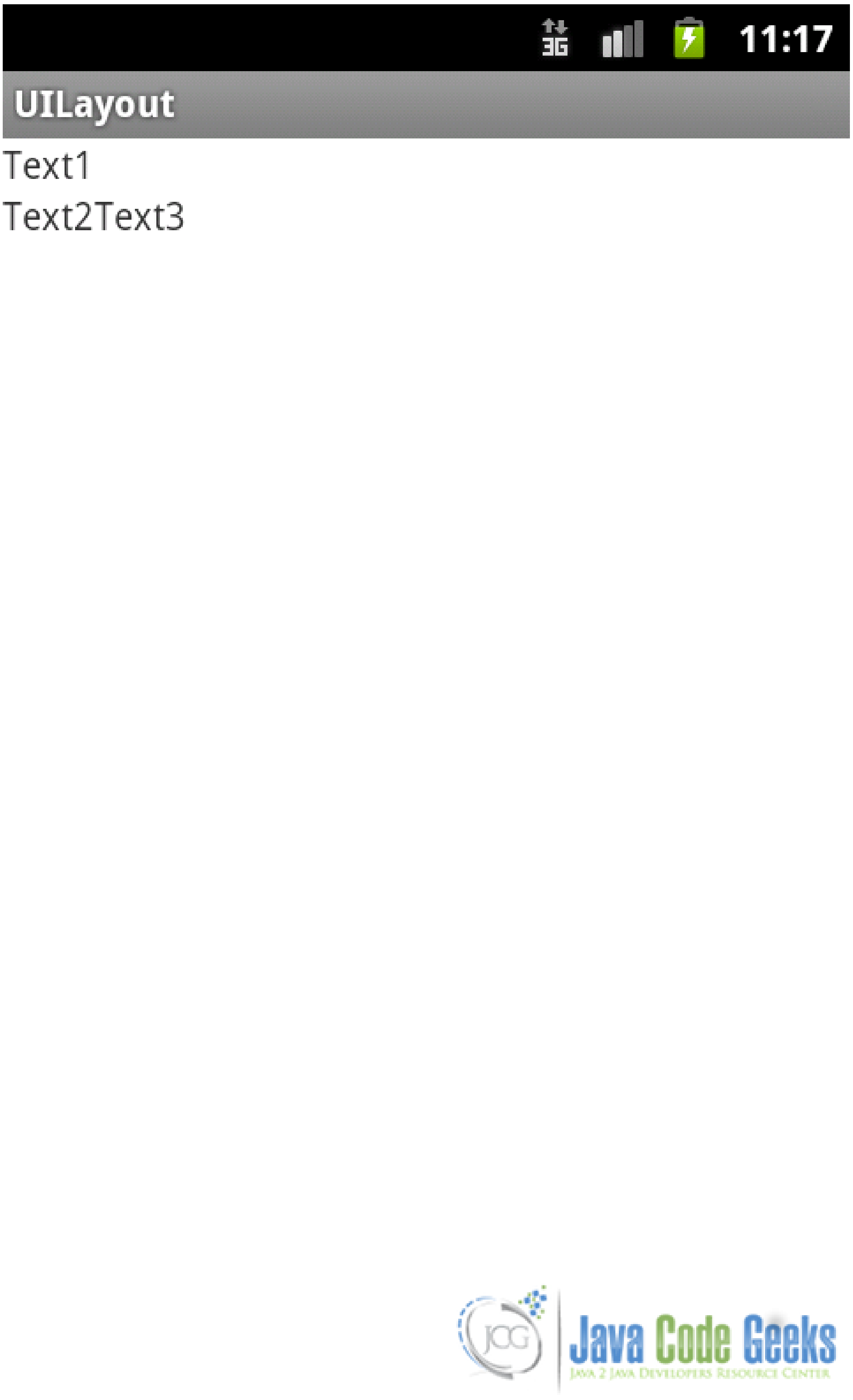
There are other parameters that are used to place the view respect to its container:

- `layout_alignParentLeft`: the left edge of this view matches the left edge of its container
- `layout_alignParentRight`: the right edge of this view matches the right edge of its container
- `layout_alignParentTop`: the top edge of this view matches the top edge of its container
- `layout_alignParentBottom`: the bottom edge of this view matches the bottom edge of its container

There are some other parameters that can be used and you are advised to have a look at the documentation. For example:

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

    <TextView
        android:id="@+id/t1"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Text1" />

    <TextView
        android:id="@+id/t2"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_below="@id/t1"
        android:text="Text2" />

    <TextView
        android:id="@+id/t3"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_below="@id/t1"
        android:layout_toRightOf="@id/t2"
        android:text="Text3" />

</RelativeLayout>
```

In the example above, we placed the `TextView` with id t2 below the `TextView` with id t1, the `TextView` with id t3 is places to the left of t2 and below t1. Running the example we have:



Text1
Text2Text3

Figure 3.6: screenshot

Let us suppose we want to add another `TextView` to the layout above and this `TextView` has to be placed in bottom right corner, so we have:

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

    <TextView
        android:id="@+id/t1"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Text1" />

    <TextView
        android:id="@+id/t2"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_below="@id/t1"
        android:text="Text2" />

    <TextView
        android:id="@+id/t3"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_below="@id/t1"
        android:layout_toRightOf="@id/t2"
        android:text="Text3" />

    <TextView
        android:id="@+id/t4"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_alignParentBottom="true"
        android:layout_alignParentRight="true"
        android:text="Text4" />

</RelativeLayout>
```

Running the example, we have:

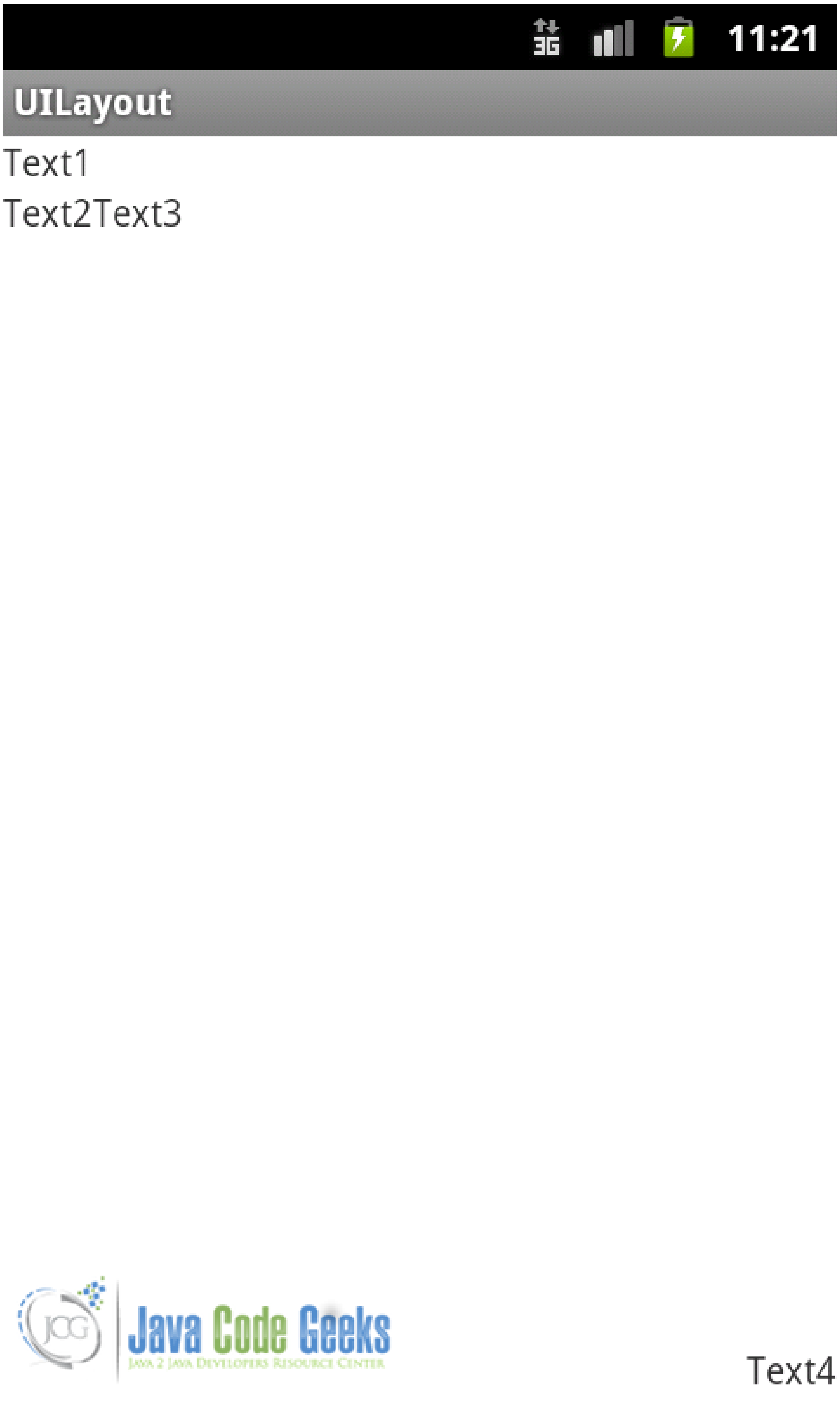


Figure 3.7: screenshot

3.2.4 FrameLayout

`FrameLayout` is a special layout that we will cover in more detail later. We saw different layout manager that implements some specific strategies to place views on the UI. `FrameLayout` is used when we want to display dynamically a view. It is very useful when we use `Fragments`.

3.2.5 GridLayout

This is the last layout manager that we cover in this article. It is very similar to `TableLayout` and it was introduced since Android 4.0. This layout manager disposed its views in a grid form, but respect to the `TableLayout` it is easier to use. `GridLayout` divides the screen area into cells. Its children occupy one or more cells.

3.2.6 Layout and multiple screen support

A special consideration is needed when we want to support multiple screen size. When we want to build an app for multiple devices, we have to implement several layouts. In Android, we have different directories and there we can implement the layouts. The default layout is the one under `res/layout`. We can provide some size qualifier that we append to the layout:

- **small**
- **large**
- **xlarge**

Moreover, we can even provide the screen orientation:

- **land**
- **portrait**

So, for example, if we want to create a layout for an extra large screen, we create a directory under `res` called `layout-xlarge` and here we implement our layout.

If we want to provide a layout for landscape mode we create another dir, under `res`, called `layout-land land` and so on. In Android 3.2 there were introduced other size qualifiers to better adapt the layout to the screen size. If you want to have more information you can look here.

3.3 Fragments

By now, we have seen how we can build UIs using layouts and components. In this situation, we have an `Activity` with its layout and components. We know how we can adapt our layout for multiple screens but this technique sometimes is not enough, especially when we want to support tablets and smartphones. We have talked about this already, and we know that the smartphone screen size is very different from the screen of a tablet. Moreover, it is necessary to make the UI more dynamic and flexible in a large screen tablets. For these reasons, Android 3.0 (API level 11) has introduced the fragments concept.

What is a fragment? A fragment is a piece of user interface in an `Activity`. We can combine multiple fragments to create complex UIs. Each fragment has its own lifecycle and we can manage the lifecycle of each fragment independently. A fragment exists just inside an `Activity` that acts as its container. When we add a fragment inside a layout, it lives inside the ViewGroup that represents the layout. A fragment is a very powerful component that helps developers to create dynamic UI and support, at the same time, multiple screen size.

A fragment can be seen as a reusable piece of code with its own interface. It is crucial to differentiate when we need to use fragments and when we can use just “simple” layouts. Well, as said before, there are some situations where simple layouts, even if they are adapted to the screen size, are not enough.

A classic example is an app that has a list of contacts where, when a user clicks on a contact, the app shows the contact’s details. In a smartphone we can move from one activity to another showing a different layout, list and details. However, in a tablet this behavior would result in a poorly appealing app that does not use all the screen size available. In the tablet, we would like to have a list and the details at the same time on the screen.

For example, in a smartphone we would have:

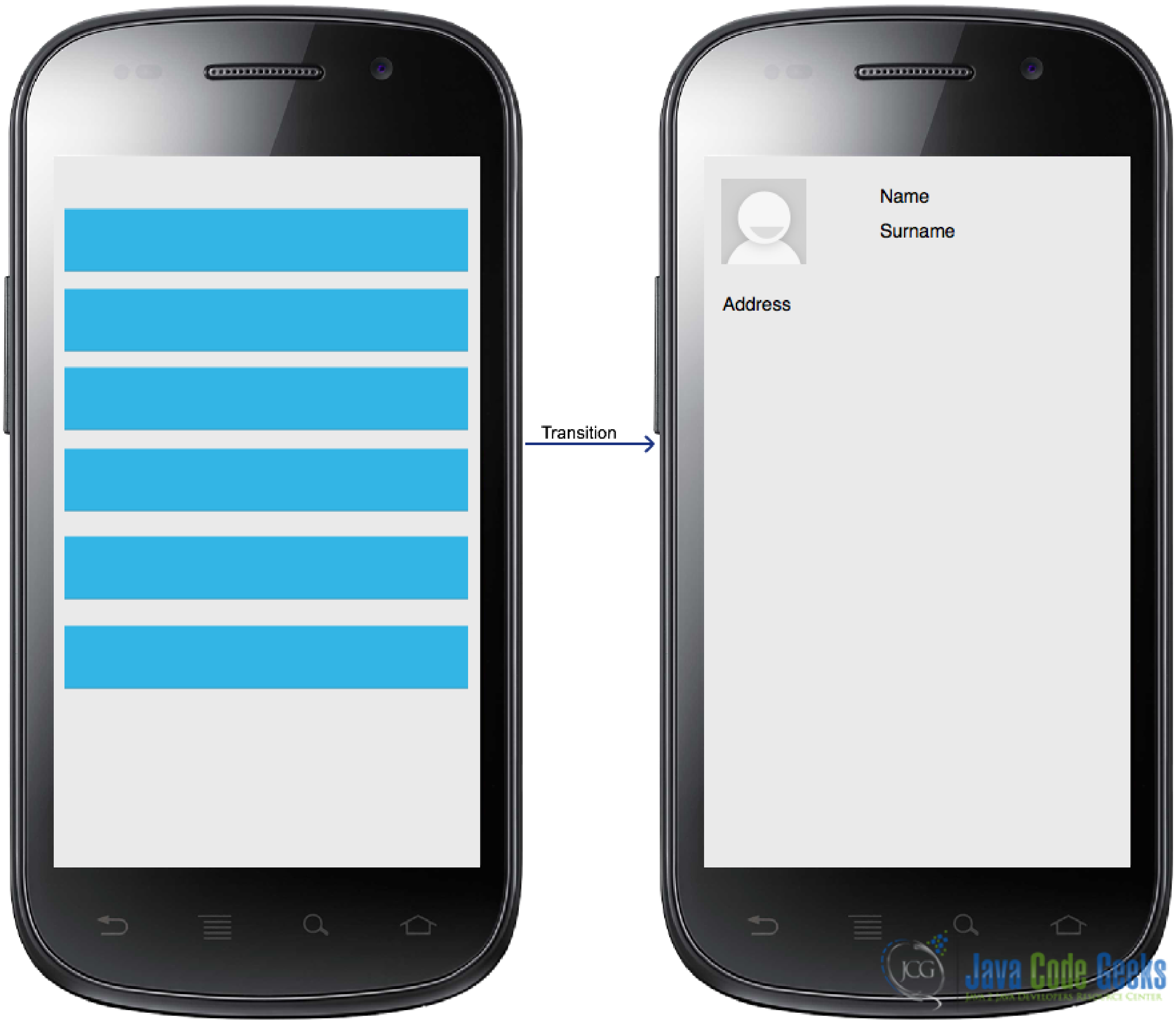


Figure 3.8: screenshot

while in a tablet we would have:

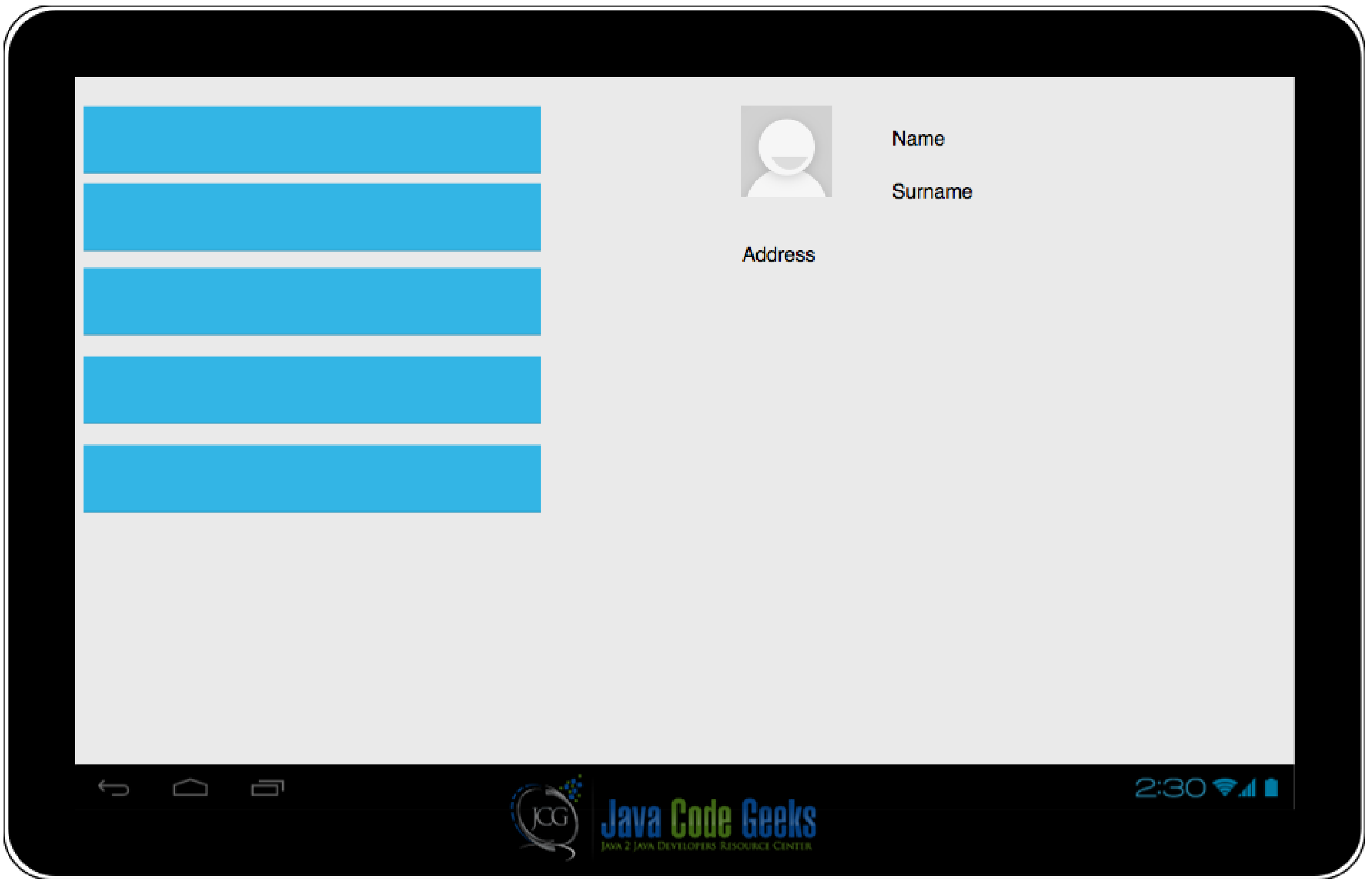


Figure 3.9: screenshot

In this case, fragments help us. We can create two fragments: one that handles the contact list and another one the contact details. So we have:

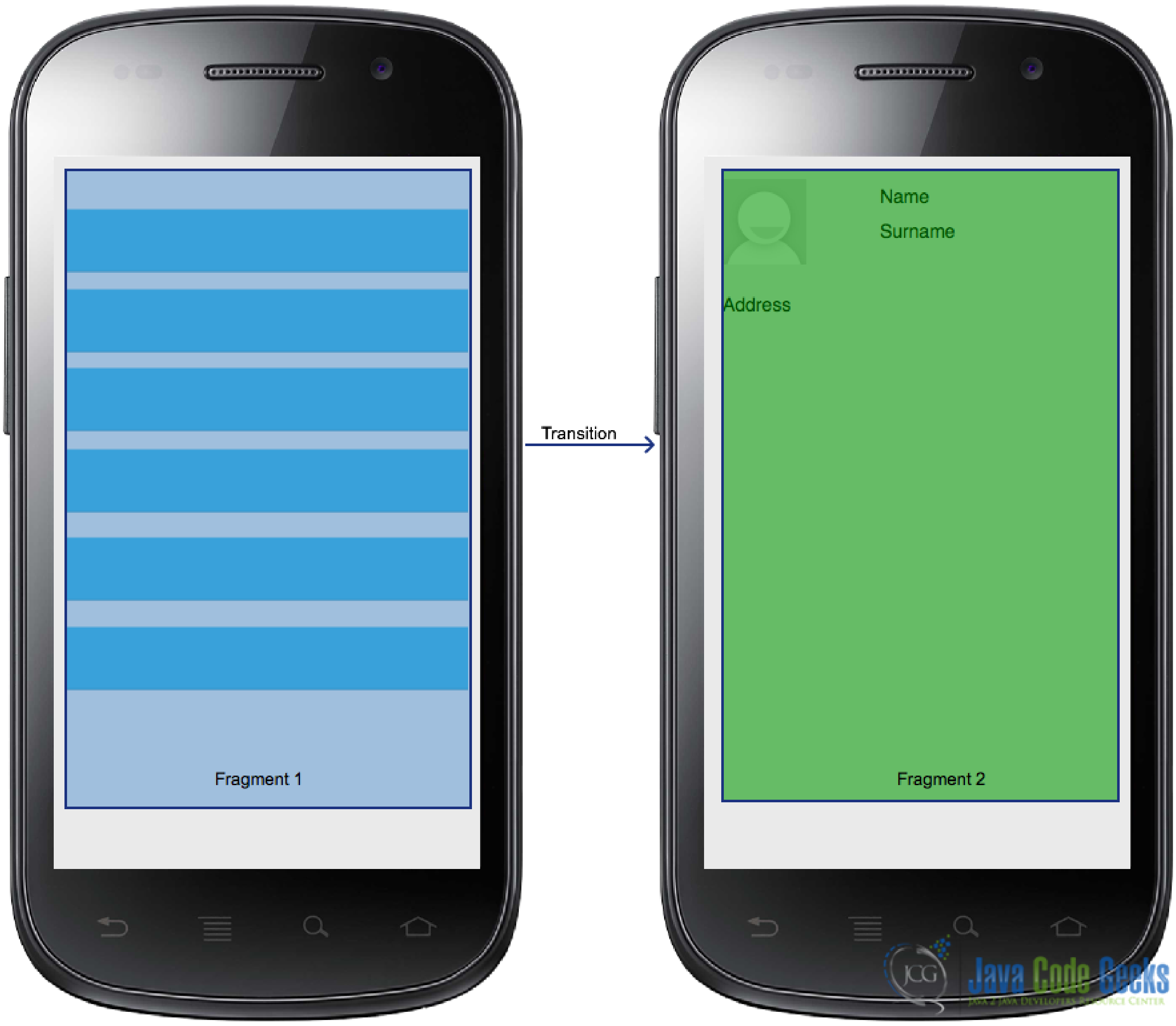


Figure 3.10: screenshot

while in a tablet:

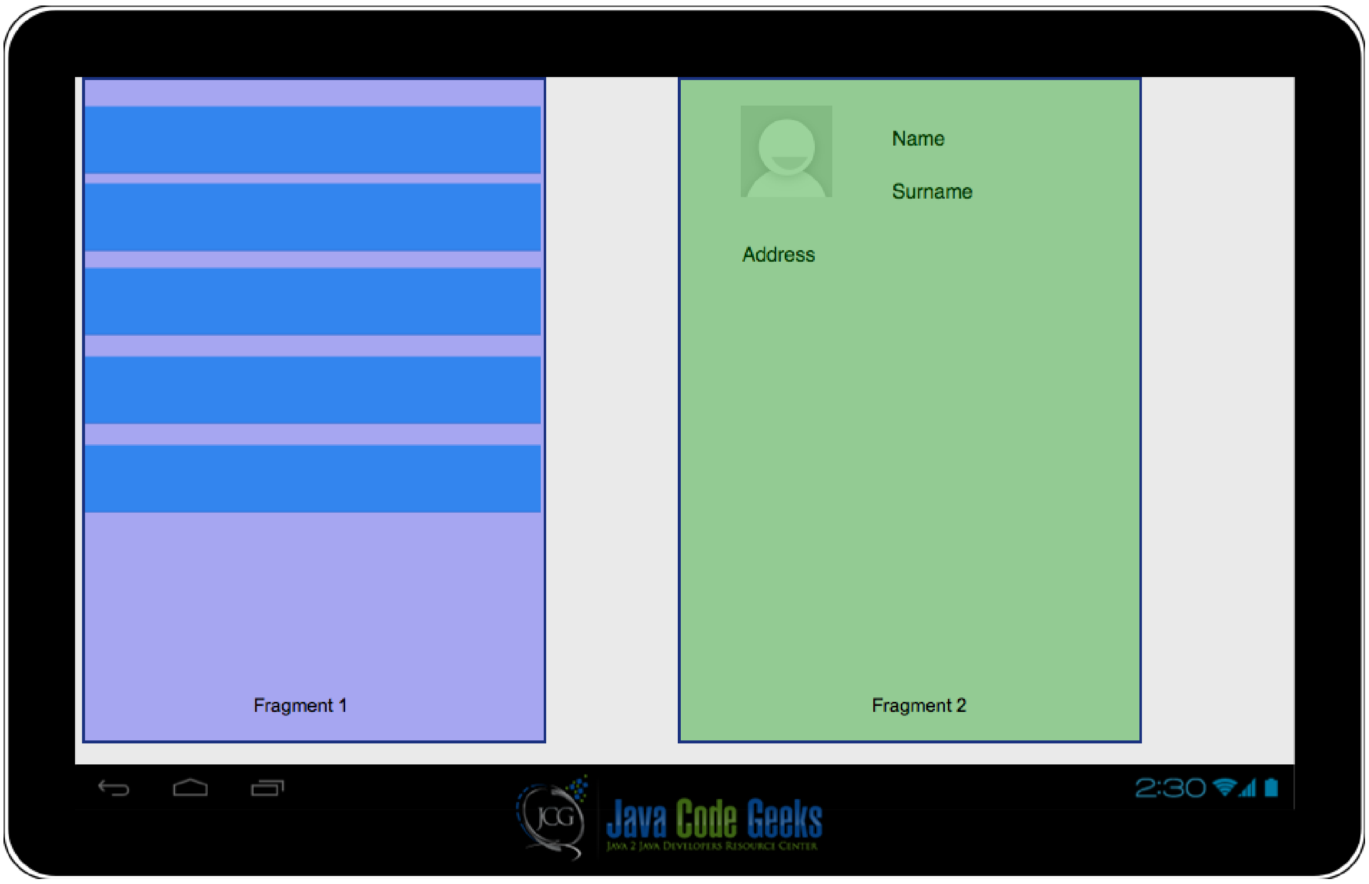


Figure 3.11: screenshot

3.3.1 Fragment lifecycle

Now that we know when we should use fragments, we need to know how they work before using them. A fragment has its own lifecycle as an `Activity` has, but it is more complex in comparison to the activity lifecycle. Moreover, a fragment lives only inside an `Activity` that acts as its container. Below, the fragment lifecycle is shown:

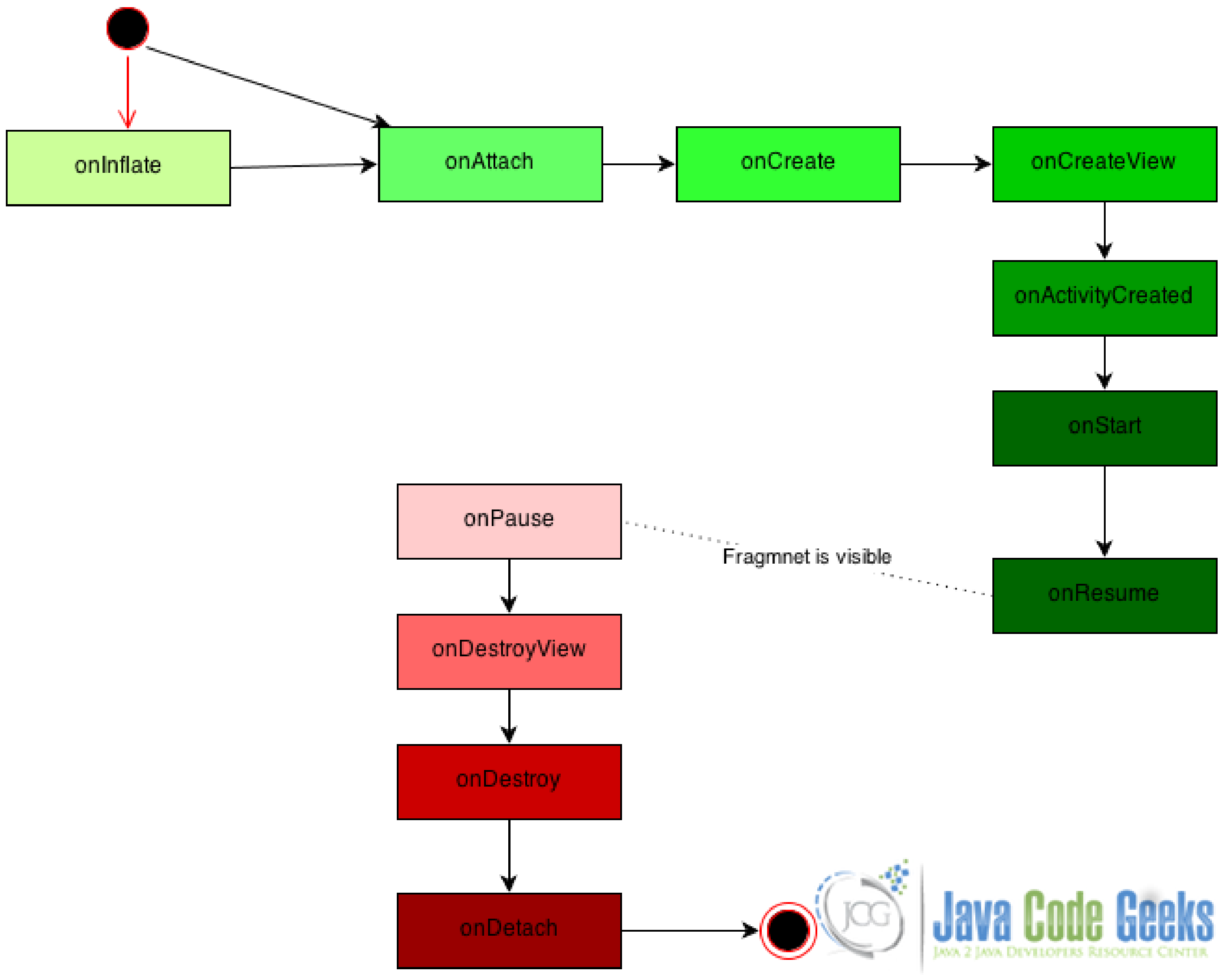


Figure 3.12: screenshot

As we can see, this lifecycle has more states in comparison with the activity lifecycle. Moving from the top to the bottom, we have:

- `onInflate`: This method is called only if we define fragment directly in our layout using the tag. In this method we can save some configuration parameter and some attributes define in the XML layout file.
- `onAttach`: This method is called as soon as the fragment is "attached" to the "father" activity and we can use this method to store the reference about the activity.
- `onCreate`: It is one of the most important steps, our fragment is in the creation process. This method can be used to start some thread to retrieve data information, maybe from a remote server.
- `onCreateView`: It is the method called when the fragment has to create its view hierarchy. During this method we will inflate our layout inside the fragment. During this phase we cannot be sure that our activity is still created so we cannot count on it for some operation.
- `OnActivityCreated`: In this method, we are notified when the "father" activity is created and ready for use. From now on, our activity is active and created and we can use it when we need.

- onStart: Here we do the common things as in the activity onStart, during this phase our fragment is visible but it isn't still interacting with the user.
- onResume: This method is called when the fragment is ready to interact with user. At the end of this phase our fragment is up and running!

Then, it is possible that the activity might be paused and so the activity's onPause is called. Well, in this case the onPause fragment method is called too. After that, it is possible that the OS decides to destroy our fragment view and so the onDestroyView is called. After that, if the system decides to dismiss our fragment, it calls the onDestroy method. Here we should release all the active connections because our fragment is close to shutting down. Even during the destroy phase, it is still attached to the father activity. The last step is to detach the fragment from the activity and it happens when the onDetach is called.

3.3.2 How to use fragments

Once we know the fragment lifecycle, we need to know how we can create a fragment and how we attach it to the Activity. Creating a fragment is very simple, we have just to extend an Android class called android.app.Fragment . We can suppose, we want to create a fragment called Fragment1. In this case we have:

```
public class Fragment1 extends Fragment {
}
```

In this way, we have created a fragment. In this class we can override callback methods so that we handle different states in the fragment lifecycle. By now, we can suppose that this fragment has a very simple layout:

```
<LinearLayout 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="Test Fragment1" />

</LinearLayout>
```

In this case this layout just has a TextView component that writes a text on the screen. If we want to "attach" this layout to our fragment we can do it in the onCreateView method, because according to the fragment lifecycle, this method is called when the fragment creates its view hierarchy. So in our Fragment1 class we can override the onCreateView method and implement our logic:

```
@Override
public View onCreateView(LayoutInflater inflater, ViewGroup container,
        Bundle savedInstanceState) {
        View v = inflater.inflate(R.layout.text_frag, container, false);
        return v;
}
```

In this method, we inflate our fragment layout called text_frag (the one shown above). Now we have created the fragment and we need to attach it to the Activity that holds it, because we know that a fragment exists only inside an Activity. We can do this operation in two ways:

- declaring the fragment inside the activity layout
- declaring a place holder inside the activity layout and attach the fragment inside the activity

In the first case, this operation is static, meaning we attach the fragment permanently to the activity. In the second case we can manage fragments at runtime, so we can replace a fragment with another one for example.

If we declare the fragment inside the layout we have to use the fragment tag.

Moreover, we have to specify the full qualified class name. For example, if the Fragment1 class is under the package com.swa, for the activity layout we will have:

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

    <fragment
        android:id="@+id/f1"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        class="com.sswa.Fragment1" />

</LinearLayout>
```

Running the example above, we will get us:

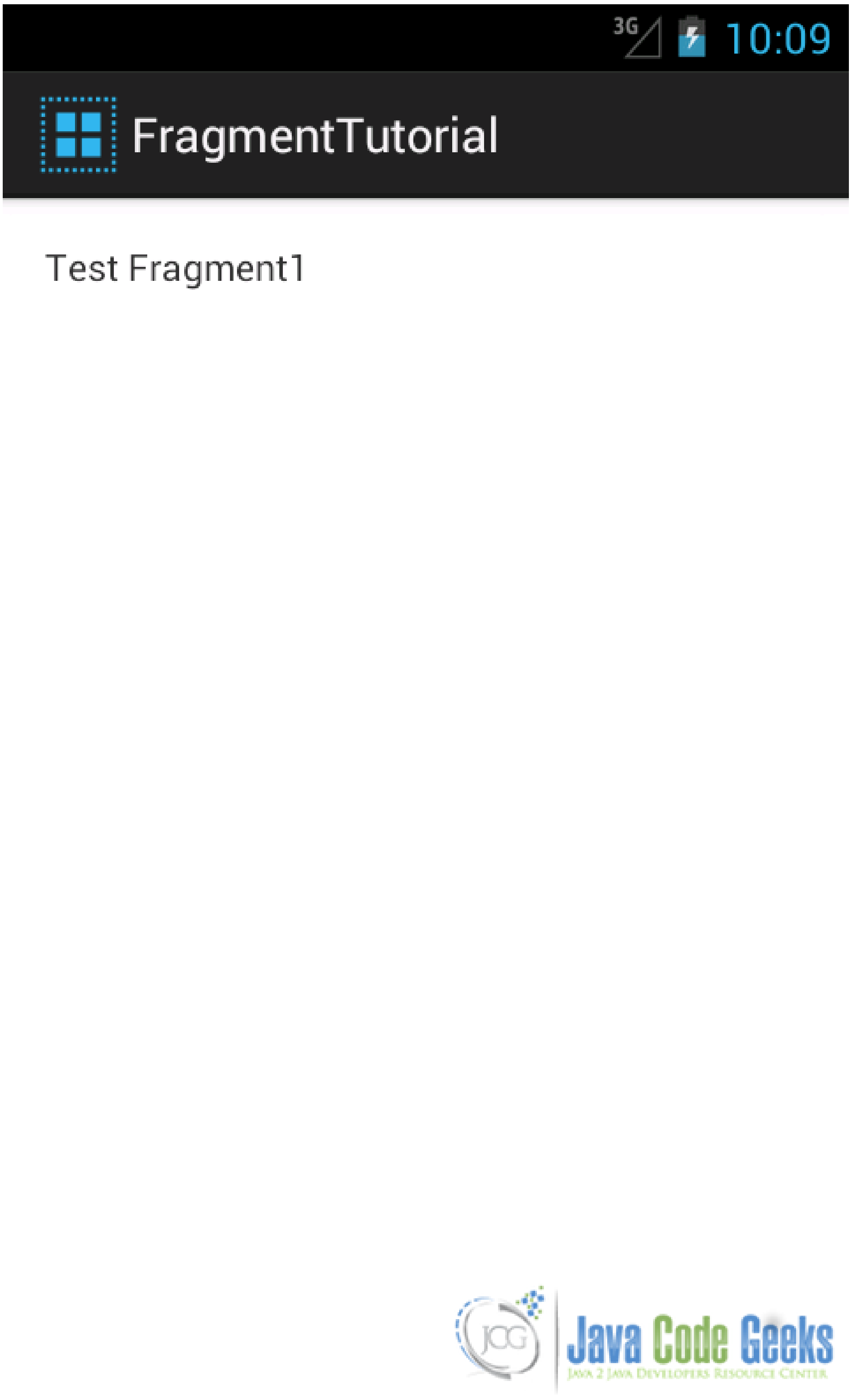


Figure 3.13: screenshot

A bit more complex is the second option where we want to manage fragments dynamically.

3.3.3 FrameLayout and FragmentManager

When we want to handle fragments dynamically inside our `Activity` we cannot use fragment tag in the activity layout but we have to use a place holder that we will “fill” with the fragment UI. In this case, the activity layout becomes:

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

    <FrameLayout
        android:id="@+id/fl1"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content" />

</LinearLayout>
```

In the `Activity`, we have to use a component that helps us to handle fragments. This component is called `FragmentManager`. Using this component we can add, remove, replace fragments at runtime and also to find fragments.

Note that before doing any kind of operations on a fragment, we need to activate a transaction. Using the `FragmentManager` we can create a transaction, start it and commit it at the end, when the fragment operation is done. So in the `Activity` we have:

```
public class MainActivity extends Activity {

        @Override
        protected void onCreate(Bundle savedInstanceState) {
                super.onCreate(savedInstanceState);
                setContentView(R.layout.activity_main_din);

                // Create fragment
                Fragment1 f1 = new Fragment1();
                FragmentManager fm = getFragmentManager();
                FragmentTransaction ft = fm.beginTransaction();
                ft.add(R.id.fl1, f1);
                ft.commit();
        }

}
```

3.3.4 Fragment Activity communication

Often we need to exchange information between a fragment and the activity that holds it. We need to consider that we can re-use fragments with different activities, so we cannot bind the fragment to a specific activity.

Inside a fragment, we can retrieve the activity instance using the `getActivity()` method. If we want to exchange data, it is a good practice to create a callback interface inside the fragment and require that the activity implements it.

For example, we can suppose to add a button to the fragment1 UI and when the user clicks the button we want to pass this information to the activity. The first thing we need to do is to modify our fragment layout:

```
<LinearLayout 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="Test Fragment Dynamic" />

    <Button
        android:id="@+id/btn1"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Click me" />

</LinearLayout>
```

The second step is to create an interface like this:

```
public class Fragment1 extends Fragment {
        public static interface FragmentListener {
                public void onClickButton() ;
        }
}
```

Now that we have the interface, our activity has to implement it:

```
public class MainActivity extends Activity implements
                Fragment1.FragmentListener {
        @Override
        public void onClickButton() {
                // Handle here the event
        }
}
```

Now we have to check, in the fragment class, if the activity implements the interface so that we can notify the event, when it occurs. The best place to do it, remembering the fragment lifecycle, is in the `onAttach` method, so we have:

```
public class Fragment1 extends Fragment {

        @Override
        public void onAttach(Activity activity) {
                super.onAttach(activity);
                if (!(activity instanceof FragmentListener))
                        throw new ClassCastException();

        }
}
```

We are, finally, ready to notify the event when the user clicks on the button and we can do it in the `onCreateView` method:

```
public class Fragment1 extends Fragment {

        @Override
        public View onCreateView(LayoutInflater inflater, ViewGroup container,
                        Bundle savedInstanceState) {
                View v = inflater.inflate(R.layout.text_frag, container, false);
                Button b = (Button) v.findViewById(R.id.btn1);
                b.setOnClickListener(new View.OnClickListener() {
                        @Override
                        public void onClick(View v) {
                                ((FragmentListener) getActivity()).onClickButton();
                        }
                });
                return v;
        }
}
```

3.3.5 Inter fragment communication

When developing an android app using fragments, we not only need to exchange data between the fragment and the activity that contains it, but we also need to exchange information between two fragments. In this case, the best practice is to avoid a fragment to fragment communication, so that we can guarantee that fragments are decoupled.

If we want to exchange data in ths manner, we can pass it from the first fragment to the activity, using the method described before, and then from the activity to the destination fragment. In this way, we can be sure that our fragments can be re-used in different situations.

3.3.6 Multiple screen support using fragments

At the beginning of this article, we explained when we should use fragments. We specified that they are very useful when we want to support different screen sizes, for example when we build an app that runs on a smartphone and on a tablet too. In this case just using layouts is not enough.

It is time ti explore how to create an UI that works on a tablet and a smartphone. In order to achieve this, we can revisit the example described before. In this case, we can suppose that when a user clicks on the button then the app shows a message. If we have a smartphone, when user clicks on the button then we move to another activity that shows the message, while if we use a tablet we want to show this message on the same screen.

First things first, we code a second fragment with a very simple layout that just shows a message and we call this fragment `Fragment2`:

```
public class Fragment2 extends Fragment {
        @Override
        public View onCreateView(LayoutInflater inflater, ViewGroup container,
                        Bundle savedInstanceState) {
                View v = inflater.inflate(R.layout.frag_message, container, false);
                return v;
        }
}
```

The layout is:

```
<LinearLayout 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="Hello world" />

</LinearLayout>
```

If the app runs on a smartphone, then the layout is similar to the one described previously, while some special consideration has to be done in the case of a tablet. We can suppose, for simplicity, that tablet has a landscape layout, so we can create, under `res` directory, a new directory called `layout-land`:

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

    <FrameLayout
        android:id="@+id/fl1"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content" />
    <FrameLayout
        android:id="@+id/fl2"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content" />

</LinearLayout>
```

Notice that we have two `FrameLayout` in this case. The “heavy” work is done by the `Activity`. If in the current layout attached to it there is no `FrameLayout` with id equal to `f2`, then we have a single fragment layout, meaning we are using a smartphone, otherwise we are using a fragment.

In the callback method in the activity we have:

```
import wkkwc.com;

@Override
        public void onClickButton() {
                // Handle here the event
                if (findViewById(R.id.fl2) != null) {
                        // We have dual fragment layout

                        Fragment2 f2 = (Fragment2) getFragmentManager().findFragmentById(R. ↩
                            id.fl2);

                        if (f2 == null) {
                                // Fragment2 is not inizialized
                                f2 = new Fragment2();
                                FragmentTransaction ft = getFragmentManager(). ↩
                                    beginTransaction();
                                ft.replace(R.id.fl2, f2);
                                ft.commit();
                        }

                }
                else {
                        // Single fragment layout
                        Fragment2 f2 = new Fragment2();
                        FragmentTransaction ft = getFragmentManager().beginTransaction();
                        ft.replace(R.id.fl1, f2);
                        ft.commit();
                }
        }
```

Running the app, we have in the smartphone:

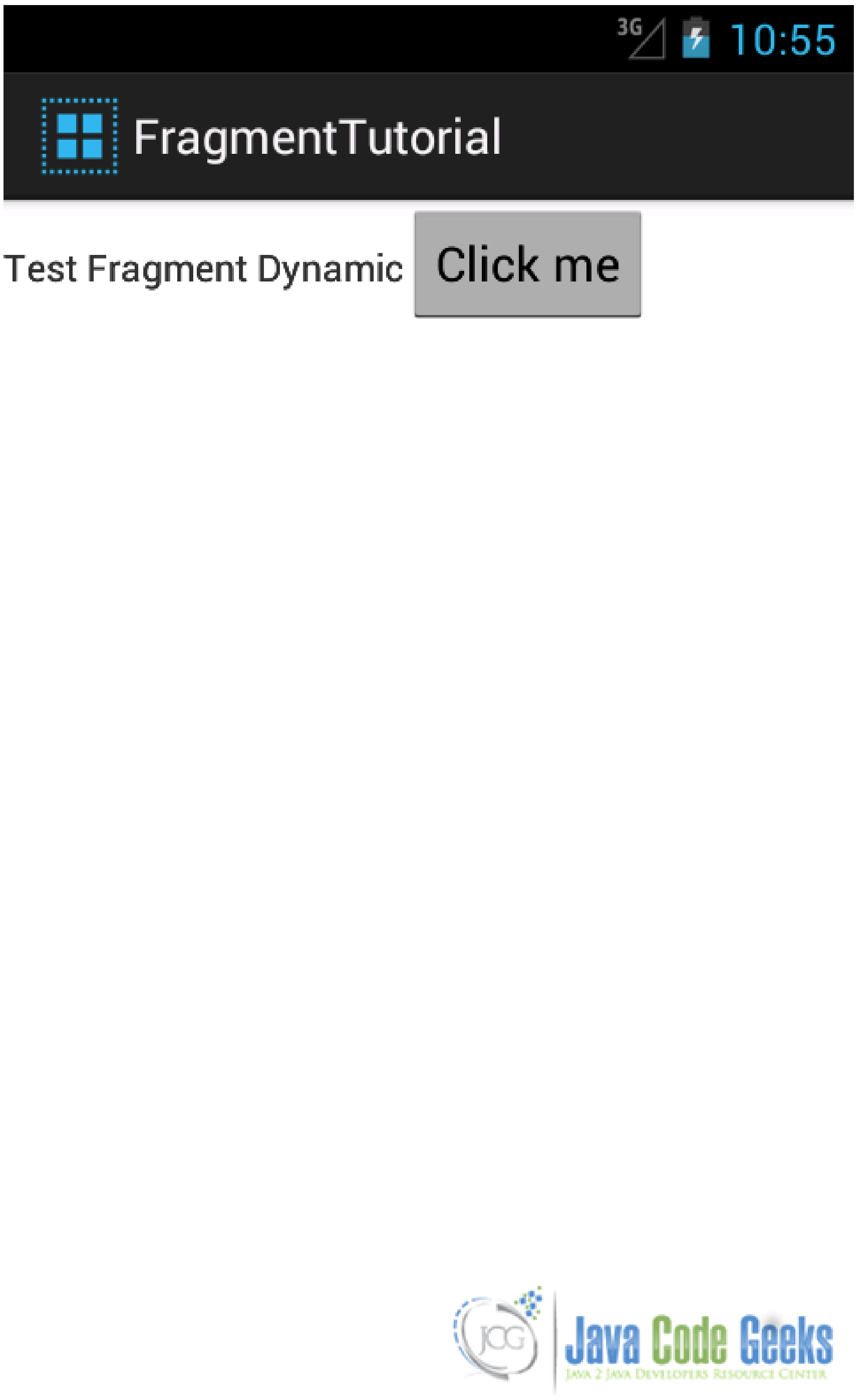


Figure 3.14: screenshot

10:57
FragmentTutorial
Hello world
Java Code Geeks
Java 2 Java Developers Resource Center

Figure 3.15: screenshot

while, if we run the app on a tablet, we have:

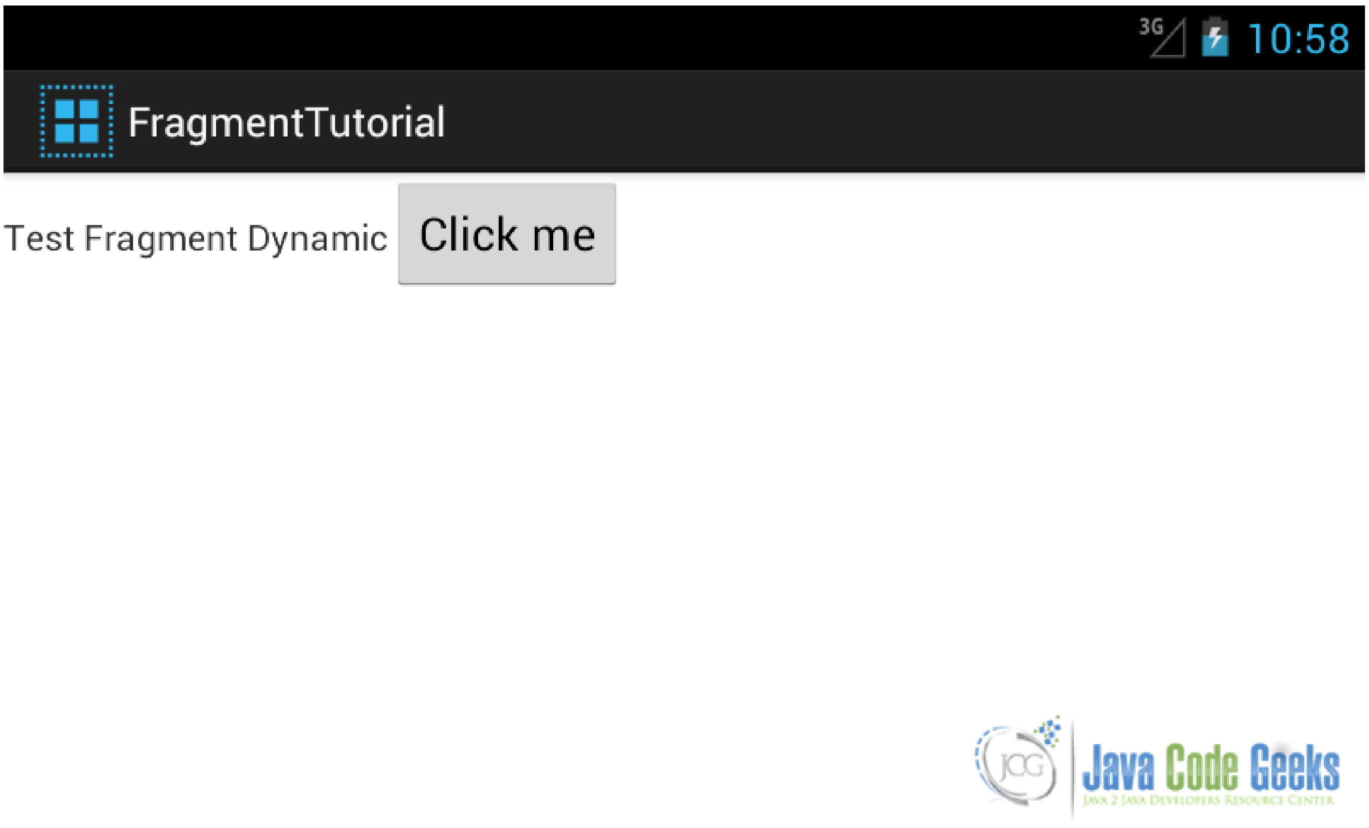


Figure 3.16: screenshot

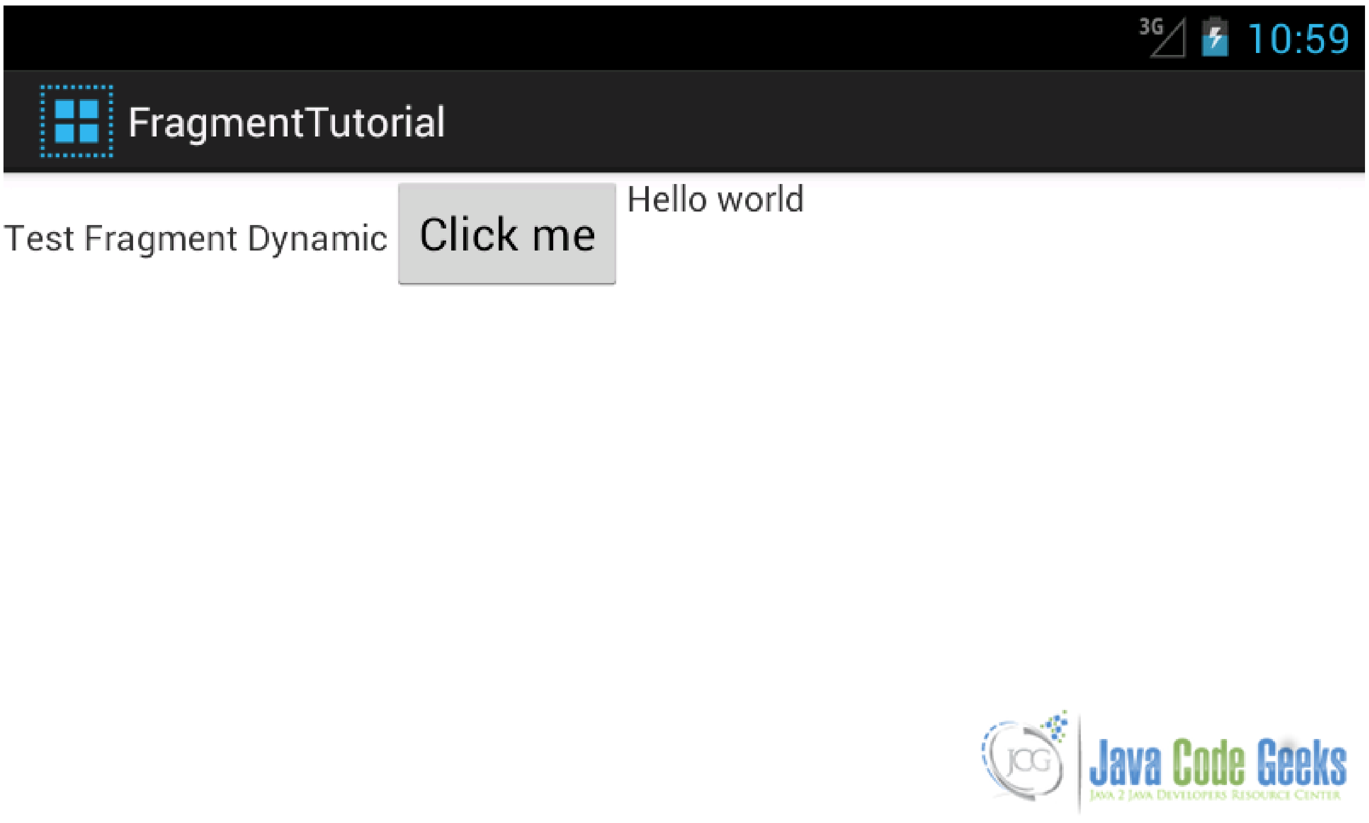


Figure 3.17: screenshot

3.4 Download the Source Code

This was a lesson of how to create Android Layouts with ViewGroups and Fragments. You may download the source code here: AndroidLayout.zip

Chapter 4

Android UI: Adding Multimedia to an App

4.1 Overview

In this article, we will cover some multimedia and graphic aspects in Android. The Android SDK provides a set of APIs to handle multimedia files, such as audio, video and images. Moreover, the SDK provides other API sets that help developers to implement interesting graphics effects, like animations and so on.

The modern smart phones and tablets have an increasing storage capacity so that we can store music files, video files, images etc. Not only the storage capacity is important, but also the high definition camera makes it possible to take impressive photos. In this context, the Multimedia API plays an important role.

4.2 Multimedia API

Android supports a wide list of audio, video and image formats. You can give a look here to have an idea; just to name a few formats supported:

Audio

- MP3
- MIDI
- Vorbis (es: mkv)

Video

- H.263
- MPEG-4 SP

Images

- JPEG
- GIF
- PNG

Android, moreover, can handle local files, meaning files that are stored inside the smart phone or tablet or remote file using data streaming. We can leverage these capabilities in order to build very interesting apps.

All the classes provided by the Android SDK that we can use to add multimedia capabilities to our apps are under the `android.media` package. In this package, the heart class is called `MediaPlayer`. This class has several methods that we can use to play audio and video file stored in our device or streamed from a remote server.

This class implements a state machine with well-defined states and we have to know them before playing a file. Simplifying the state diagram, as shown in the official documentation, we can define these macro-states:

- **Idle state:** When we create a new instance of the MediaPlayer class.
- **Initialization state:** This state is triggered when we use `setDataSource` to set the information source that `MediaPlayer` has to use.
- **Prepared state:** In this state, the preparation work is completed. We can enter in this state calling `prepare` method or `prepareAsync`. In the first case after the method returns the state moves to `Prepared`. In the async way, we have to implement a listener to be notified when the system is ready and the state moves to `Prepared`. We have to keep in mind that when calling the `prepare` method, the entire app could hang before the method returns because the method can take a long time before it completes its work, especially when data is streamed from a remote server. We should avoid calling this method in the main thread because it might cause a ANR (Application Not Responding) problem. Once the `MediaPlayer` is in prepared state we can play our file, pause it or stop it.
- **Completed state:** Te end of the stream is reached.

We can play a file in several ways:

```
// Raw audio file as resource
MediaPlayer mp = MediaPlayer.create(this, R.raw.audio_file);
// Local file
MediaPlayer mp1 = MediaPlayer.create(this, Uri.parse("file:///...."));
// Remote file
MediaPlayer mp2 = MediaPlayer.create(this, Uri.parse("http://website.com"));
```

or we can use `setDataSource` in this way:

```
MediaPlayer mp3 = new MediaPlayer();
mp3.setDataSource("http://www.website.com");
```

Once we have created our `MediaPlayer` we can “prepare” it:

```
mp3.prepare();
```

and finally we can play it:

```
mp3.start();
```

Please keep in mind the observations above regarding preparing the state. According to them, we can use an async operation so that we will not stop the main thread. In this case, we have:

```
// Remote file
MediaPlayer mp2 = MediaPlayer.create(this, Uri.parse("http://website.com"));
mp2.setAudioStreamType(AudioManager.STREAM_MUSIC);
mp2.setOnCompletionListener(new MediaPlayer.OnCompletionListener() {
        @Override
        public void onCompletion(MediaPlayer mp) {
                mp.start();
        }
});
mp2.prepareAsync();
```

We used a listener to be notified when the `MediaPlayer` is in the prepared state so we can start playing. At the end, when we don’t need the instance of `MediaPlayer` anymore, we should release it:

```
mp2.release();
```

4.2.1 Using Android Camera

If we want to add to our apps the capability to take photos using the integrated smart phone camera, then the best way is to use an `Intent`. For example, let us suppose we want to start the camera as soon as we press a button and show the result in our app.

In the `onCreate` method of our `Activity`, we have to setup a listener of the `Button` and when clicked to fire the intent:

```
Button b = (Button) findViewById(R.id.btn1);
b.setOnClickListener(new View.OnClickListener() {
        @Override
        public void onClick(View v) {
                 // Here we fire the intent to start the camera
                Intent i = new Intent(MediaStore.ACTION_IMAGE_CAPTURE);
                startActivityForResult(i, 100);
          }
});
```

In the `onActivityResult` method, we retrieve the picture taken and show the result:

```
@Override
protected void onActivityResult(int requestCode, int resultCode, Intent data) {
        // This is called when we finish taking the photo
           Bitmap bmp = (Bitmap) data.getExtras().get("data");
           iv.setImageBitmap(bmp);
}
```

Running the app we have:

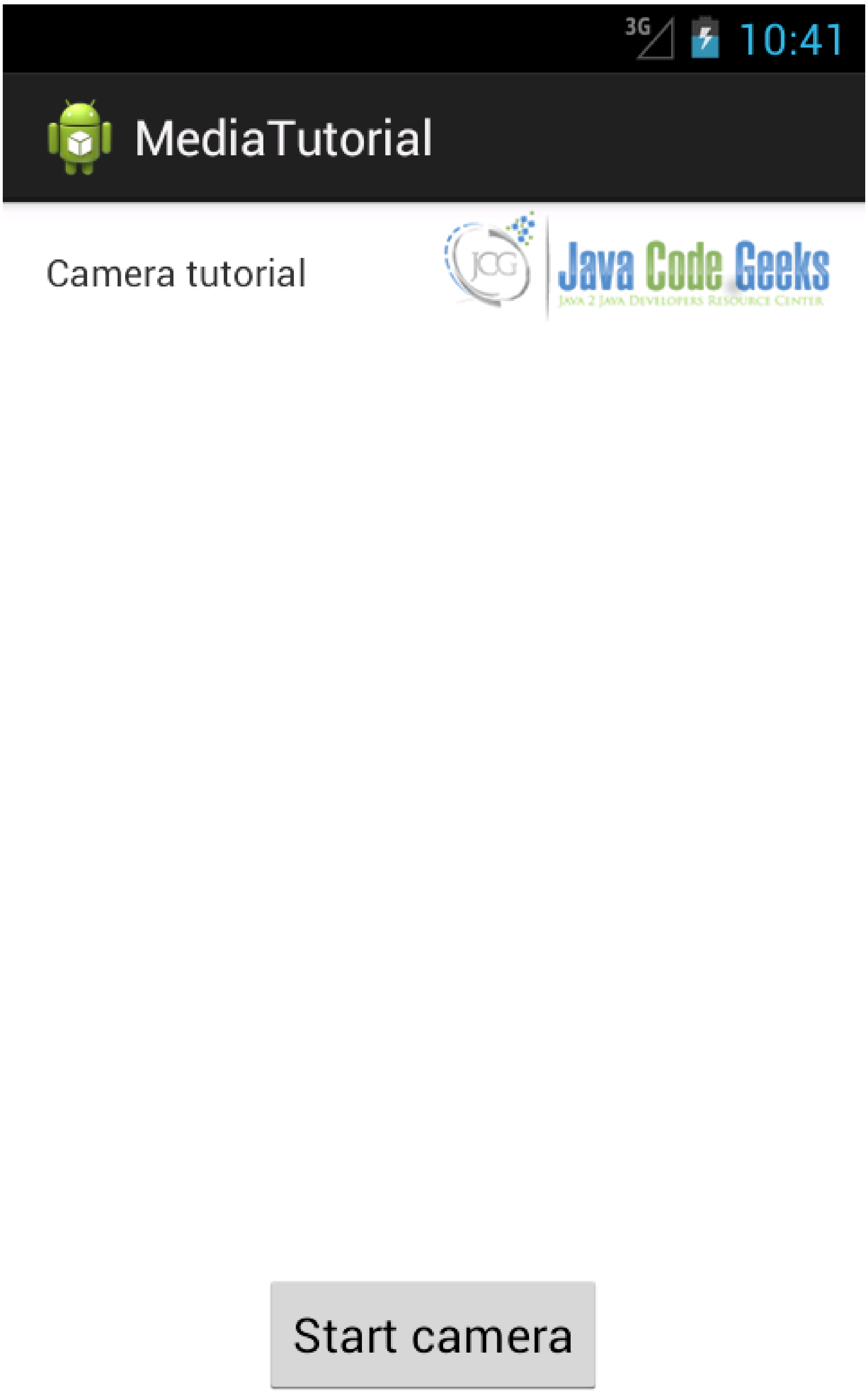


Figure 4.1: screenshot

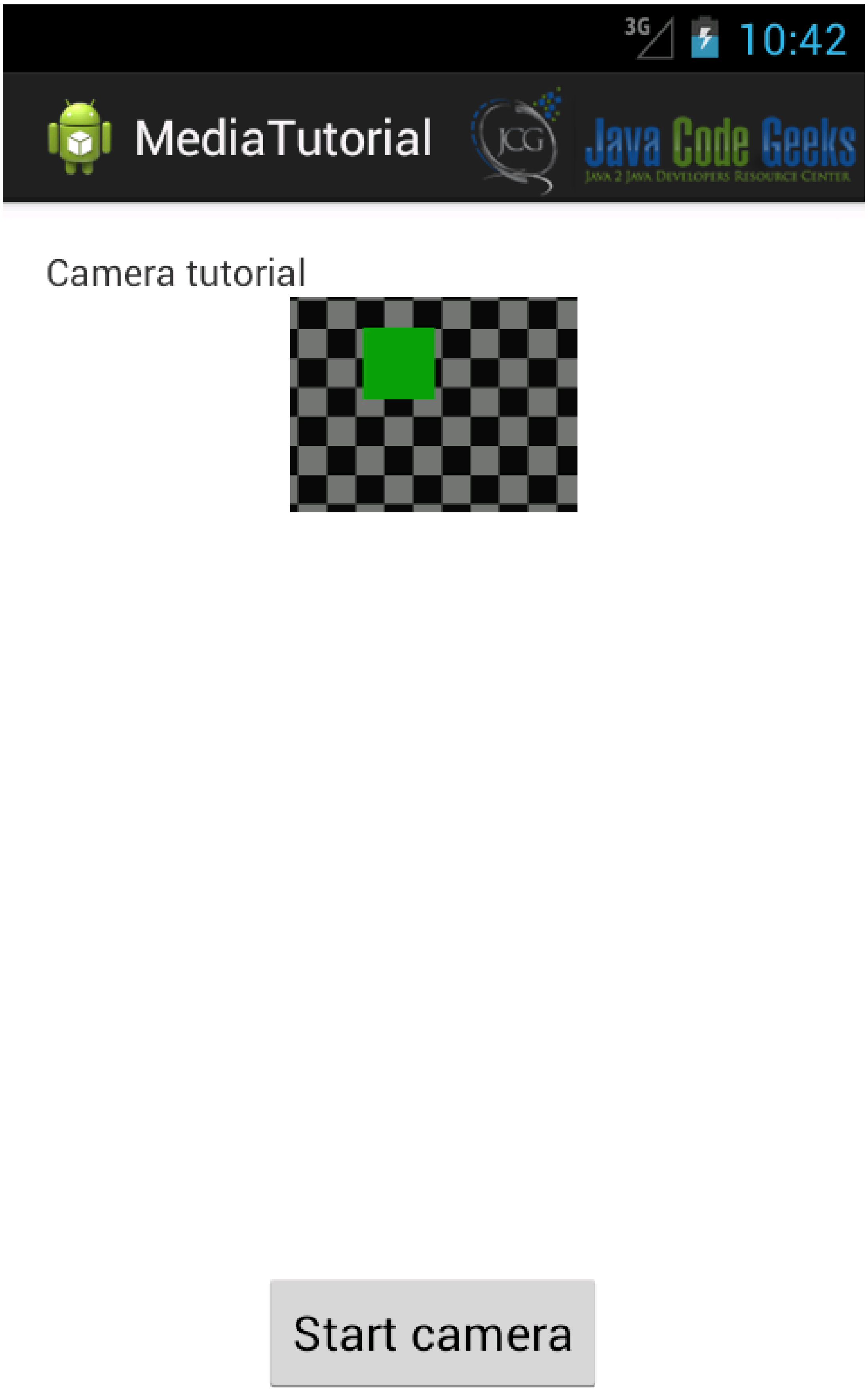


Figure 4.2: screenshot

In the example above, we used an emulated camera.

4.3 Graphics

By now, we talked about standard components that we can be used in our UI. This is good but it is not enough when we want to develop a game or an app that requires graphic contents. Android SDK provides a set of API for drawing custom 2D and 3D graphics. When we write an app that requires graphics, we should consider how intensive the graphic usage is. In other words, there could be an app that uses quite static graphics without complex effects and there could be other app that uses intensive graphical effects like games.

According to this usage, there are different techniques we can adopt:

- **Canvas and Drawable:** In this case, we can extend the existing UI widgets so that we can customize their behavior or we can create custom 2D graphics using the standard method provided by the `Canvas` class.
- **Hardware acceleration:** We can use hardware acceleration when drawing with the `Canvas` API. This is possible from Android 3.0.
- **OpenGL:** Android supports OpenGL natively using NDK. This technique is very useful when we have an app that uses intensively graphic contents (i.e games).

The easiest way to use 2D graphics is extending the `View` class and overriding the `onDraw` method. We can use this technique when we do not need a graphics intensive app.

In this case, we can use the `Canvas` class to create 2D graphics. This class provides a set of method starting with `draw*` that can be used to draw different shapes like:

- lines
- circle
- rectangle
- oval
- picture
- arc

For example let us suppose we want do draw a rectangle. We create a custom view and then we override `onDraw` method. Here we draw the rectangle:

```
public class TestView extends View {
        public TestView(Context context) {
                super(context);
        }
        public TestView(Context context, AttributeSet attrs, int defStyle) {
                super(context, attrs, defStyle);

        }
        public TestView(Context context, AttributeSet attrs) {
                super(context, attrs);
        }

        @Override
        protected void onDraw(Canvas canvas) {
                super.onDraw(canvas);
                Paint p = new Paint();
                p.setColor(Color.GREEN);
                p.setStrokeWidth(1);
                p.setStyle(Paint.Style.STROKE);
                canvas.drawRect(5, 5, 120, 120, p);
                invalidate();
        }
}
```

As it is clear from the code above, in the `onDraw` method, we used the `drawRect Canvas` method. Notice that we used another class called `Paint`. This class specifies how the shape will be drawn; it specifies its color, if it has to be filled, the border width and so on.

In this case the layout looks like:

```
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:paddingBottom="@dimen/activity_vertical_margin"
    android:paddingLeft="@dimen/activity_horizontal_margin"
    android:paddingRight="@dimen/activity_horizontal_margin"
    android:paddingTop="@dimen/activity_vertical_margin"
    tools:context=".MainActivity" >

    <com.swa.customview.TestView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content" />

</RelativeLayout>
```

Running the app, we have:

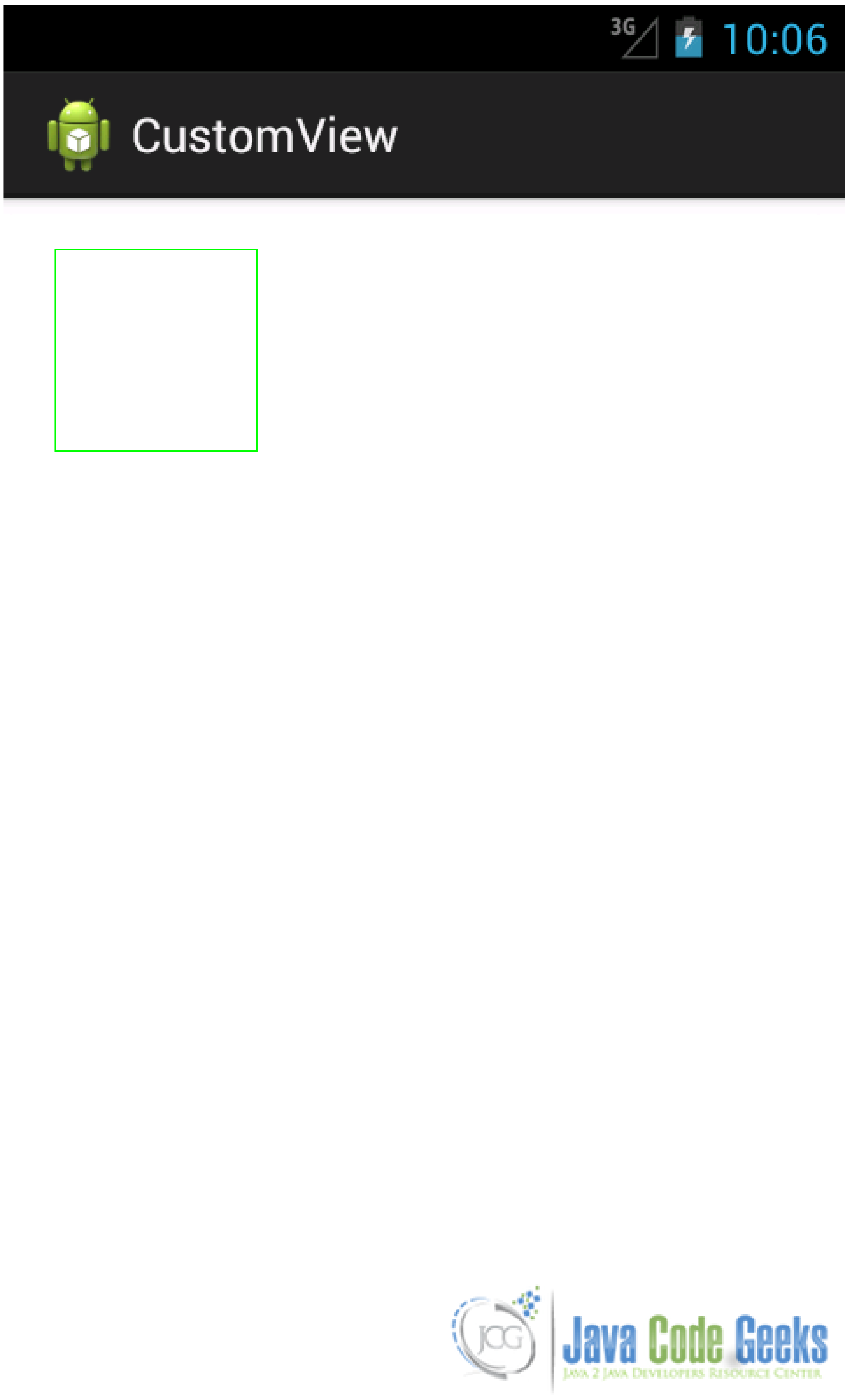


Figure 4.3: screenshot

Suppose we want to fill the rectangle with a gradient color, so the `onDraw` method becomes:

```
protected void onDraw(Canvas canvas) {
        super.onDraw(canvas);
        Paint p = new Paint();
        p.setColor(Color.GREEN);
        p.setStrokeWidth(1);
        p.setStyle(Paint.Style.FILL_AND_STROKE);
        LinearGradient lg = new LinearGradient(0F, 0F, 115F,115F, Color.GREEN,  Color. ←
            YELLOW, TileMode.CLAMP);
        p.setShader(lg);
        canvas.drawRect(5, 5, 120, 120, p);
        invalidate();
}
```

Running the app we have:

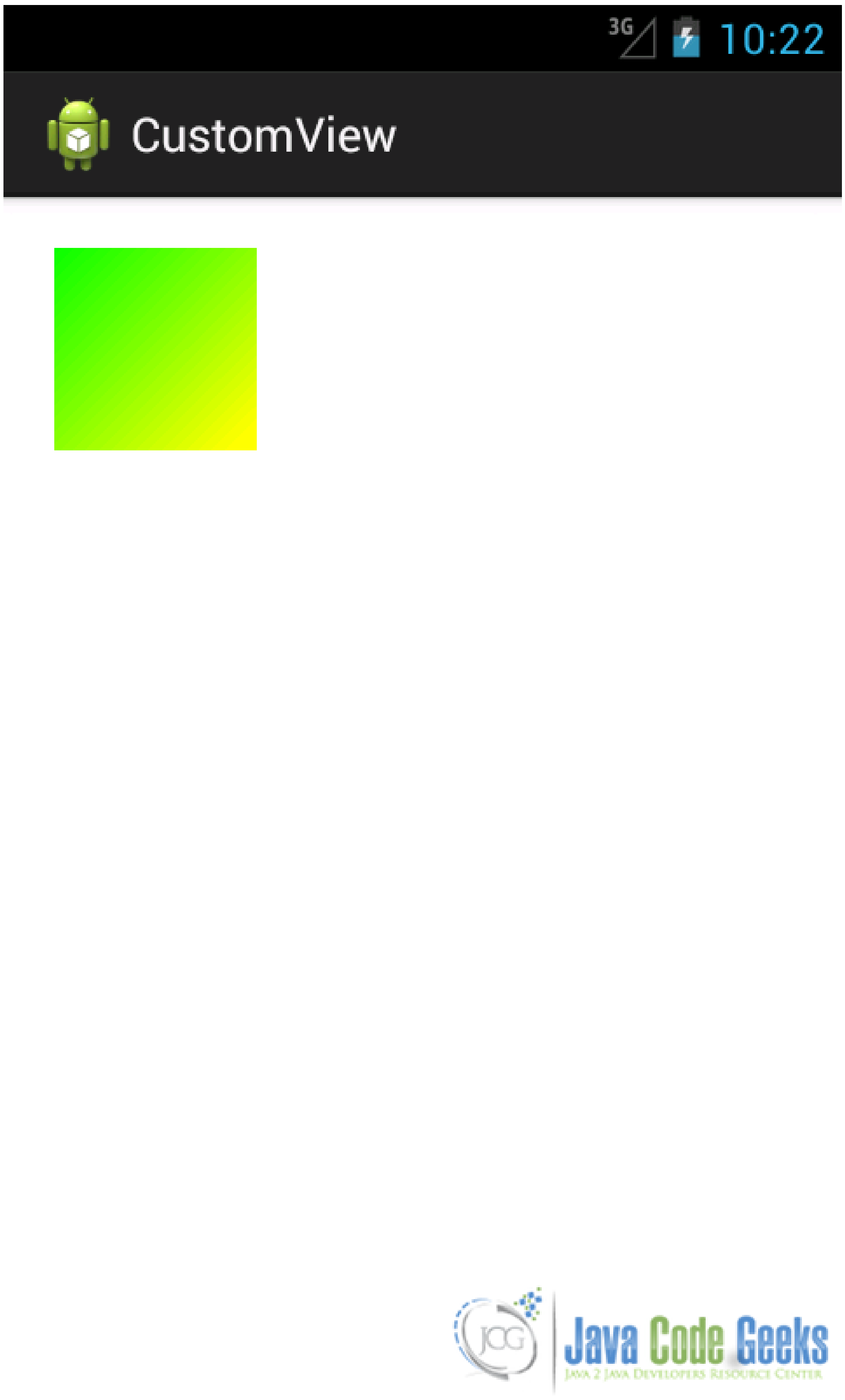


Figure 4.4: screenshot

As we told before, beginning from Android 3.0 (API 11), we can use hardware acceleration. In this case, if we want to use it, we have to modify the `Manifest.xml` and add the following line:

```
<application android:hardwareAccelerated="true" >
```

or we can use it at `Activity` level.

4.4 Drawable

In Android, a `Drawable` is a graphical object that can be shown on the screen. From API point of view all the `Drawable` objects derive from `Drawable` class. They have an important role in Android programming and we can use XML to create them. They differ from standard widgets because they are not interactive, meaning that they do not react to user touch.

Images, colors, shapes, objects that change their aspect according to their state, object that can be animated are all drawable objects. In Android under `res` directory, there is a sub-dir reserved for `Drawable`, it is called `res/drawable`.

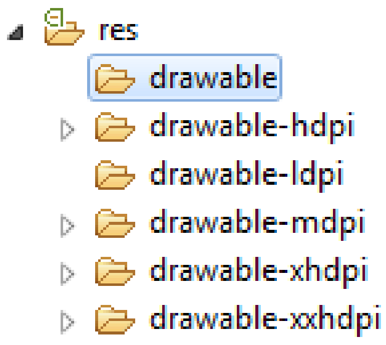


Figure 4.5: screenshot

Under the `drawable` dir we can add binary files like images or XML files.

As we saw in the previous chapters, we can create several directories according to the screen density we want to support. These directories have a name like `drawable-<>`.

This is very useful when we use images; in this case, we have to create several image versions: for example, we can create an image for the high dpi screen or another one for medium dpi screen. Once we have our file under `drawable` directory, we can reference it, in our class, using `R.drawable.file_name`. While it is very easy add a binary file to one of these directory, it is a matter of copy and paste, if we want to use a XML file we have to create it.

There are several types of drawable:

- Bitmap
- Nine-patch

- Layer list
- State list
- Level list
- Transition drawable
- Inset drawable
- Clip drawable
- Scale drawable
- Shape drawable

An interesting aspect is that we can create such elements using XML or directly from code. There is a correspondence between the elements shown above and the API class. We can add the `Drawable` suffix and we create the corresponding class name: for example if the corresponding class of `Bitmap` drawable is `BitmapDrawable` and so on.

You can have a look here if you want to have more information. We will not cover all these objects in this article but only the most popular.

4.4.1 Shape drawable

This is a generic shape. Using XML we have to create a file with shape element as root. This element as an attribute called `android:shape` where we define the type of shape like rectangle, oval, line and ring. We can customize the shape using child elements like:

Table 4.1: datasheet

Element name	**Description**
`gradient`	Specify the gradient color for the shape.
`solid`	Specify the background shape color.
`stroke`	Specify the border of the shape.
`corners`	Specify the corner radius of the shape.
`padding`	Specify the padding for the shape.
`size`	Specify the width and the height of the shape.

For example, let us suppose we want to create an oval with solid background color. We create a XML file called for example `oval.xml`:

```
<shape xmlns:android="http://schemas.android.com/apk/res/android"
    android:shape="oval" >

    <solid android:color="#FF0000" />

    <size
        android:height="100dp"
        android:width="120dp" />

</shape>
```

In this way, we create an oval shape having red as background color and with size 120dpx100dp. Then we can reference it in our layout file:

```
   <ImageView
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_centerInParent="true"
    android:src="@drawable/oval" />
```

Running the app, we obtain:

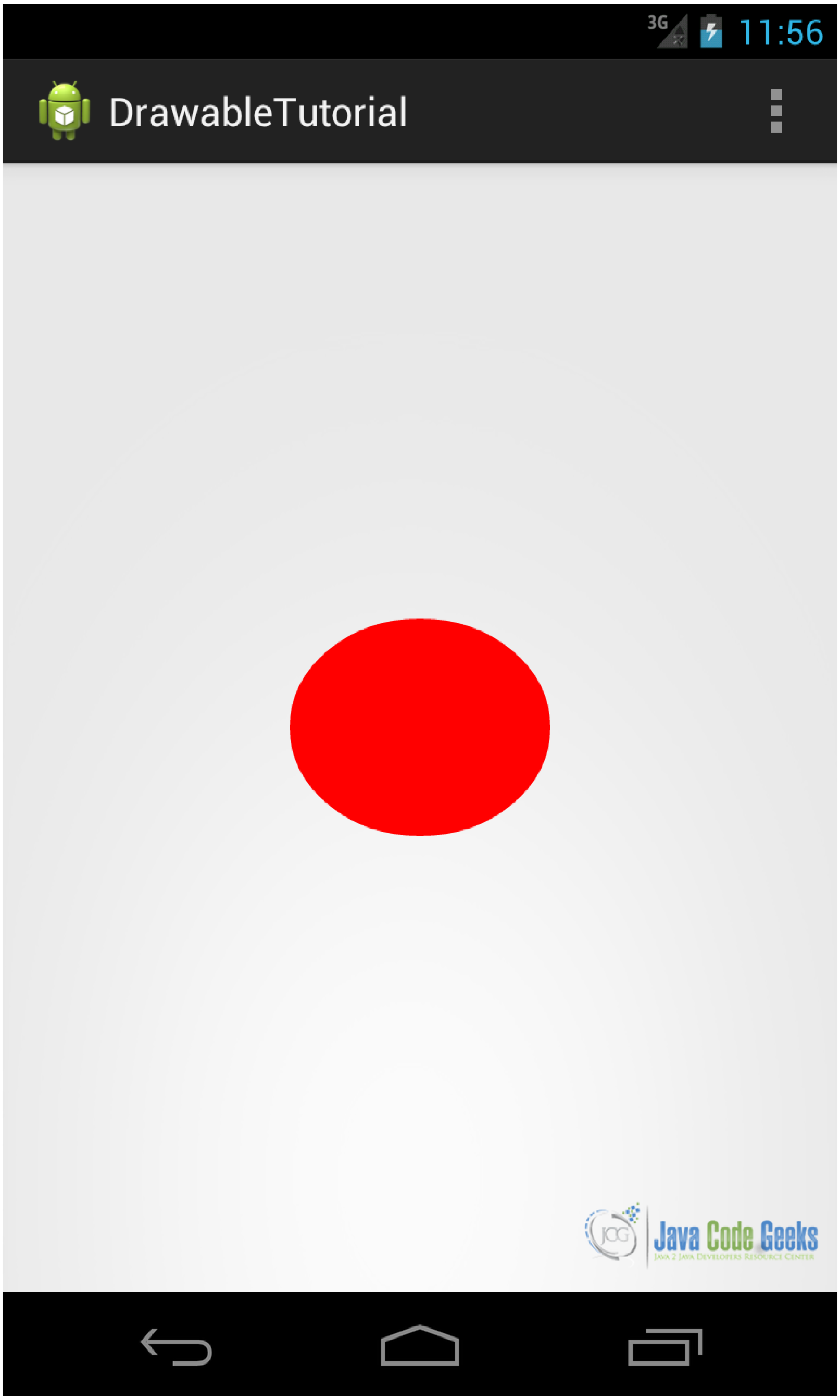


Figure 4.6: screenshot

For example, we can suppose we want to change the Button widget look. We want to create a rectangle with rounded corners and as background a gradient color. We define a shape in XML file called round_corner.xml and we add it to drawable dir:

```
<shape xmlns:android="http://schemas.android.com/apk/res/android"
    android:shape="rectangle" >

    <stroke
        android:width="2dp"
        android:color="#00FF00" />

    <gradient
        android:angle="-90"
        android:endColor="#FFFFFF"
        android:startColor="#00FF00"
        android:type="linear" />

    <corners android:radius="3dp" />

    <padding
        android:bottom="4dp"
        android:left="4dp"
        android:right="4dp"
        android:top="4dp" />

</shape>
```

and in the layout file we have:

```
<Button
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_centerInParent="true"
    android:background="@drawable/round_corner"
    android:text="Click me" />
```

Running the app, we have:

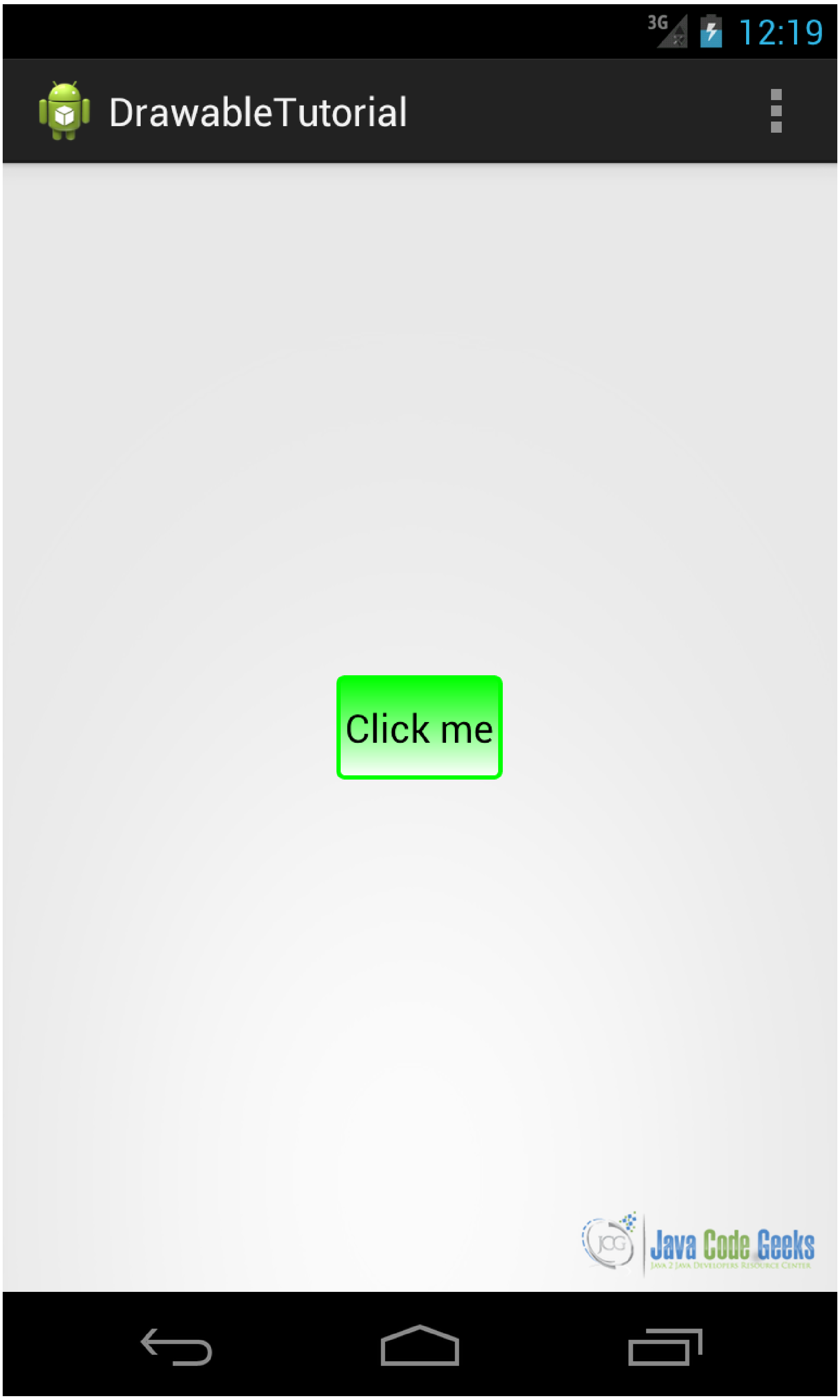


Figure 4.7: screenshot

As we can see, just using XML we can modify the widget background or create shapes.

4.4.2 State list

This drawable object can display several drawables depending on the object state. It is very useful when we want to customize some object that has an internal states. For example, the `Button` widget is one of these objects, it has several states: pressed, focused and so on.

In XML this drawable is represented by selector tag. This tag has item child elements:

```
<selector xmlns:android="http://schemas.android.com/apk/res/android">

    <item android:drawable=""drawable1" android:state_pressed="true"/>
    <item android:drawable=""drawable2" android:state_focused="true"/>

</selector>
```

Let's suppose we want to customize a `Button` widget in our layout when it is pressed. Additionally, we want to change its background to a red gradient color. So the first thing we do is to define two shapes:

```
<shape xmlns:android="http://schemas.android.com/apk/res/android" >

    <solid android:color="#00FF00" />

</shape>
```

green.xml

```
<shape xmlns:android="http://schemas.android.com/apk/res/android" >

    <gradient
        android:angle="-90"
        android:endColor="#FFFFFF"
        android:startColor="#AA0000"
        android:type="linear" />

</shape>
```

Once we have our shapes we can assign them to different object states:

```
<selector xmlns:android="http://schemas.android.com/apk/res/android">

    <item android:drawable="@drawable/red_gradient" android:state_pressed="true"/>
    <item android:drawable="@drawable/green"/>

</selector>
```

In this way, we assign the red_gradient drawable when the button is pressed and the green drawable in the default state. Running the app we have:

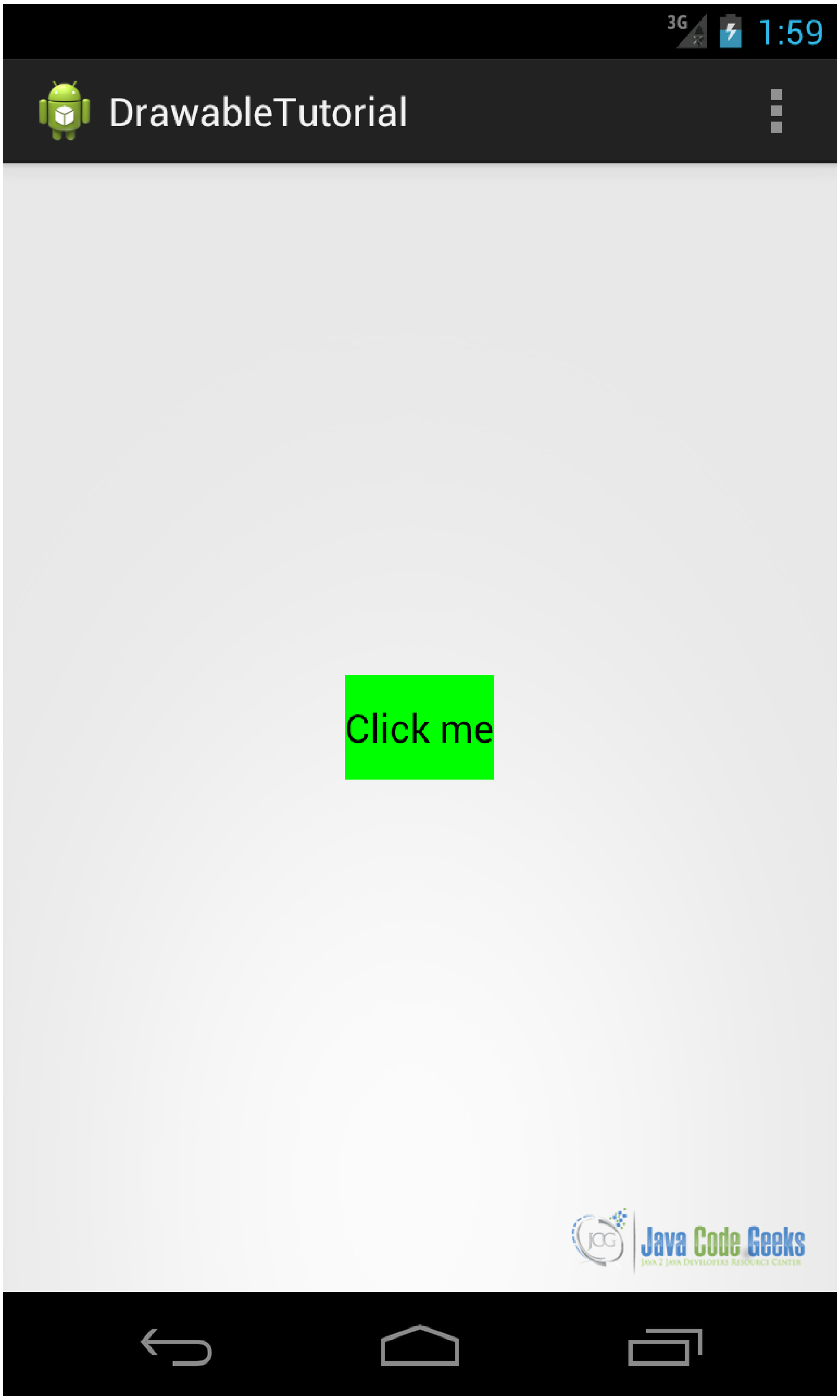


Figure 4.8: screenshot

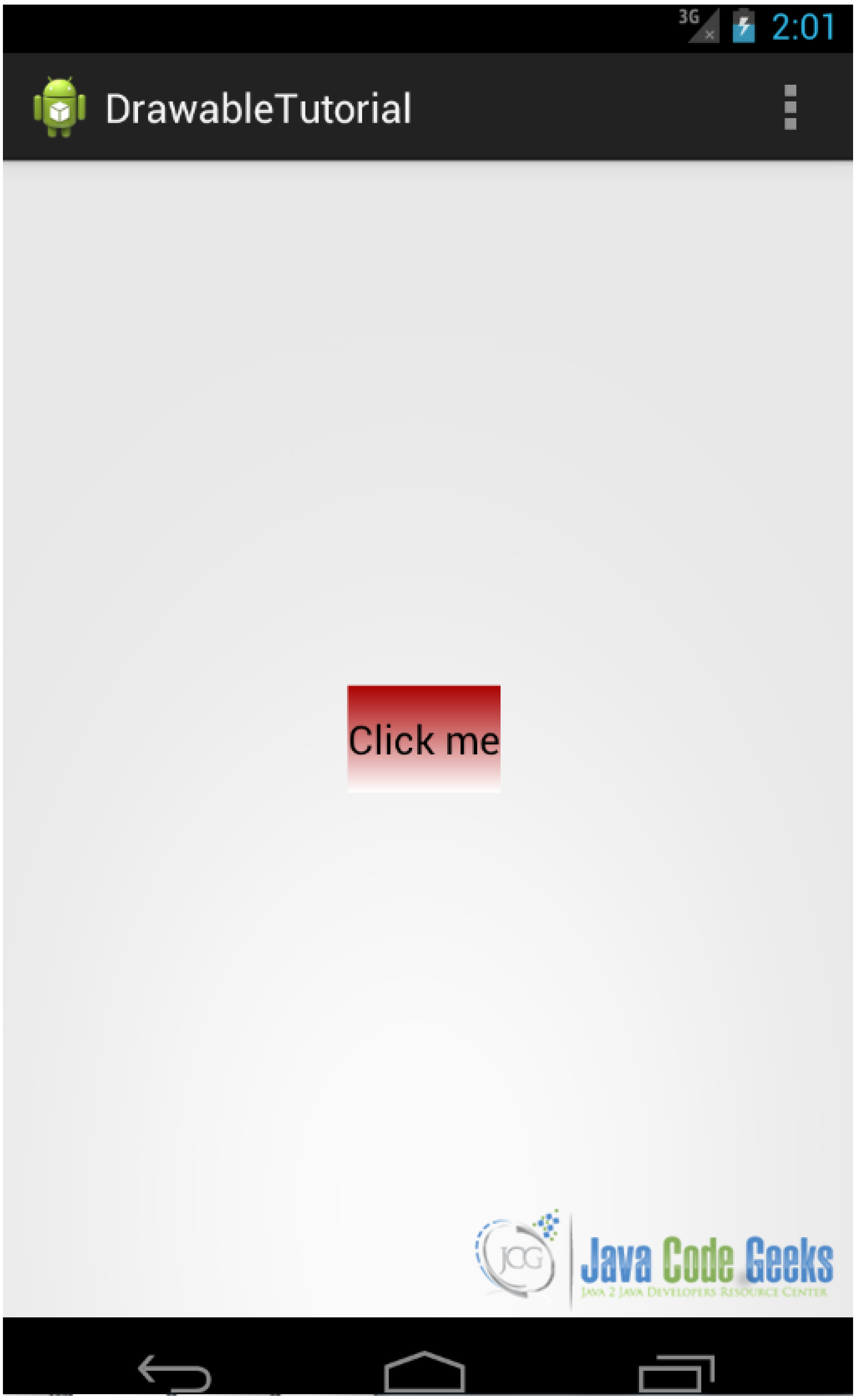


Figure 4.9: screenshot

4.4.3 Nine-patch

Nine-patch image is a special background image that can be resized, so that it can hold the `View` content. You can look here if you want to have more information. It can be used when we want to create an image but we do not know the exact size of the `View` content.

Briefly, while creating this image we define the borders that can be stretched and the static area. Android provides a tool to help us creating this kind of images located under the tools directory. Suppose we want to create a `Button` widget background, we can create an image like the one shown below:

Now we can run `draw9patch.bat` under the tools directory. Now we can drag&drop this image on the window just opened:

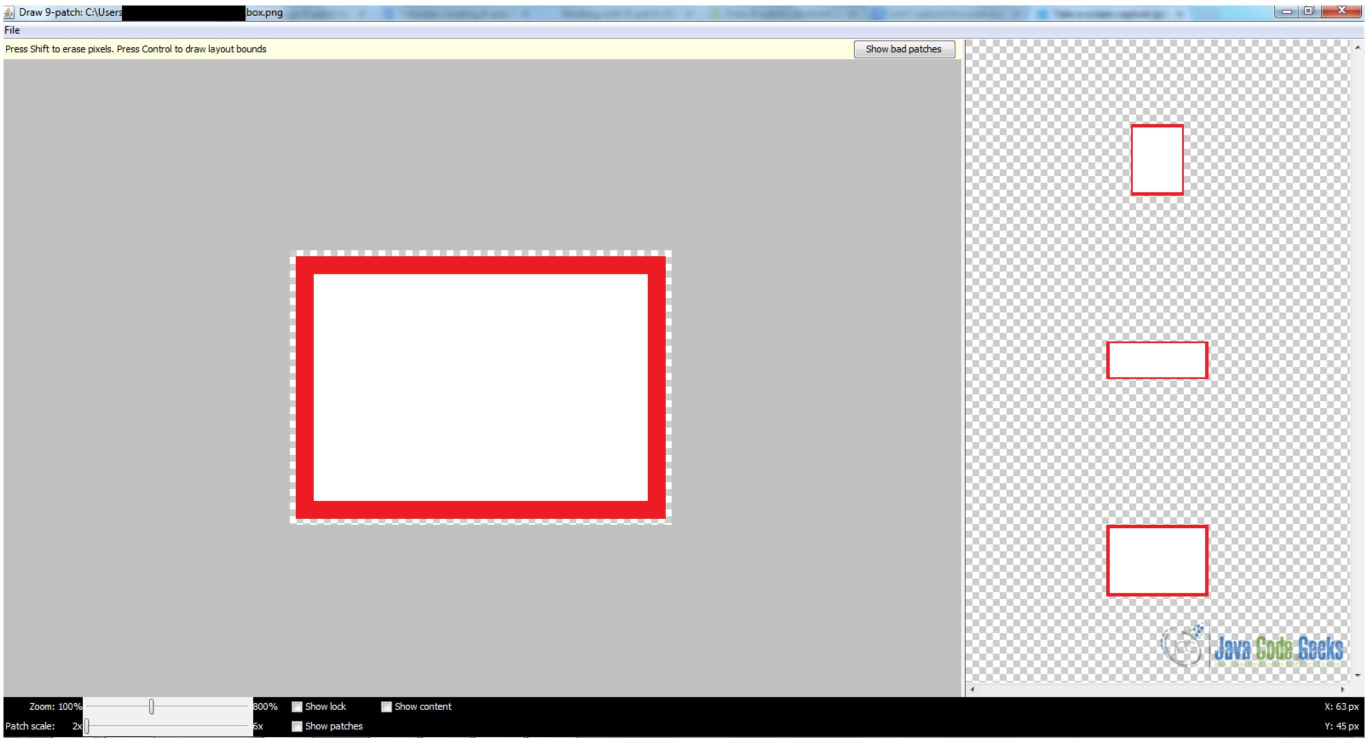


Figure 4.10: screenshot

The window is divided in two areas: the left one is the "working window" while on the right side we have the final result. Now we have to choose the area of the image that can scale, we can do it drawing lines on the left and top side, as you can see in the picture:

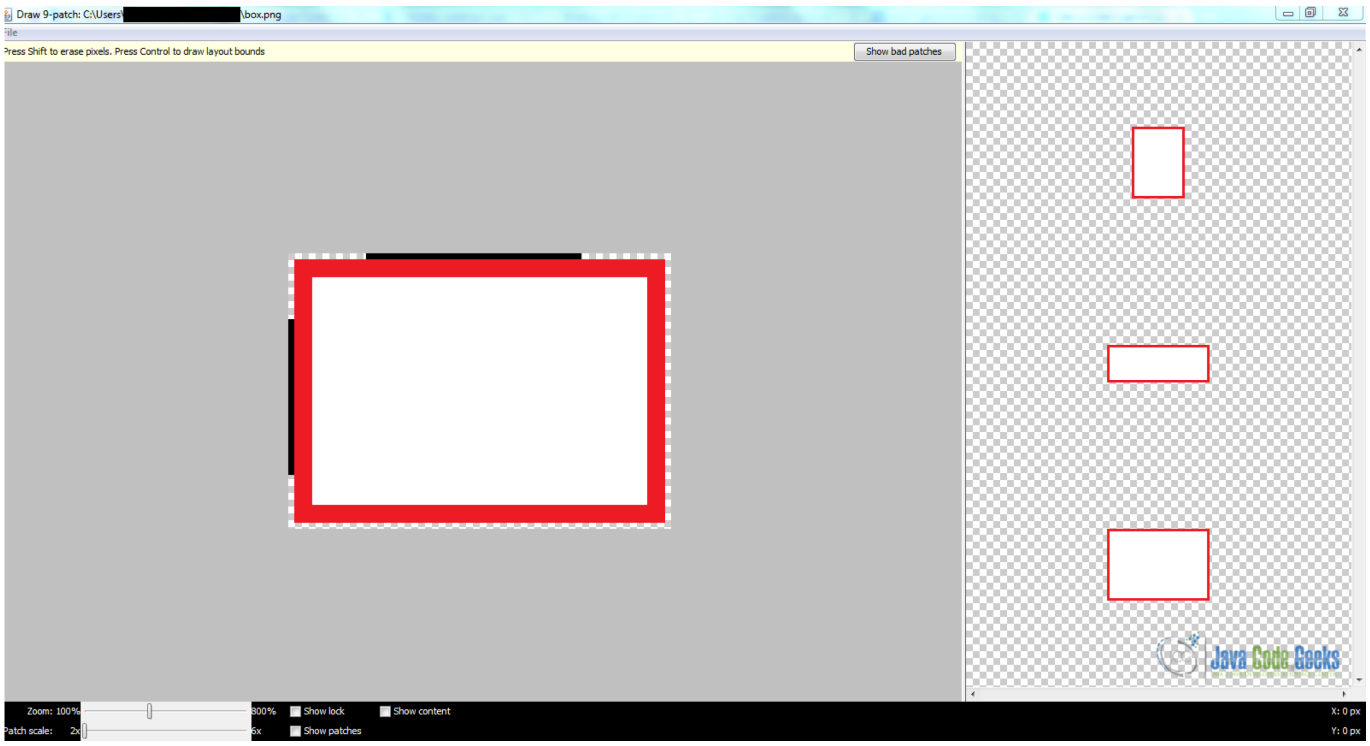


Figure 4.11: screenshot

Now we set the content area, selecting the right and bottom side of the image.

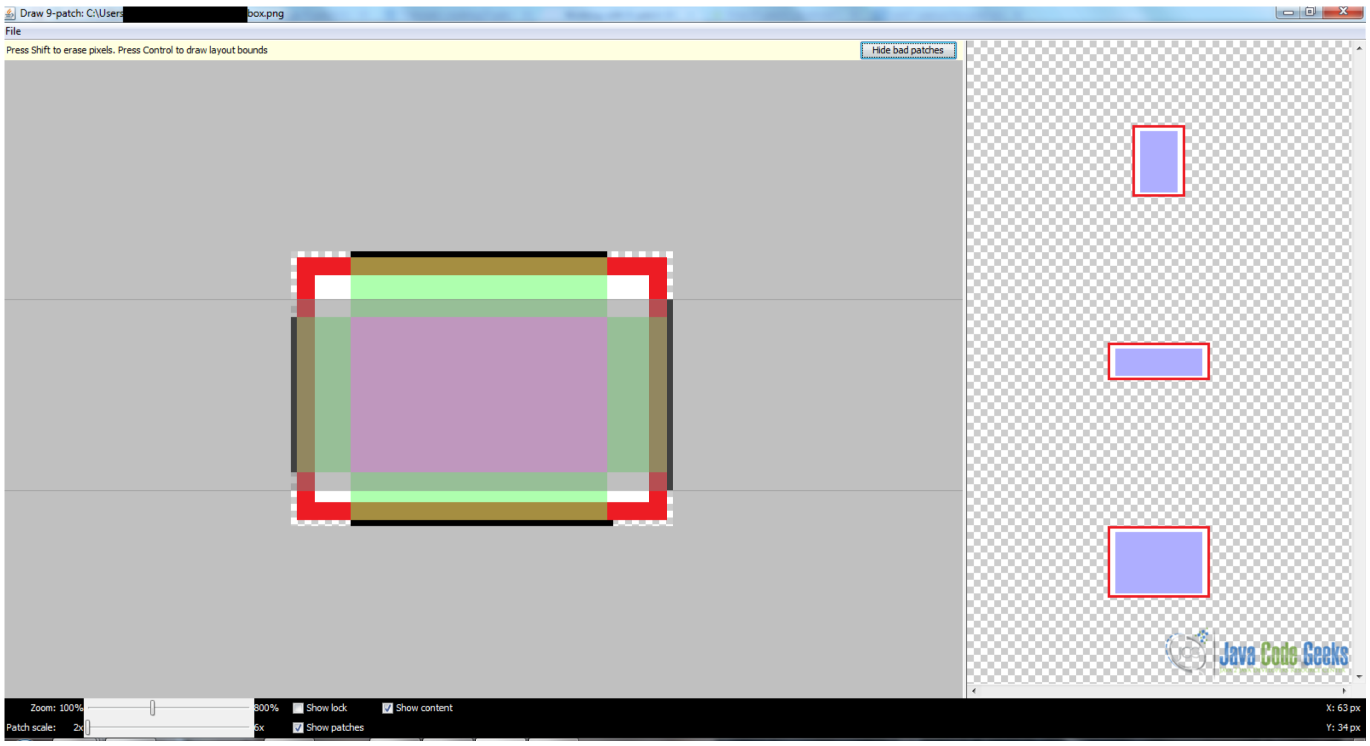


Figure 4.12: screenshot

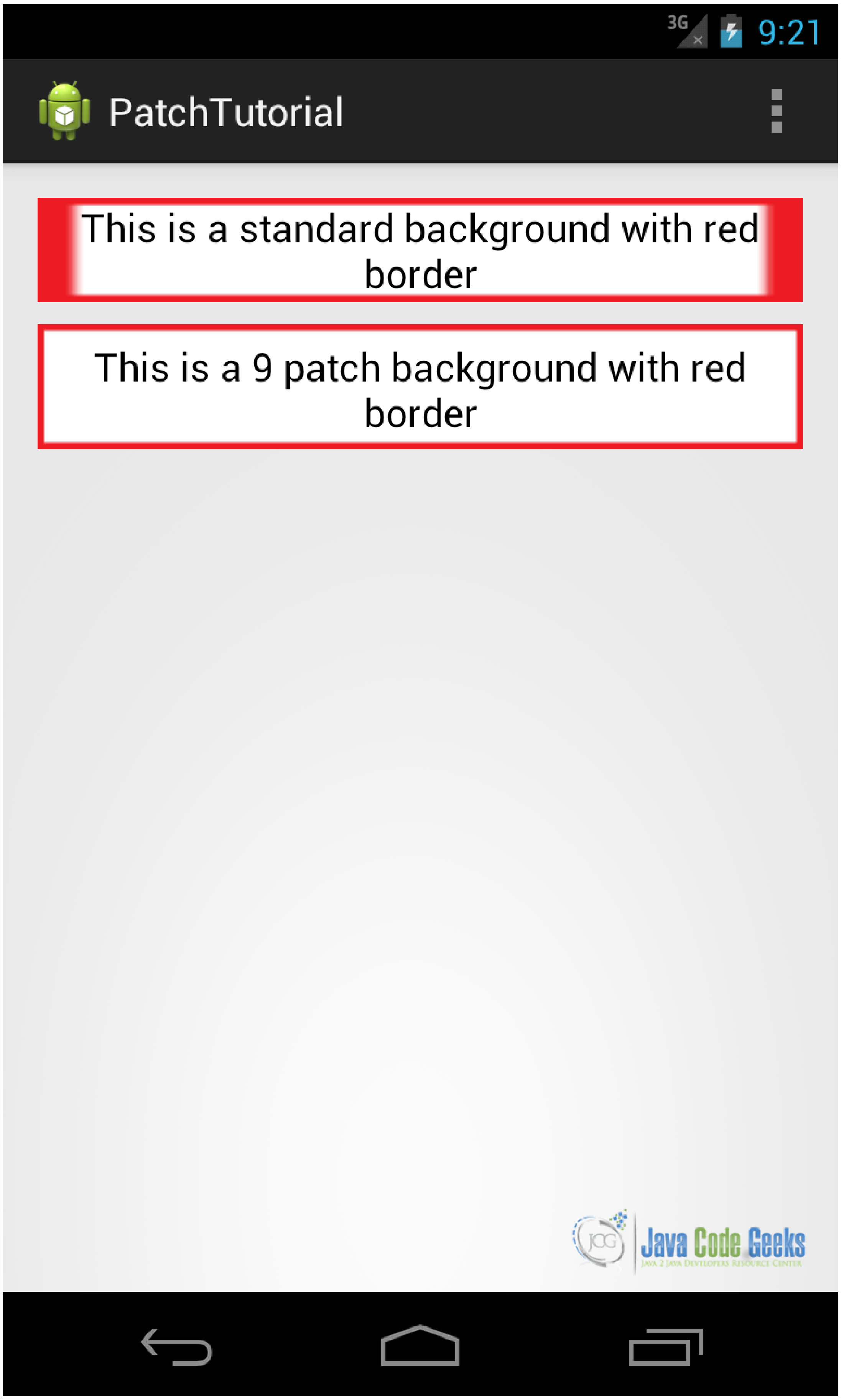
We can see on the right side the final result. Now we can save our work. When we have finished, we can copy this image under `res/drawable` of our Android project.

To see the final effect we can create a layout like the one shown below:

```
<Button
    android:id="@+id/btn1"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:background="@drawable/stdbox"
    android:text="This is a standard background with red border" />
<Button
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_below="@id/btn1"
    android:layout_marginTop="10dp"
    android:background="@drawable/box"
    android:text="This is a 9 patch background with red border" />
```

In the first button we used a standard image the simple red box as shown above, while in the second button we use a `9-patch` image.

Running the example we have:



This is a standard background with red border

This is a 9 patch background with red border

Figure 4.13: screenshot

You can notice the `9-patch` images scales better than the standard image.

4.5 Download the Source Code

This was a lesson on how to use Multimedia with Android. You may download the source code here:

- CustomView.zip
- DrawableTutorial.zip
- MediaTutorial.zip
- PatchTutorial.zip

Chapter 5

Android UI: Themes and Styles

5.1 Introduction

In the previous chapters, we talked about layout and how we can organize our Views so that we can create appealing user interfaces. We saw there are several layout manager and View provided by the Android SDK and that, if they are not enough, we can create custom layout manager and View. We also talked about drawable, which is very useful when we want to customize some widget.

In this article we will describe how we can use style and theme. This is an important aspect when developing an app because using style and theme we can brand our app and have an harmonious look.

While the layout and the Views specify the structure of the user interface, styles and themes define their look. What is a theme in Android?

A theme is a set of styles and can be applied to the whole app or to a single Activity. It helps to create an appealing user interface. A style, on the other hand, specifies a set of properties that have effects on how a View or Window looks like. These properties specify the font size, the font family, the background color, font color and so on. We can imagine that a style is like a CSS in a web page, it separates the content structure from how it is displayed on the screen.

When we create a layout in XML we use layout managers and Views. When using Views we could specify for example the background color, the font color, View size and so on. For example let us suppose we have a simple TextView like the one shown below:

```
<TextView
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:background="#FF0000"
    android:text="Hello style"
    android:textColor="#FFFFFF"
    android:textSize="14sp" >

</TextView>
```

Looking at the code above we notice there are some mistakes: we added to the View not only the information regarding its content but also information about how it looks like and we hard-coded it directly. This is a very bad practice, because if we want to change the text color we should look for each TextView in our user interface definition.

Moreover, in a consistent and appealing user interface where text messages have all the same size, we will have to replicate the text size information for each TextView in our layout. This is a classic example where using a style would avoid duplication of data, separate the layout structure from its appearance and improve the XML file readability.

Styles and themes are very similar concepts: when we apply a style to an Activity or an application, instead of applying it to a single View, we call it a theme. In this case every property of a View that belongs to the Activity/application will be specified by the Activity/application style. For example we could define text size and color in a style and apply it to an Activity. In this way all the texts will look like as defined in the style.

5.2 Style definition

Now that we know what a style is, we have to define it inside our app. To do this, we have to create an XML file under res/values. We can choose the file name as we prefer but it must have the .xml extension. A style is defined inside the resource tag. So we have:

```
<resources xmlns:android="http://schemas.android.com/apk/res/android" >

    <style
        name="MyTextStyle"
        parent="@android:style/TextAppearance" >

        <item name="android:textColor">#FF0000</item>

        <item name="android:textSize">14sp</item>
    </style>

</resources>
```

In this way we have defined a new style called MyTextStyle and we have defined some properties as specified in item tags. Notice that in the style definition we used the parent attribute. This is an optional attribute which can be usedf if we want the defined style to inherit its properties from the parent style. Then, inside the style definition, we can override the parent properties. Under the style tag we defined two items so we specified two properties.

An item tag has a name attribute that defines the property name. This is the name of the property we want to apply the style, and the tag content is the property value. In our case, we defined a property called android:textColor assigning to it the value #FF0000 and another one called android:textSize and we assigned the value 14sp.

5.2.1 Inheritance

As we have seen before, style can inherit from a parent style, using parent attribute in style tag. So we can create an hierarchical structure with styles. There is another way to implement an hierarchical structure inside the same XML style file: we can simply use . notation.

For example, we can suppose we want to create another style called MyTextStyleHuge that inherits from MyTextStyle but overrides the text size:

```
<style name="MyTextStyle.MyTextStyleHuge" >
    <item name="android:textSize">30sp</item>
</style>
```

and we can continue in this way adding new inheritance.

5.2.2 Style properties

Each style defines the values of a set of properties using item tag. The property names are the names of the properties defined by a View. For example, looking at TextView we notice it has a set of properties described in the documentation in the XML attributes section. If we want to define the style of one of this property, we have to use the same name in the attribute name of item tag.

Looking at the example above, we used the properties android:textSize and android:textColor, they are two properties of the TextView according to the documentation: the first one used to define the text size and the other one for text color.

Once we have defined our style we have to apply it. As said before we can apply it to a single View, to an Activity or to an application. For example if we want to apply the style to a single View we use style attribute in this way:

```
<TextView
    style="@style/MyTextStyle"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="Text with style" />
```

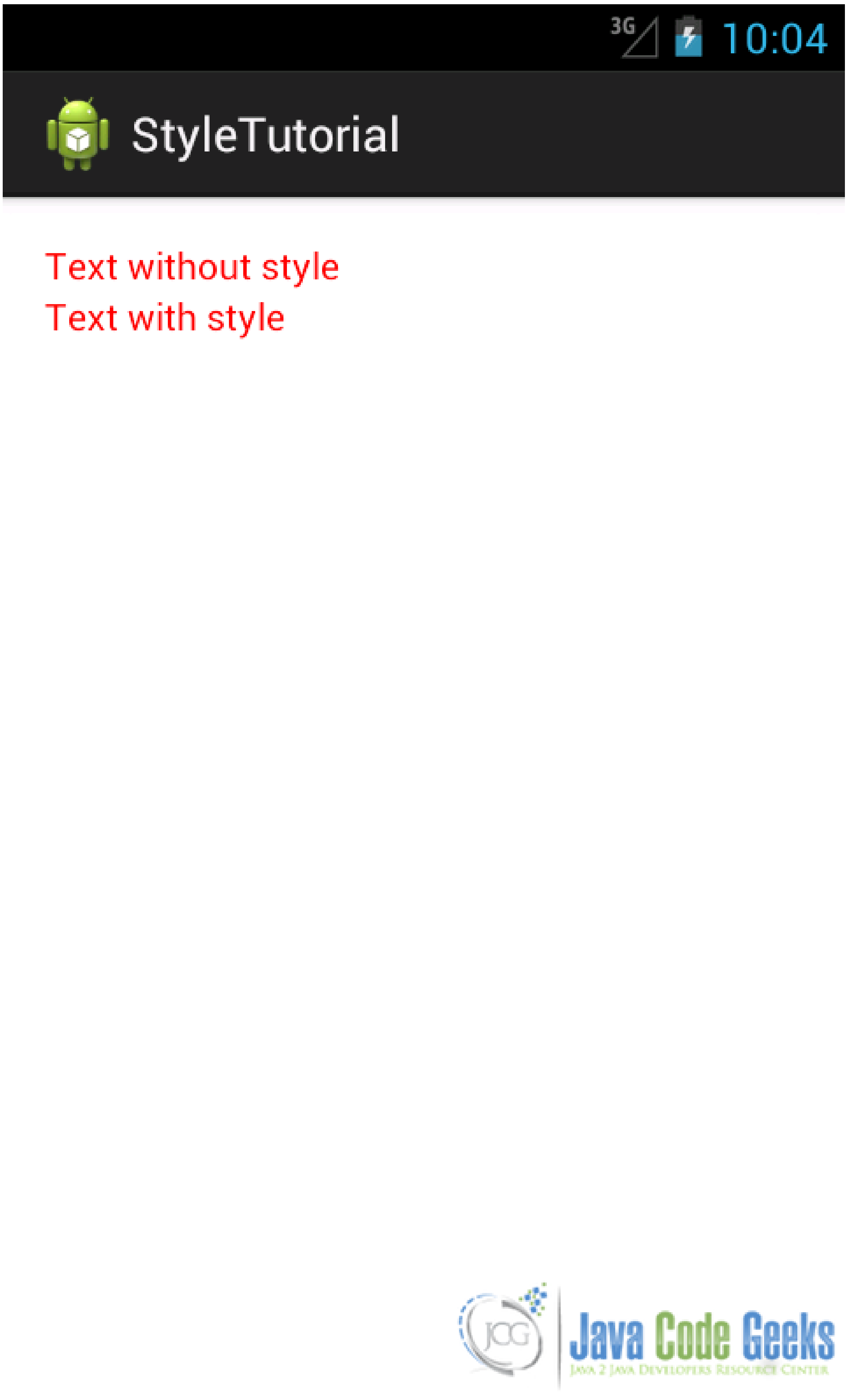
Now we applied `MyTextStyle` to the `TextView`.

These two ways are equivalent:

```
<TextView
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="Hello without style"
    android:textColor="#FF0000"
    android:textSize="14dp" >
```

```
<TextView
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="Text with style"
    style="@style/MyTextStyle"/>
```

In the left side we used a bad practice that we should avoid, while on right side we used a best practice. Running an app with these two `TextView` in the layout we have:



Text without style
Text with style

Figure 5.1: screenshot

As you can notice the result is the same for both TextView.

5.3 Creating and applying themes

We know a theme is a special style that is applied to an Activity or to an application instead of a single View. A theme is always a style and when we want to define a theme we use the same technique used before when defining a style, so we create a XML file under res/values. Android provides a large collection of themes and styles that we can use or override customizing them.

Let us suppose we want to create a theme for all the application that sets the background to green. In this scenario, we define a style called AllGreen in this way:

```
<style
    name="AllGreen"
    parent="@android:style/Theme.Black" >

    <item name="android:background">#00FF00</item>

</style>
```

and then we apply it to the application in the Manifest.xml:

```
<application
    android:icon="@drawable/ic_launcher"
    android:label="@string/app_name"
    android:theme="@style/AllGreen" >

</application>
```

Running an app like that we will have the following result:

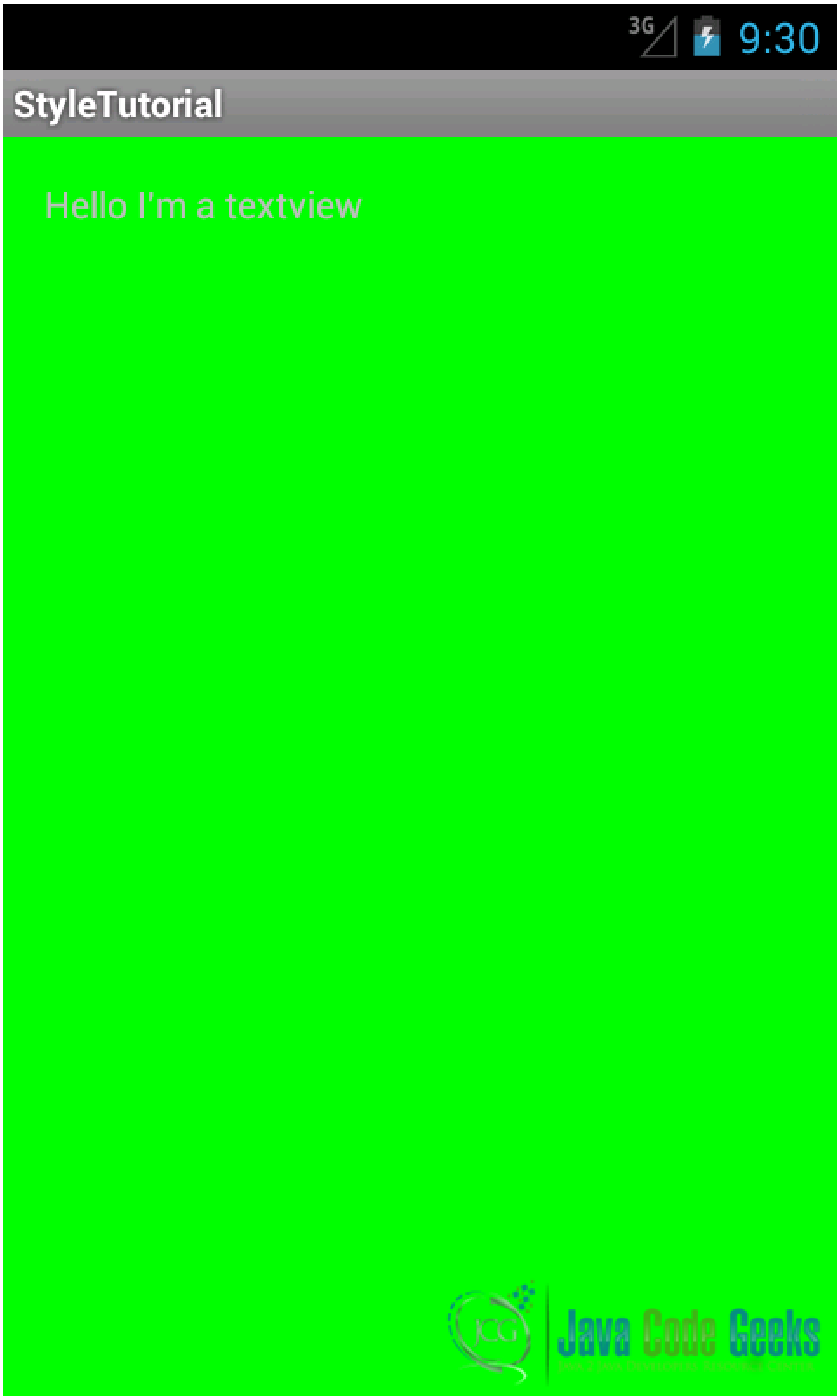


Figure 5.2: screenshot

As a result the `Activity` background is green as we wanted. We have defined before a style for `TextView` and now we would like to use it in our theme. To do it, we have to modify it a little bit:

```xml
<resources xmlns:android="http://schemas.android.com/apk/res/android" >

    <style
        name="MyTextStyle"
        parent="@android:style/Widget.Text<code>View</code>" >

        <item name="android:textColor">#FF0000</item>

        <item name="android:textSize">14sp</item>
    </style>

    <style
        name="AllGreen"
        parent="@android:style/Theme.Black" >

        <item name="android:background">#00FF00</item>

        <item name="android:textViewStyle">@style/MyTextStyle</item>
    </style>

</resources>
```

Running the app we have:

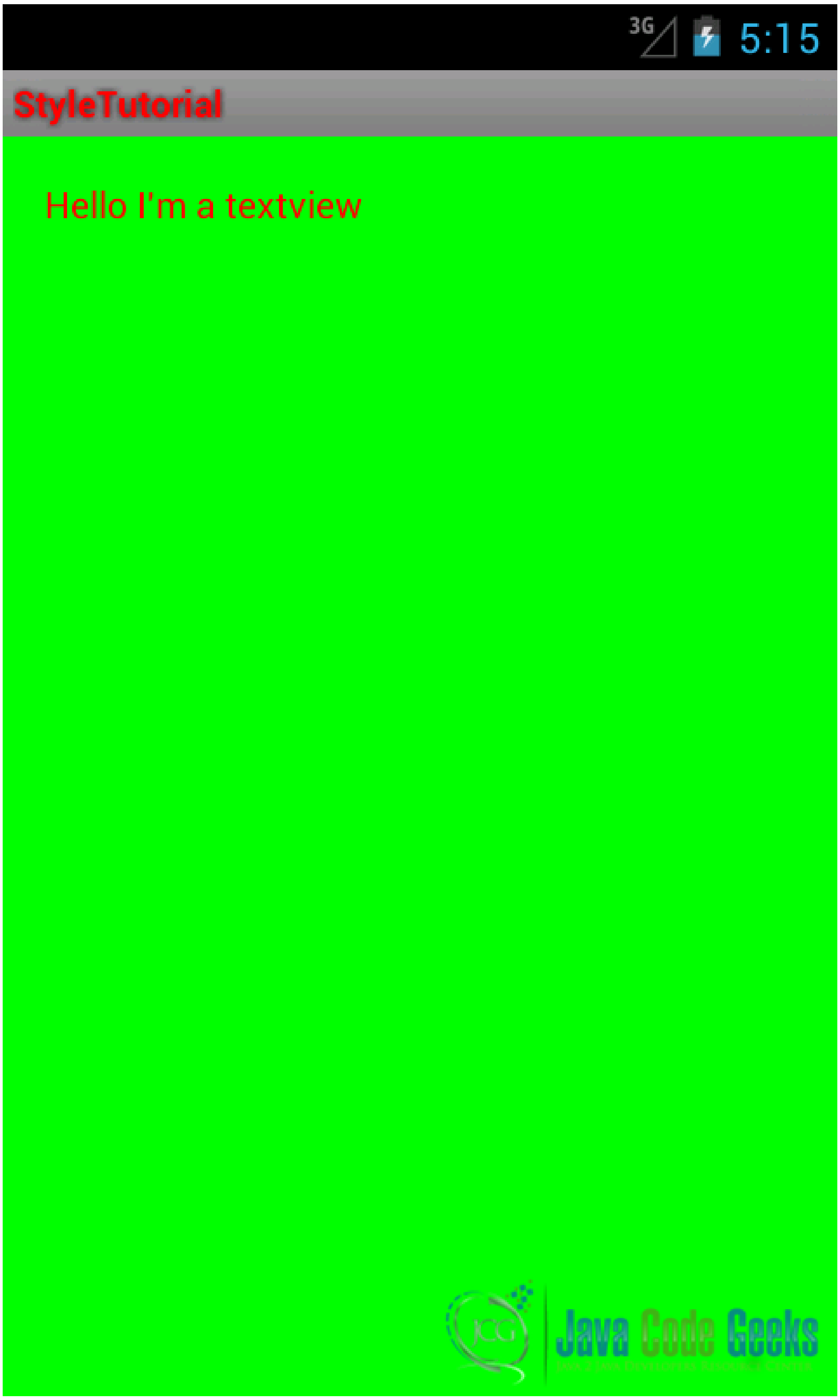


Figure 5.3: screenshot

By now we used just solid color for the background and we have seen it is easy to change the background, but what if we want to use a gradient color?

In the previous chapter we described how to create a gradient: we have to create a file with XML extension under `res/drawable` directory; we call it `grandient_red.xml`:

```
<shape xmlns:android="http://schemas.android.com/apk/res/android" >

    <gradient
        android:endColor="#AA0000"
        android:startColor="#FF0000"
        android:type="linear" />

</shape>
```

Now we have to reference it in our style and for simplicity we can create another style file called `style_gradient.xml`:

```
<resources xmlns:android="http://schemas.android.com/apk/res/android" >

    <style
        name="RedGrad"
        parent="@android:style/Theme.Black" >

        <item name="android:background">@drawable/gradient_red</item>
    </style>

</resources>
```

As you can notice, we referenced the `drawable` we created under the `drawable` directory, using `@drawable/file_name`. Running an app with this style we have:

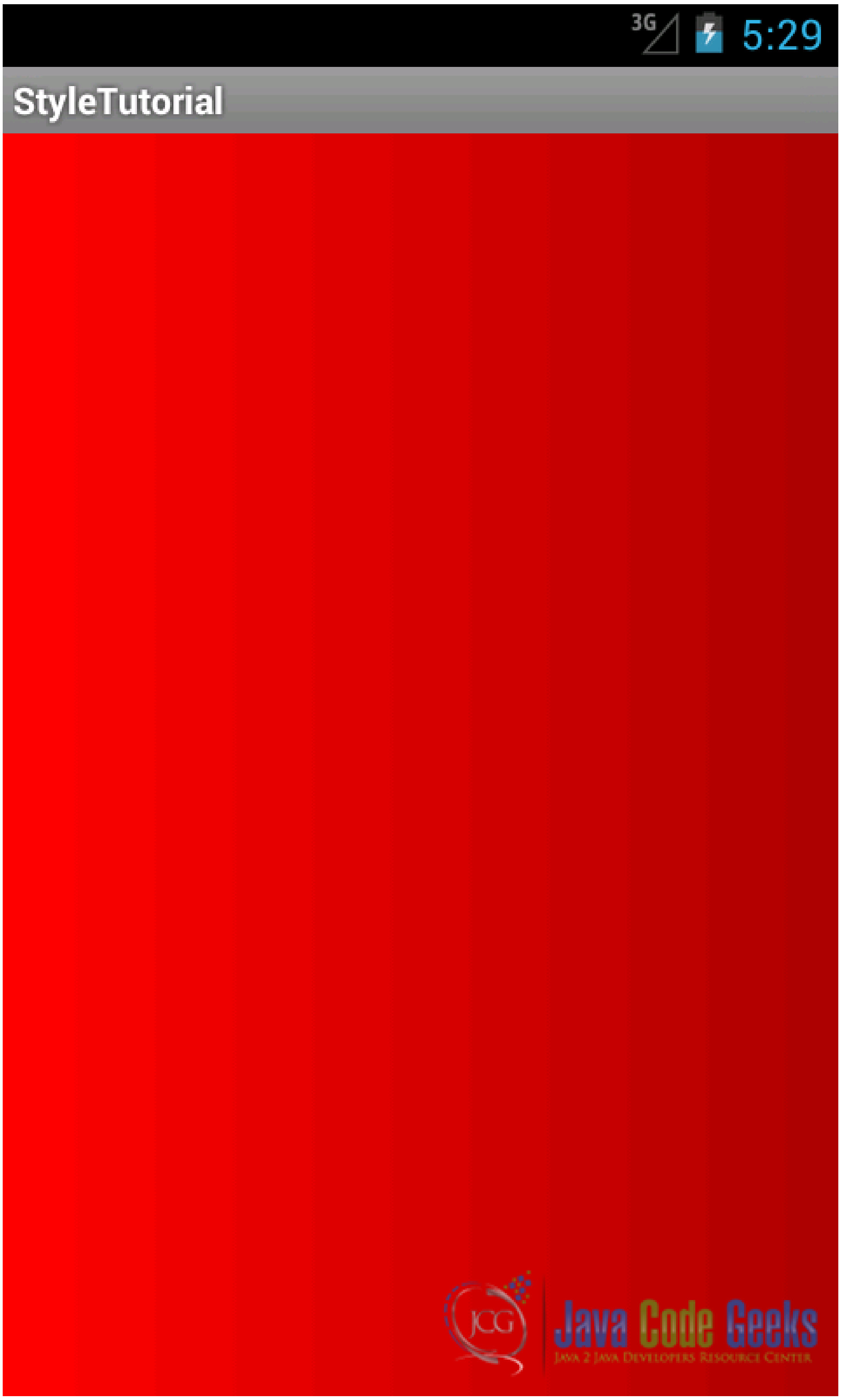


Figure 5.4: screenshot

Until now, whenever we defibed colors or font sizes, we used some hard coded values. Even if this is possible, it is not recommended for several reasons:

- If we want to change the color or a font size for example we have to look for these values in the style file.
- We would like to change the font size and colors according to the platform version for example or according to the screen dimensions.

So it is a good practice to move the color definition and/or the font size and so on to an external resources and reference them in the style.

So let us rewrite the text style we used before where this time we write the style in an XML file called `style_text.xml`:

```
<resources>

    <style name="MyTextStyle1" >

        <item name="android:textColor">@color/red</item>

        <item name="android:textSize">@dimen/myTextSize</item>
    </style>

</resources>
```

In this style we referenced an external resource that sets the text color and the text size. In this case, we have to create two files: one that we can call `colors.xml` containing the color definition and another one containing the dimension called `dimens.xml`:

```
<resources>
    <color name="red">#FF0000</color>
</resources>
<resources>
    <dimen name="myTextSize">14sp</dimen>
</resources>
```

We now know that we can reference external resources using the `@` symbol, but sometimes we want to reference a single style element not the whole style.

For example, we want to reference the current text color, but we do not know which one will be used by the OS. How can we do it? Well Android provides the `?` symbol, which we can use to access a single element. For example we can suppose we want to set the text color value to one defined in Android OS:

```
<style name="TextRef" >

    <item name="android:textColor">?android:attr/textColorLink</item>

</style>
```

Using this notation, we set the current text color to the default text color used for links. Running the app we have:

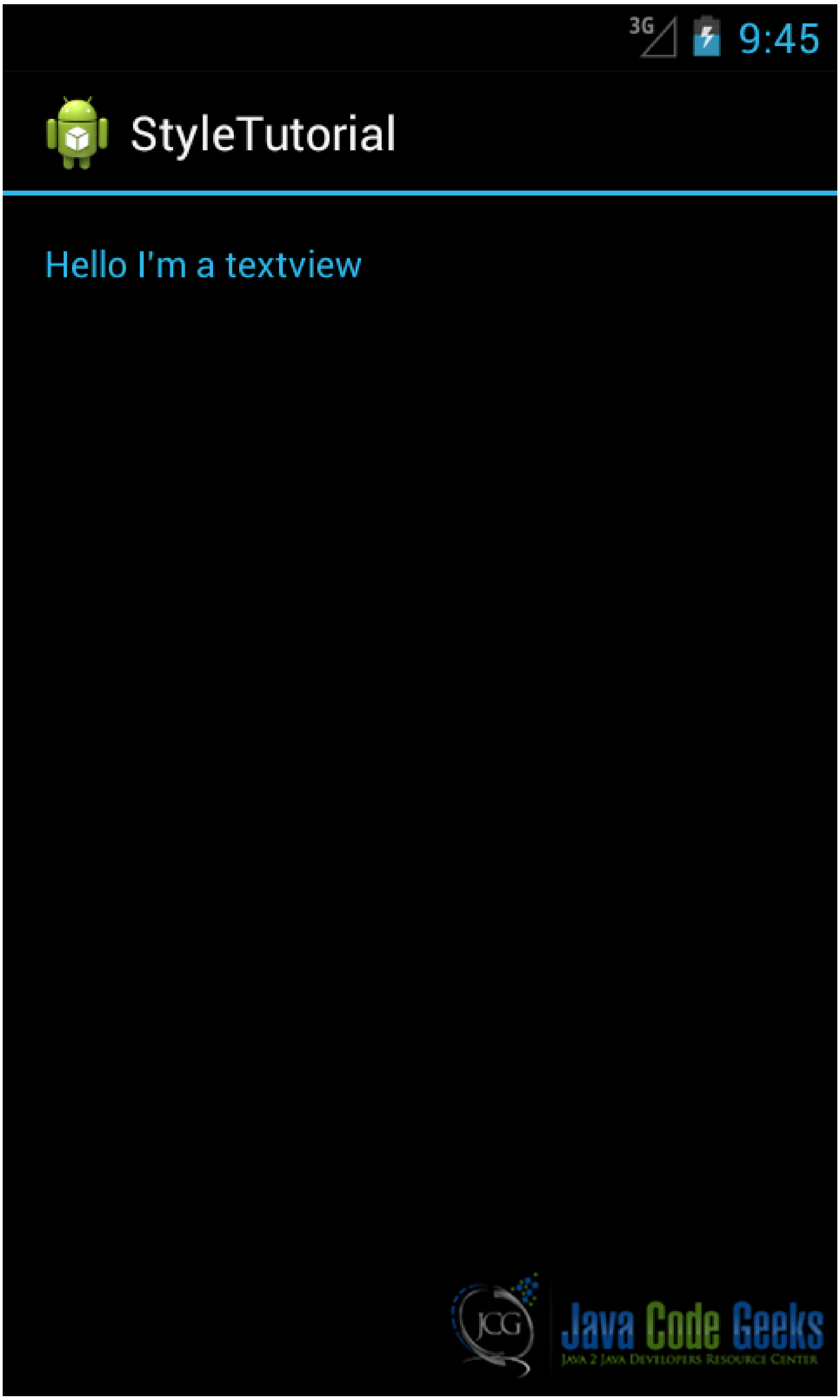


Figure 5.5: screenshot

5.3.1 Themes and platforms

Android provides an interesting feature that is useful when we want to create an app for different devices that can run on different Android versions.

Until now we used just `res/values` and we created several files that represent styles, dimensions, colors and so on. Android can use different resources according to the platform version, so we could adapt, for example, our themes and styles to the platform version, or we can even create a different style with different colors depending on the platform.

We know that newer Android version introduced some new features, APIs and resources including styles and themes. We could use newer themes when our app runs on devices that have a new Android version. At the same time, we want to keep the compatibility towards older Android versions. Using platform dependent resources we can achieve this goal.

In order to create a resource set that is dependent on the platform version we have to create a directory under `res` called `values-vxx` where `xx` is the API level; for example if we want to create a resource set that is suitable for Android 3.0 (API level 11) or higher, we can create a directory called `values-v11`. Under this directory we create style and themes. If we notice when we use Eclipse as IDE it creates two themes by default, one under `res/values`:

```
<style name="AppBaseTheme" parent="android:Theme.Light"></style>
```

and another one under `res/values-v11`:

```
 <style name="AppBaseTheme" parent="android:Theme.Holo.Light"></style>
```

As you can see, these two themee have the same name but inherit from different parents.

We can not only specify our theme according to the platform version but also according to the screen size. For example, if we want to apply our theme when the smallest screen size is at least `600dp`, we can create a directory called `values-sw600dp` and create under it our style file. This style and theme will be applied only when the device smallest screen size is at least `600dp`. This technique is useful when we want to provide different style/theme according to the screen size. We could, for example, have only one style/theme defined in `res/values` and provide different text size according to the screen size. In this case we can only create a file called `dimens.xml`.

5.4 Conclusion

We now know how to use styles and themes and it is useful to recap some basic rules we should follow:

- Do not use hard coded values for colors, size and so on.
- Define a style or a theme and use `style="..."` in the widget definition to reference the style.
- In the style definition do not use values directly in the XML file but reference theme using external resources. For example reference a color using `@color/color_name`.
- Provides different style or theme according to the platform version.
- Provides different resources according to the screen size.

5.5 Download the Source Code

This was a lesson on how to use Android Themes and Styles. You may download the source code here:

themeStyle.zip

Chapter 6

Android UI: Full Sample App

6.1 Introduction

In this last chapter about Android UI, we will build an Android app that uses almost all the concepts we talked about in the previous chapters. We talked about the most important aspects we need to consider when developing an Android app. We saw how to create a UI structure using layout managers and how we can place widgets; we described some best practices, we should use, while developing an app. Well, the app we will build will be based on the topics covered previously, so have another look at them to refresh your memory.

As an example, we will build a `To Do app`: this is a simple app where we can add todo items and manage them. We will cover how to create the UI layout structure, how to add widgets so that we can show text messages to the user and how to accept user input. An important aspect we will consider is how to build an app that can be used on several devices with different screen size and resolutions.

6.2 App structure

Before digging into the code details, the first thing we should consider when building an app is making some sketches that help us to understand the app navigation and user interaction. There are several tools we can use, some of them are free. Moreover, these sketches help us to have an idea how our app will look like and we could show them to our customers so that they can realize if the app we want to build respects their needs.

Coming back to our To do app, we can image we have these requirements we have to satisfy:

- There should be a list of items (to do items).
- A user can add an item to the existing ones.
- Items should have a priority color.
- The app should run on smart phone and tablets.

In a real app, the requirements will be much more complex of course, but this is just a stepping stone. We can imagine a simple navigation like this one:

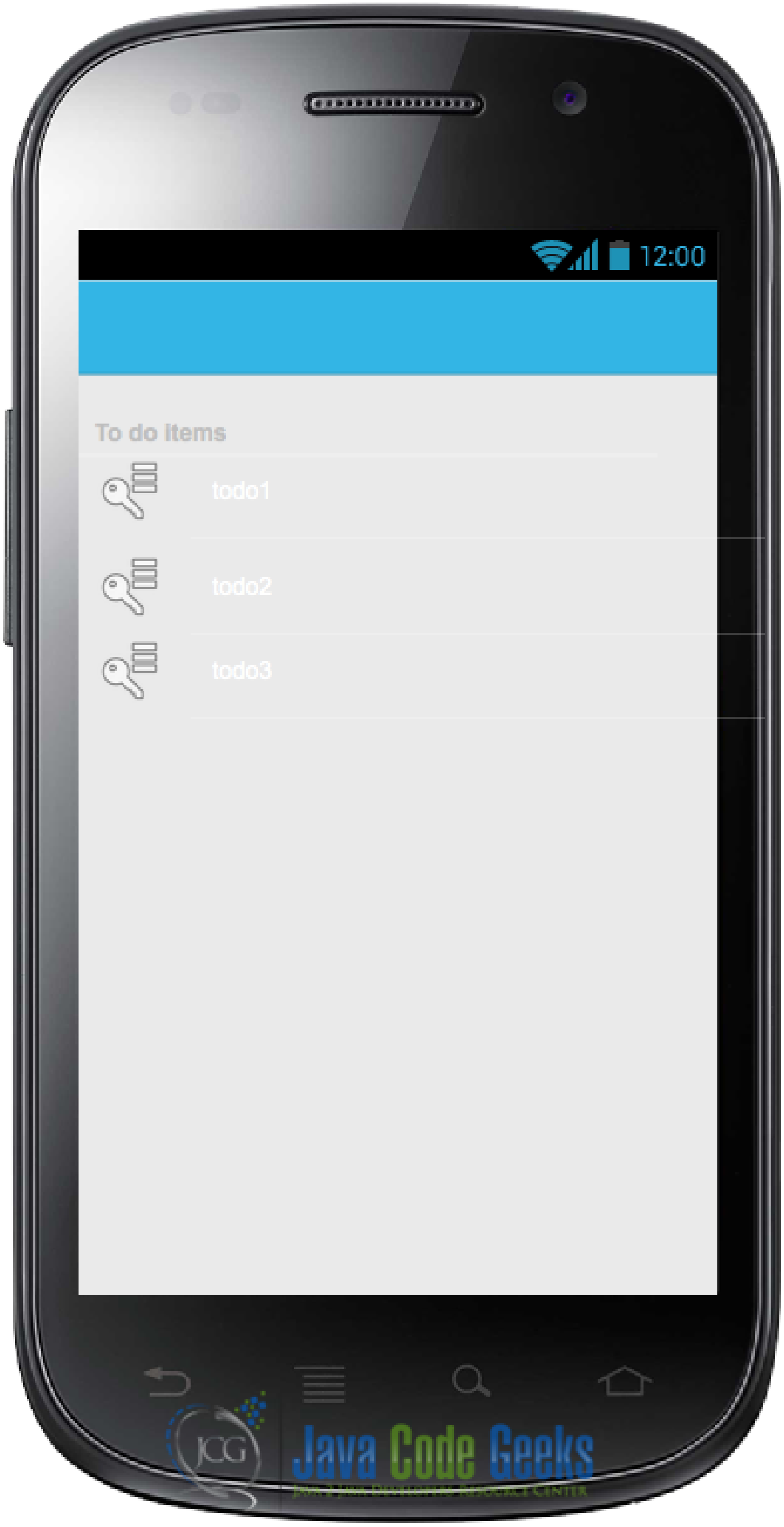


Figure 6.1: screenshot

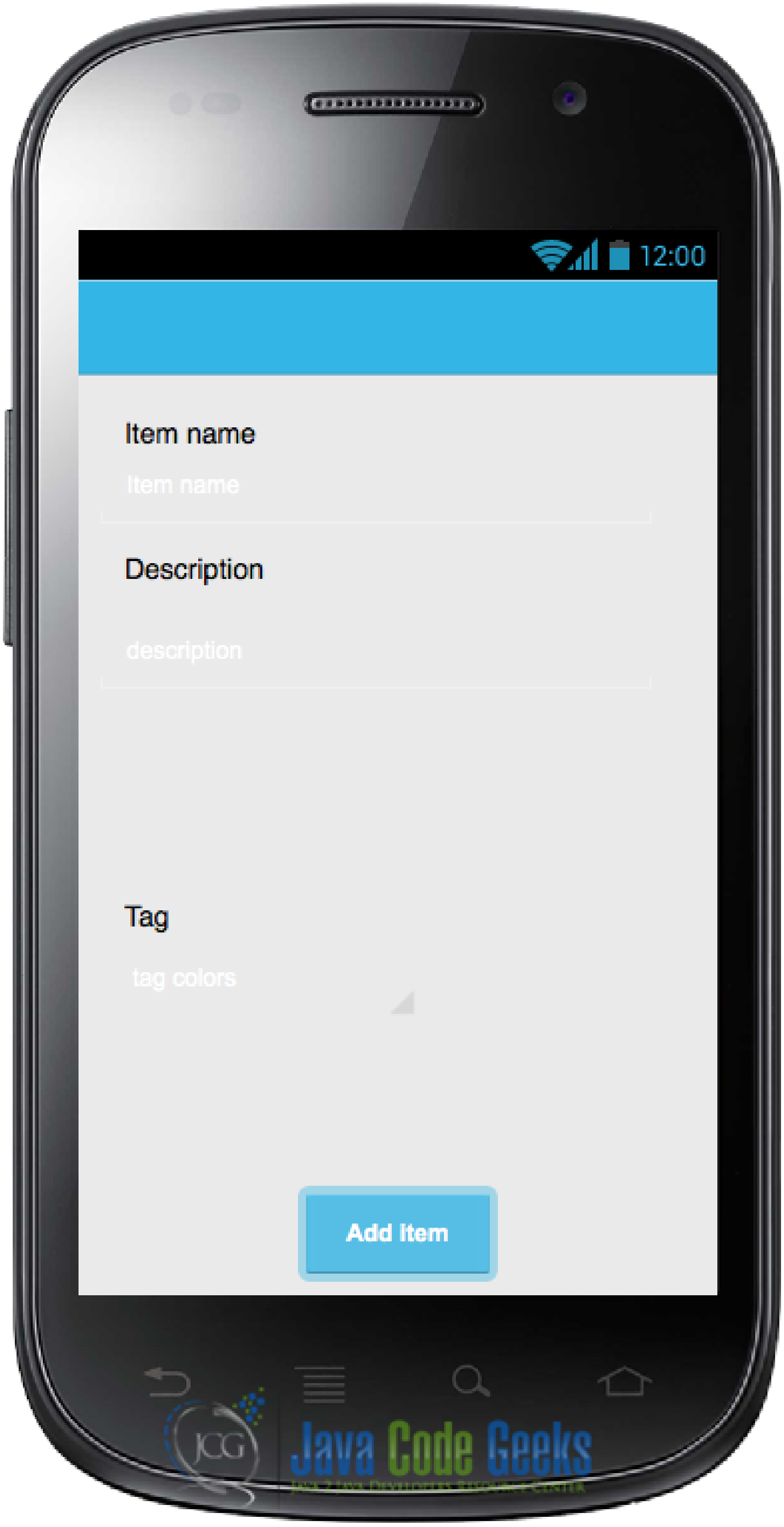


Figure 6.2: screenshot

This is a very simple navigation: at the start up, the app shows the current items list and when the user clicks on "Add Item" on the action bar. the app will show the add item screen. To keep things simple and get focused on the UI aspects we can suppose that the app will not save the items. It could be an interesting exercise for the reader to extend the app so that it saves the items.

Now that we roughly know what the navigation will be and how the app user interface will look, we can start creating our app using our IDE. In this case we will use Eclipse + ADT. We create a new Android project that we can call `Todo`. We will not cover how to create an Android project using Eclipse so we assume you are already familiar with this IDE. Check out our "Android Hello World" example if you are not familiar with the process.

6.2.1 Item List with ListView and Object Model

Now we have our project structure, we can focus our mind on designing the model that stands behind the app. In this case the model is very simple, it is just a class that holds the information about a new todo item:

```
public class Item implements Serializable {
        private String name;
        private String descr;
        private Date date;
        private String note;
        private TagEnum tag;
            // Set and get methods
}
```

This will be the basic class that we will handle in our app. Looking at the Android project, we have just created, we can notice that under the layout directory there is a default layout called `activity_main.xml`. This is the default layout created by the tool.

By now we can suppose we have just a list of items in this layout: this layout will be used just for smart phone and we will consider later when the app runs on a tablet. The list item layout is very simple, it is just built by a standard `ListView` widget:

```
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:paddingBottom="@dimen/activity_vertical_margin"
    android:paddingLeft="@dimen/activity_horizontal_margin"
    android:paddingRight="@dimen/activity_horizontal_margin"
    android:paddingTop="@dimen/activity_vertical_margin"
    tools:context=".MainActivity" >

    <ListView
        android:id="@+id/listItmes"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content" />

</RelativeLayout>
```

If you notice, we instructed Android to use just the space necessary to hold the items in the list. We know that to use a `ListView` we have to implement an adapter. We could use a standard adapter provided by Android, but in this case these standard adapters are not enough, we want to implement a custom adapter because we would like to show some information for each row in the list. We would like that a row in the list looks like:

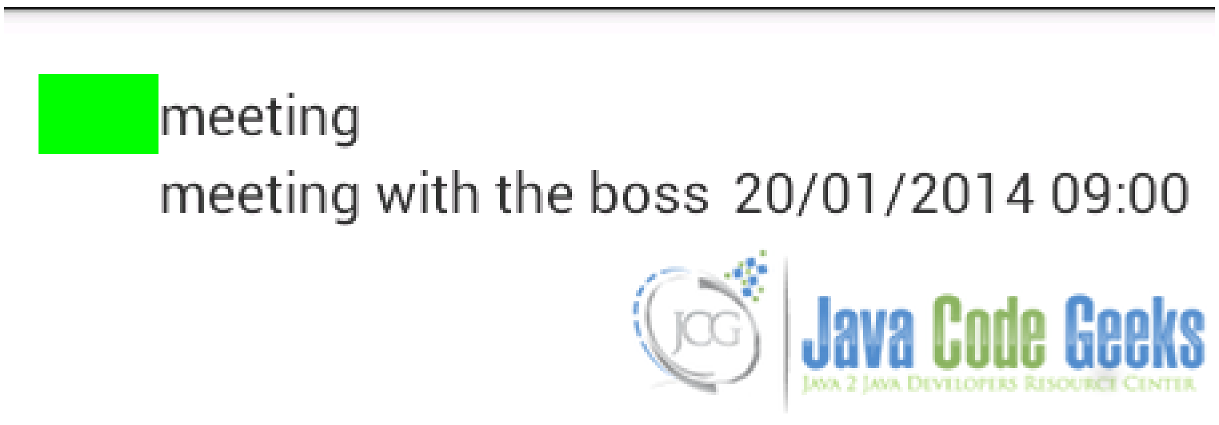


Figure 6.3: screenshot

As you can notice, each row has an image on the left side that represents the todo priority and some information. By now we do not consider applying any style to our rows. To have a row in our `ListView` like that, we have to create a row layout that we will use in our custom adapter. So we can create a new file called `item_layout.xml` under `layout` directory. This file looks like this:

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

    <ImageView
        android:id="@+id/tagView"
        android:layout_width="30dp"
        android:layout_height="20dp"
        android:background="@color/red" />

    <TextView
        android:id="@+id/nameView"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_alignParentTop="true"
        android:layout_toRightOf="@id/tagView" />

    <TextView
        android:id="@+id/descrView"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_below="@id/nameView"
        android:layout_toRightOf="@id/tagView" />

    <TextView
        android:id="@+id/dateView"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_alignParentBottom="true"
        android:layout_alignParentRight="true" />

</RelativeLayout>
```

In this layout, we use a `RelativeLayout` manager to place easily widgets where we want. As you can see in this layout manager, views are placed according to other view positions. For example we want our todo name to be placed just after the image, so we use the attribute:

```
android:layout_toRightOf="@id/tagView"
```

Moreover, we can place views respect to the parent, for example we want that the date information will be placed on the right side of the row and at the bottom:

```
android:layout_alignParentRight="true"
android:layout_alignParentBottom="true"
```

Now that we have the layout, we have to build the adapter. We will extend an `ArrayAdapter` and override some methods so we can handle our model data and the new layout. We call this adapter `ToDoItemAdaper`, so we have:

```
public class ToDoItemAdapter extends ArrayAdapter<Item> {
        private Context ctx;
        private List<Item> itemList;
        public ToDoItemAdapter(Context context, List<Item> itemList) {
                super(context, R.layout.item_layout);
                this.ctx = context;
                this.itemList = itemList;
        }
}
```

The constructor receives the `Context` and the `itemList` as parameters, the last one param holds the todo item list. You can notice that when we call the super method, we pass out custom layout called `R.layout.item_layout`. Now we have to override one of the most important method called `getView`, used to create the `View` and render the row layout:

```
        @Override
        public View getView(int position, View convertView, ViewGroup parent) {
                View v = convertView;
                ItemHolder h = null;
                if (v == null) {
                        // Inflate row layout
                        LayoutInflater inf = (LayoutInflater) ctx.getSystemService(Context. ↩
                            LAYOUT_INFLATER_SERVICE);
                        v = inf.inflate(R.layout.item_layout, parent, false);
                        // Look for Views in the layout
                        ImageView iv = (ImageView) v.findViewById(R.id.tagView);
                        TextView nameTv = (TextView) v.findViewById(R.id.nameView);
                        TextView descrView = (TextView) v.findViewById(R.id.descrView);
                        TextView dateView = (TextView) v.findViewById(R.id.dateView);
                        h = new ItemHolder();
                        h.tagView = iv;
                        h.nameView = nameTv;
                        h.descrView = descrView;
                        h.dateView = dateView;
                        v.setTag(h);
                }
                else
                   h = (ItemHolder) v.getTag();

                h.nameView.setText(itemList.get(position).getName());
                h.descrView.setText(itemList.get(position).getDescr());
                h.tagView.setBackgroundResource(itemList.get(position).getTag().getTagColor ↩
                    ());
                h.dateView.setText(sdf.format(itemList.get(position).getDate()));

                return v;
        }
```

In this method, we check at the beginning if the View we receive as parameter is null. In this case, we have to inflate our layout. If you notice, we used the `ViewHolder` pattern to make the `ListView` scrolling smoother. We have created a small inner class called `ItemHolder` that holds the references to the `View` inside our custom layout:

```
// ViewHolder pattern
static class ItemHolder {
        ImageView tagView;
        TextView nameView;
        TextView descrView;
        TextView dateView;
}
```

One thing you should notice is how we handled the background color of the ImageView. We used setBackgroundResource to set the imageview background. This method accepts an int representing the resource id we want to use as background:

```
h.tagView.setBackgroundResource(itemList.get(position).getTag().getTagColor());
```

Looking at our model class we can notice that the getTag() method returns an instance of TagEnum class. This is an enumeration defined in this way:

```
public enum TagEnum {
        BLACK(R.color.black,"Black"), RED(R.color.red, "Red"),
        GREEN(R.color.green, "Green"), BLUE(R.color.blue, "Blue"),YELLOW(R.color.yellow," ↩
            Yellow");
        private int code;
        private String name;
        private TagEnum(int code, String name) {
                this.code = code;
                this.name = name;
        }
        public int getTagColor() {
                return this.code;
        }
}
```

In the enumeration we define the different colors we want to support and as first parameter we pass a resource id. If you remember in a previous chapter we talked about how to define resource color in XML format. We know, already, we have to create a XML file under res/values that we can call colors.xml:

```
<resources>

    <color name="red" >#FF0000
    </color>

    <color name="green" >#00FF00
    </color>

    <color name="blue" >#0000FF
    </color>

    <color name="black" >#000000
    </color>

    <color name="yellow" >#FFAA00
    </color>

</resources>
```

In the enumeration color definition we referenced this color using R.color.color_name, so that when we use getTagColor method in the custom adapter getView, we receive the resource id that will be used by the image background. An important aspect to understand is that we did not hard-code the colors in the constructor: for example, we could use directly the color hex code like #FF0000 for red and so on.

Even if the result would be the same, it is not recommended to use hard-coded values in the source code. For example, if we want to change the red color to another one, we will have to find the hex color in the source code and change it, but if we had used resources to define colors, we would go directly to the file holding the color definitions and change the color we like.

Notice that in the enumeration, we used a bad practice: we wrote directly the color name. We used it purposely to show to you something you should not do. In this case, if we want to support multi-language app, we have to change the way we initialize the enumeration using name written in a string resource file.

6.2.2 Multi-device support and layout considerations

Remember that one of our requirements is that we have to build an app that supports both smart phones and tablets. Thinking about tablet screen dimensions, we realize that the screen is too big to hold just a list of items, so we could consider splitting the screen in two areas: one that holds the list and another one that we can use to show item details or even to show the user interface to add a new item. This is true if we use tablets, but if we have a smart phone the screen dimensions are not big enough to be divided in two areas.

At the same time, we do not want to develop two different code branches: one for smart phone and one for tablet. We would rewrite the same code changing just some details and dimensions. Android helps us to solve this problem: we talked about `Fragment` in a previous chapter. So we could create a fragment that handles the user interface to add a new item to the list. A fragment encapsulates a set of components and activity behaviors so that we can reuse this piece of code in different activities. The figures below depict the situation we have to handle:

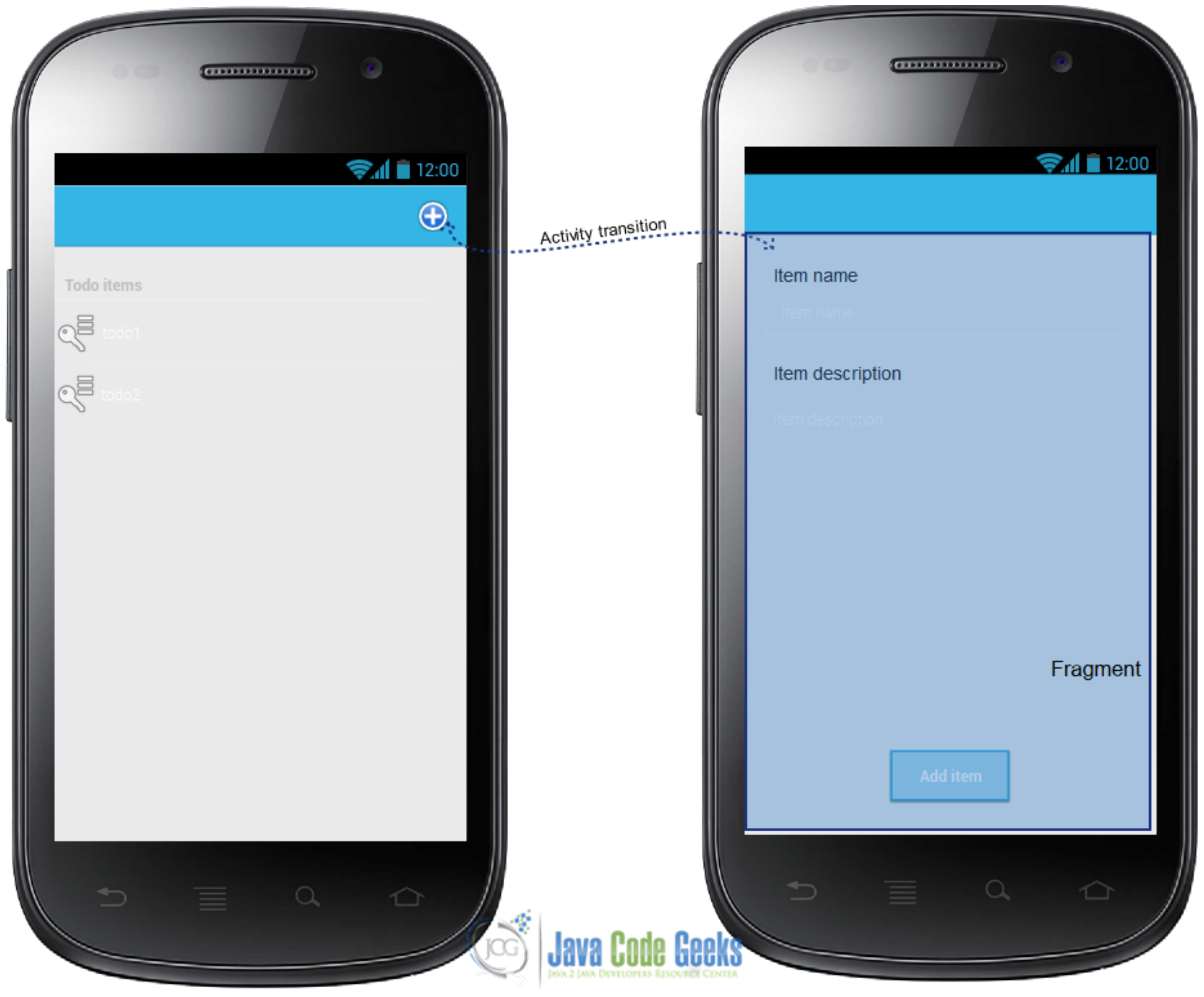


Figure 6.4: screenshot

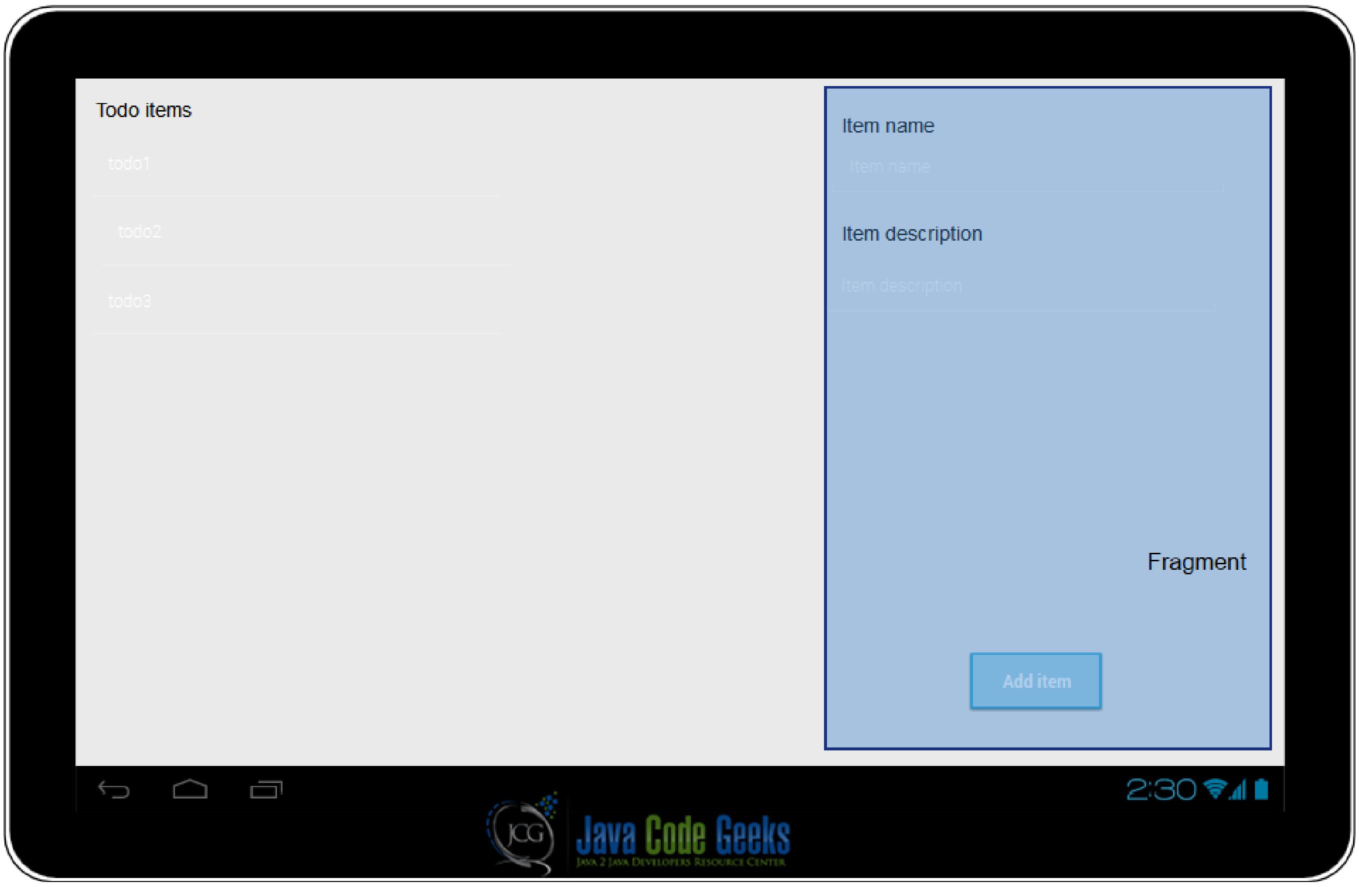


Figure 6.5: screenshot

When the app runs in a smart phone we have to handle two activities one for the list item and another that handles the user input, while in a tablet we can have only one activity.

We can suppose that the screen size is at least 600dp so we want to split the screen in a different areas and we define a new layout under `res/layout-sw600dp`:

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:paddingBottom="@dimen/activity_vertical_margin"
    android:paddingLeft="@dimen/activity_horizontal_margin"
    android:paddingRight="@dimen/activity_horizontal_margin"
    android:paddingTop="@dimen/activity_vertical_margin"
    tools:context=".MainActivity" >

    <ListView
        android:id="@+id/listItmes"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_weight="1" />

    <FrameLayout
        android:id="@+id/frm1"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_weight="1" />
```

```
</LinearLayout>
```

where the `FrameLayout` will be "filled" dynamically depending on the navigation flow.

Of course, we can customize the layout in more details, according to the different screen sizes. In this case, we can simply create different layouts in different directories under res.

6.2.3 Add item user interface layout

If we run the app we have as a result an empty list with no items. We have to create a new user interface. So keeping in mind the considerations we made before, we create a `Fragment` that handles the add item functionality, calling it as `NewItemFragment`. A fragment has a complex life cycle but for this purpose we can override just `onCreateView` method. This method is responsible for creating the user interface. As always, we have to create the layout first. In our IDE under `res/layout` we create another xml file that we call `add_item_layout.xml`:

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

    <TextView
        android:id="@+id/txtTitle"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_centerHorizontal="true"
        android:text="@string/addItem" />

    <TextView
        android:id="@+id/itemName"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_below="@id/txtTitle"
        android:layout_marginStart="10dp"
        android:text="@string/addItemName" />

    <EditText
        android:id="@+id/edtName"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:layout_alignParentLeft="true"
        android:layout_below="@id/itemName"
        android:hint="@string/addItemNameHint" />

    <TextView
        android:id="@+id/itemDescr"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_alignParentLeft="true"
        android:layout_below="@id/edtName"
        android:layout_marginTop="10dp"
        android:text="@string/addItemDescr" />

    <EditText
        android:id="@+id/edtDescr"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:layout_alignParentLeft="true"
        android:layout_below="@id/itemDescr"
        android:hint="@string/addItemDescrHint" />

    <TextView
        android:id="@+id/itemNote"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_alignParentLeft="true"
        android:layout_below="@id/edtDescr"
        android:layout_marginTop="10dp"
        android:text="@string/addItemNote" />

    <EditText
        android:id="@+id/edtNote"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:layout_alignParentLeft="true"
        android:layout_below="@id/itemNote"
        android:hint="@string/addItemNoteHint" />

    <TextView
        android:id="@+id/itemDate"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_alignParentLeft="true"
        android:layout_below="@id/edtNote"
        android:layout_marginTop="10dp"
        android:text="@string/addItemDate" />

    <TextView
        android:id="@+id/inDate"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_alignParentLeft="true"
        android:layout_below="@id/itemDate"
        android:layout_marginTop="10dp" />

    <TextView
        android:id="@+id/itemTime"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_alignParentLeft="true"
        android:layout_below="@id/inDate"
        android:layout_marginTop="10dp"
        android:text="@string/addItemTime" />

    <TextView
        android:id="@+id/inTime"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_alignParentLeft="true"
        android:layout_below="@id/itemTime"
        android:layout_marginTop="10dp" />

    <TextView
        android:id="@+id/itemTag"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_alignParentLeft="true"
        android:layout_below="@id/inTime"
        android:layout_marginTop="10dp"
        android:text="@string/addItemTag" />

    <Spinner
        android:id="@+id/tagSpinner"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_alignParentLeft="true"
        android:layout_below="@id/itemTag" />

    <!-- ADD button -->

    <Button
        android:id="@+id/addBtn"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_alignParentBottom="true"
        android:layout_centerHorizontal="true"
        android:text="@string/addButton" />

</RelativeLayout>
```

It is a very simple layout made by `TextView` and `EditText`: the first one is used to show text messages on the user interface and the other one is used to accept user inputs. There are two components that are important in this layout: the `Spinner` and the `Date/Time picker`.

A **spinner** is a UI component that shows only one item at time and lets the user to select one item among them. We use this component to show different tag colors/priorities. It suits perfectly our purpose, in fact we want the user to select one color among a color list.

6.2.4 Tag Color/Priority Spinner

To work correctly, a `Spinner` needs an array adapter behind it. Android provides a list of adapters we can use, but we want to customize them because we want to show an image with a color. Then, we have to create a custom adapter in the same way we did for the `ListView`. First we create the row layout called `spinner_tag_layout.xml`:

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

    <ImageView
        android:id="@+id/tagSpinnerImage"
        android:layout_width="30dp"
        android:layout_height="20dp" />

    <TextView
        android:id="@+id/tagNameSpinner"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_toRightOf="@id/tagSpinnerImage" />

</RelativeLayout>
```

and finally we create our adapter:

```
public class TagSpinnerAdapter extends ArrayAdapter<TagEnum> {

        private Context ctx;
        private List<TagEnum> tagList;

        public TagSpinnerAdapter(Context ctx, List<TagEnum> tagList) {
                super(ctx, R.layout.spinner_tag_layout);
                this.ctx = ctx;
                this.tagList = tagList;
        }
        @Override
        public View getDropDownView(int position, View convertView, ViewGroup parent) {
                return _getView(position, convertView, parent);
        }

        @Override
        public View getView(int position, View convertView, ViewGroup parent) {
                return _getView(position, convertView, parent);
        }

        private View _getView(int position, View convertView, ViewGroup parent) {
                View v = convertView;
                if (v == null) {
                        // Inflate spinner layout
                        LayoutInflater inf = (LayoutInflater) ctx.getSystemService(Context. ↩
                            LAYOUT_INFLATER_SERVICE);
                        v = inf.inflate(R.layout.spinner_tag_layout, parent, false);
                }

                // We should use ViewHolder pattern
                ImageView iv = (ImageView) v.findViewById(R.id.tagSpinnerImage);
                TextView tv = (TextView) v.findViewById(R.id.tagNameSpinner);

                TagEnum t = tagList.get(position);
                iv.setBackgroundResource(t.getTagColor());
                tv.setText(t.getName());
                return v;
        }

}
```

Analyzing the code above, we notice that this adapter handles `TagEnum` objects and we override two methods `getView` and `getDropDownView`. We handle these methods in the same way. As you can notice we did almost the same things already done for the ListView.

In the fragment that holds this UI components, we have to find the reference to the `Spinner` and set the custom layout we defined above:

```
Spinner sp = (Spinner) v.findViewById(R.id.tagSpinner);
TagSpinnerAdapter tsa = new TagSpinnerAdapter(getActivity(), tagList);
sp.setAdapter(tsa);
```

When user selects one item in the `Spinner`, we have to find a way to know which one item was selected.

As you remember, we can use listener to be informed when some events occur in a component. In this case we are interested on item selection event, so we create a listener for it:

```
sp.setOnItemSelectedListener(new AdapterView.OnItemSelectedListener() {
        @Override
        public void onItemSelected(AdapterView<?> adptView, View view,
                int pos, long id) {
                currentTag = (TagEnum) adptView.getItemAtPosition(pos);
        }

        @Override
        public void onNothingSelected(AdapterView<?> arg0) {
                // TODO Auto-generated method stub

                }
        });
```

and we store the result in a class attribute.

6.2.5 Date and Time picker

When we add a new todo item to the list, we want the user to select the date and the time. Android provides two components called DatePickerDialog and TimePickerDialog. As the name implies, these are two dialogs that can be opened to select the date and the time.

We are using fragments so we have to create two inner class that present to the user the date and time pickers. In this case we extend for both pickers the DialogFragment class and override the onCreateDialog method. In this method, we simply initialize the Date picker and return it as result:

```
public static class DatePickerFragment extends DialogFragment implements DatePickerDialog. ↩
    OnDateSetListener {
        @Override
            public Dialog onCreateDialog(Bundle savedInstanceState) {
                // Use the current date as the default date in the picker
                final Calendar c = Calendar.getInstance();
                c.setTime(selDate);
                int year = c.get(Calendar.YEAR);
                int month = c.get(Calendar.MONTH);
                int day = c.get(Calendar.DAY_OF_MONTH);

                // Create a new instance of DatePickerDialog and return it
                return new DatePickerDialog(getActivity(), this, year, month, day);
            }

        @Override
        public void onDateSet(DatePicker view, int year, int monthOfYear,
                                int dayOfMonth) {

                Calendar c = Calendar.getInstance();
                c.set(year, monthOfYear, dayOfMonth, 9, 0);
                selDate = c.getTime();
                tvDate.setText(sdfDate.format(selDate));
        }
}
```

From the code above you can notice that we simply set the current date to the DatePickerDialog in the onCreateDialog. We implement the DatePickerDialog.OnDateSetListener to be notified when user selects the date. In the callback method of this interface we simply store the date selected by the user.

In the same way, we handle the TimePickerFragment, but in this case we extend TimePickerDialog:

```
public static class TimePickerFragment extends DialogFragment implements TimePickerDialog. ↩
    OnTimeSetListener {
        @Override
         public Dialog onCreateDialog(Bundle savedInstanceState) {
                // Use the current date as the default date in the picker
                final Calendar c = Calendar.getInstance();
                c.setTime(selDate);
                int hour = c.get(Calendar.HOUR_OF_DAY);
                int minute = c.get(Calendar.MINUTE);
            // Create a new instance of TimePickerDialog and return it
             return new TimePickerDialog(getActivity(), this, hour, minute,
                                DateFormat.is24HourFormat(getActivity()));
            }

          @Override
          public void onTimeSet(TimePicker view, int hourOfDay, int minute) {
                Calendar c = Calendar.getInstance();
                c.setTime(selDate);
                c.set(Calendar.HOUR_OF_DAY, hourOfDay);
                c.set(Calendar.MINUTE, minute);
                selDate = c.getTime();
                // We set the hour
                tvTime.setText(sdfTime.format(selDate));
        }
}
```

These are two dialogs that they do not appear by themselves, but we have to check if the user clicks on a date/time textview to open these dialogs, so we have:

```
tvDate = (TextView) v.findViewById(R.id.inDate);
tvTime = (TextView) v.findViewById(R.id.inTime);
```

to get the reference to the `TextView` and then we have simply implement a listener:

```
tvDate.setOnClickListener(new View.OnClickListener() {
        @Override
        public void onClick(View v) {
                DatePickerFragment dpf = new DatePickerFragment();
                dpf.show(getFragmentManager(), "datepicker");
        }
});
```

As result we have:

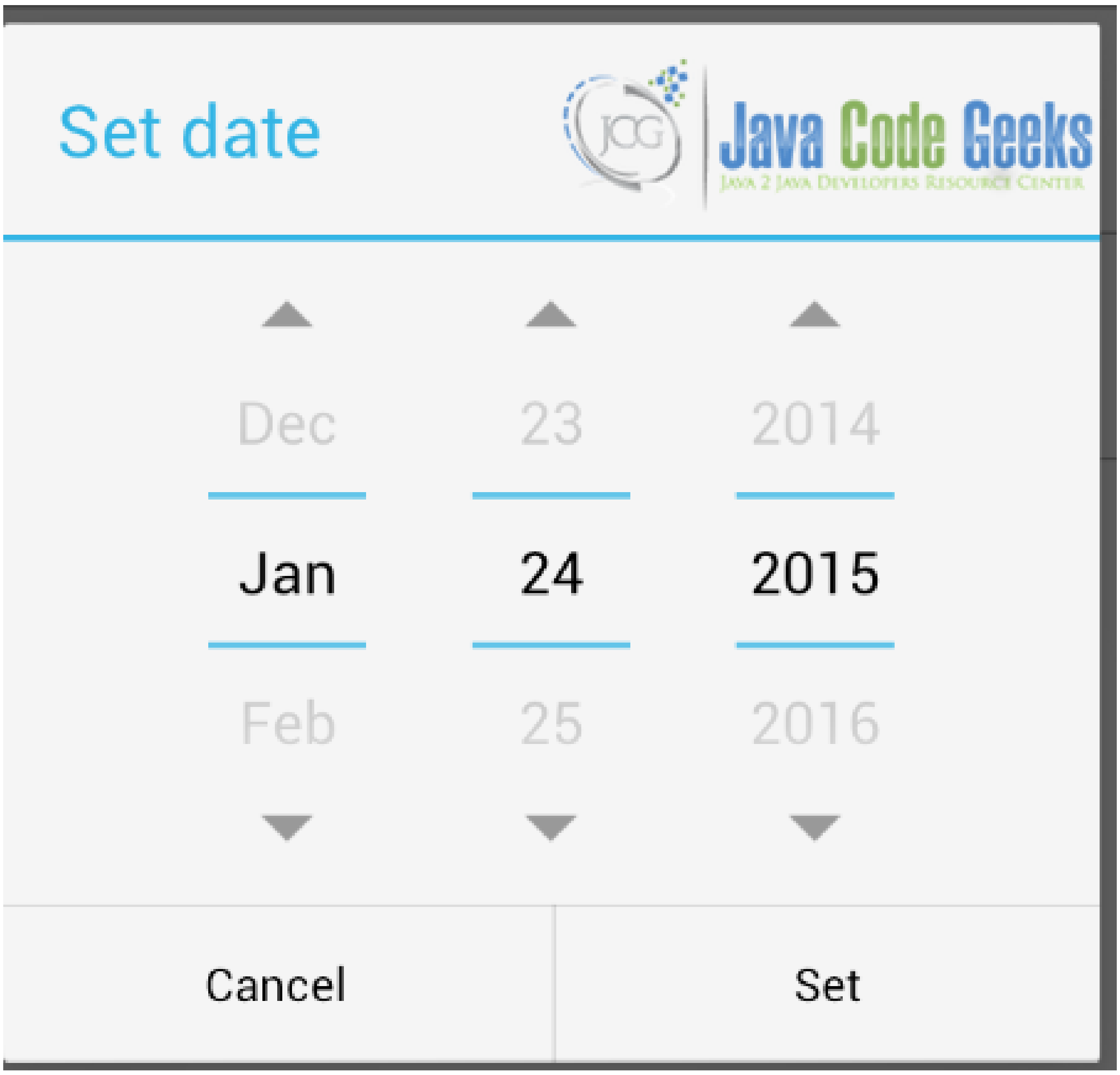


Figure 6.6: screenshot

The last part in developing this fragment is dedicated to handling the event when the user clicks on the add button. When this event occurs, the fragment should create an instance of Item class holding the data inserted by user and send it back to the activity that holds the fragment. The best practice suggests to use an interface and callback methods.

We can define this interface in the fragment class:

```
public interface AddItemListener {
        public void onAddItem(Item item);
}
```

It is a very simple interface made by just only one method. Now we have:

```
Button addBtn = (Button) v.findViewById(R.id.addBtn);
addBtn.setOnClickListener(new View.OnClickListener() {
        @Override
        public void onClick(View v) {
                // We retrieve data inserted
                Item i = new Item();
                i.setName(edtName.getText().toString());
                i.setDescr(edtDescr.getText().toString());
                i.setNote(edtNote.getText().toString());
                i.setTag(currentTag);
                i.setDate(selDate);
                // Safe cast
                ( (AddItemListener) getActivity()).onAddItem(i);
        }
});
```

By using an interface we decouple the fragment from the activity that holds it and this will be very useful as we will see later.

Coming back to the note made regarding smart phones and tablets and remembering the picture shown above, we know that if the app runs in a smart phone we have to use two activities so we create another one called `NewItemActivity` that will hold the fragment described above. This activity is very simple because it acts as a fragment container.

```
public class NewItemActivity extends Activity implements NewItemFragment.AddItemListener{
        @Override
        protected void onCreate(Bundle savedInstanceState) {
                super.onCreate(savedInstanceState);
                NewItemFragment nif = new NewItemFragment();
                getFragmentManager().beginTransaction().add(android.R.id.content, nif). ↩
                    commit();
        }

        @Override
        public void onAddItem(Item item) {
                // We get the item and return to the main activity
                Log.d("TODO", "onAddItem");
                Intent i = new Intent();
                i.putExtra("item", item);
                setResult(RESULT_OK,i);
                finish();
        }
}
```

In the `onCreate` method, we create a new instance of our fragment class and the using `FragmentManager` we show the fragment on the screen.

Notice that this activity implements the interface defined in the fragment because it has to be notified when the user wants to add a new item, so it implements the `onAddItem` method. We will cover the functionality of this method later.

6.2.6 Main Activity

The main activity is the heart of the app, it is the activity that is called at the start up. It sets up the initial layout and shows the item list that we previously described:

```
itemListView = (ListView) findViewById(R.id.listItmes);
adpt = new ToDoItemAdapter(this, itemList);
itemListView.setAdapter(adpt);
```

Now we have to check if our app runs on a smart phone or a tablet so that we can change the activity behavior. We can do it by verifying if there is a `FrameLayout` component in the layout definition:

```
if (findViewById(R.id.frm1) != null)
   isTablet = true;
```

6.2.7 Action bar

In this activity we add a action bar (a well-known Android pattern) with an action: add new item. To define it, we create (if not exists) a XML file under `res/menu`:

```
<menu xmlns:android="http://schemas.android.com/apk/res/android" >

    <item
        android:id="@+id/action_add"
        android:icon="@android:drawable/ic_menu_add"
        android:orderInCategory="0"
        android:showAsAction="always"/>

</menu>
```

As a result we obtain:

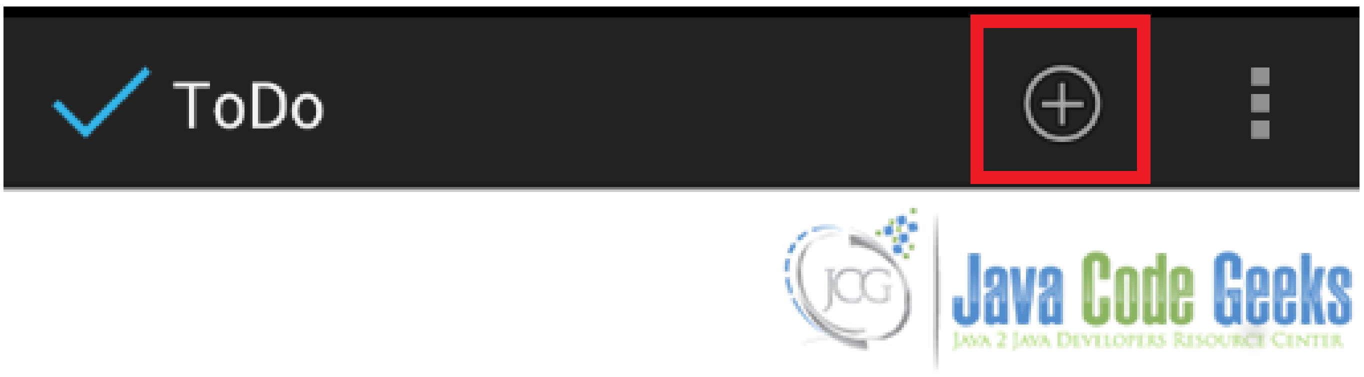


Figure 6.7: screenshot

As a user clicks on the plus sign, we should show the user interface for adding a new item. The first thing we have to handle is the event when user clicks on *plus* icon so we have to override a method in the activity class:

```
@Override
public boolean onOptionsItemSelected(MenuItem item) {
        int menuId = item.getItemId();
        switch (menuId) {
                case R.id.action_add: {
                        if (!isTablet) {
                                Intent i = new Intent(this, NewItemActivity.class);
                                startActivityForResult(i, ADD_ITEM);
                                break;
                        }
                        else {
                                Log.d("TODO", "Tablet");
                                FragmentTransaction ft = getFragmentManager(). ↩
                                    beginTransaction();
                                NewItemFragment nif = new NewItemFragment();
                                ft.replace(R.id.frm1, nif);
                                ft.commit();
                        }
                }
        }
}
```

This method is very important because we define how our activity should behave. If the button clicked by the user is our add button, we first verify if our app is running on a smart phone. In this case, we know we have to start another activity and we start it waiting for a result.

If we start an activity waiting for its result in the called activity we should return the result and we do it in the `NewItemActivity` in this way:

```
@Override
public void onAddItem(Item item) {
        // We get the item and return to the main activity
        Log.d("TODO", "onAddItem");
        Intent i = new Intent();
        i.putExtra("item", item);
        setResult(RESULT_OK,i);
        finish();
}
```

In this method we create an Intent that holds the result and pass it back to the calling activity (`MainActivity`) and we finish the activity. In the `MainActivity`, then, we have to be ready to handle the result:

```
protected void onActivityResult(int requestCode, int resultCode, Intent data) {
        Log.d("TODO", "OnResult");
        if (requestCode == ADD_ITEM) {
                if (resultCode == RESULT_OK) {
                        Log.d("TODO", "OK");
                        Item i = (Item) data.getExtras().getSerializable("item");
                        itemList.add(i);
                        adpt.notifyDataSetChanged();
                }
        }
}
```

In this activity we extract the item object stored inside the `Intent` that we receive as result and add it to the item list. When we finish we call `notifyDataSetChange` method so that the `ListView` can be updated.

If our app is running on a tablet we simply “fill” the `FrameLayout` with the `NewItemFragment` described above. In this case we start a transaction and replace the `FrameLayout` with the fragment and at the end we commit the transaction. In this case, we do not need to start another activity because we use the `FrameLayout` to show the fragment that handles the user interface for adding a new item. So we have:

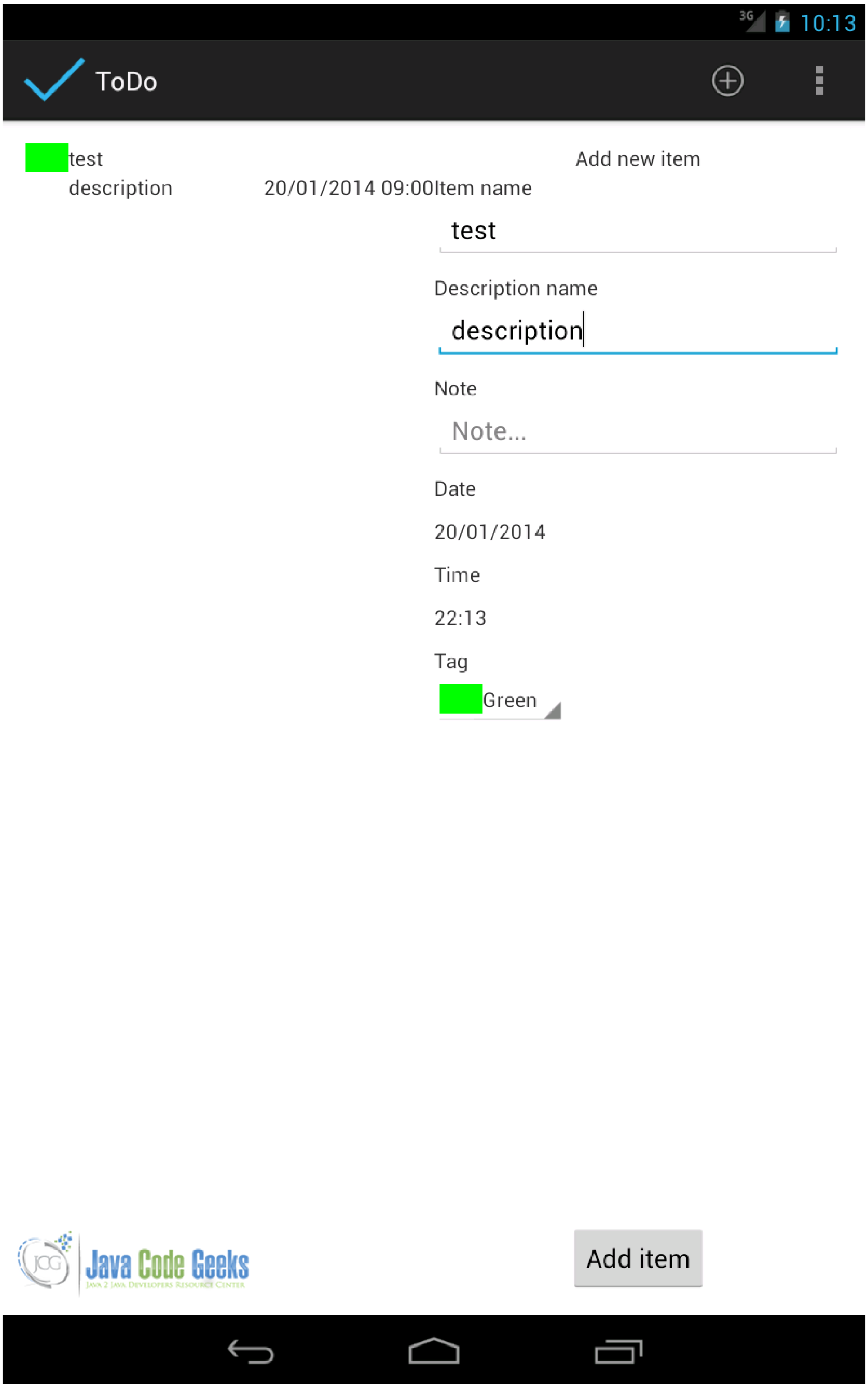


Figure 6.8: screenshot

In this case, we have to simply implement the interface specified by the fragment as we did before:

```
@Override
public void onAddItem(Item item) {
        itemList.add(item);
        adpt.notifyDataSetChanged();
        NewItemFragment nif = (NewItemFragment) getFragmentManager().findFragmentById(R.id. ↩
            frm1);
        getFragmentManager().beginTransaction().remove(nif).commit();
}
```

Notice in the last part that we removed the fragment at the end.

6.3 Styling the app

By now we have not considered the application's style, but we know we can apply style to every UI component. We can modify all the UI colors and looks, implementing our style and branding the app. For example we want to apply a style to the listView row making them a little more appealing. We create (if it does not already exist), a file called `style.xml` under `res/values` and here we can define our style as we discussed in a previous chapter:

```
<style name="dateStyle">
        <item name="android:textAppearance">?android:textAppearanceSmall</item>
        <item name="android:textColor">@color/red</item>
    </style>

     <style name="descrStyle">
        <item name="android:textStyle">italic</item>
        <item name="android:textAppearance">?android:textAppearanceSmall</item>
    </style>

     <style name="nameStyle">
         <item name="android:textAppearance">?android:textAppearanceMedium</item>
         <item name="android:textStyle">bold</item>
     </style>
```

We defined three different styles and we want to apply them to our listview row:

```
<TextView
    android:id="@+id/nameView"
    style="@style/nameStyle"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_alignParentTop="true"
    android:layout_toRightOf="@id/tagView" />
<TextView
    android:id="@+id/descrView"
    style="@style/descrStyle"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_below="@id/nameView"
    android:layout_toRightOf="@id/tagView" />
<TextView
    android:id="@+id/dateView"
    style="@style/dateStyle"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_alignParentBottom="true"
    android:layout_alignParentRight="true" />
```

Running the app with this style we have:

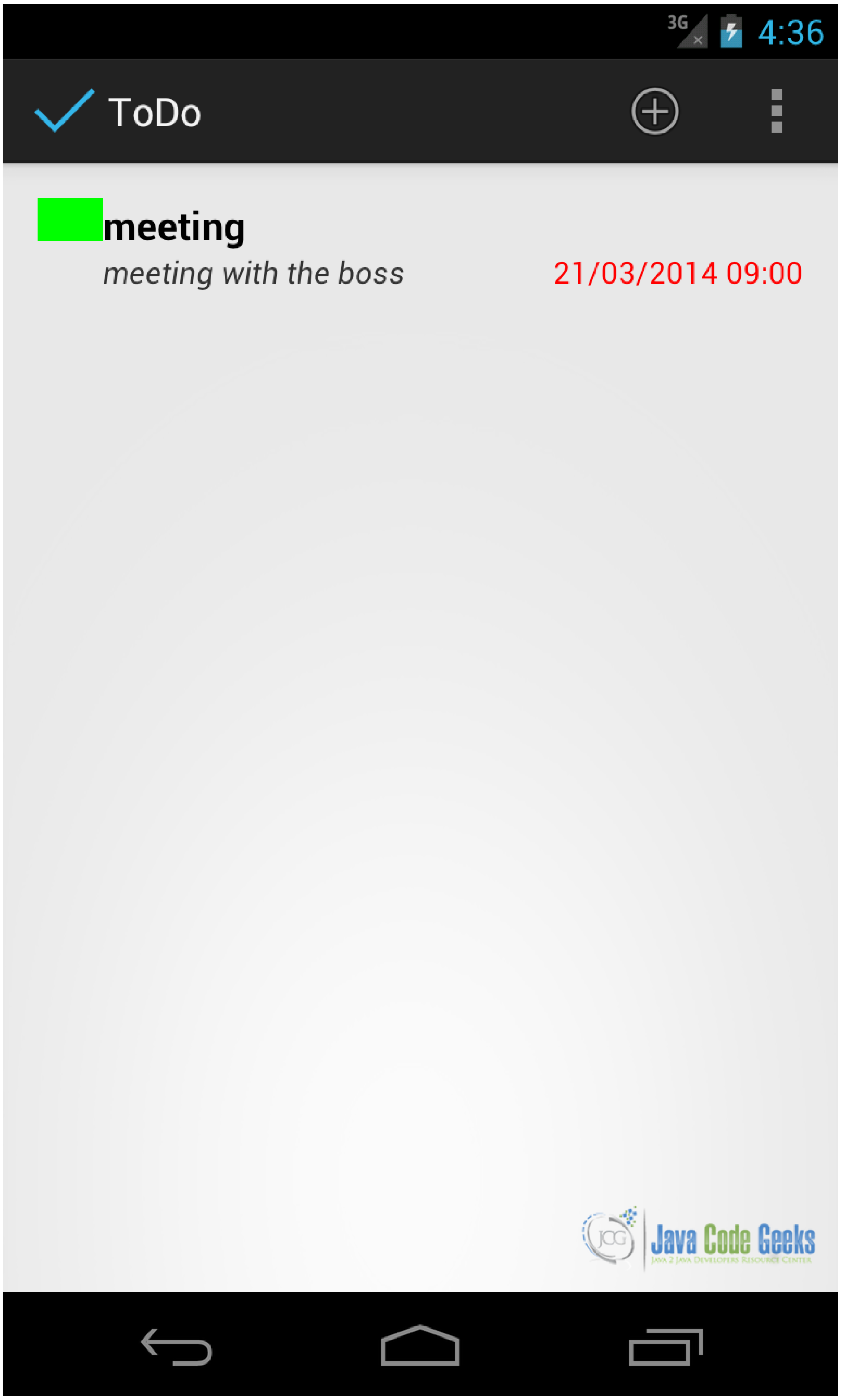


Figure 6.9: screenshot

6.4 Conclusion

In this Android UI course we covered some important UI aspects, concepts and best practices we should follow when developing an Android app. In this last article we saw how to apply all this knowledge to build a real app using all the topics covered in the previous chapters. Of course, this simple app can be improved and you can have it as an exercise to try to add new functionalities. For example, we did not cover how to modify or delete the items in the list. In this case we could listen when a user clicks on an item and you could show a menu with several options, or we could use the action bar changing it according to the user actions. For example we could create a multiple selection list view and when user clicks on “trash” icon in the action bar we remove all the items selected.

There are different aspects we can improve in this app and it will be a good exercise if you want to go deeper in Android development aspects.

6.5 Download the Source Code

This was a lesson on how to create an Android app UI from scratch. You may download the source code here: AndroidApp.zip