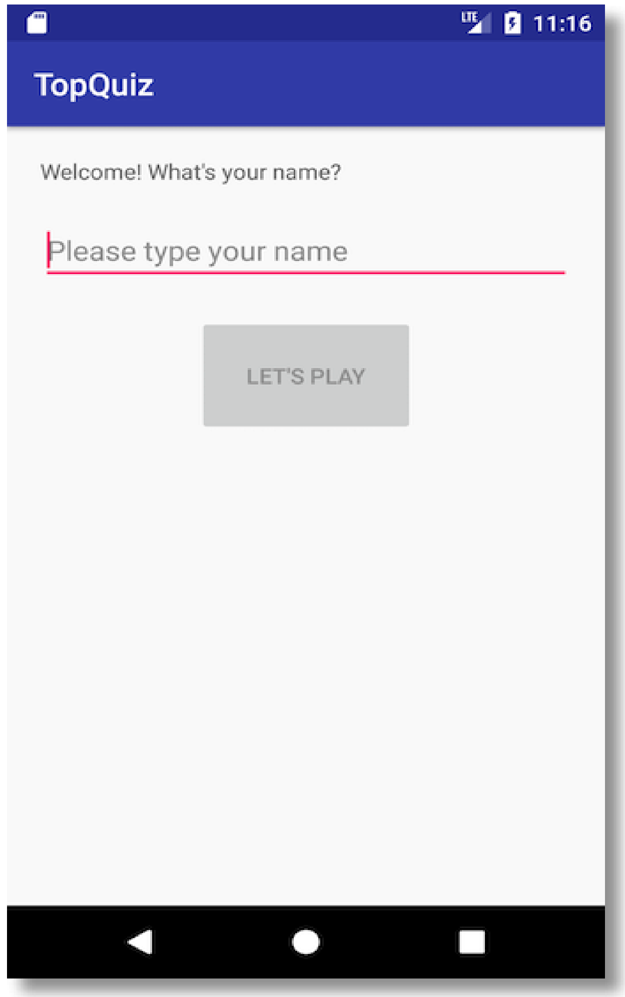
Develop Your First Android Application

Design Your First Screen

- we will draw the user interface for the application’s first screen.
- In other words, we will determine which graphic elements we need and how to position them on the screen.

In this first screen, we want to welcome the user by asking them to enter their first name. This screen will require a text field, an input area, and a button. The expected result is this:

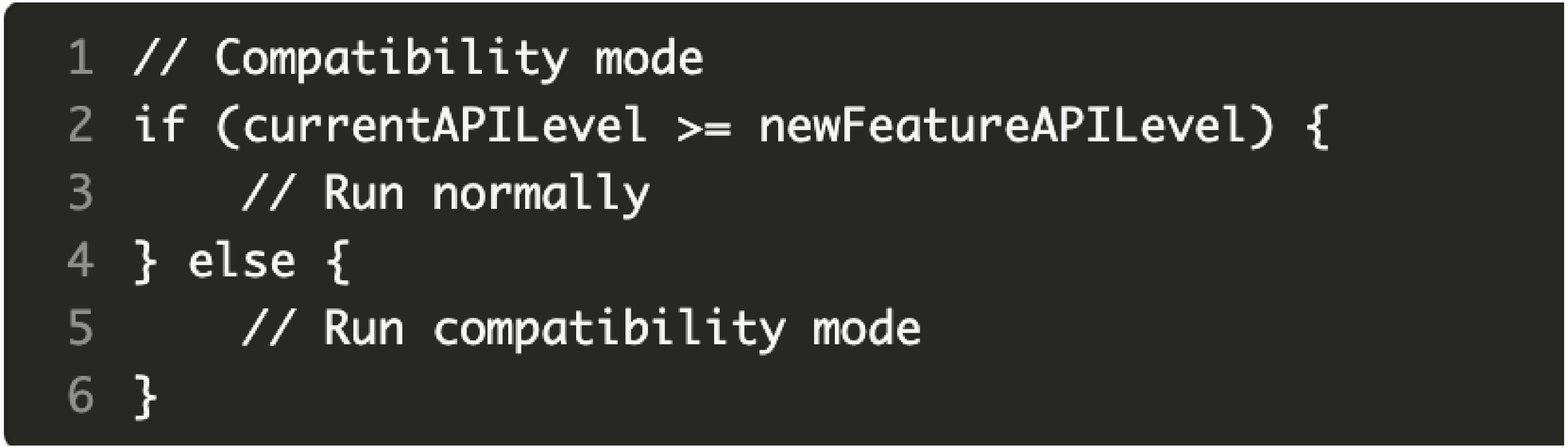


Activity and Layout

- An Activity, as noted previously, is a fundamental building block of Android (one of Android's core app components). Activities also provide the entry point to all Android applications.

- One Activity in particular is responsible for acting as the entry point for the user. This Activity is a class which inherits from either the Android Activity class or the **AppCompatActivity** class.

To bring new features to old versions of Android, Google began releasing support libraries. These libraries provide classes that imitate new features on old devices by running in “compatibility” mode. The behavior looks like this:



```
// Compatibility mode
if (currentAPILevel >= newFeatureAPILevel) {
    // Run normally
} else {
    // Run compatibility mode
}
```

The AppCompatActivity behaves like an Activity but also brings modern design features, like the ActionBar, to older API versions. Google does not offer compatibility modes for every new feature: they primarily focus on **backporting** new user interface classes.

Backporting means taking parts from a newer version of software systems or components and adapting them to an older version of the same software.

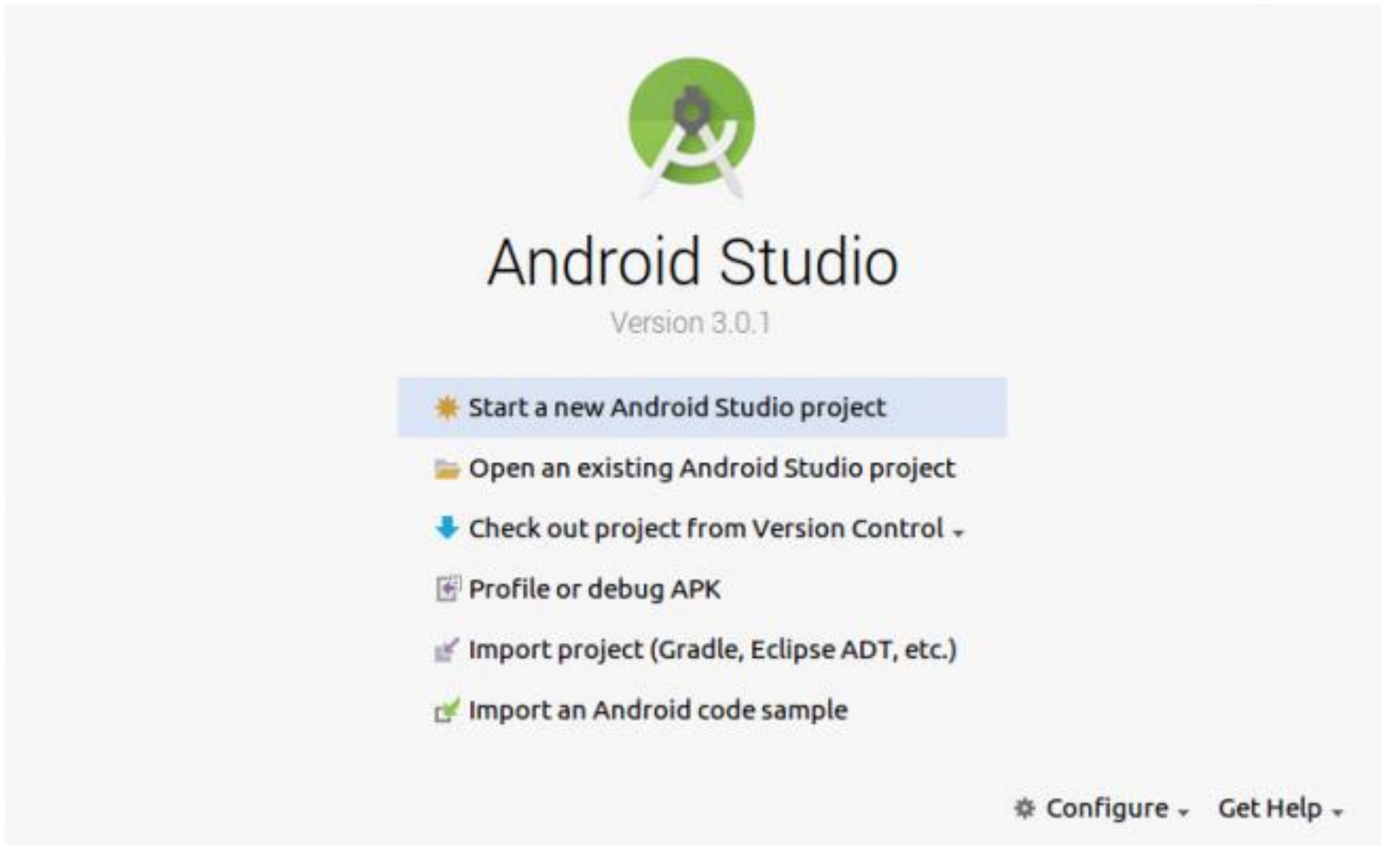
If you want to have two screens in your application (e.g. a login screen and a tutorial screen), you will generally use two Activities. The first Activity handles the login flow and the second manages the tutorial. By convention, the name of an Activity is always followed by the word “Activity” and is written in CamelCase. Therefore, you might name your two Activities "LoginActivity" and "TutorialActivity," respectively.

Why did I say “you will *generally* use two activities?”

- it’s not mandatory to have a separate Activity for each screen, but it’s not essential to look into that this early in the game. 😉
- To allow the user to interact with our application, we have to present graphic and control elements (aka. **widgets**) so that they can see and touch our app. These widgets can be buttons, input fields, drop-down menus, images, and a lot more.
- To choose which graphic elements to use and where to position them on screen, we use **a layout file**. A layout file is an XML file which the Activity loads upon creation. This XML file is always stored in the **res/layout** directory of your project.

Android ListView

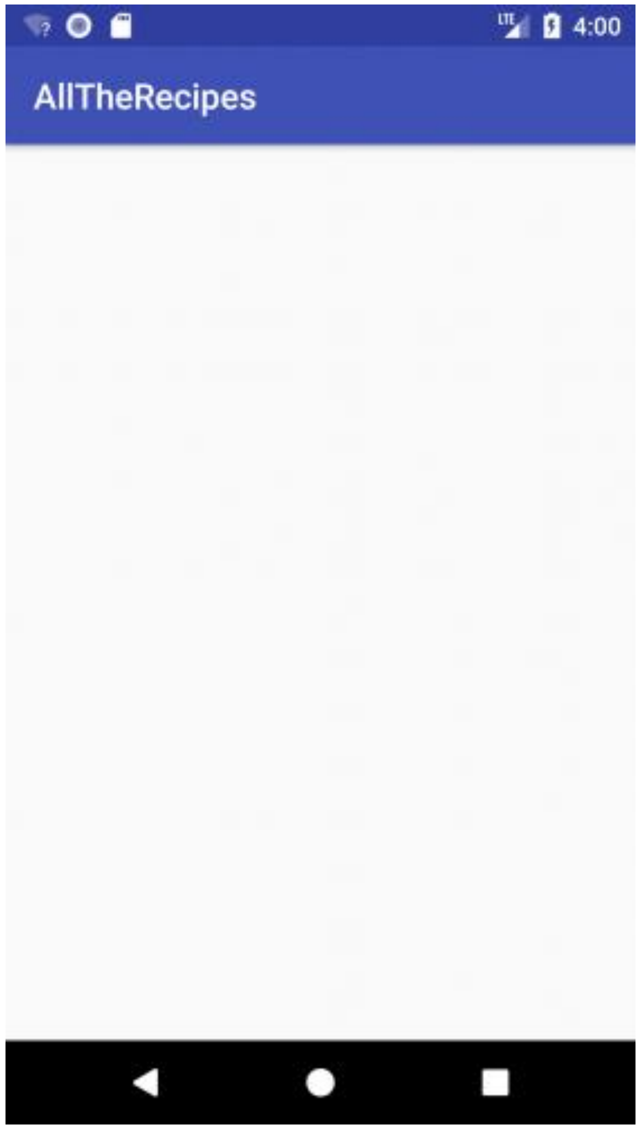
- In the **Welcome to Android Studio** dialog, select **Open an existing Android Studio project**.


Android Studio
Version 3.0.1
Start a new Android Studio project
Open an existing Android Studio project
Check out project from Version Control
Profile or debug APK
Import project (Gradle, Eclipse ADT, etc.)
Import an Android code sample
Configure
Get Help

- In the following dialog, select the top-level directory of the starter project our given file and click **OK**.

  Inside the imported project, you'll find some assets and resources that you'll use to create your app, such as strings, colors, XML layout files, and fonts. Additionally, there's some boilerplate code modeling a Recipe and a bare bones MainActivity class.

Build and run. You should see something like this:



Add Your First ListView

The first order of business is to add a ListView to MainActivity.

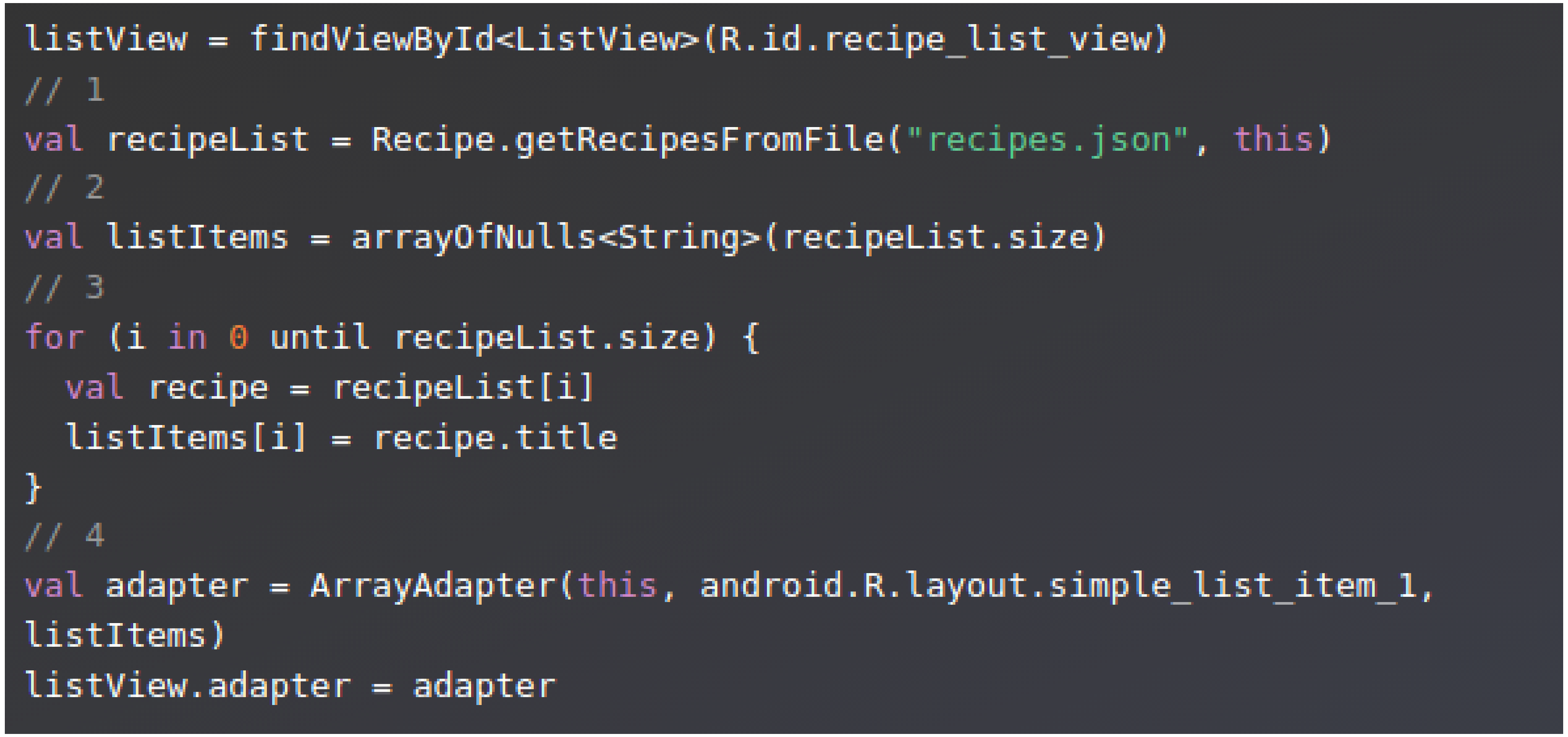
Open **res/layout/activity_main.xml**. As you may know, this is the file that describes the layout of MainActivity. Add a ListView to MainActivity by inserting the following code snippet inside the ConstraintLayout tag:

```
<ListView
  android:id="@+id/recipe_list_view"
  android:layout_width="0dp"
  android:layout_height="0dp"
  app:layout_constraintBottom_toBottomOf="parent"
  app:layout_constraintEnd_toEndOf="parent"
  app:layout_constraintStart_toStartOf="parent"
  app:layout_constraintTop_toTopOf="parent" />
```

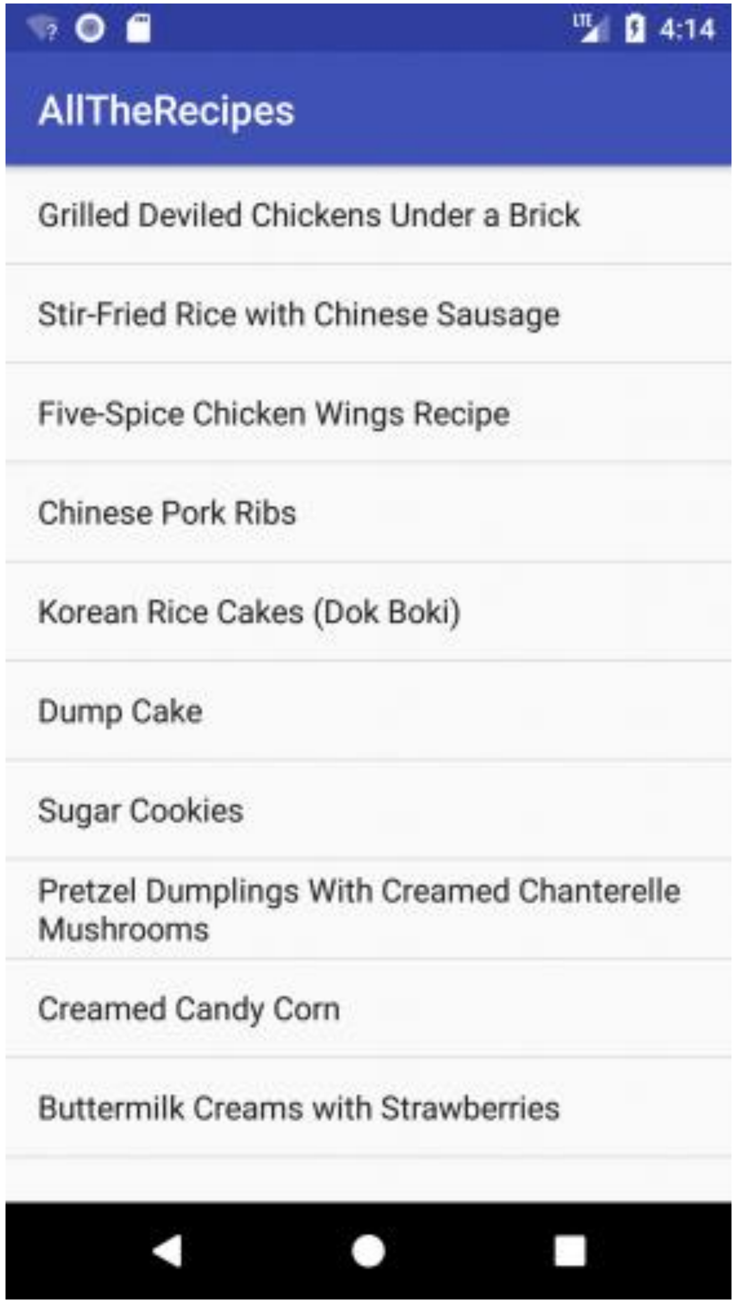
Open **MainActivity** and add an instance variable for your ListView with the following line:

private lateinit var listView ListView

Add the following snippet below the existing code inside the onCreate method:



```
listView = findViewById<ListView>(R.id.recipe_list_view)
// 1
val recipeList = Recipe.getRecipesFromFile("recipes.json", this)
// 2
val listItems = arrayOfNulls<String>(recipeList.size)
// 3
for (i in 0 until recipeList.size) {
  val recipe = recipeList[i]
  listItems[i] = recipe.title
}
// 4
val adapter = ArrayAdapter(this, android.R.layout.simple_list_item_1,
listItems)
listView.adapter = adapter
```


AllTheRecipes

Grilled Deviled Chickens Under a Brick

Stir-Fried Rice with Chinese Sausage

Five-Spice Chicken Wings Recipe

Chinese Pork Ribs

Korean Rice Cakes (Dok Boki)

Dump Cake

Sugar Cookies

Pretzel Dumplings With Creamed Chanterelle Mushrooms

Creamed Candy Corn

Buttermilk Creams with Strawberries

Here's a breakdown of what's happening in there:

- This loads a list of Recipe objects from a JSON asset in the app. Notice that the starter project contains a Recipe class that models and stores the information about the recipes that will be displayed.
- This creates an array of strings that'll contain the text to be displayed in the ListView.
- This populates the ListView's data source with the titles of the recipes loaded in section one.
- This creates and sets a simple adapter for the ListView. The ArrayAdapter takes in the current context, a layout file specifying what each row in the list should look like, and the data that will populate the list as arguments.

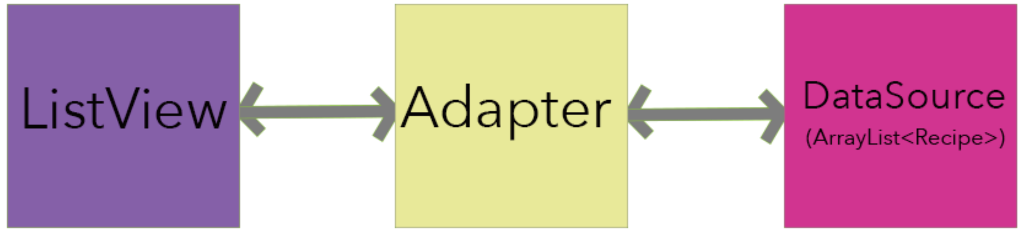
Enough talk! Your ListView has all that it needs to function. Build and run the project. You should see something like this:

Adapters: Servants of the ListView

Your recipe app is starting to look functional, but not all that appetizing…yet.
In the previous section, you successfully built a list of recipe titles. It works, but it’s nothing to get excited about. What if you needed to show more than just the titles? More than just text? Maybe even add some screen-licking worthy thumbnails?
For these cases, the simple ArrayAdapter you just used won’t cut it. You’ll have to take matters into your own hands and write your own adapter. Well, you won’t actually write your own adapter, per se; you’ll simply extend a regular adapter and make some tweaks.

What Exactly is an Adapter?

An adapter loads the information to be displayed from a data source, such as an array or database query, and creates a view for each item. Then it inserts the views into the ListView.
Adapters not only exist for ListViews, but for other kinds of views as well; ListView is a subclass of AdapterView, so you can populate it by binding it to an adapter.



The adapter acts as the middle man between the ListView and data source, or its provider. It works kind of like this:

The ListView asks the adapter what it should display, and the adapter jumps into action:

- It fetches the items to be displayed from the data source
- It decides how they should be displayed
- It passes this information on to the ListView

In short, The ListView isn’t very smart, but when given the right inputs it does a fine job. It fully relies on the adapter to tell it what to display and how to display it.

Building Adapters:

Okay, now that you’ve dabbled in theory, you can get on with building your very own adapter.
Create a new class by **right-clicking** on the **com.raywenderlich.alltherecipes** package and selecting **New > Kotlin File/Class**. Name it **RecipeAdapter** and define it with the following:

```
class RecipeAdapter : BaseAdapter() { }
```

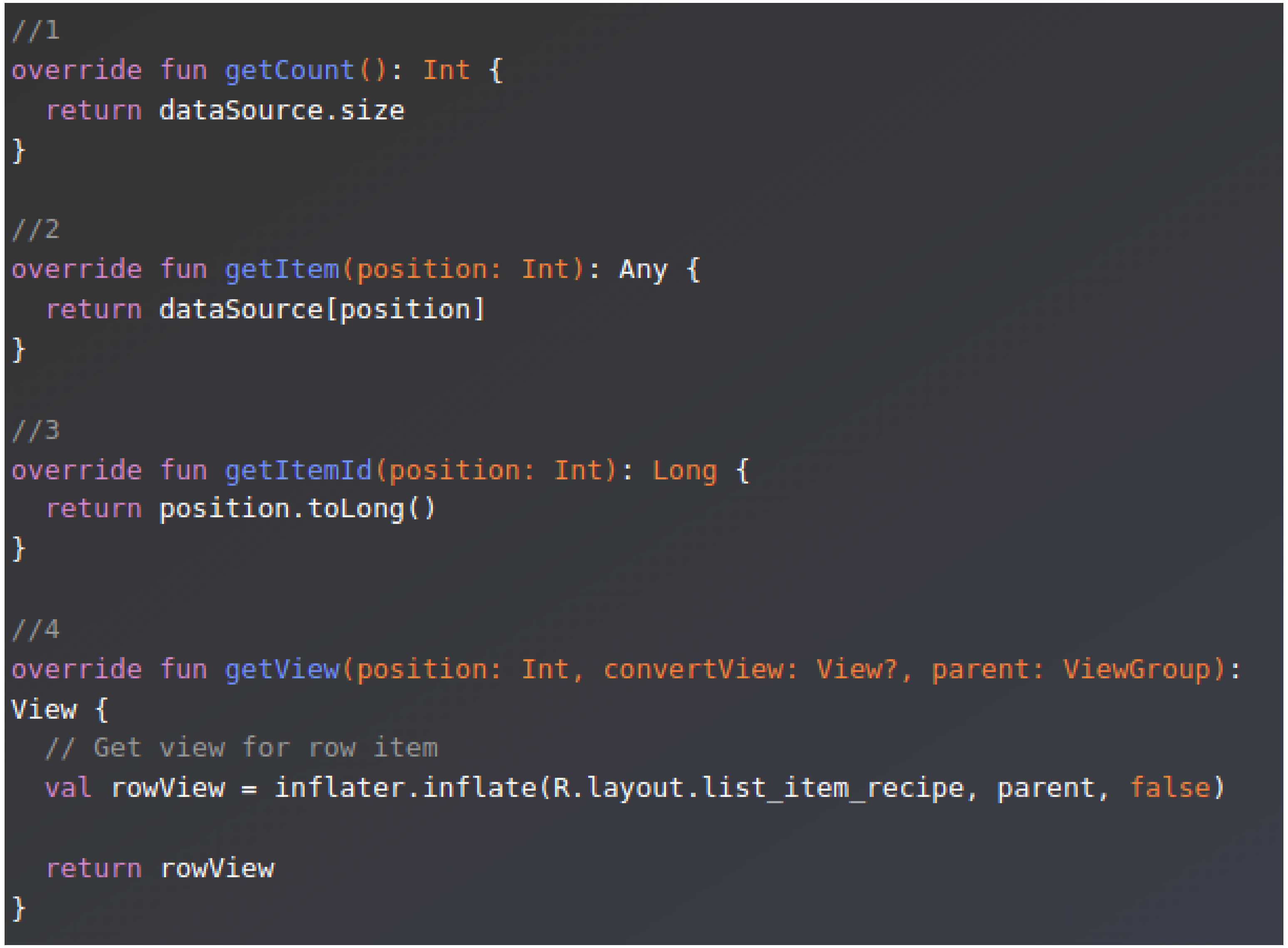
You’ve made the skeleton of the adapter. It extends the BaseAdapter class, which requires several inherited methods you’ll implement after taking care of one more detail. Update the RecipeAdapter class as follows:

```
class RecipeAdapter(private val context: Context, private val dataSource:

ArrayList<Recipe>) : BaseAdapter() { private val inflater: LayoutInflater =

context.getSystemService(Context.LAYOUT_INFLATER_SERVICE) as LayoutInflater }
```

Your next step is to implement the adapter methods. Kick it off by placing the following code at the bottom of RecipeAdapter:



```kotlin
//1
override fun getCount(): Int {
  return dataSource.size
}

//2
override fun getItem(position: Int): Any {
  return dataSource[position]
}

//3
override fun getItemId(position: Int): Long {
  return position.toLong()
}

//4
override fun getView(position: Int, convertView: View?, parent: ViewGroup):
View {
  // Get view for row item
  val rowView = inflater.inflate(R.layout.list_item_recipe, parent, false)

  return rowView
}
```

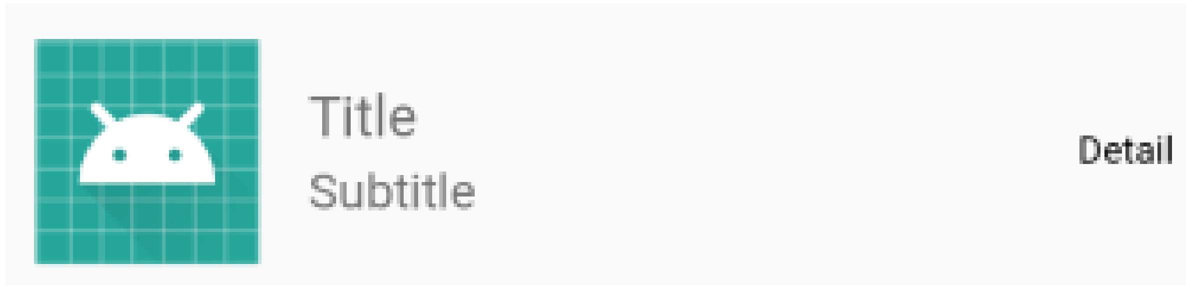
Here’s a step-by-step breakdown:

- getCount() lets ListView know how many items to display, or in other words, it returns the size of your data source.
- getItem() returns an item to be placed in a given position from the data source, specifically, Recipe objects obtained from dataSource.
- This implements the getItemId() method that defines a unique ID for each row in the list. For simplicity, you just use the position of the item as its ID.
- Finally, getView() creates a view to be used as a row in the list. Here you define what information shows and where it sits within the ListView. You also inflate a custom view from the XML layout defined in res/layout/list_item_recipe.xml — more on this in the next section.

Defining the Layout of the ListView's Rows

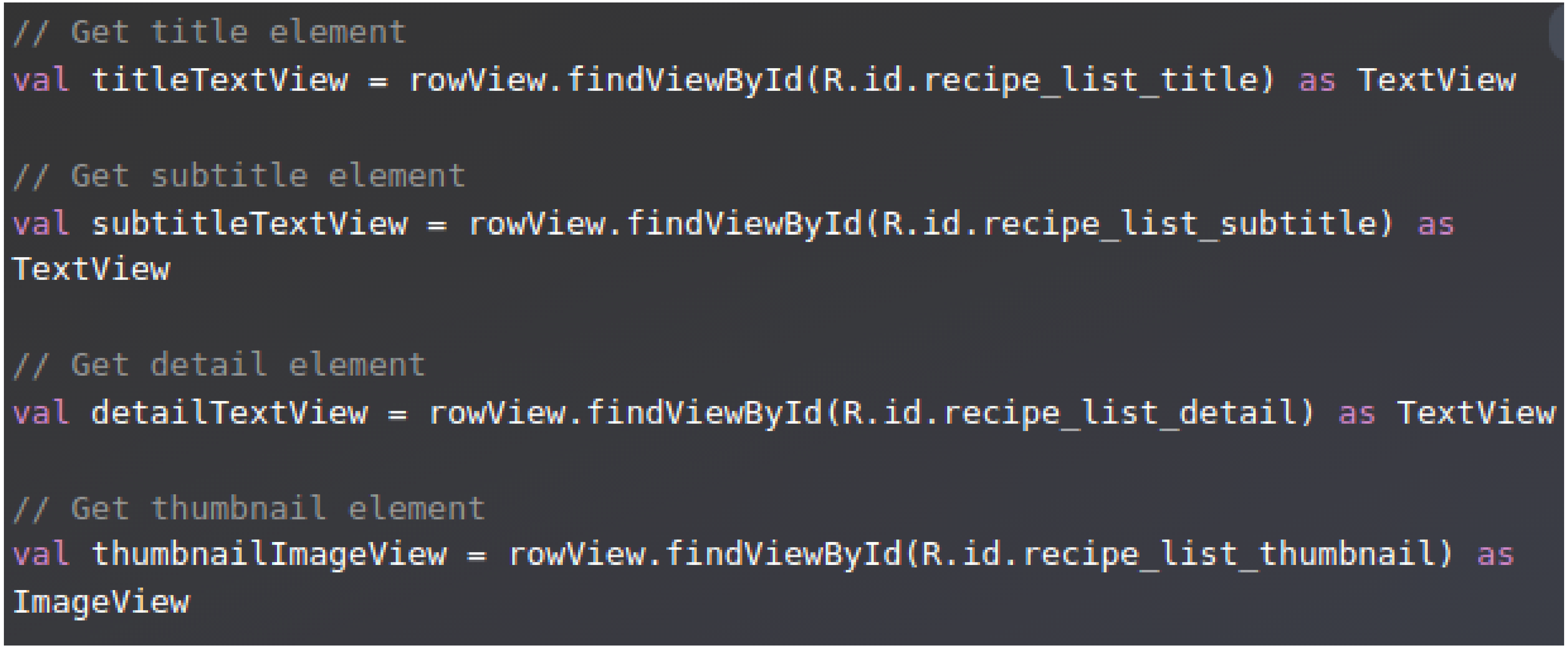
You probably noticed that the starter project comes with the file res/layout/list_item_recipe.xml that describes how each row in the ListView should look and be laid out.

Below is an image that shows the layout of the row view and its elements:



Your task is to populate each element of the row view with the relevant recipe data, hence, you'll define what text goes in the "title" element, the "subtitle" element and so on.

In the getView() method, add the following code snippet just before the return statement:



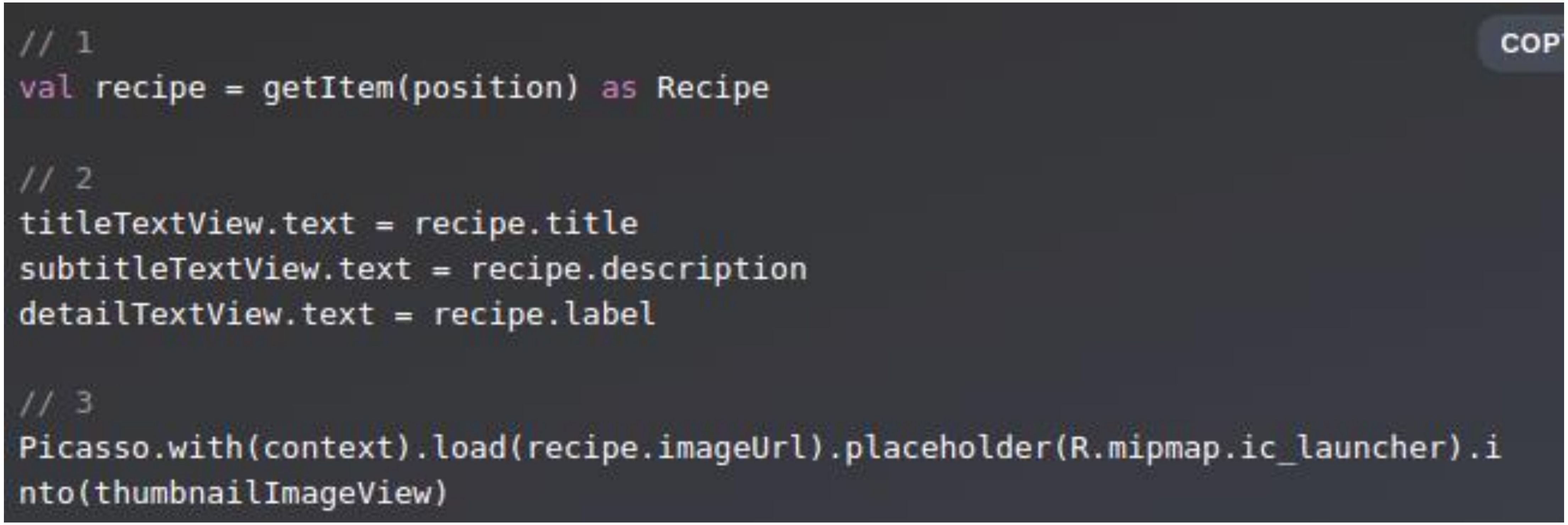
```
// Get title element
val titleTextView = rowView.findViewById(R.id.recipe_list_title) as TextView

// Get subtitle element
val subtitleTextView = rowView.findViewById(R.id.recipe_list_subtitle) as
TextView

// Get detail element
val detailTextView = rowView.findViewById(R.id.recipe_list_detail) as TextView

// Get thumbnail element
val thumbnailImageView = rowView.findViewById(R.id.recipe_list_thumbnail) as
ImageView
```

Now that you’ve got the references sorted out, you need to populate each element with relevant data. To do this, add the following code snippet under the previous one but before the return statement:



```
// 1
val recipe = getItem(position) as Recipe

// 2
titleTextView.text = recipe.title
subtitleTextView.text = recipe.description
detailTextView.text = recipe.label

// 3
Picasso.with(context).load(recipe.imageUrl).placeholder(R.mipmap.ic_launcher).i
nto(thumbnailImageView)
```

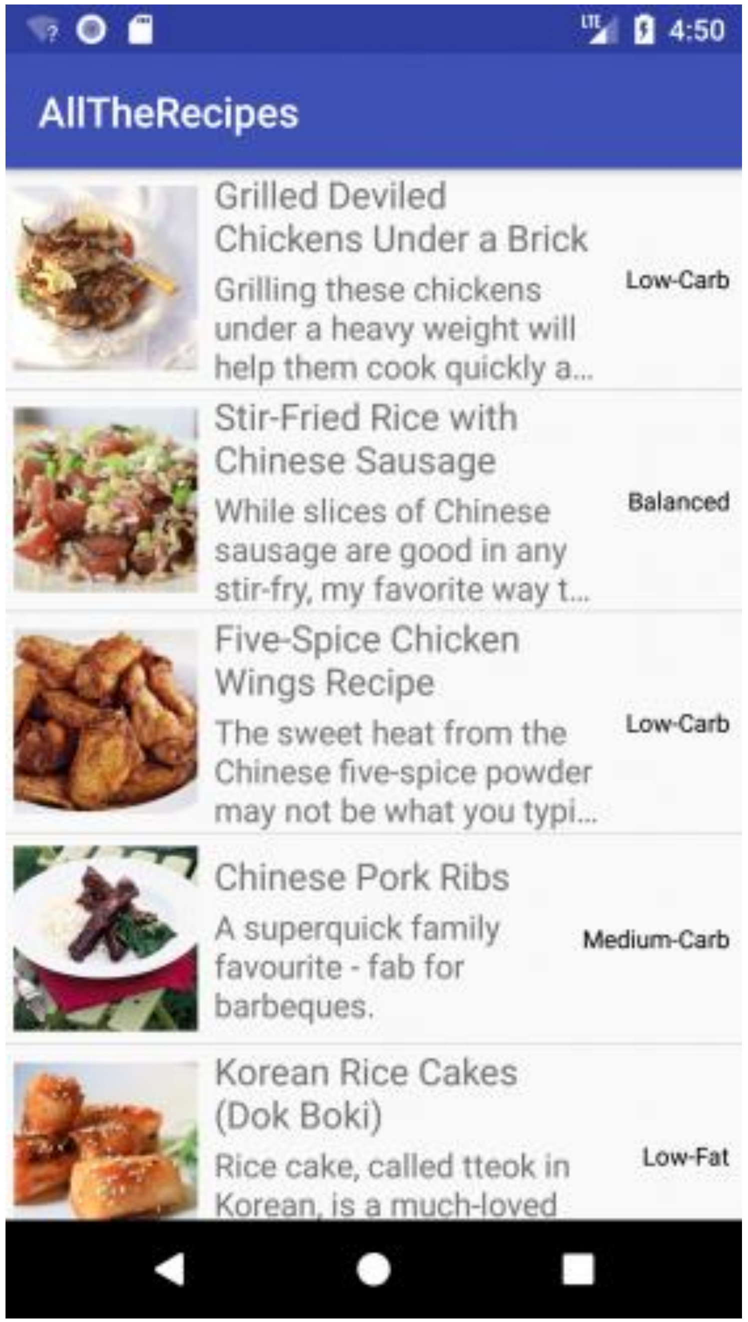
- Getting the corresponding recipe for the current row.
- Updating the row view’s text views so they are displaying the recipe.
- Making use of the open-source Picasso library for asynchronous image loading — it helps you download the thumbnail images on a separate thread instead of the main thread. You’re also assigning a temporary placeholder for the ImageView to handle slow loading of images

Now open up **MainActivity** so that you can get rid of the old adapter. In onCreate, replace everything below (but *not* including) this line:

```
val recipeList = Recipe.getRecipesFromFile("recipes.json", this)
```

With:

```
val adapter = RecipeAdapter(this, recipeList) listView.adapter = adapter
```


4:50

AllTheRecipes

Grilled Deviled Chickens Under a Brick
Grilling these chickens under a heavy weight will help them cook quickly a...
Low-Carb

Stir-Fried Rice with Chinese Sausage
While slices of Chinese sausage are good in any stir-fry, my favorite way t...
Balanced

Five-Spice Chicken Wings Recipe
The sweet heat from the Chinese five-spice powder may not be what you typi...
Low-Carb

Chinese Pork Ribs
A superquick family favourite - fab for barbeques.
Medium-Carb

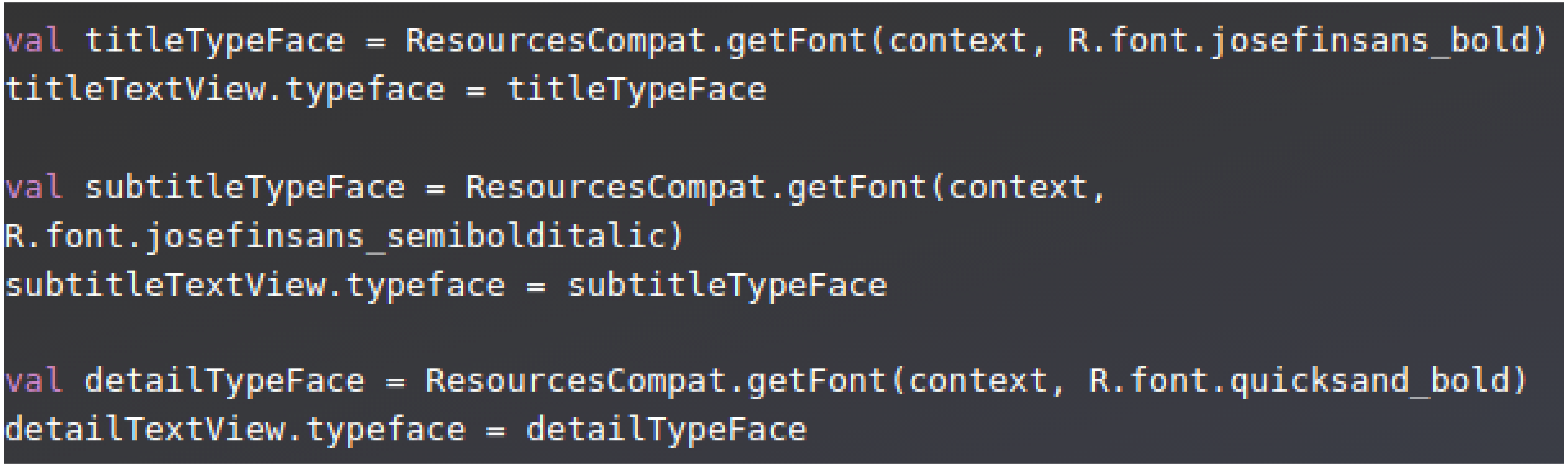
Korean Rice Cakes (Dok Boki)
Rice cake, called tteok in Korean, is a much-loved
Low-Fat

You just replaced the rather simple ArrayAdapter with your own RecipeAdapter to make the list more informative.

Build and run and you should see something like this:

Styling:

Now that you’ve got the functionality under wraps, it’s time to turn your attention to the finer things in life. In this case, your finer things are elements that make your app more snazzy, such as compelling colors and fancy fonts.
Start with the fonts. Look for some custom fonts under **res/font**. You’ll find three font
files: **josefinsans_bold.ttf**, **josefinsans_semibolditalic.ttf** and **quicksand _bold.otf**.
Open **RecipeAdapter.java** and go to the getView() method. Just before the return statement, add the following:



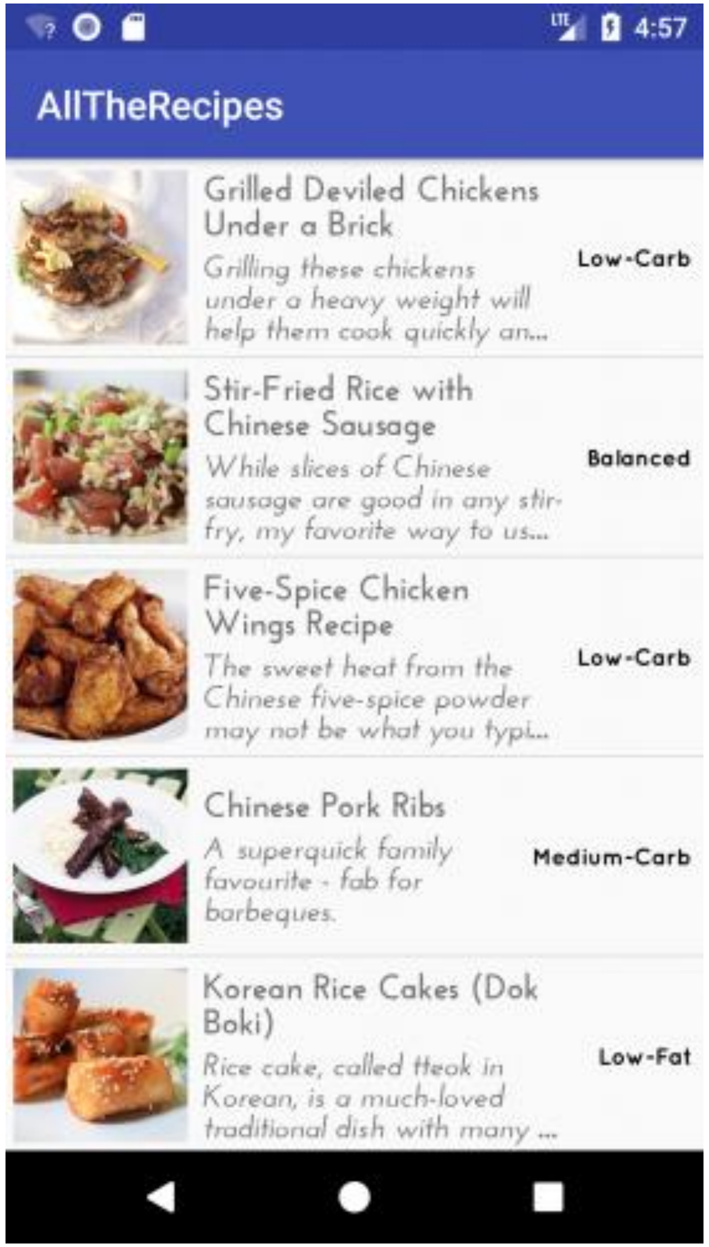
```kotlin
val titleTypeFace = ResourcesCompat.getFont(context, R.font.josefinsans_bold)
titleTextView.typeface = titleTypeFace

val subtitleTypeFace = ResourcesCompat.getFont(context,
R.font.josefinsans_semibolditalic)
subtitleTextView.typeface = subtitleTypeFace

val detailTypeFace = ResourcesCompat.getFont(context, R.font.quicksand_bold)
detailTextView.typeface = detailTypeFace
```

In here, you're assigning a custom font to each of the text views in your rows' layout. You access the font by creating a Typeface, which specifies the intrinsic style and typeface of the font, by using ResourcesCompat.getFont(). Next you set the typeface for the corresponding TextView to set the custom font.

Now build and run. Your result should look like this:



4:57

AllTheRecipes

Grilled Deviled Chickens Under a Brick
Grilling these chickens under a heavy weight will help them cook quickly an...
Low-Carb

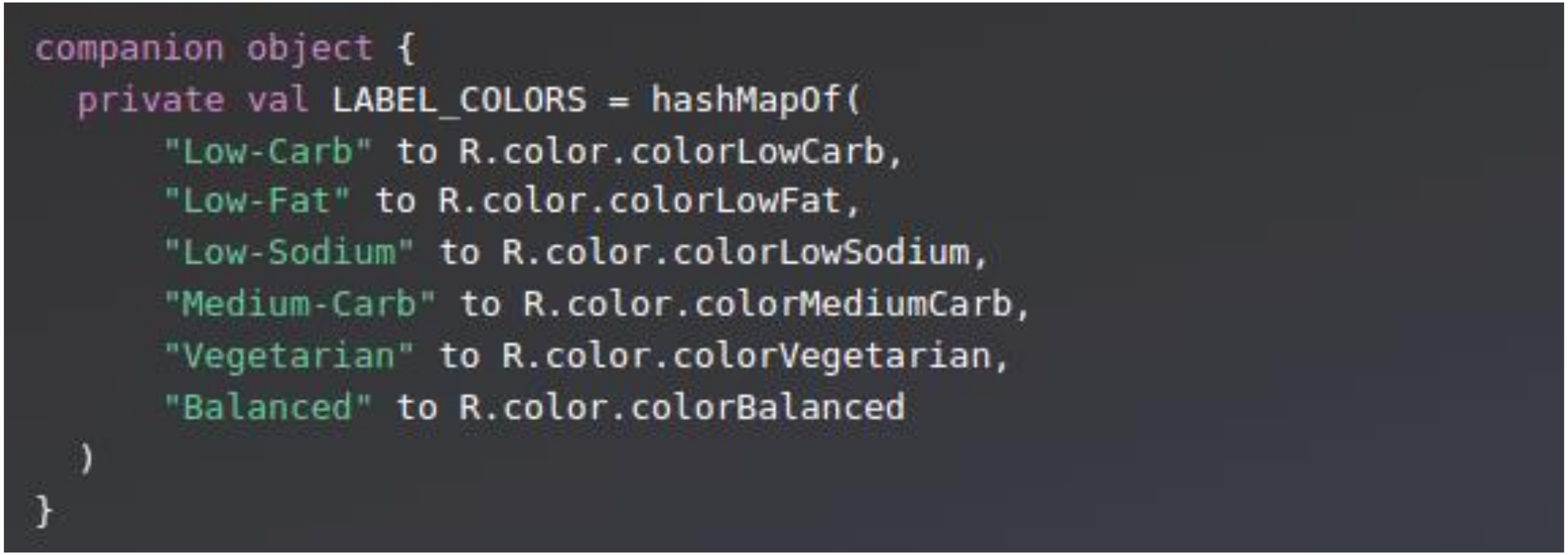
Stir-Fried Rice with Chinese Sausage
While slices of Chinese sausage are good in any stir-fry, my favorite way to us...
Balanced

Five-Spice Chicken Wings Recipe
The sweet heat from the Chinese five-spice powder may not be what you typi...
Low-Carb

Chinese Pork Ribs
A superquick family favourite - fab for barbeques.
Medium-Carb

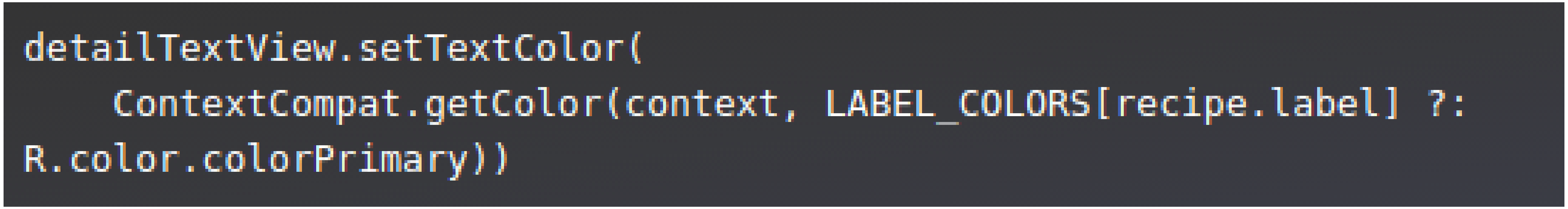
Korean Rice Cakes (Dok Boki)
Rice cake, called tteok in Korean, is a much-loved traditional dish with many ...
Low-Fat

On to sprucing up the colors, which are defined in **res/values/colors.xml**. Open up **RecipeAdapter** and add the following below the inflater declaration:



```
companion object {
  private val LABEL_COLORS = hashMapOf(
      "Low-Carb" to R.color.colorLowCarb,
      "Low-Fat" to R.color.colorLowFat,
      "Low-Sodium" to R.color.colorLowSodium,
      "Medium-Carb" to R.color.colorMediumCarb,
      "Vegetarian" to R.color.colorVegetarian,
      "Balanced" to R.color.colorBalanced
  )
}
```

You’ve created a hash map that pairs a recipe detail label with the resource id of a color defined in **colors.xml**.
Now go to the getView() method, and add this line just above the return statement:



```
detailTextView.setTextColor(
    ContextCompat.getColor(context, LABEL_COLORS[recipe.label] ?:
R.color.colorPrimary))
```

Working from the inside out:

•Here you get the resource id for the color that corresponds to the recipe.label from the LABEL_COLORS hash map.

•getColor() is used inside of ContextCompat to retrieve the hex color associated with that resource id.

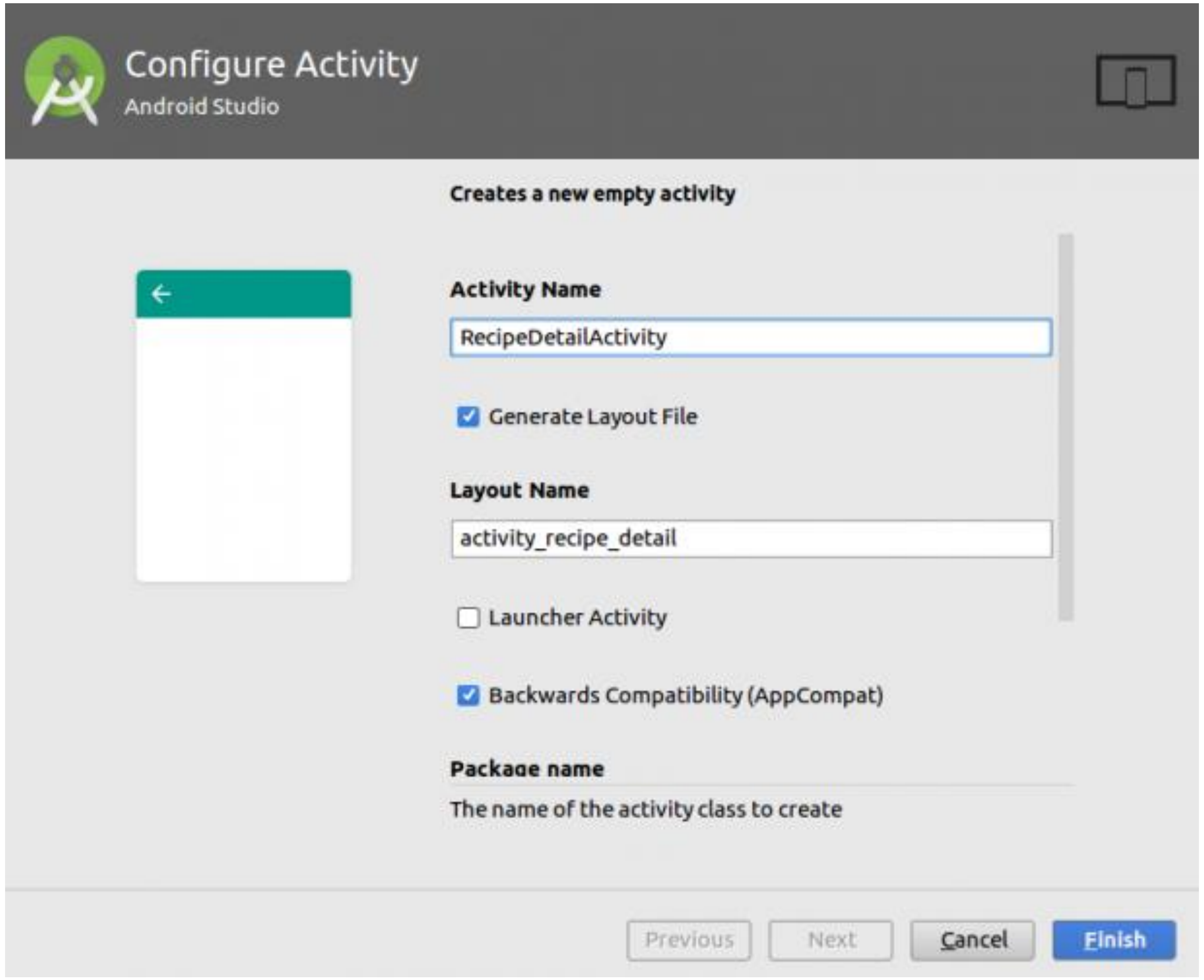
•Then you set the color property of the detailTextView to the hex color.

User Interaction:

Now your list has function and style. What’s it missing now? Try tapping or long pressing it. There’s not much to thrill and delight the user.
What could you add here to make the user experience *that much more satisfying*? Well, when a user taps on a row, don’t you think it’d be nice to show the full recipe, complete with instructions?
You’ll make use of **AdapterView.onItemClickListener** and a brand spanking new activity to do this with elegance.

Make a New Activity:

This activity will display when the user selects an item in the list.
Right-click on **com.raywenderlich.alltherecipes** then select **New > Activity > EmptyActivity** to bring up a dialog. Fill in the **Activity Name** with **RecipeDetailActivity**. Leave the automatically populated fields as-is. Check that your settings match these:


Configure Activity
Android Studio

Creates a new empty activity

Activity Name
RecipeDetailActivity

[✓] Generate Layout File

Layout Name
activity_recipe_detail

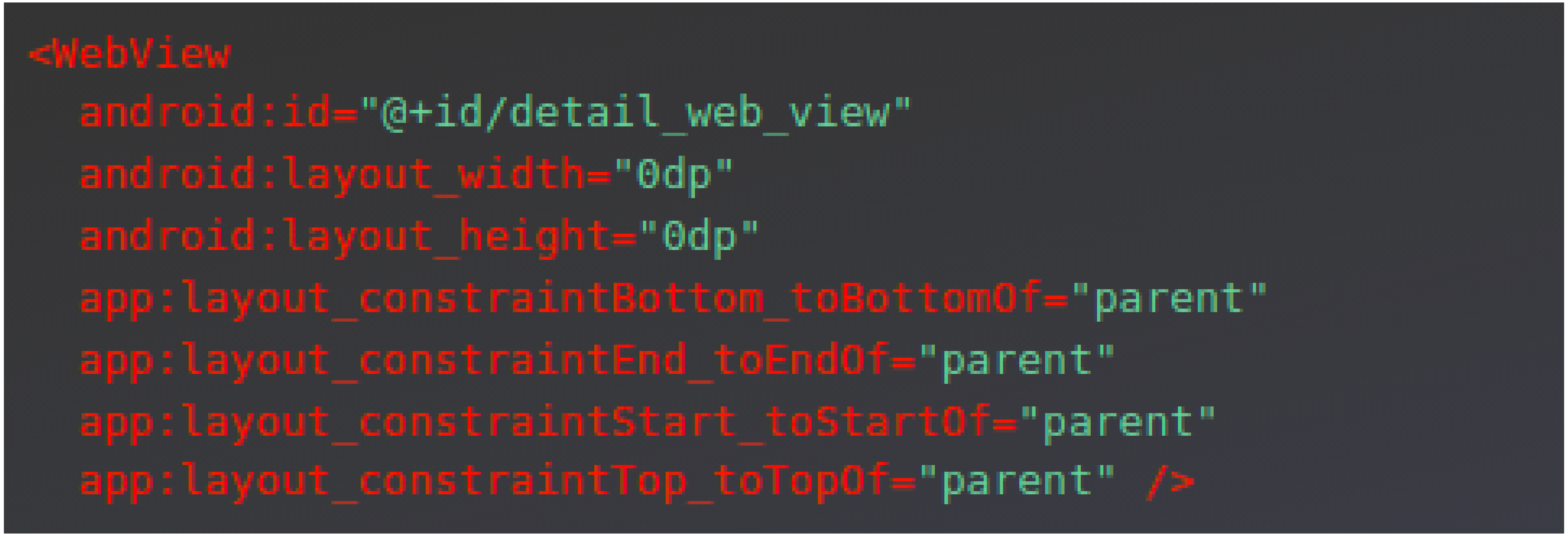
[ ] Launcher Activity

[✓] Backwards Compatibility (AppCompat)

Package name
The name of the activity class to create

Open **res/layout/activity_recipe_detail.xml** and add a WebView by inserting the following snippet inside the ConstraintLayout tag:

Open **res/layout/activity_recipe_detail.xml** and add a WebView by inserting the following snippet inside the ConstraintLayout tag:



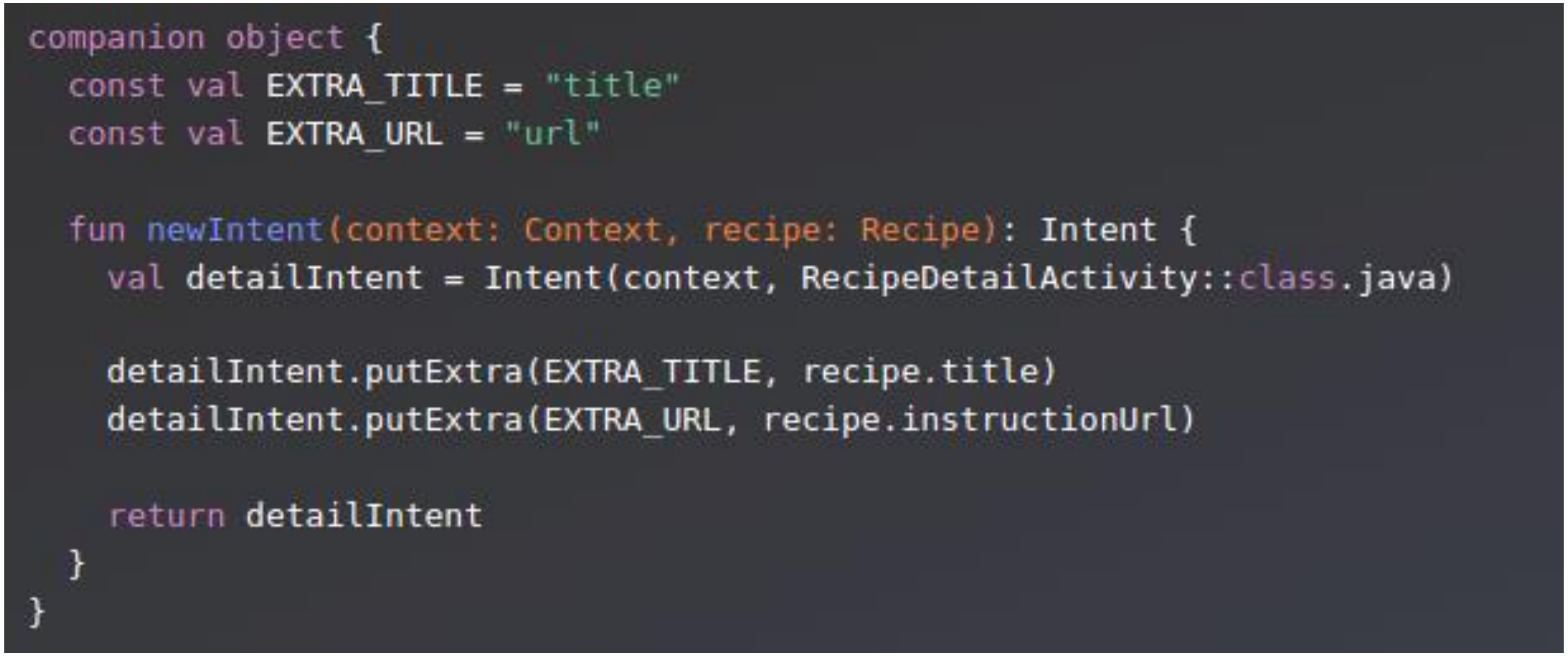
```
<WebView
  android:id="@+id/detail_web_view"
  android:layout_width="0dp"
  android:layout_height="0dp"
  app:layout_constraintBottom_toBottomOf="parent"
  app:layout_constraintEnd_toEndOf="parent"
  app:layout_constraintStart_toStartOf="parent"
  app:layout_constraintTop_toTopOf="parent" />
```

WebView will be used to load and display a webpage containing the selected recipe’s instructions.

Open **res/layout/activity_recipe_detail.xml** and add a WebView by inserting the following snippet inside the ConstraintLayout tag:

```
private lateinit var webView: WebView
```

Add the following below the **webView** property declaration:



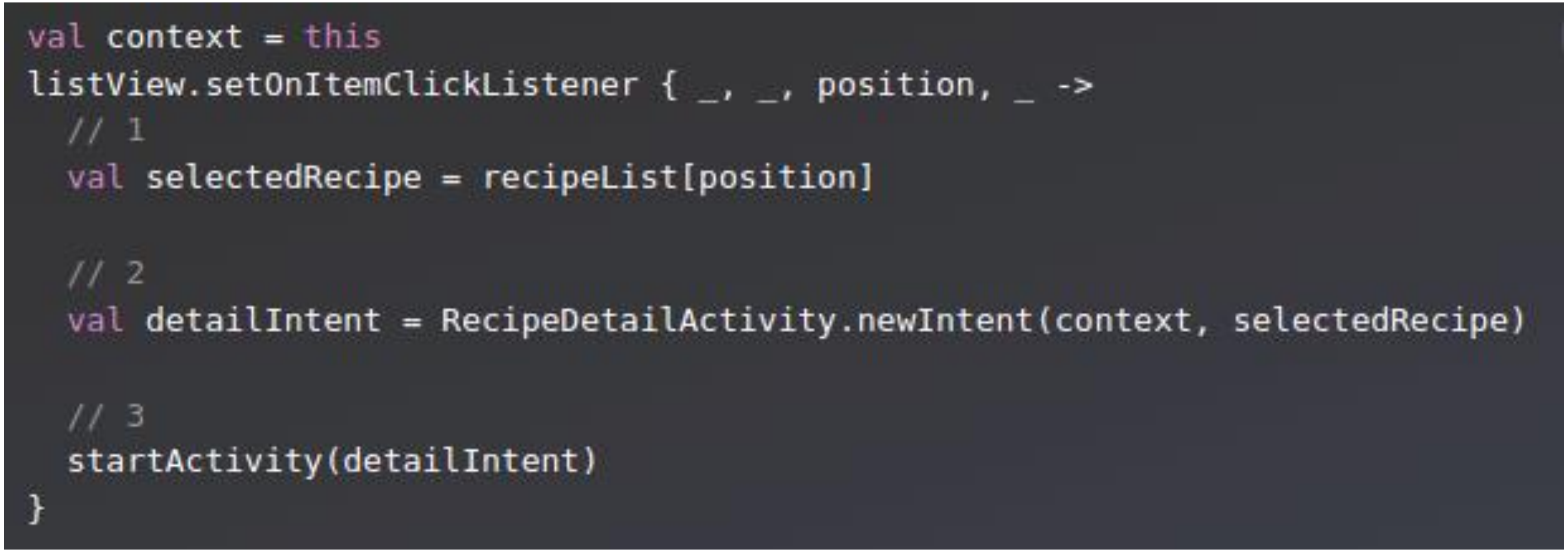
```
companion object {
  const val EXTRA_TITLE = "title"
  const val EXTRA_URL = "url"

  fun newIntent(context: Context, recipe: Recipe): Intent {
    val detailIntent = Intent(context, RecipeDetailActivity::class.java)

    detailIntent.putExtra(EXTRA_TITLE, recipe.title)
    detailIntent.putExtra(EXTRA_URL, recipe.instructionUrl)

    return detailIntent
  }
}
```

This adds a companion object method to return an Intent for starting the detail activity, and sets up title and url extras in the Intent.
Head back to **MainActivity** and add the following to the bottom of the onCreate method:

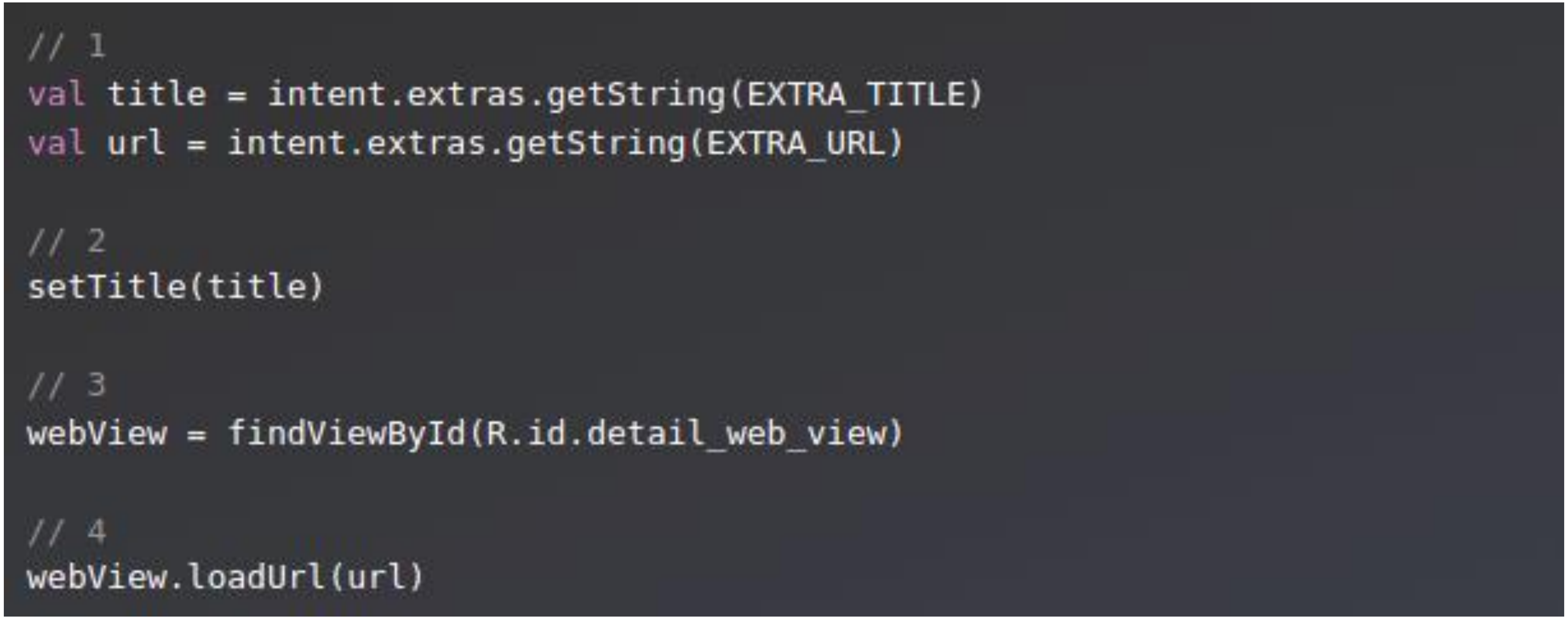


```
val context = this
listView.setOnItemClickListener { _, _, position, _ ->
  // 1
  val selectedRecipe = recipeList[position]

  // 2
  val detailIntent = RecipeDetailActivity.newIntent(context, selectedRecipe)

  // 3
  startActivity(detailIntent)
}
```

Once again, open **RecipeDetailActivity** and add the following snippet at the bottom of the onCreate method:



```
// 1
val title = intent.extras.getString(EXTRA_TITLE)
val url = intent.extras.getString(EXTRA_URL)

// 2
setTitle(title)

// 3
webView = findViewById(R.id.detail_web_view)

// 4
webView.loadUrl(url)
```

You can see a few things happening here:

1. You retrieve the recipe data from the Intent passed from MainActivity by using the extras property.
2. You set the title on the action bar of this activity to the recipe title.
3. You initialize webView to the web view defined in the XML layout.
4. You load the recipe web page by calling loadUrl() with the corresponding recipe's URL on the web view object.

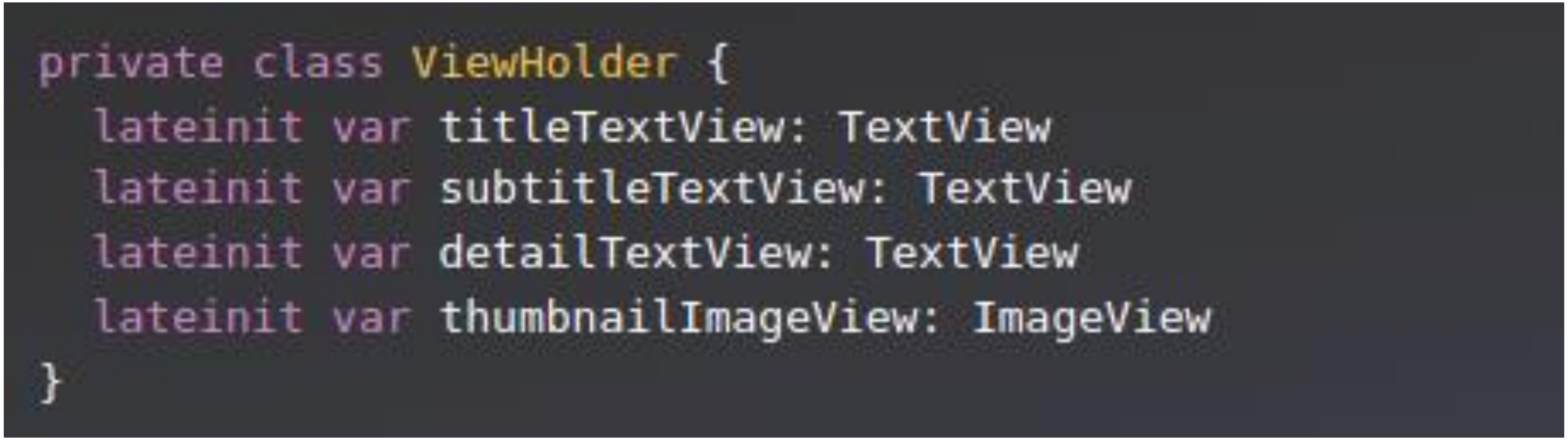
Build and run. When you click on the first item in the list, you should see something like this:

Optimizing Performance:

Whenever you scroll the ListView, its adapter’s getView() method is called in order to create a row and display it on screen.
Now, if you look in your getView() method, you’ll notice that each time this method is called, it performs a lookup for each of the row view’s elements by using a call to the findViewById() method.
These repeated calls can seriously harm the ListView’s performance, especially if your app is running on limited resources and/or you have a very large list. You can avoid this problem by using the *View Holder Pattern*.

Implement a ViewHolder Pattern

To implement the ViewHolder pattern, open **RecipeAdapter** and add the following after the getView() method definition:



```
private class ViewHolder {
  lateinit var titleTextView: TextView
  lateinit var subtitleTextView: TextView
  lateinit var detailTextView: TextView
  lateinit var thumbnailImageView: ImageView
}
```

As you can see, you create a class to hold your exact set of component views for each row view. The ViewHolder class stores each of the row's subviews, and in turn is stored inside the tag field of the layout.

Now, in getView(), replace everything above (but NOT including) this line:

```
val recipe = getItem(position) as Recipe
```

With:

```
val view: View
val holder: ViewHolder

// 1
if (convertView == null) {

  // 2
  view = inflater.inflate(R.layout.list_item_recipe, parent, false)

  // 3
  holder = ViewHolder()
  holder.thumbnailImageView = view.findViewById(R.id.recipe_list_thumbnail) as ImageView
  holder.titleTextView = view.findViewById(R.id.recipe_list_title) as TextView
  holder.subtitleTextView = view.findViewById(R.id.recipe_list_subtitle) as TextView
  holder.detailTextView = view.findViewById(R.id.recipe_list_detail) as TextView
  // 4
  view.tag = holder
} else {
  // 5
  view = convertView
  holder = convertView.tag as ViewHolder
}

// 6
val titleTextView = holder.titleTextView
val subtitleTextView = holder.subtitleTextView
val detailTextView = holder.detailTextView
val thumbnailImageView = holder.thumbnailImageView
```

Here's the play-by-play of what's happening above.

1.Check if the view already exists. If it does, there's no need to inflate from the layout and call findViewById() again.

2.If the view doesn't exist, you inflate the custom row layout from your XML.

3.Create a new ViewHolder with subviews initialized by using findViewById().

4.Hang onto this holder for future recycling by using setTag() to set the tag property of the view that the holder belongs to.

5.Skip all the expensive inflation steps and just get the holder you already made.

6.Get relevant subviews of the row view.

Finally, update the return statement of getView() with the line below.

```
return view
```

Build and run. If your app was running a bit slow on the last build, you should see it running smoother now. :]

What does LayoutInflater in Android do?

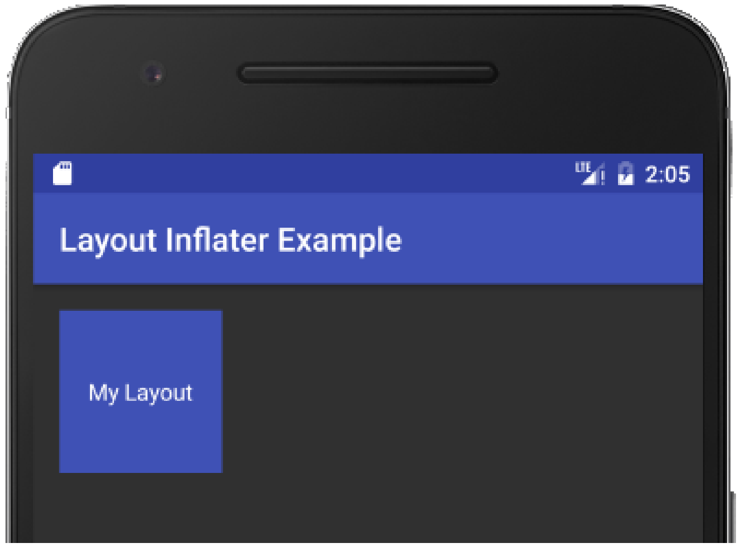
When Starting with Android new programmers mostly confused by LayoutInflater and findViewById. Sometimes we used one and sometimes the other.

- *LayoutInflater* is used to create a new *View* (or *Layout*) object from one of your xml layouts.

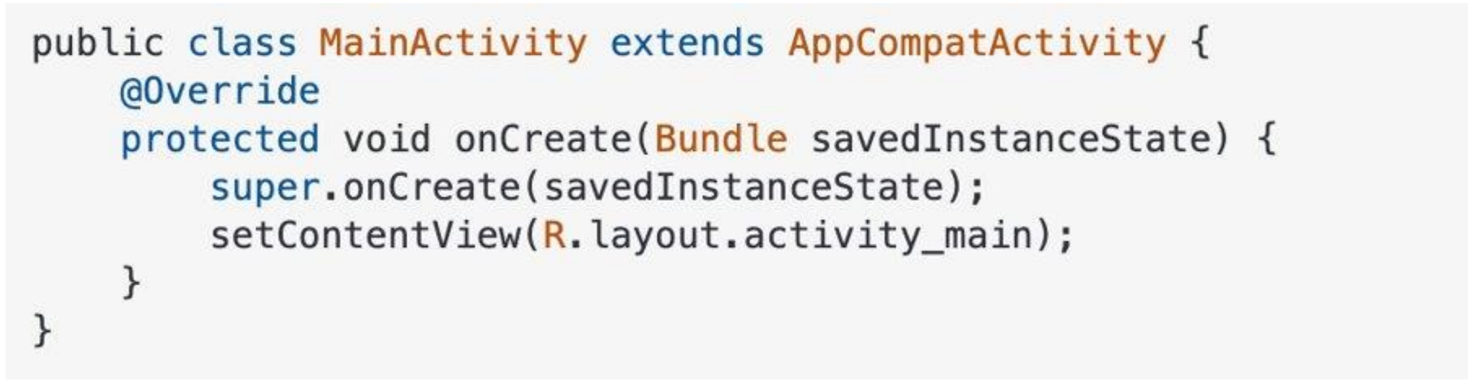
- findViewById just gives you a reference to a view than has already been created. You might think that you haven't created any views yet, but whenever you call *setContentView* in *onCreate*, the activity's layout along with its subviews gets inflated (created) behind the scenes.

So if the view already exists, then use *findViewById.* If not, then create it with a *LayoutInflater.*

Here is a mini project I made that shows both *LayoutInflater* and *findViewById* in action. With no special code, the layout looks like this.

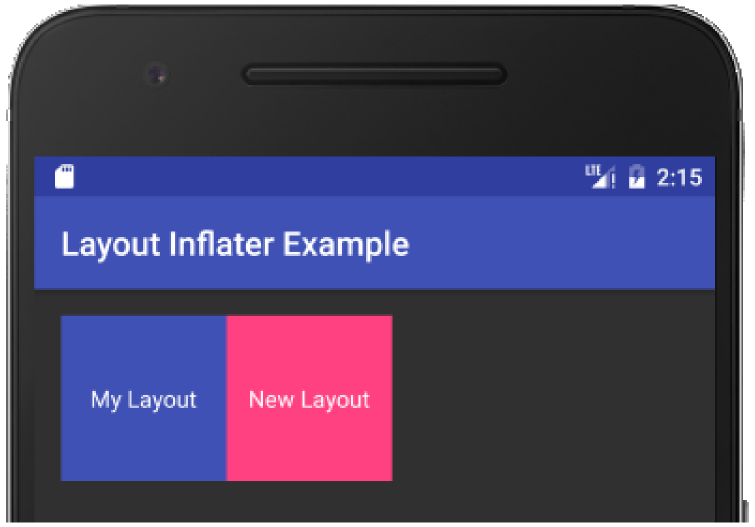


The blue square is a custom layout inserted into the main layout with *include* . It was inflated automatically because it is part of the content view. As you can see, there is nothing special about the code.



```
public class MainActivity extends AppCompatActivity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
    }
}
```

Now let's inflate (create) another copy of our custom layout and add it in.

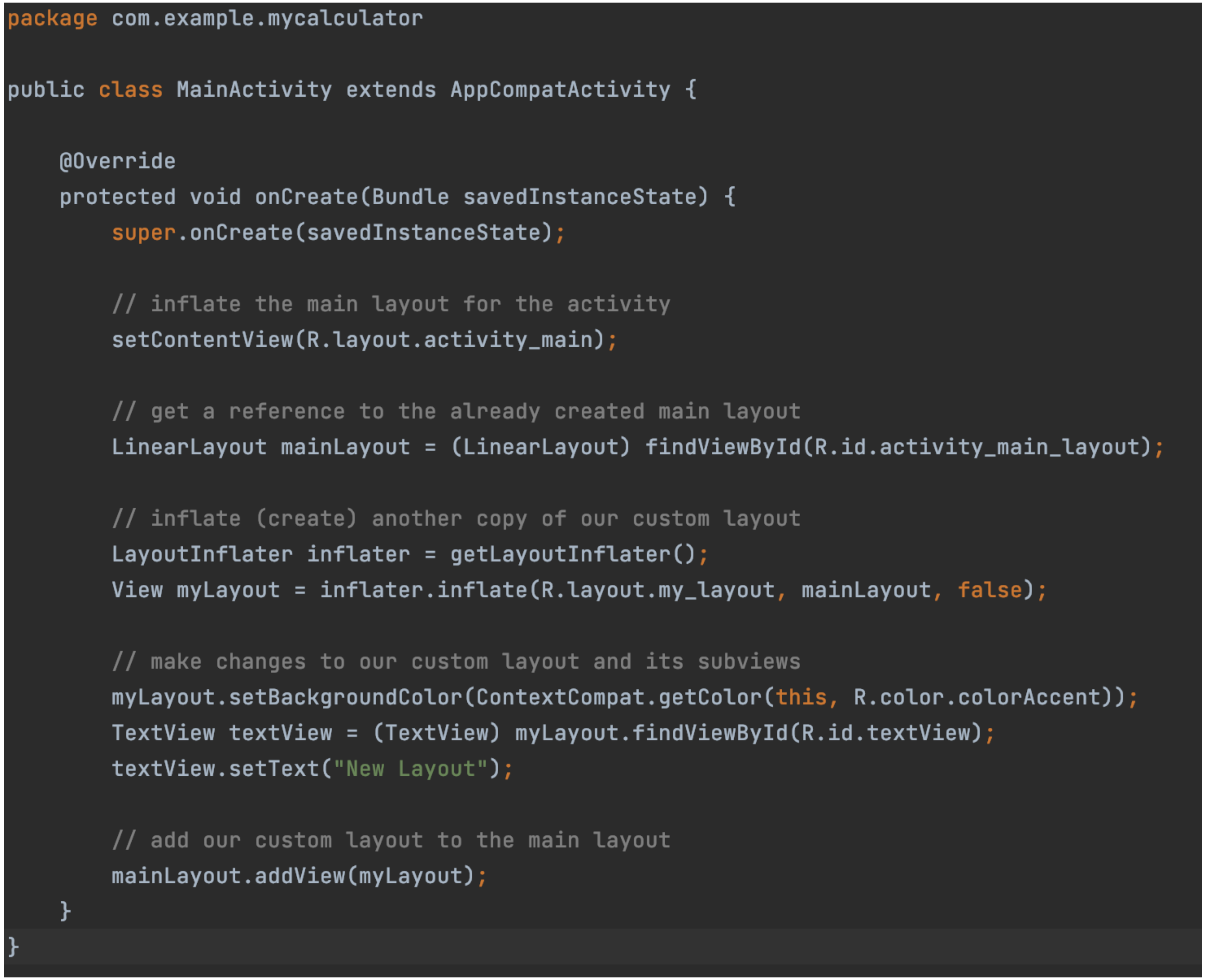


```
LayoutInflater inflater = getLayoutInflater();
View myLayout = inflater.inflate(R.layout.my_layout, mainLayout, false);
```

To inflate the new view layout, all I did was tell the inflater the name of my xml file (*my_layout*), the parent layout that I want to add it to (*mainLayout*), and that I don't want to add it yet (false). (I could also set the parent to null, but then the layout parameters of my custom layout's root view would be ignored.)

Here it is again in context.

Notice how findViewById is used only after a layout has already been inflated.



```java
package com.example.mycalculator

public class MainActivity extends AppCompatActivity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);

        // inflate the main layout for the activity
        setContentView(R.layout.activity_main);

        // get a reference to the already created main layout
        LinearLayout mainLayout = (LinearLayout) findViewById(R.id.activity_main_layout);

        // inflate (create) another copy of our custom layout
        LayoutInflater inflater = getLayoutInflater();
        View myLayout = inflater.inflate(R.layout.my_layout, mainLayout, false);

        // make changes to our custom layout and its subviews
        myLayout.setBackgroundColor(ContextCompat.getColor(this, R.color.colorAccent));
        TextView textView = (TextView) myLayout.findViewById(R.id.textView);
        textView.setText("New Layout");

        // add our custom layout to the main layout
        mainLayout.addView(myLayout);
    }
}
```

Supplemental Code

Here is the xml for the example above.

activity_main.xml

```xml
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:id="@+id/activity_main_layout"
    android:orientation="horizontal"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:padding="16dp">

    <!-- Here is the inserted layout -->
    <include layout="@layout/my_layout"/>

</LinearLayout>
```

my_layout.xml

```xml
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="100dp"
    android:layout_height="100dp"
    android:background="@color/colorPrimary">

    <TextView
        android:id="@+id/textView"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_centerInParent="true"
        android:padding="5dp"
        android:textColor="@android:color/white"
        android:text="My Layout"/>

</RelativeLayout>
```

When do you need LayoutInflater

- The most common time most people use it is in a RecyclerView. (See these RecyclerView examples for a list or a grid.) You have to inflate a new layout for every single visible item in the list or grid.

- You also can use a layout inflater if you have a complex layout that you want to add programmatically (like we did in our example). You could do it all in code, but it is much easier to define it in xml first and then just inflate it.