**Chapter 2 Layouts**

In this chapter, we will cover the following topics:

1. **Defining and inflating a layout**
2. **Using RelativeLayout**
3. **Using LinearLayout**
4. **Creating tables—TableLayout and GridLayout**
5. **RecyclerView replaces ListView**
6. Changing layout properties during runtime

**Introduction**

In Android, the user interface is defined in a layout. A layout can be declared in XML or created dynamically in code. (It's recommended to declare the layout in XML rather than in code to keep the presentation layer separate from the implementation layer.)

A layout can define an individual ListItem, a fragment, or even the entire activity. Layout files are stored in the /res/layout folder and referenced in code with the following identifier: R.layout.<filename\_without\_extension>.

Android provides a useful variety of Layout classes that contain and organize individual elements of an activity (such as buttons, checkboxes, and other Views). The ViewGroup object is a container object that serves as the base class for Android's family of Layout classes. The Views placed in a layout form a hierarchy, with the topmost layout being the parent.

Android provides several built-in layout types designed for specific purposes, such as RelativeLayout, which allows Views to be positioned with respect to other elements. The LinearLayout can stack Views or align them horizontally, depending on the orientation specified. The TableLayout can be used for laying out a grid of Views.

Within various layouts, we can also justify Views with Gravity and provide proportional size with Weight control. Layouts and ViewGroups can be nested within each other to create complex configurations. Over a dozen different Layout objects are provided for managing widgets, lists, tables, galleries, and other display formats, plus you can always derive from base classes to create your own custom layouts.

Google has released a new layout called ConstraintLayout. This layout is similar to ReleativeLayout in that Views are positioned relative to each other and to the parent, as well as a new element called guidelines.

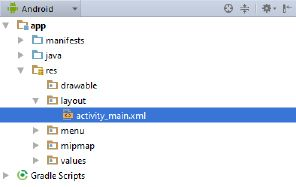
The focus of the layout is to keep the layout itself as flat as possible (deeply nested layouts can cause performance issues) and for a visual layout editor. Giving the best visual editing experience while keeping the editor in sync with the underlying class is such a priority for Google, that the same team develops both.

ConstraintLayout is now the default layout created when using the Android Studio and is the basis for most of the examples in this book. (The other layouts are still available and are used when their layout provides the cleanest XML.) Here's the link to the ConstraintLayout class, but for the best experience, it's recommended to use the visual editor in Android Studio:

<https://developer.android.com/reference/android/support/constraint/ConstraintLayout>

**1 Defining and inflating a layout**

When using the Android Studio wizard to create a new project, it automatically creates the res/layout/activity\_main.xml file (as shown in the following screenshot). It then inflates the XML file in the onCreate() callback with setContentView (R.layout.activity\_main):



For this recipe, we will create two slightly different layouts and switch between them with a button.

**Getting ready**

Create a new project in Android Studio and call it InflateLayout. Once the project is created, expand the res/layout folder so we can edit the activity\_main.xml file. Use the default Phone & Tablet settings on the Target Android devices and select Empty Activity on the Add an Activity to Mobile dialog.

**How to do it…**

1. Edit the **res/layout/activity\_main.xml** file so it includes a button as defined here:

<Button

android:id="@+id/buttonLeft"

android:layout\_width="wrap\_content"

android:layout\_height="wrap\_content"

android:layout\_alignParentLeft="true"

android:layout\_centerVertical="true"

android:onClick="onClickLeft"

android:text="Left Button"

app:layout\_constraintBottom\_toBottomOf="parent"

app:layout\_constraintHorizontal\_bias="0.0"

app:layout\_constraintLeft\_toLeftOf="parent"

app:layout\_constraintRight\_toRightOf="parent"

app:layout\_constraintTop\_toTopOf="parent"/>

2. Now make a copy of **activity\_main.xml** and call it **activity\_main2.xml**. Change the button so it matches the following:

<Button

android:id="@+id/buttonLeft"

android:layout\_width="wrap\_content"

android:layout\_height="wrap\_content"

android:layout\_alignParentLeft="true"

android:layout\_centerVertical="true"

android:onClick="onClickRight"

android:text="Right Button"

app:layout\_constraintBottom\_toBottomOf="parent"

app:layout\_constraintHorizontal\_bias="1.0"

app:layout\_constraintLeft\_toLeftOf="parent"

app:layout\_constraintRight\_toRightOf="parent"

app:layout\_constraintTop\_toTopOf="parent"/>

3. Open **MainActivity.java** and add the following **two methods** to handle the button clicks:

public void **onClickLeft**(View view) {

setContentView(R.layout.activity\_main2);

}

public void **onClickRight**(View view) {

setContentView(R.layout.activity\_main);

}

4. Run this application on a device or emulator to see it in action.

**How it works...**

The key here is the call to setContentView(), which we have come across before in the autogenerated onCreate() code. Just pass a layout ID to setContentView() and it automatically inflates the layout.

This code is meant to make the concept easy to understand but would be overkill for simply changing the property of a button (in this example, we could just change the alignment on the button click). Inflating the layout is usually needed once, in the onCreate() method, but there are times when you may want to manually inflate a layout, as we did here. (If you were manually handling orientation changes, it would be a good example.)

**There’s more…**

As well as identifying a layout using a resource ID, as we did here, setContentView() can also take a View as an argument, for example:

findViewById(R.id.myView)

setContentView(myView);

**See also**

As mentioned previously, read about Fragments in Chapter 5, Fragments, for creating reusable screen components for your activities.

**2 Using RelativeLayout**

As mentioned in the Introduction section, RelativeLayout allows Views to be position-relative to each other and the parent.

RelativeLayout is particularly useful for reducing the number of nested layouts, which is very important for reducing memory and processing requirements.

**Getting ready**

Create a new project and call it RelativeLayout. Android Studio defaults to using a ConstraintLayout, which we will replace with a RelativeLayout for this example. Use the default Phone & Tablet settings on the Target Android devices and select Empty Activity on the Add an Activity to Mobile dialog.

**How to do it...**

1. Open the **res/layout/activity\_main.xml** file and change it as follows:

<?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/textView1"

android:layout\_width="wrap\_content"

android:layout\_height="wrap\_content"

android:text="Centered"

android:layout\_centerVertical="true"

android:layout\_centerHorizontal="true" />

<TextView

android:id="@+id/textView2"

android:layout\_width="wrap\_content"

android:layout\_height="wrap\_content"

android:text="Below Left"

android:layout\_below="@+id/textView1"

android:layout\_toLeftOf="@id/textView1" />

<TextView

android:id="@+id/textView3"

android:layout\_width="wrap\_content"

android:layout\_height="wrap\_content"

android:text="Bottom Right"

android:layout\_alignParentBottom="true"

android:layout\_alignParentRight="true" />

</RelativeLayout>

2. **Run the code, or view the layout in the Design** tab

**How it works...**

This is a very straightforward exercise but it demonstrates several of the RelativeLayout options:

layout\_centerVertical, layout\_centerHorizontal, layout\_below, layout\_alignParentBottom, and so on.

The most commonly used RelativeLayout layout attributes include the following:

* layout\_below: This View should be below the View specified.
* layout\_above: This View should be above the View specified.
* layout\_alignParentTop: Align this View to the top edge of the parent.
* layout\_alignParentBottom: Align this View to the bottom edge of the parent.
* layout\_alignParentLeft: Align this View to the left edge of the parent.
* layout\_alignParentRight: Align this View to the right edge of the parent.
* layout\_centerVertical: Center this View vertically within the parent.
* layout\_centerHorizontal: Center this View horizontally within the parent.
* layout\_center: Center this View both horizontally and vertically within the parent.

For the complete list of RelativeLayout parameters, visit

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

**There's more...**

In contrast to what we saw earlier, **here is an example using LinearLayout just to center TextView** (creating the same effect as the layout\_center parameter of RelativeLayout):

<?xml version="1.0" encoding="utf-8"?>

<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"

android:orientation="horizontal"

android:layout\_width="match\_parent"

android:layout\_height="match\_parent"

android:gravity="center">

<LinearLayout

android:layout\_width="0dp"

android:layout\_height="wrap\_content"

android:layout\_weight="1"

android:gravity="center" >

<TextView

android:id="@+id/imageButton\_speak"

android:layout\_width="wrap\_content"

android:layout\_height="wrap\_content"

android:text="Centered" />

</LinearLayout>

</LinearLayout>

Notice **this layout is one level deeper than the equivalent RelativeLayout** (which is LinearLayout nested within the parent LinearLayout.) Though a simple example, it's a good idea **to avoid unnecessary nesting as it can impact performance**, especially when a layout **is being repeatedly inflated (such as ListItem).**

**See also**

The next recipe, Using LinearLayout, will give you an alternative layout.

See the Optimizing layouts with the **Hierarchy Viewer recipe** for more information on efficient layout design.

**3 Using LinearLayout**

Another common layout option is LinearLayout, which arranges the child Views in a single column or single row, depending on the orientation specified. The default orientation (if not specified) is vertical, which aligns the Views in a single column.

LinearLayout has a key feature not offered in RelativeLayout—the weight attribute. We can specify a layout\_weight parameter when defining a View to allow the View to dynamically size based on the available space.

Options include having a View fill all the remaining space (if a View has a higher weight), having multiple Views fit within the given space (if all have the same weight), or spacing the Views proportionally by their weight.

We will create LinearLayout with three EditText Views to demonstrate how the weight attribute can be used.

For this example, we will use three EditText Views-one to enter a To Address parameter, another to enter Subject, and the third to enter Message. The To and Subject Views will be a single line each, with the remaining space given to the Message View.

**Getting ready**

Create a new project and call it **LinearLayout**. We will replace the default RelativeLayout created in activity\_main.xml with LinearLayout. Use the default Phone & Tablet settings on the Target Android devices and select Empty Activity on the Add an Activity to Mobile dialog.

**How to do it...**

1. Open the **res/layout/activity\_main.xml** file and replace it as follows:

<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"

android:orientation="vertical"

android:layout\_width="match\_parent"

android:layout\_height="match\_parent">

<EditText

android:id="@+id/editTextTo"

android:layout\_width="match\_parent"

android:layout\_height="wrap\_content"

android:hint="To" />

<EditText

android:id="@+id/editTextSubject"

android:layout\_width="match\_parent"

android:layout\_height="wrap\_content"

android:hint="Subject" />

<EditText

android:id="@+id/editTextMessage"

android:layout\_width="match\_parent"

android:layout\_height="0dp"

android:layout\_weight="1"

android:gravity="top"

android:hint="Message" />

</LinearLayout>

2. **Run the code, or view the layout in the Design** tab.

**How it works…**

When using vertical orientation with LinearLayout, the child Views are created in a single column (stacked on top of each other). The first two Views use the android:layout\_height="wrap\_content" attribute, giving them a single line each. To specify the height, editTextMessage uses the following:

**android:layout\_height="0dp"**

**android:layout\_weight="1"**

When using LinearLayout, it tells Android to calculate the height based on the weight. A weight of 0 (the default if not specified) indicates the View should not expand. In this example, editTextMessage is the only View defined with a weight, so it alone will expand to fill any remaining space in the parent layout.

When using the horizontal orientation, specify android:layout\_height="0dp" (along with the weight) to have Android calculate the width.

It might be helpful to think of the weight attribute as a percentage. In this case, the total weight defined is 1, so this View gets 100 percent of the remaining space. If we assigned a weight of 1 to another View, the total would be 2, so this View would get 50 percent of the space.

Try adding a weight to one of the other Views (make sure to change the height to 0dp as well) to see it in action.

If you added a weight to one (or both) of the other Views, did you notice the text position? Without specifying a value for gravity, the text just remains in the center of the View space.

The editTextMessage View specifies android:gravity="top", which forces the text to the top of the View.

**There’s more…**

Multiple attribute options can be combined using bitwise OR. (Java uses the pipe character (|) for OR).

For example, we could combine two gravity options to both align along the top of the parent and center within the available space:

**android:layout\_gravity="top|center"**

It should be noted that the layout\_gravity and gravity tags are not the same thing. Where layout\_gravity dictates where in its parent a View should lie, gravity controls the positioning of the contents within a View, for example, the alignment of text on a button.

**See also**

The previous recipe, Using RelativeLayout.

RECONVERTIDO Y ENVIAR CORREOS CHECADO

**4 Creating tables – TableLayout and GridLayout**

When you need to create a table in your UI, Android provides two convenient layout options:

TableLayout (along with TableRow) and GridLayout (added in API 14). Both layout options can create similar looking tables, but each using a different approach. With TableLayout, rows and columns are added dynamically as you build the table. With GridLayout, row and column sizes are defined in the layout definition.

Neither layout is better, it's just a matter of using the best layout for your needs. We'll create a 3 x 3 grid using each layout to give a comparison, as you could easily find yourself using both layouts, even within the same application.

**Getting ready**

To stay focused on the layouts and offer an easier comparison, we will create two separate applications for this recipe. Create two new Android projects, **the first called TableLayout and the other called GridLayout**. Use the default Phone & Tablet settings on the Target Android devices and select Empty Activity on the Add an Activity to Mobile dialog.

**How to do it...**

1. Starting with the TableLayout project, open **activity\_main.xml**. Change the root layout to TableLayout.

2. Add **three TableRow objects with three sets of TextView objects to each TableRow to create a 3 x 3 matrix**.

For demonstration purposes, the columns are labeled A-C and the rows 1-3, **so the first row of TextView objects will be A1, B1, and C1**. The final result will look like this:

<TableLayout

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">

<TableRow

android:layout\_width="match\_parent"

android:layout\_height="match\_parent">

<TextView

android:layout\_width="wrap\_content"

android:layout\_height="wrap\_content"

android:text="A1"

android:id="@+id/textView1" />

<TextView

android:layout\_width="wrap\_content"

android:layout\_height="wrap\_content"

android:text="B1"

android:id="@+id/textView2" />

<TextView

android:layout\_width="wrap\_content"

android:layout\_height="wrap\_content"

android:text="C1"

android:id="@+id/textView3" />

</TableRow>

<TableRow

android:layout\_width="match\_parent"

android:layout\_height="match\_parent">

<TextView

android:layout\_width="wrap\_content"

android:layout\_height="wrap\_content"

android:text="A2"

android:id="@+id/textView4" />

<TextView

android:layout\_width="wrap\_content"

android:layout\_height="wrap\_content"

android:text="B2"

android:id="@+id/textView5" />

<TextView

android:layout\_width="wrap\_content"

android:layout\_height="wrap\_content"

android:text="C2"

android:id="@+id/textView6" />

</TableRow>

<TableRow

android:layout\_width="match\_parent"

android:layout\_height="match\_parent">

<TextView

android:layout\_width="wrap\_content"

android:layout\_height="wrap\_content"

android:text="A3"

android:id="@+id/textView7" />

<TextView

android:layout\_width="wrap\_content"

android:layout\_height="wrap\_content"

android:text="B3"

android:id="@+id/textView8" />

<TextView

android:layout\_width="wrap\_content"

android:layout\_height="wrap\_content"

android:text="C3"

android:id="@+id/textView9" />

</TableRow>

</TableLayout>

3. Now, open the **GridLayout** project to edit **activity\_main.xml**. Change the root layout to GridLayout. Add the columnCount=3 and rowCount=3 attributes to the GridLayout element.

4. Now, **add nine TextView objects to GridLayout**. We will use the same text as the preceding TableLayout for a consistent comparison. Since GridView does not use TableRow objects, **the first three TextView objects are in row 1, the next three are in row 2, and so on. The final result will look like this**:

<GridLayout

xmlns:android="http://schemas.android.com/apk/res/android"

android:layout\_width="match\_parent"

android:layout\_height="match\_parent"

android:columnCount="3"

android:rowCount="3">

<TextView

android:layout\_width="wrap\_content"

android:layout\_height="wrap\_content"

android:text="A1"

android:id="@+id/textView1" />

<TextView

android:layout\_width="wrap\_content"

android:layout\_height="wrap\_content"

android:text="B1"

android:id="@+id/textView2" />

<TextView

android:layout\_width="wrap\_content"

android:layout\_height="wrap\_content"

android:text="C1"

android:id="@+id/textView3" />

<TextView

android:layout\_width="wrap\_content"

android:layout\_height="wrap\_content"

android:text="A2"

android:id="@+id/textView4" />

<TextView

android:layout\_width="wrap\_content"

android:layout\_height="wrap\_content"

android:text="B2"

android:id="@+id/textView5" />

<TextView

android:layout\_width="wrap\_content"

android:layout\_height="wrap\_content"

android:text="C2"

android:id="@+id/textView6" />

<TextView

android:layout\_width="wrap\_content"

android:layout\_height="wrap\_content"

android:text="A3"

android:id="@+id/textView7" />

<TextView

android:layout\_width="wrap\_content"

android:layout\_height="wrap\_content"

android:text="B3"

android:id="@+id/textView8" />

<TextView

android:layout\_width="wrap\_content"

android:layout\_height="wrap\_content"

android:text="C3"

android:id="@+id/textView9" />

</GridLayout>

5. You can either **run the application** or use the Design tab to see the results.

**How it works...**

As you can see when viewing the tables created, the tables basically look the same on screen. The main difference is the code to create them.

In the TableLayout XML, each row is added to the table using TableRow. Each View becomes a column. This is not a requirement as cells can be skipped or left empty. (See how to specify the cell location in TableRow in the following section.)

GridLayout uses the opposite approach. The number of rows and columns are specified when creating the table. We don't have to specify the row or column information (though we can, as discussed later). Android will automatically add each View to the cells in order.

**There's more...**

First, let's **see more similarities** between the layouts. Both layouts **have the ability to stretch columns to use the remaining screen space**. For TableLayout, add the following attribute to the XML declaration:

**android:stretchColumns="1"**

The stretchColumns attribute specifies the (zero-based) index of the columns to stretch (android:shrinkColumns is a zero-based index of columns that can shrink, so the table can fit the screen).

To achieve the same effect with GridLayout, add the following attribute to all the Views in the B column (textView2, textView5, and textView8):

**android:layout\_columnWeight="1"**

All cells in a given column must define the weight or it will not stretch.

Now, let's look at **some of the differences**, as this is really the key to determining which layout to use for a given task. The first item to note is how the columns and rows are actually defined. In TableLayout, **the rows are specifically defined**, using TableRow. (Android will **determine the number of columns in the table based on the row with the most cells**.) Use the android:layoutColumn attribute when defining the View to specify the column.

In contrast, with GridLayout, **the row and column counts are specified when defining the table** (using columnCount and rowCount as shown previously).

In the preceding example, we just added TextView objects to GridLayout **and let the system position them automatically**. We can alter this behavior **by specifying the row and column position when defining the View**, such as the following:

**android:layout\_row="2"**

**android:layout\_column="2"**

Android automatically increments the cell counter after adding each View, so the next View should also specify the row and column, otherwise, you may not get the intended result.

Like LinearLayout, shown in the Using LinearLayout recipe, GridLayout also offers the orientation attribute of supporting both horizontal (the default) and vertical. The orientation determines how the cells are placed. (Horizontal fills the columns first, then moves down to the next row. Vertical fills the first column on each row, then moves to the next column.)

**5 RecyclerView replaces ListView**

As the name implies, ListView is designed for displaying lists of information. If you have prior experience on Android, you’ve probably come across the ListView and possibly GridView controls before.

If not while coding, most likely you’ve used it as an app, as it’s one of the most commonly used controls available. For most applications and users, the old ListView was probably sufficient and didn’t pose any problems.

As an example, most users could probably see their list of emails in their inbox without any problems. But for some, they might have so many emails in their inbox that when scrolling through their list, their device would stutter (slight pauses when scrolling).

Unfortunately, ListView has many such performance problems. The most significant performance issue with ListView is caused by creating new item objects for each item when scrolling.

Though much of the performance problem could be eliminated with a properly implemented data adapter, the implementation was optional. As the name implies, RecyclerView is based on recycling the list items (the part that was optional in the ListView adapter).

There are other changes to the control as well. Whereas ListView has many features built-in, RecyclerView is very basic and relies on additional helper classes to achieve the same functionality. For some, this feels like a step backward with the new control but this design allows it to be expanded much easier.

Where RecylerView really shines is with the flexibility when extending it and animations. Our example here uses a static list so it doesn’t show off the built-in animations, but with dynamic data, your list will take advantage of the Material Design look and feel.

Though ListView is not officially deprecated, it is recommended to use RecyclerView for new projects. It’s a bit more work to get started, but this recipe will give you all the code to get set up.

**Getting ready**

Create a new project in Android Studio called **RecyclerView**. Use the default Phone & Tablet settings on the Target Android devices and select Empty Activity on the Add an Activity to Mobile dialog.

**How to do it…**

Creating RecyclerView is as simple as **placing the control on the screen.** Most of the work is **with the adapter**, **which we'll create from a static list**. RecyclerView is distributed in a separate library so it **needs to be added to the project** as a dependency. The steps are as follows:

1. Either **add the dependency through the Android Studio** UI or add the following code to the dependencies section of the **build.gradle (Module: app) file**:

**implementation 'com.android.support:recyclerview-v7:27.1'**

NOTE: v7:27.1 is **current at the time of this writing**, but should be updated to the latest version. (The IDE will likely give you a warning if you're not using the latest version.)

2. Open **activity\_main.xml** and replace the existing <TextView /> block with the **following RecyclerView widget**:

<android.support.v7.widget.RecyclerView

android:id="@+id/recyclerView"

android:layout\_width="match\_parent"

android:layout\_height="match\_parent"

app:layout\_constraintBottom\_toBottomOf="parent"

app:layout\_constraintLeft\_toLeftOf="parent"

app:layout\_constraintRight\_toRightOf="parent"

app:layout\_constraintTop\_toTopOf="parent" />

3. We **need another layout for the adapter to create the individual items in the list**. To do this, create a new file in the res\layout folder called **item.xml** as follows:

<?xml version="1.0" encoding="utf-8"?>

<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"

android:layout\_width="match\_parent"

android:layout\_height="wrap\_content"

android:orientation="vertical">

<TextView

android:id="@+id/textView"

android:layout\_width="match\_parent"

android:layout\_height="wrap\_content"

android:text="TextView" />

</LinearLayout>

4. Now comes the heart of **RecyclerView – the adapter**. Create a **new Java file called MyAdapter.java**. Our new class **will extend from the RecylerView.Adapter class** so there are several key methods we need to **override**. We’ll discuss the details of this class later, but the full code is as follows:

public class MyAdapter extends RecyclerView.Adapter<MyAdapter.MyViewHolder> {

private List<String> nameList;

public MyAdapter(List<String> list) {

nameList = list;

}

@Override

public MyViewHolder onCreateViewHolder(ViewGroup parent, int viewType) {

LayoutInflater inflater = LayoutInflater.from(parent.getContext());

View view = inflater.inflate(R.layout.item, parent, false);

MyViewHolder myViewHolder = new MyViewHolder(view);

return myViewHolder;

}

@Override

public void onBindViewHolder(@NonNull MyViewHolder holder, final int position) {

final String name = nameList.get(position);

holder.textView.setText(name);

}

@Override

public int getItemCount() {

if (nameList==null) {

return 0;

} else {

return nameList.size();

}

}

public class MyViewHolder extends RecyclerView.ViewHolder {

public TextView textView;

public MyViewHolder(View itemVieww) {

super(itemVieww);

textView = itemView.findViewById(R.id.textView);

}

}

}

5. With all the pieces set up, the final step is to put it all together. Open **the MainActivity.java** file and **add the following code to the existing onCreate() method**:

List<String> list = new ArrayList<>();

list.add("China");

list.add("France");

list.add("Germany");

list.add("India");

list.add("Russia");

list.add("United Kingdom");

list.add("United States");

RecyclerView recyclerView = findViewById(R.id.recyclerView);

recyclerView.setHasFixedSize(true);

LinearLayoutManager linearLayoutManager = new LinearLayoutManager(this);

linearLayoutManager.setOrientation(LinearLayoutManager.VERTICAL);

recyclerView.setLayoutManager(linearLayoutManager);

MyAdapter myAdapter = new MyAdapter(list);

recyclerView.setAdapter(myAdapter);

**How it works…**

We’ve purposely kept this recipe basic, but as you can see, there are **still many steps even for this basic implementation**. The good news is, with this foundation set, **you can easily expand and modify RecyclerView as needed.**

Want your list to **scroll sideways** instead? You can easily accomplish this by using **LinearLayoutManager.HORIZONTAL** in the setOrientation() call.

If you have ever worked with Android ListView before, then the preceding steps will look very familiar.

The concept is the same: **we create an adapter to hold a list of items**. Steps 1 and 2 **set up RecyclerView on the activity**. In step 3, **we specify the visual layout and pass it to the adapter**. In step 4, **we created the adapter by extending the RecycerView.Adapter** class.

As you can see from the code, there are **three methods we need to override**: onCreateViewHolder(), onBindViewHolder(), and getItemCount(). The key concept behind RecylerView **is to recycle or reuse the item Views**.

This means, when you have a very large list of items, **instead of creating a new view object for each item** (which is very costly in terms of performance and memory usage), **the item Views are reused**.

So as a **user scrolls through a long list, as a view goes off the screen, it's reused for the next item being shown**. Even if we added all the countries in the world to our list, there wouldn't be enough items to see the performance difference, **but when you’re working with a list of thousands of items, especially if those items include images**, the performance when scrolling will be noticeable.

Now that you understand the concept behind RecyclerView, hopefully the methods we need to override are self-explanatory. **The adapter only calls onCreateViewHolder() to create enough items to show on the screen (plus a few extra for scrolling),** whereas **onBindViewHolder() is called for each item as it’s displayed**.

**There’s more…**

If you ran the code, then you saw it’s a very simple app. In fact, it doesn’t do anything more **than just display the list in a scrollable container**.

Most apps will require some interaction with the list so how do we respond to click events? **Unlike the older ListView, RecyclerView does not have any click events built-in.**

It's up to you, the programmer, **to create the events you need**. (For basic items like in our example, this may seem like more work for the programmer, but when you get to complex list items with buttons and other interactive controls, ListView would often get in your way and you’d need to implement custom events anyway.)

**To respond to item clicks, add the following code to the MyAdapter** class:

private void remove(int position) {

nameList.remove(position);

notifyItemRemoved(position);

}

Then **add the following code to the onBindViewHolder()** method created in step 4:

holder.itemView.setOnClickListener(new View.OnClickListener() {

@Override

public void onClick(View v) {

remove(position);

}

});

**//FALLA LA ACTUALIZACION DE LA LISTA**

Usemos mejor **notifyDataSetChanged()**

//FUNCIONA LA RETIRADA VISUAL DEL ELEMENTO

Now, when you run the code, **the app will respond to the click event by removing the clicked item**.

You may also notice **the smooth animation when removing the item**.

By calling the **notifyItemRemoved() and notifyItemInserted()** methods of RecyclerView, we can take advantage of the widget’s built-in Material Design animations.

**6 Changing layout properties during runtime**

In Android development, it's generally the preferred practice to define the UI with XML and the application code in Java, **keeping the user interface code separate from the application code**.

There are times where it is **much easier or more efficient, to alter (or even build) the UI from the Java code**. Fortunately, this is easily supported in Android.

In this recipe, **we will obtain a reference to the LayoutParams** object to change the margin during runtime.

**Getting ready**

Here, we will set up a simple layout with XML and use a **LinearLayout.LayoutParams object to change the margins of a View during runtime**. Create a new project using with an Empty Activity called **RuntimeProperties**. Use the default Phone & Tablet settings on the Target Android devices and select **Empty Activity** on the Add an Activity to Mobile dialog.

**How to do it...**

We **can create or manipulate any of the standard layouts or controls through code**. For this example, we will work with **LinearLayout**:

1. Open the **activity\_main.xml** file and change the layout as follows:

<?xml version="1.0" encoding="utf-8"?>

<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"

android:layout\_width="match\_parent"

android:layout\_height="match\_parent" >

</LinearLayout>

2. **Add TextView with an ID value of textView**, as follows:

<TextView

android:id="@+id/textView"

android:layout\_width="wrap\_content"

android:layout\_height="wrap\_content"

android:text="TextView" />

3. **Add Button with an ID value of button**, as follows:

<Button

android:id="@+id/button"

android:layout\_width="wrap\_content"

android:layout\_height="wrap\_content"

android:text="Button" />

4. Open **MainActivity.java and add the following code to the onCreate() method to respond to the button click**:

Button button = (Button)findViewById(R.id.button);

button.setOnClickListener(new View.OnClickListener() {

@Override

public void onClick(View view) {

((TextView)findViewById(

R.id.textView)).setText("Changed at runtime!");

LinearLayout.LayoutParams params = (LinearLayout.

LayoutParams)view.getLayoutParams();

params.leftMargin += 5;

}

});

5. **Run the program** on a device or emulator.

**How it works...**

Every View (and therefore ViewGroup) **has a set of layout parameters associated with it**. In particular, all Views have parameters **to inform their parent of their desired height and width**.

These are defined with the **layout\_height and layout\_width parameters**. We can **access this layout information from the code with the getLayoutParams() method**. The layout information includes the layout **height, width, margins**, and any class-specific parameters.

In this example, **we moved the button on each click by obtaining the current button LayoutParams and increasing the margin**.