

16

Adapters and Recyclers

We will achieve much in this brief chapter. We will first go through the theory of adapters and lists. We will then look at how we can use a `RecyclerViewAdapter` instance in Kotlin code and add a `RecyclerView` widget to the layout, which acts as a list for our UI, and then, through the apparent magic of the Android API, bind them together so that the `RecyclerView` instance displays the contents of the `RecyclerViewAdapter` instance and allows the user to scroll through the contents of an `ArrayList` instance full of `Note` instances. You have probably guessed that we will be using this technique to display our list of notes in the Note to self app.

In this chapter, we will do the following:

- Explore another type of Kotlin class – the **inner class**
- Look at the theory of adapters and examine binding them to our UI
- Implement the layout with `RecyclerView`
- Lay out a list item for use in `RecyclerView`
- Implement the adapter with `RecyclerViewAdapter`
- Bind the adapter to `RecyclerView`
- Store notes in `ArrayList` and display them in `RecyclerView` via `RecyclerViewAdapter`

Soon, we will have a self-managing layout that holds and displays all our notes, so let's get started.

Inner classes

In this project, we will use a type of class we have not seen yet – an **inner** class. Suppose that we have a regular class called `SomeRegularClass`, with a property called `someRegularProperty`, and a function called `someRegularFunction`, as shown in this next code:

```
class SomeRegularClass{
    var someRegularProperty = 1

    fun someRegularFunction(){
    }
}
```

An inner class is a class that is declared inside of a regular class, like in this next highlighted code:

```
class SomeRegularClass{
    var someRegularProperty = 1

    fun someRegularFunction(){
    }

    inner class MyInnerClass {
        val myInnerProperty = 1

        fun myInnerFunction() {
        }
    }
}
```

The preceding highlighted code shows an inner class called `MyInnerClass`, with a property called `myInnerProperty`, and a function called `myInnerFunction`.

One advantage is that the outer class can use the properties and functions of the inner class by declaring an instance of it, as shown highlighted in the next code snippet:

```
class SomeRegularClass{
    var someRegularProperty = 1

    val myInnerInstance = MyInnerClass()

    fun someRegularFunction(){
        val someVariable = myInnerInstance.myInnerProperty
    }
}
```

```

        myInnerInstance.myInnerFunction()
    }

    inner class MyInnerClass {
        val myInnerProperty = 1

        fun myInnerFunction() {
        }

    }
}

```

Furthermore, the inner class can also access the properties of the regular class, perhaps from the `myInnerFunction` function. This next code snippet shows this in action:

```

fun myInnerFunction() {
    someRegularProperty ++
}

```

This ability to define a new type within a class and create instances and share data is very useful in certain circumstances and for encapsulation. We will use an inner class in the Note to self app later in this chapter.

RecyclerView and RecyclerViewAdapter

In *Chapter 5, Beautiful Layouts with CardView and ScrollView*, we used a `ScrollView` widget and we populated it with a few `CardView` widgets so that we could see it scrolling. We could take what we have just learned about `ArrayList` and create a container of `TextView` objects, use them to populate a `ScrollView` widget, and, within each `TextView`, place the title of a note. This sounds like a perfect solution for showing each note so that it is clickable in the Note to self app.

We could create the `TextView` objects dynamically in Kotlin code, set their `text` property to be the title of a note, and then add the `TextView` objects to a `LinearLayout` contained in `ScrollView`. But this is imperfect.

The problem with displaying lots of widgets

This might seem fine, but what if there were dozens, hundreds, or even thousands of notes? We couldn't have thousands of `TextView` objects in memory because the Android device might simply run out of memory, or, at the very least, grind to a halt as it tries to handle the scrolling of such a vast amount of data.

Now, also imagine that we wanted (which we do) each note in the `ScrollView` widget to show whether it was important, a to-do, or an idea. And how about a short snippet from the text of the note as well?

We would need to devise some clever code that loads and destroys `Note` objects and `TextView` objects from `ArrayList`. It can be done – but to do it efficiently is far from straightforward.

The solution to the problem with displaying lots of widgets

Fortunately, this is a problem faced so commonly by mobile developers that the Android API has a solution built in.


We can add a single widget, called `RecyclerView` (like an environmentally friendly `ScrollView`, but with boosters too), to our UI layout. The `RecyclerView` class was designed as a solution to the problem we have been discussing. In addition, we need to interact with `RecyclerView` with a special type of class that understands how `RecyclerView` works. We will interact with it using an **adapter**. We will use the `RecyclerViewAdapter` class, inherit from it, customize it, and then use it to control the data from our `ArrayList` and display it in the `RecyclerView` class.

Let's find out a bit more about how the `RecyclerView` and `RecyclerViewAdapter` classes work.

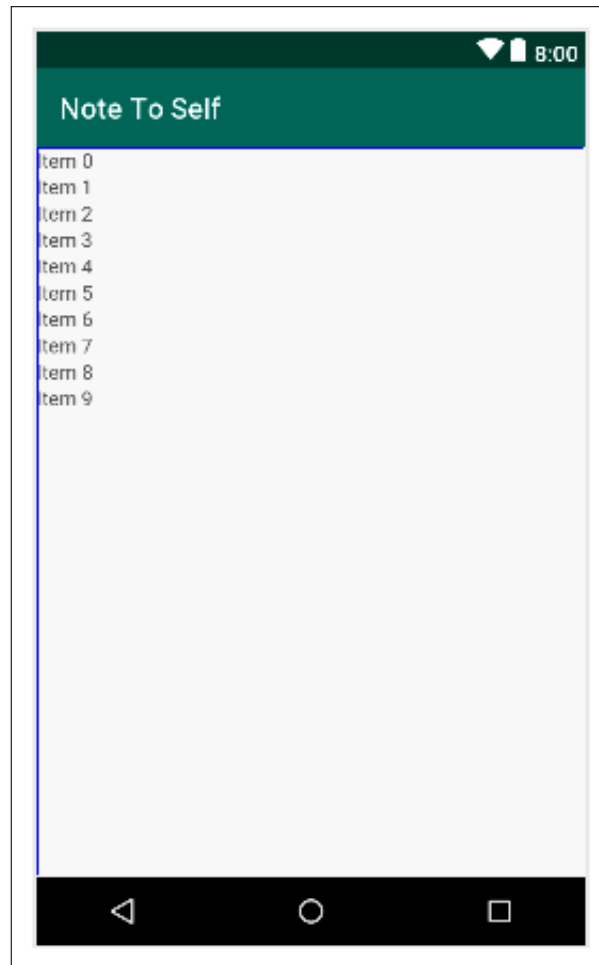
How to use RecyclerView and RecyclerViewAdapter

We already know how to store almost unlimited notes – we can do so in `ArrayList`, although we haven't implemented it yet. We also know that there is a UI layout called `RecyclerView` that is specifically designed to display potentially long lists of data. We just need to see how to put it all into action.

To add a `RecyclerView` widget to our layout, we can simply drag and drop it from the palette onto our UI in the usual way.

[ Don't do it yet. Let's just discuss it for a while first.]

The RecyclerView class will look like this in the UI designer:



This appearance, however, is more a representation of the possibilities than the actual appearance in an app. If we run the app at once after adding a RecyclerView widget, we will just get a blank screen.

The first thing we need to do to make practical use of a RecyclerView widget is decide what each item in the list will look like. It could be just a single TextView widget, or it could be an entire layout. We will use `LinearLayout`. To be clear and specific, we will use a `LinearLayout` instance that holds three `TextView` widgets for each item in our RecyclerView widget. This will allow us to display the note status (important/idea/to-do), the note title, and a short snippet of text from the actual note contents.

A list item needs to be defined in its own XML file, then the `RecyclerView` widget can hold multiple instances of this list item layout.

Of course, none of this explains how we overcome the complexity of managing what data is shown in which list item and how it is retrieved from `ArrayList`.

This data handling is taken care of by our own customized implementation of `RecyclerViewAdapter`. The `RecyclerViewAdapter` class implements the `Adapter` interface. We don't need to know how `Adapter` works internally, we just need to override some functions, and then `RecyclerViewAdapter` will do all the work of communicating with our `RecyclerView` widget.

Wiring up an implementation of `RecyclerViewAdapter` to a `RecyclerView` widget is certainly more complicated than dragging 20 `TextView` widgets onto a `ScrollView` widget, but once it is done we can forget about it, and it will keep on working and manage itself regardless of how many notes we add to `ArrayList`. It also has built-in features for handling things such as neat formatting and detecting which item in a list was clicked.

We will need to override some functions of `RecyclerViewAdapter` and add a little code of our own.

What we will do to set up RecyclerView with RecyclerViewAdapter and an ArrayList of notes

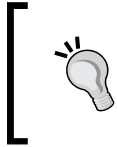
Look at this outline of the required steps so we know what to expect. To get the whole thing up and running, we would do the following:

1. Delete the temporary button and related code and then add a `RecyclerView` widget to our layout with a specific `id` property.
2. Create an XML layout to represent each item in the list. We have already mentioned that each item in the list will be a `LinearLayout` that contains three `TextView` widgets.
3. Create a new class that inherits from `RecyclerViewAdapter`, and add code to several overridden functions to control how it looks and behaves, including using our list item layout and `ArrayList` full of `Note` instances.
4. Add code in `MainActivity` to use `RecyclerViewAdapter` and the `RecyclerView` widget and bind it to our `ArrayList` instance.
5. Add an `ArrayList` instance to `MainActivity` to hold all our notes, and update the `createNewNote` function to add any new notes created in the `DialogNewNote` class to this `ArrayList`.

Let's go through and implement each of those steps in detail.

Adding RecyclerView, RecyclerViewAdapter, and ArrayList to the Note to Self project

Open the Note to self project. As a reminder, if you want to see the completed code and working app based on completing this chapter, it can be found in the Chapter16/Note to self folder.



As the required action in this chapter jumps around between different files, classes, and functions, I encourage you to follow along with the files from the download bundle open in your preferred text editor for reference.

Removing the temporary "Show Note" button and adding RecyclerView

These next few steps will get rid of the temporary code we added in *Chapter 14, Android Dialog Windows*, and set up our RecyclerView ready for binding to RecyclerViewAdapter later in the chapter:

1. In the `content_main.xml` file, remove the temporary Button with an id of `button`, which we added previously for testing purposes.
2. In the `onCreate` function of `MainActivity.kt`, delete the Button instance declaration and initialization along with the lambda that handles its clicks, as this code now creates an error. We will delete some more temporary code later in this chapter. Delete the code shown next:

```
// Temporary code
val button = findViewById<View>(R.id.button) as Button
button.setOnClickListener {
    // Create a new DialogShowNote called dialog
    val dialog = DialogShowNote()

    // Send the note via the sendNoteSelected function
    dialog.sendNoteSelected(tempNote)

    // Create the dialog
    dialog.show(supportFragmentManager, "123")
}
```

3. Now, switch back to `content_main.xml` in design view and drag a **RecyclerView** widget from the **Common** category of the palette onto the layout.
4. Set its `id` property to `recyclerView`.

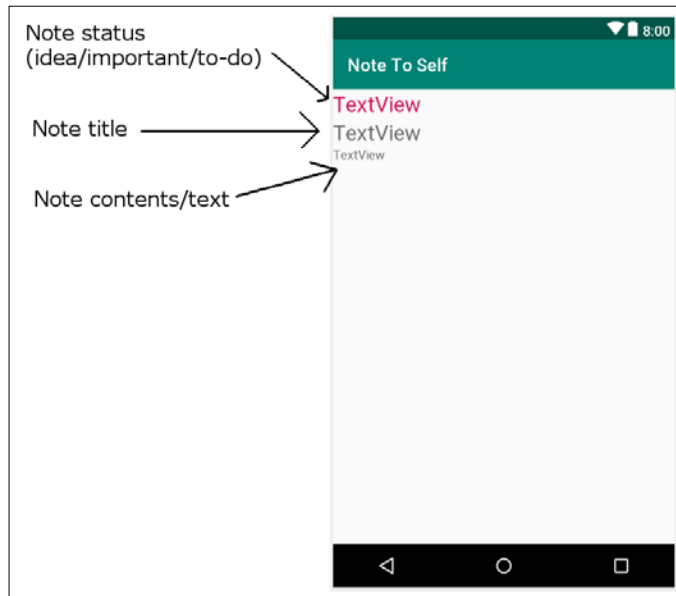
Now we have removed the temporary UI aspects from our project, and we have a `RecyclerView` widget complete with a unique `id` attribute ready to be referenced from our Kotlin code.

Creating a list item for RecyclerView

Next, we need a layout to represent each item in our `RecyclerView` widget. As previously mentioned, we will use a `LinearLayout` instance that holds three `TextView` widgets.

These are the steps needed to create a list item for use within `RecyclerView`:

1. Right-click on the `layout` folder in the project explorer and select **New | Layout resource file**. Enter `listitem` in the **Name:** field and make the **Root element:** `LinearLayout`. The default orientation attribute is vertical, which is just what we need.
2. Look at the next screenshot to see what we are trying to achieve with the remaining steps of this section. I have annotated it to show what each part will be in the finished app:



3. Drag three `TextView` instances onto the layout, one above the other, as per the reference screenshot. The first (top) will hold the note status/type (idea/important/to-do), the second (middle) will hold the note title, and the third (bottom) will hold a snippet of the note itself.
4. Configure the various attributes of the `LinearLayout` instance and the `TextView` widgets as shown in the following table:

Widget type	Property	Value to set to
<code>LinearLayout</code>	<code>layout_height</code>	<code>wrap_contents</code>
<code>LinearLayout</code>	<code>Layout_Margin all</code>	<code>5dp</code>
<code>TextView (top)</code>	<code>id</code>	<code>textViewStatus</code>
<code>TextView (top)</code>	<code>textSize</code>	<code>24sp</code>
<code>TextView (top)</code>	<code>textColor</code>	<code>@color/colorAccent</code>
<code>TextView (middle)</code>	<code>id</code>	<code>textViewTitle</code>
<code>TextView (middle)</code>	<code>textSize</code>	<code>24sp</code>
<code>TextView (top)</code>	<code>id</code>	<code>textViewDescription</code>

Now we have a `RecyclerView` widget for the main layout and a layout to use for each item in the list. We can go ahead and code our `RecyclerViewAdapter` implementation.

Coding the `RecyclerViewAdapter` class

We will now create and code a brand-new class. Let's call our new class `NoteAdapter`. Create a new class called `NoteAdapter` in the same folder as the `MainActivity` class (and all the other classes) in the usual way.

Edit the code for the `NoteAdapter` class by adding these `import` statements and inheriting from the `RecyclerView.Adapter` class, then add the two properties as shown. Edit the `NoteAdapter` class to be the same as the following code that we have just discussed:

```
import android.view.LayoutInflater
import android.view.View
import android.view.ViewGroup
import android.widget.TextView
import androidx.recyclerview.widget.RecyclerView


class NoteAdapter(
    private val mainActivity: MainActivity,
    private val noteList: List<Note>)
    : RecyclerView.Adapter<NoteAdapter.ListItemHolder>() {

}
```

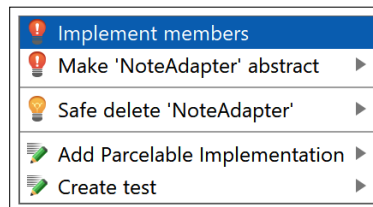
In the previous code, we declare and initialize two properties of the `NoteAdapter` class using the primary constructor. Notice the parameters of the constructor. It receives a `MainActivity` reference as well as a `List` reference. This implies that, when we use this class, we will need to send in a reference to the main activity of this app (`MainActivity`) as well as a `List` reference. We will see what use we put the `MainActivity` reference to shortly, but we can sensibly guess that the reference to a `List` with a type of `<Note>` will be a reference to our `Note` instances, which we will soon code in the `MainActivity` class. `NoteAdapter` will then hold a permanent reference to all the users' notes.

You will notice, however, that the class declaration and other areas of the code are underlined in red, showing that there are errors in our code.

The first error is because the `RecyclerView.Adapter` class (which we are inheriting from) needs us to override some of its abstract functions.

[ We discussed abstract classes and their functions in Chapter 11, *Inheritance in Kotlin*.]

The quickest way to do this is to click the class declaration, hold the *Alt* key, and then tap the *Enter* key. Choose **Implement members**, as shown in the next screenshot:



In the window that follows, hold down *Shift* and left-click all three options (functions to add) and then click **OK**. This process adds the following three functions:

- The `onCreateViewHolder` function, which is called when a layout for a list item is required
- The `onBindViewHolder` function, which is called when the `RecyclerView.Adapter` instance is bound to (connected/associated with) the `RecyclerView` instance in the layout
- The `getItemCount` function, which will be used to return the number of `Note` instances in `ArrayList`

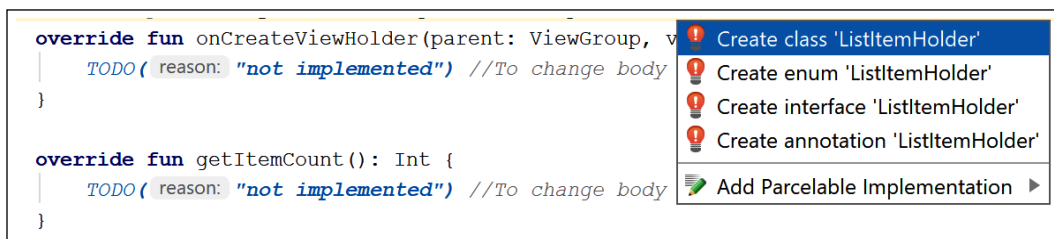
We will soon add code to each of these functions to do the required work at the specific time.

Note, however, that we still have multiple errors in our code, including in the newly autogenerated functions as well as the class declaration. We need to do some work to resolve these errors.

The errors are because the `NoteAdapter.ListItemHolder` class does not exist. `ListItemHolder` was added by us when we extended `NoteAdapter`. It is our chosen class type that will be used as the holder for each list item. Currently, it doesn't exist – hence the error. The two functions that also have the same error for the same reason were autogenerated when we asked Android Studio to implement the missing functions.

Let's solve the problem by making a start on the required `ListItemHolder` class. It is useful to us for `ListItemHolder` instances to share data/variables with `NoteAdapter`; therefore, we will create `ListItemHolder` as an inner class.

Click the error in the class declaration and select **Create class 'ListItemHolder'**, as shown in this next screenshot:



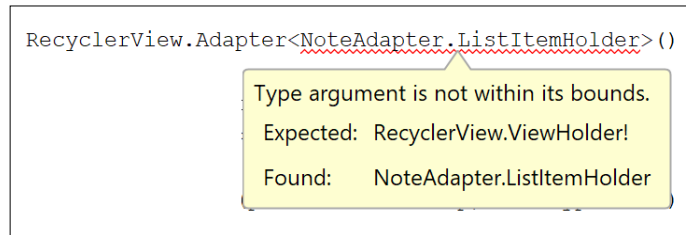
In the pop-up window that follows, choose **NoteAdapter** to generate `ListItemHolder` inside `NoteAdapter`.

The following code has been added to the `NoteAdapter` class:

```
class ListItemHolder {

}
```

But we still have multiple errors. Let's fix one of them now. Hover your mouse over the red-underlined error in the class declaration as shown in the next screenshot:



The error message reads **Type argument is not within its bounds. Expected: RecyclerView.ViewHolder! Found: NoteAdapter.ListItemHolder**. The reason for this is because we may have added `ListItemHolder`, but `ListItemHolder` must also implement `RecyclerView.ViewHolder` in order to be used as the correct type.

Amend the declaration of the `ListItemHolder` class to match this code:

```
inner class ListItemHolder(view: View) :
    RecyclerView.ViewHolder(view),
    View.OnClickListener {
```

Now the error is gone from the `NoteAdapter` class declaration, but because we also implemented `View.OnClickListener`, we need to implement the `onClick` function. Furthermore, `ViewHolder` doesn't provide a default constructor, so we need to do it. Add the following `onClick` function (empty for now) and this `init` block (empty for now) to the `ListItemHolder` class:

```
init {
}

override fun onClick(view: View) {
}
```



Be sure you added the code to the inner `ListItemHolder` class and not the `NoteAdapter` class.

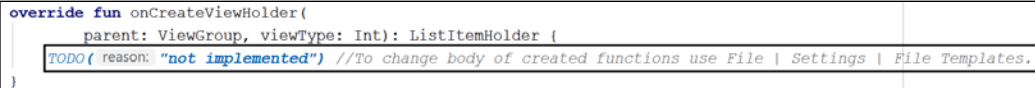
Let's clear up the final remaining errors. When the `onBindViewHolder` function was autogenerated, Android Studio didn't add the type for the holder parameter. This is causing an error in the function and an error in the class declaration. Update the `onBindViewHolder` function's signature, as shown in the next code:

```
override fun onBindViewHolder(
    holder: ListItemHolder, position: Int) {
```

In the `onCreateViewHolder` function signature, the return type has not been autogenerated. Amend the signature of the `onCreateViewHolder` function, as shown in this next code:

```
override fun onCreateViewHolder(
    parent: ViewGroup, viewType: Int): ListItemHolder {
```

As a last bit of good housekeeping, let's delete the three `// TODO...` comments that were autogenerated but not required. There is one in each of the autogenerated functions. They look like the one highlighted in this next screenshot:



```
override fun onCreateViewHolder(
    parent: ViewGroup, viewType: Int): ListItemHolder {
    TODO( reason: "not implemented") //To change body of created functions use File | Settings | File Templates.
}
```

As you delete the `TODO...` comments, more errors will appear. We need to add `return` statements to some of the autogenerated functions. We will do this as we proceed with coding the class.

After much tinkering and autogenerated, we finally have an almost error-free `NoteAdapter` class, complete with overridden functions and an inner class that we can code to get our `RecyclerView.Adapter` instance working. In addition, we can write code to respond to clicks (in `onClick`) on each of our `ListItemHolder` instances.

What follows is a complete listing of what the code should look like at this stage (excluding the import statements):

```
class NoteAdapter(
    private val mainActivity: MainActivity,
    private val noteList: List<Note>)
    : RecyclerView.Adapter<NoteAdapter.ListItemHolder>() {

    override fun onCreateViewHolder(
        parent: ViewGroup, viewType: Int):
        ListItemHolder {

    }

    override fun getItemCount(): Int {

    }

    override fun onBindViewHolder(
        holder: ListItemHolder,
```

```
        position: Int) {  
  
    }  
  
    inner class ListItemHolder(view: View) :  
        RecyclerView.ViewHolder(view),  
        View.OnClickListener {  
  
        init {  
  
        }  
  
        override fun onClick(view: View) {  
        }  
    }  
}
```



You could have just copy and pasted the preceding code instead of enduring the machinations of the previous pages, but then you wouldn't have experienced the process of implementing interfaces and inner classes so closely.

Now, let's code the functions and get this class operational.

Coding the onCreateViewHolder function

Next, we will adapt the autogenerated `onCreateViewHolder` function. Add the highlighted lines of code to the `onCreateViewHolder` function and study them:

```
override fun onCreateViewHolder(  
    parent: ViewGroup, viewType: Int):  
    ListItemHolder {  
  
    val itemView = LayoutInflater.from(parent.context)  
        .inflate(R.layout.listitem, parent, false)  
  
    return ListItemHolder(itemView)  
}
```

This code works by initializing `itemView` using `LayoutInflater` and our newly designed `listitem` layout. It then returns a new `ListItemHolder` instance, complete with an inflated and ready-to-use layout.

Coding the onBindViewHolder function

Next, we will adapt the `onBindViewHolder` function. Add the highlighted code to make the function the same as this code, and be sure to study the code as well:

```
override fun onBindViewHolder(
    holder: ListItemHolder, position: Int) {

    val note = noteList[position]
    holder.title.text = note.title

    // Show the first 15 characters of the actual note
    holder.description.text =
        note.description!!.substring(0, 15)

    // What is the status of the note?
    when {
        note.idea -> holder.status.text =
            MainActivity.resources.getString(R.string.idea_text)

        note.important -> holder.status.text =
            MainActivity.resources.getString(R.string.important_text)

        note.todo -> holder.status.text =
            MainActivity.resources.getString(R.string.todo_text)
    }
}
```

First, the code truncates the text to 15 characters so that it looks sensible in the list. Note that if the user enters a very short note below 15 characters this will cause a crash. It is left as an exercise for the reader to come back to this project and discover a solution to this imperfection.

It then checks what type of note it is (idea/to-do/important) and assigns the appropriate label from the string resources using a `when` expression.

This new code has left some errors in the code with `holder.title`, `holder.description`, and `holder.status`, because we need to add them to our `ListItemHolder` inner class. We will do this very soon.

Coding getItemCount

Amend the code in the getItemCount function, as shown next:

```
override fun getItemCount(): Int {  
    if (noteList != null) {  
        return noteList.size  
    }  
    // error  
    return -1  
}
```

This function is used internally by the class, and it supplies the current number of items in List.

Coding the ListItemHolder inner class

Now we can turn our attention to the ListItemHolder inner class. Adapt the ListItemHolder inner class by adding the following highlighted code:

```
inner class ListItemHolder(view: View) :  
    RecyclerView.ViewHolder(view),  
    View.OnClickListener {  
  
    internal var title =  
        view.findViewById<View>(  
            R.id.textViewTitle) as TextView  
  
    internal var description =  
        view.findViewById<View>(  
            R.id.textViewDescription) as TextView  
  
    internal var status =  
        view.findViewById<View>(  
            R.id.textViewStatus) as TextView  
  
    init {  
  
        view.isClickable = true  
        view.setOnClickListener(this)  
    }  
  
    override fun onClick(view: View) {  
        mainActivity.showNote(adapterPosition)  
    }  
}
```


The `ListItemHolder` properties get a reference to each of the `TextView` widgets in the layout. The `init` block code sets the whole view as clickable so that the OS will call the next function we discuss, `onClick`, when a holder is clicked.

In `onClick`, the call to `mainActivity.showNote` has an error because the function doesn't exist yet, but we will fix that in the next section. The call will simply show the clicked note using our custom `DialogFragment` instance.

Coding MainActivity to use the RecyclerView and RecyclerViewAdapter classes

Now, switch over to the `MainActivity` class in the editor window. Add these three new properties to the `MainActivity` class and remove the temporary code:

```
// Temporary code
//private var tempNote = Note()

private val noteList = ArrayList<Note>()
private val recyclerView: RecyclerView? = null
private val adapter: NoteAdapter? = null
```

These three properties are our `ArrayList` instance for all our `Note` instances, our `RecyclerView` instance, and an instance of our `NoteAdapter` class.

Adding code to onCreate

Add the following highlighted code in the `onCreate` function after the code that handles the user pressing on the floating action button (shown again for context):

```
fab.setOnClickListener { view ->
    val dialog = DialogNewNote()
    dialog.show(supportFragmentManager, "")
}

recyclerView =
    findViewById<View>(R.id.recyclerView)
    as RecyclerView

adapter = NoteAdapter(this, noteList)
val layoutManager =
    LinearLayoutManager(applicationContext)

recyclerView!!.layoutManager = layoutManager
```

```
recyclerView!!.itemAnimator = DefaultItemAnimator()

// Add a neat dividing line between items in the list
recyclerView!!.addItemDecoration(
    DividerItemDecoration(this,
        LinearLayoutManager.VERTICAL))

// set the adapter
recyclerView!!.adapter = adapter
```

Here, we initialize `recyclerView` with the `RecyclerView` widget from the layout. Our `NoteAdapter` (`adapter`) instance is initialized by calling the constructor we coded. Note that a reference to `MainActivity` (`this`) and the `ArrayList` instance is passed in, just as required by the class we have coded previously.

Next, we create a new object – a `LayoutManager` object. In the next four lines of code, we configure some properties of `recyclerView`.

The `itemAnimator` property and `addItemDecoration` function make each list item a little more visually enhanced with a separator line between each item in the list. Later, when we build a "Settings" screen, we will give the user the option to add and remove this separator.

The last thing we do is initialize the `adapter` property of `recyclerView` with our `adapter`, which combines our `adapter` with our view.

Now, we will make some changes to the `createNewNote` function.

Modifying the `createNewNote` function

In the `createNewNote` function, delete the temporary code we added in *Chapter 14, Android Dialog Windows* (shown commented out), and add the new highlighted code shown next:

```
fun createNewNote(n: Note) {
    // Temporary code
    // tempNote = n
    noteList.add(n)
    adapter!!.notifyDataSetChanged()
}
```

The new highlighted code adds a note to the `ArrayList` instance instead of simply initializing a solitary `Note` object, which has now been commented out. Then, we need to call `notifyDataSetChanged`, which lets our adapter know that a new note has been added.

Coding the `showNote` function

Add the `showNote` function, which is called from the `NoteAdapter` class using the reference to this class that was passed into the `NoteAdapter` constructor. Or, more accurately, it is called from the `ListItemClickListener` inner class when one of the items in the `RecyclerView` widget is tapped by the user. Add the `showNote` function to the `MainActivity` class:

```
fun showNote(noteToShow: Int) {  
    val dialog = DialogShowNote()  
    dialog.sendNoteSelected(noteList [noteToShow])  
    dialog.show(supportFragmentManager, "")  
}
```

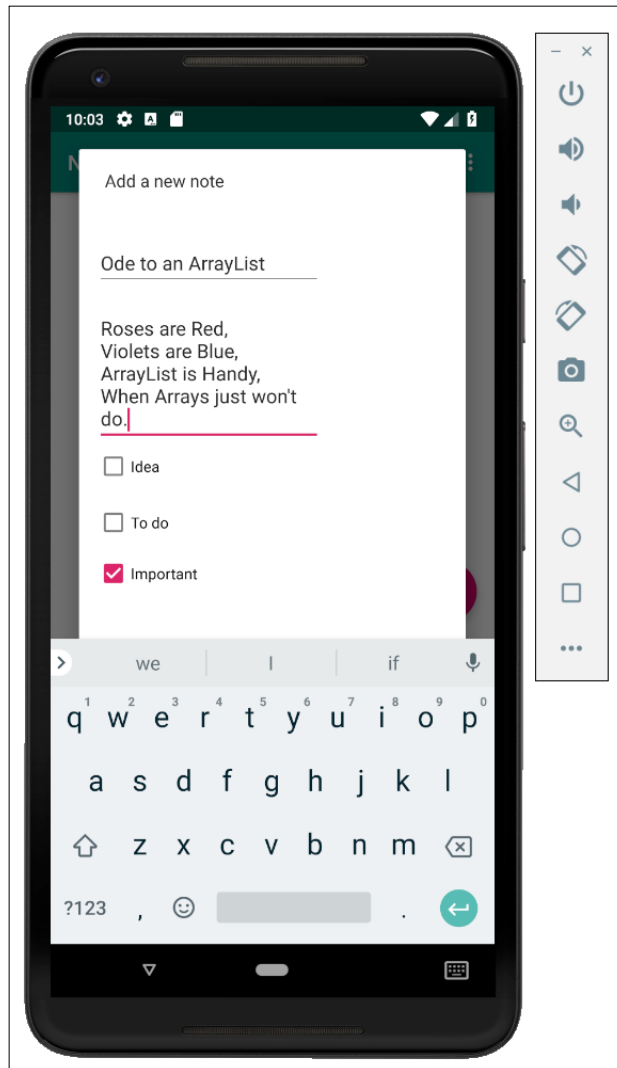


All the errors in the `NoteAdapter.kt` file are now gone.

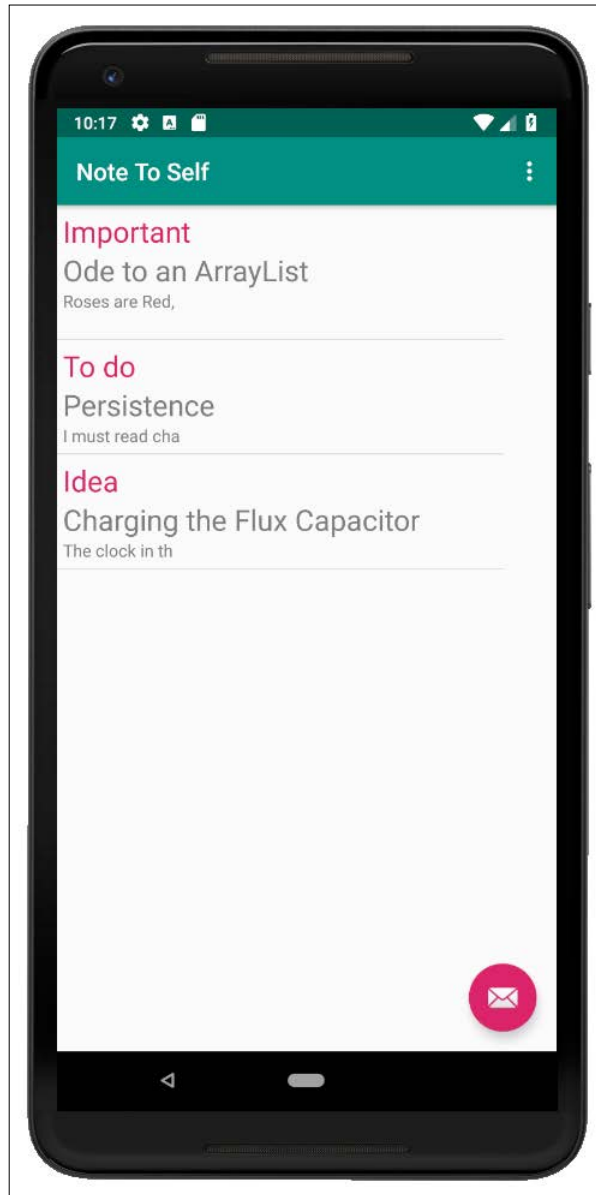
The code just added will launch a new instance of `DialogShowNote`, passing in the specific required note as referenced by `noteToShow`.

Running the app

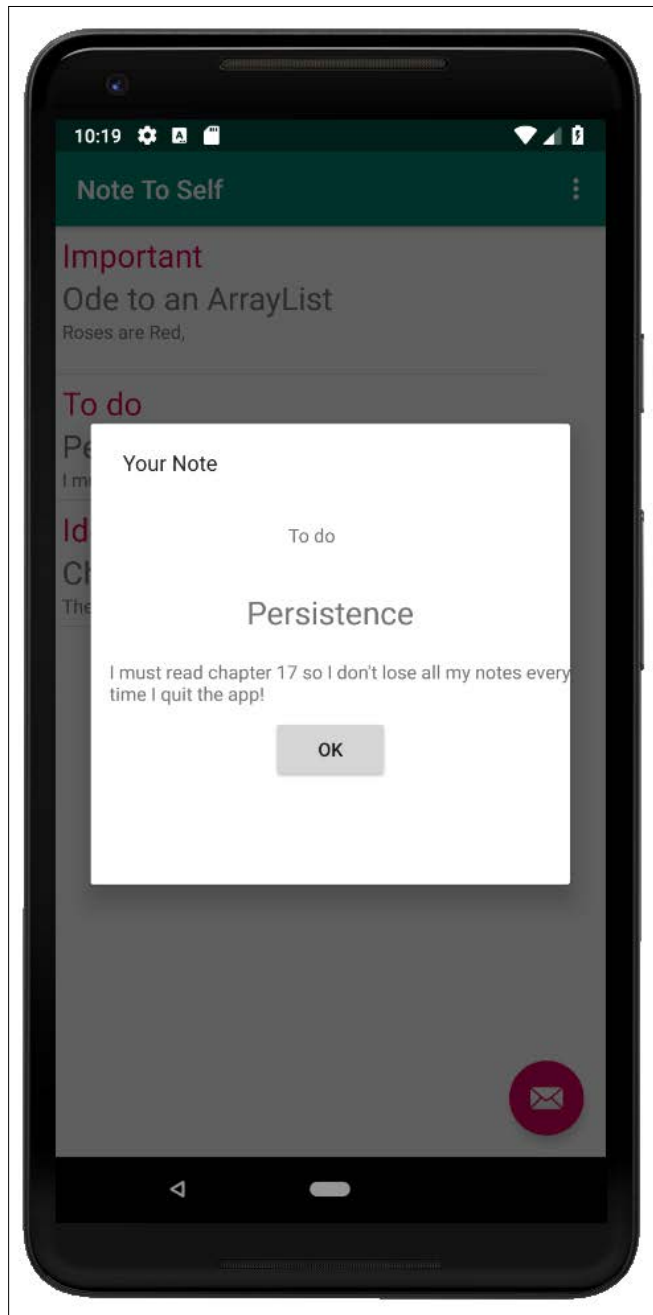
You can now run the app and enter a new note, as shown in this next screenshot:



After you have entered several notes of several types, the list (`RecyclerView`) will look something like this next screenshot:



And, if you click to view one of the notes, it will look like this:





Reader challenge

We could have spent more time formatting the layouts of our two dialog windows. Why not refer to *Chapter 5, Beautiful Layouts with CardView and ScrollView*, as well as the Material Design website, <https://material.io/design/>, and do a better job than this. Furthermore, you could enhance the `RecyclerView` list of notes by using `CardView` instead of `LinearLayout`.

Don't spend too long adding new notes, however, because there is a slight problem: close and restart the app. Uh oh, all the notes are gone!

Frequently asked questions

Q.1) I still don't understand how `RecyclerView.Adapter` works?

A) That's because we haven't really discussed it. The reason we have not discussed the behind-the-scenes details is because we don't need to know them. If we override the required functions, as we have just seen, everything will work. This is how `RecyclerView.Adapter` and most other classes we use are meant to be: hidden implementation with public functions to expose the necessary functionality.

Q.2) I feel like I *need* to know what is going on inside `RecyclerView.Adapter` and other classes as well. How can I do this?

A) It is true that there are more details for `RecyclerView.Adapter` (and almost every class that we use in this book) that we don't have the space to discuss. It is good practice to read the official documentation of the classes you use. You can read more about it at <https://developer.android.com/reference/android/support/v7/widget/RecyclerView.Adapter>.

Summary

Now we have added the functionality to hold multiple notes and implemented the ability to display them.

We achieved this by learning about and using the `RecyclerViewAdapter` class, which implements the `Adapter` interface, which allows us to bind together a `RecyclerView` instance and an `ArrayList` instance, allowing for the seamless display of data without us (the programmer) having to worry about the complex code that is part of these classes, and which we don't even see.

In the next chapter, we will start with making the user's notes persist when they quit the app or switch off their device. In addition, we will create a "Settings" screen, and see how we can make the settings persist as well. We will use different techniques to achieve each of these goals.