# 6. RecyclerView

*Overview*

*In this chapter, you will learn how to add lists and grids of items to your apps and effectively leverage the recycling power of* **`RecyclerView`**. *You'll also learn how to handle user interaction with the item views on the screen and support different item view types—for example, for titles. Later in the chapter, you'll add and remove items dynamically.*

*By the end of the chapter, you will have the skills required to present your users with interactive lists of rich items.*

## Introduction

In the previous chapter, we learned how to fetch data, including lists of items and image URLs, from APIs, and how to load images from URLs. Combining that knowledge with the ability to display lists of items is the goal of this chapter.

Quite often, you will want to present your users with a list of items. For example, you might want to show them a list of pictures on their device, or let them select their country from a list of all countries. To do that, you would need to populate multiple views, all sharing the same layout but presenting different content.

Historically, this was achieved by using **`ListView`** or **`GridView`**. While both are still viable options, they do not offer the robustness and flexibility of **`RecyclerView`**. For example, they do not support large datasets well, they do not support horizontal scrolling, and they do not offer rich divider customization. Customizing the divider between items in **`RecyclerView`** can be easily achieved using **`RecyclerView.ItemDecorator`**.

So, what does **`RecyclerView`** do? **`RecyclerView`** orchestrates the creation, population, and reuse (hence the name) of views representing lists of items. To use **`RecyclerView`**, you need to familiarize yourself with two of its dependencies: the adapter (and through it, the view holder) and the layout manager. These dependencies provide our **`RecyclerView`** with the content to show, as well as telling it how to present that content and how to lay it out on the screen.

The adapter provides **`RecyclerView`** with child views (nested Android views within **`RecyclerView`** used to represent individual data items) to draw on the screen, binds those views to data (via **`ViewHolder`** instances), and reports user interaction with those views. The layout manager tells **`RecyclerView`** how to lay its children out. We are provided with three layout types by default: linear, grid, and staggered grid—managed by **`LinearLayoutManager`**, **`GridLayoutManager`**, and **`StaggeredGridLayoutManager`**, respectively.

In this chapter, we will develop an app that lists secret agents and whether they are currently active or sleeping (and thus unavailable). The app will then allow us to add new agents or delete existing ones by swiping them away. There is a twist, though—as you saw in *Chapter 5, Essential Libraries: Retrofit, Moshi, and Glide*, all our agents will be cats.

## Adding RecyclerView to Our Layout

In *Chapter 3, Screens and UI*, we saw how we can add views to our layouts to be inflated by activities, fragments, or custom views.

**RecyclerView** is just another such view. To add it to our layout, we need to add the following tag to our layout:

```
<androidx.recyclerview.widget.RecyclerView
    android:id="@+id/recycler_view"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    tools:listitem="@layout/item_sample" />
```

You should already be able to recognize the **android:id** attribute, as well as the **android:layout_width** and **android:layout_height** ones.

We can use the optional **tools:listitem** attribute to tell Android Studio which layout to inflate as a list item in our preview toolbar. This will give us an idea of how **RecyclerView** might look in our app.

Adding a **RecyclerView** tag to our layout means we now have an empty container to hold the child views representing our list items. Once populated, it will handle the presenting, scrolling, and recycling of child views for us.

## Exercise 6.01: Adding an Empty RecyclerView to Your Main Activity

To use **RecyclerView** in your app, you first need to add it to one of your layouts. Let's add it to the layout inflated by our main activity:

1. Start by creating a new empty activity project (**File** | **New** | **New Project** | **Empty Activity**). Name your application **My RecyclerView App**. Make sure your package name is **com.example.myrecyclerviewapp**.
2. Set the save location to where you want to save your project. Leave everything else at its default values and click **Finish**. Make sure you are on the **Android** view in your **Project** pane:

Figure 6.1: Android view in the Project pane

3. Open your **activity_main.xml** file in **Text** mode.
4. To turn your label into a title at the top of the screen under which you can add your **RecyclerView**, add an ID to **TextView** and align it to the top, like so:

```
<TextView
    android:id="@+id/hello_label"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="Hello World!"
    app:layout_constraintLeft_toLeftOf="parent"
    app:layout_constraintRight_toRightOf="parent"
    app:layout_constraintTop_toTopOf="parent" />
```

5. Add the following after **TextView** tag to add an empty **RecyclerView** element to our layout, constrained below your **hello_label** **TextView** title:

```
<androidx.recyclerview.widget.RecyclerView
    android:id="@+id/recycler_view"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    app:layout_constraintTop_toBottomOf="@+id/hello_l
abel" />
```

Your layout file should now look something like this:

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.ConstraintLayout
xmlns:android="http://schemas.android.com/apk/res/and
roid"
```

```
    xmlns:app="http://schemas.android.com/apk/res-
auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".MainActivity">
    <TextView
        android:id="@+id/hello_label"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Hello World!"
        app:layout_constraintLeft_toLeftOf="parent"
        app:layout_constraintRight_toRightOf="parent"
        app:layout_constraintTop_toTopOf="parent" />
    <androidx.recyclerview.widget.RecyclerView
        android:id="@+id/recycler_view"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        app:layout_constraintTop_toBottomOf="@+id/hel
lo_label" />
</androidx.constraintlayout.widget.ConstraintLayout>
```

6. Run your app by clicking the **`Run app`** button or pressing *Ctrl* + *R* (*Shift* + *F10* on Windows). On the emulator, it should look like this:

Figure 6.2: App with an empty RecyclerView (image cropped for space)

As you can see, our app runs and our layout is presented on the screen. However, we do not see our **`RecyclerView`**. Why is that? At this stage, our **`RecyclerView`** has no content. **`RecyclerView`** with no content does not render by default—so, while our **`RecyclerView`** is indeed on the screen, it is not visible. This brings us to the next step—populating **`RecyclerView`** with content that we can actually see.

## Populating the RecyclerView

So, we added **`RecyclerView`** to our layout. For us to benefit from **`RecyclerView`**, we need to add content to it. Let's see how we go about doing that.

As we mentioned before, to add content to our **`RecyclerView`**, we would need to implement an adapter. An adapter binds our data to child views. In simpler terms, this means it tells **`RecyclerView`** how to plug data into views designed to present that data.

For example, let's say we want to present a list of employees.

First, we need to design our UI model. This will be a data object holding all the information needed by our view to present a single employee. Because this is a UI model, one convention is to suffix its name with **`UiModel`**:

```
data class EmployeeUiModel(
    val name: String,
    val biography: String,
    val role: EmployeeRole,
    val gender: Gender,
    val imageUrl: String
)
```

We will define **`EmployeeRole`** and **`Gender`** as follows:

```
enum class EmployeeRole {
    HumanResources,
    Management,
    Technology
}
enum class Gender {
    Female,
    Male,
    Unknown
}
```

The values are provided as an example, of course. Feel free to add more of your own!

Figure 6.3: The model's hierarchy

Now we know what data to expect when binding to a view—so, we can design our view to present this data (this is a simplified version of the actual layout, which we'll save as **`item_employee.xml`**). We'll start with the **`ImageView`**:

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.ConstraintLayout
xmlns:android="http://schemas.android.com/apk/res/andro
id"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:orientation="vertical"
    android:padding="8dp">
    <ImageView
        android:id="@+id/item_employee_photo"
        android:layout_width="60dp"
        android:layout_height="60dp"
        android:contentDescription="@string/item_employ
ee_photo"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toTopOf="parent"
        tools:background="@color/colorPrimary" />
```

And then add each **`TextView`**:

```
    <TextView
        android:id="@+id/item_employee_name"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_marginStart="16dp"
        android:layout_marginLeft="16dp"
        android:textStyle="bold"
        app:layout_constraintStart_toEndOf="@+id/item_e
mployee_photo"
```

```
        app:layout_constraintTop_toTopOf="parent"
        tools:text="Oliver" />
    <TextView
        android:id="@+id/item_employee_role"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:textColor="@color/colorAccent"
        app:layout_constraintStart_toStartOf="@+id/item
_employee_name"
        app:layout_constraintTop_toBottomOf="@+id/item_
employee_name"
        tools:text="Exotic Shorthair" />
    <TextView
        android:id="@+id/item_employee_biography"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        app:layout_constraintStart_toStartOf="@+id/item
_employee_role"
        app:layout_constraintTop_toBottomOf="@+id/item_
employee_role"
        tools:text="Stealthy and witty. Better avoid in
dark alleys." />
    <TextView
        android:id="@+id/item_employee_gender"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:textSize="30sp"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintTop_toTopOf="parent"
        tools:text="&#9794;" />
</androidx.constraintlayout.widget.ConstraintLayout>
```

So far, there is nothing new. You should be able to recognize all of the different view types from *Chapter 2*, *Building User Screen Flows*:

Figure 6.4: Preview of the item_cat.xml layout file

With a data model and a layout, we now have everything we need to bind our data to the view. To do that, we will implement a view holder. Usually, a view holder has two responsibilities: it holds a reference to a view (as its name implies), but it also binds data to that view. We will implement our view holder as follows:

```
private val FEMALE_SYMBOL by lazy {
    HtmlCompat.fromHtml("&#9793;",
HtmlCompat.FROM_HTML_MODE_LEGACY)
}
private val MALE_SYMBOL by lazy {
    HtmlCompat.fromHtml("&#9794;",
HtmlCompat.FROM_HTML_MODE_LEGACY)
}
private const val UNKNOWN_SYMBOL = "?"
class EmployeeViewHolder(
    containerView: View,
    private val imageLoader: ImageLoader
) : ViewHolder(containerView) {
    private val employeeNameView: TextView
        by lazy {
containerView.findViewById(R.id.item_employee_name) }
    private val employeeRoleView: TextView
```

```
        by lazy {
containerView.findViewById(R.id.item_employee_role) }
    private val employeeBioView: TextView
        by lazy {
containerView.findViewById(R.id.item_employee_bio) }
    private val employeeGenderView: TextView
        by lazy {
containerView.findViewById(R.id.item_employee_gender) }
    fun bindData(employeeData: EmployeeUiModel) {
        imageLoader.loadImage(employeeData.imageUrl,
employeePhotoView)
        employeeNameView.text = employeeData.name
        employeeRoleView.text = when
(employeeData.role) {
            EmployeeRole.HumanResources -> "Human
Resources"
            EmployeeRole.Management -> "Management"
            EmployeeRole.Technology -> "Technology"
        }
        employeeBioView.text = employeeData.biography
        employeeGenderView.text = when
(employeeData.gender) {
            Gender.Female -> FEMALE_SYMBOL
            Gender.Male -> MALE_SYMBOL
            else -> UNKNOWN_SYMBOL
        }
    }
}
```

There are a few things worth noting in the preceding code. First, by convention, we suffixed the name of our view holder with **`ViewHolder`**. Second, note that **`EmployeeViewHolder`** needs to implement the abstract **`RecyclerView.ViewHolder`** class. This is required so that the generic type of our adapter can be our view holder. Lastly, we lazily keep references to the views we are interested in. The first time

**`bindData(EmployeeUiModel)`** is called, we will find these views in the layout and keep references to them.

Next, we introduced a **`bindData(EmployeeUiModel)`** function. This function will be called by our adapter to bind the data to the view held by the view holder. The last but most important thing to note is that we always make sure to set a state for all modified views for any possible input.

With our view holder set up, we can proceed to implement our adapter. We will start by implementing the minimum required functions, plus a function to set the data. Our adapter will look something like this:

```
class EmployeesAdapter(
    private val layoutInflater: LayoutInflater,
    private val imageLoader: ImageLoader
) : RecyclerView.Adapter<EmployeeViewHolder>() {
    private val employeesData =
mutableListOf<EmployeeUiModel>()
    fun setData(employeesData: List<EmployeeUiModel>) {
        this.employeesData.clear()
        this.employeesData.addAll(employeesData)
        notifyDataSetChanged()
    }
    override fun onCreateViewHolder(parent: ViewGroup,
viewType: Int): EmployeeViewHolder {
        val view =
layoutInflater.inflate(R.layout.item_employee, parent,
false)
        return EmployeeViewHolder(view, imageLoader)
    }
    override fun getItemCount() = employeesData.size
    override fun onBindViewHolder(holder:
EmployeeViewHolder, position:Int) {
```

```
            holder.bindData(employeesData[position])
        }
    }
```

Let's go over this implementation. First, we inject our dependencies to the adapter via its constructor. This will make testing our adapter much easier—but will also allow us to change some of its behavior (for example, replace the image loading library) painlessly. In fact, we would not need to change the adapter at all in that case.

Then, we define a private mutable list of **`EmployeeUiModel`** to store the data currently provided by the adapter to **`RecyclerView`**. We also introduce a method to set that list. Note that we keep a local list and set its contents, rather than allowing **`employeesData`** to be set directly. This is mainly because Kotlin, just like Java, passes variables by reference. Passing variables by reference means changes to the content of the list passed into the adapter would change the list held by the adapter. So, for example, if an item was removed from outside the adapter, the adapter would have that item removed as well. This becomes a problem because the adapter would not be aware of that change, and so would not be able to notify **`RecyclerView`**. There are other risks around the list being modified from outside the adapter, but covering them is beyond the scope of this book.

Another benefit of encapsulating the modification of the data in a function is that we avoid the risk of forgetting to notify **`RecyclerView`** that the dataset has changed, which we do by calling **`notifyDataSetChanged()`**.

We proceed to implement the adapter's **`onCreateViewHolder(ViewGroup, Int)`** function. This function is called when the **`RecyclerView`** needs a new **`ViewHolder`** to render data on the screen. It provides us with a container **`ViewGroup`** and a view type (we'll look into view types later in this chapter). The function then expects us to return a view holder initialized with a view (in our case, an inflated one). So, we inflate the view we designed earlier, passing it to a new **EmployeeViewHolder** instance. Note that the last argument to the inflated function is **false**. This makes sure we do not attach the newly inflated view to the parent. Attaching and detaching views will be managed by the layout manager. Setting it to **true** or omitting it would result in **IllegalStateException** being thrown. Finally, we return the newly created **EmployeeViewHolder**.

To implement **getItemCount()**, we simply return the size of our **employeesData** list.

Lastly, we implement **onBindViewHolder(EmployeeViewHolder, Int)**. This is done by passing **EmployeeUiModel**, stored in **catsData**, at the given position to the **bindData(EmployeeUiModel)** function of our view holder. Our adapter is now ready.

If we tried to plug our adapter into our **RecyclerView** at this point and run our app, we would still see no content. This is because we are still missing two small steps: setting data to our adapter and assigning a layout manager to our **RecyclerView**. The complete working code would look like this:

```
class MainActivity : AppCompatActivity() {
    private val employeesAdapter by lazy {
        EmployeesAdapter(layoutInflater,
GlideImageLoader(this)) }
    private val recyclerView: RecyclerView by lazy
        { findViewById(R.id.main_recycler_view) }
    override fun onCreate(savedInstanceState: Bundle?)
{
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)
        recyclerView.adapter = employeesAdapter
        recyclerView.layoutManager =
            LinearLayoutManager(this,
LinearLayoutManager.VERTICAL,
                false)
        employeesAdapter.setData(
            listOf(
                EmployeeUiModel(
                    "Robert",
                    "Rose quickly through the
organization",
                    EmployeeRole.Management,
                    Gender.Male,
                    "https://images.pexels.com/photos/2
20453 /pexels-photo-220453.jpeg?auto
=compress&cs=tinysrgb&h=650&w=940"
                ),
                EmployeeUiModel(
                    "Wilma",
                    "A talented developer",
                    EmployeeRole.Technology,
                    Gender.Female,
                    "https://images.pexels.com/photos/3
189024 /pexels-photo-3189024.jpeg?auto=compress&cs
=tinysrgb&h=650&w=940"
                ),
                EmployeeUiModel(
                    "Curious George",
                    "Excellent at retention",
                    EmployeeRole.HumanResources,
                    Gender.Unknown,
                    "https://images.pexels.com/photos/7
71742 /pexels-photo-771742.jpeg?auto
=compress&cs=tinysrgb&h=750&w=1260"
                )
            )
        )
    }
}
```

Running our app now, we would see a list of our employees.

Note that we hardcoded the list of employees. In a production app, following a **Model-View-View-Model** (**MVVM**) pattern (we will cover this pattern in *Chapter 14*, *Architecture Patterns*), you would be provided with data to present by your `ViewModel`. It is also important to note that we kept a reference to `employeesAdapter`. This is so that we could indeed, at a later time, set the data to different values. Some implementations rely on reading the adapter from `RecyclerView` itself—this can potentially result in unnecessary casting operations and unexpected states where the adapter is not yet assigned to `RecyclerView`, and so this is generally not a recommended approach.

Lastly, note that we chose to use `LinearLayoutManager`, providing it with the activity for context, a `VERTICAL` orientation flag, and `false` to tell it that we do not want the order of the items in the list reversed.

## Exercise 6.02: Populating Your RecyclerView

`RecyclerView` is not very interesting without any content. It is time you populate `RecyclerView` by adding your secret cat agents to it.

A quick recap before you dive in: in the previous exercise, we introduced an empty list designed to hold a list of secret cat agents that the users have at their disposal. In this exercise, you will be populating that list to present the users with the available secret cat agents in the agency:

1. To keep our file structure tidy, we will start by creating a model package. Right-click on the package name of our app, then select `New` | `Package`:

Figure 6.5: Creating a new package

2. Name the new package **model**. Click **OK** to create the package.
3. To create our first model data class, right-click on the newly created model package, then select **New** | **Kotlin File/Class**.
4. Under **name**, fill in **CatUiModel**. Leave **kind** as **File** and click on **OK**. This will be the class holding the data we have about every individual cat agent.
5. Add the following to the newly created **CatUiModel.kt** file to define the data class with all the relevant properties of a cat agent:

```
data class CatUiModel(
    val gender: Gender,
    val breed: CatBreed,
    val name: String,
    val biography: String,
    val imageUrl: String
)
```

   For each cat agent, other than their name and photo, we want to know the gender, breed, and biography. This will help us choose the right agent for a mission.

6. Again, right-click on the model package, then navigate to **New** | **Kotlin File/Class**.
7. This time, name the new file **CatBreed** and set **kind** to the **Enum** class. This class will hold our different cat breeds.
8. Update your newly created enum with some initial values, as follows:

```
enum class CatBreed {
    AmericanCurl,
    BalineseJavanese,
    ExoticShorthair
}
```

9. Repeat *steps 6* and *7*, only this time call your file **`Gender`**. This will hold the accepted values for a cat agent's gender.
10. Update the **`Gender`** enum like so:

```
enum class Gender {
    Female,
    Male,
    Unknown
}
```

11. Now, to define the layout of the view holding the data about each cat agent, create a new layout resource file by right-clicking on **`layout`** and then selecting **`New`** | **`Layout resource file`**:

Figure 6.6: Creating a new layout resource file

12. Name your resource **`item_cat`**. Leave all the other fields as they are and click **`OK`**.
13. Update the contents of the newly created **`item_cat.xml`** file. (The following code block has been truncated for space. Use the link below to see the full code that you need to add.)

item_cat.xml

```
<ImageView
    android:id="@+id/item_cat_photo"
    android:layout_width="60dp"
    android:layout_height="60dp"
    android:contentDescription="@string/item_cat_photo"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toTopOf="parent"
```

```
17        tools:background="@color/colorPrimary" />
18
19    <TextView
20        android:id="@+id/item_cat_name"
21        android:layout_width="wrap_content"
22        android:layout_height="wrap_content"
23        android:layout_marginStart="16dp"
24        android:layout_marginLeft="16dp"
25        android:textStyle="bold"
26        app:layout_constraintStart_toEndOf="@+id/item_cat_photo"
27        app:layout_constraintTop_toTopOf="parent"
28        tools:text="Oliver" />
```

*The complete code for this step can be found at* **http://packt.live/3sopUjo**.

This will create a layout with an image and text fields for a name, breed, and biography to be used in our list.

14. You will notice that *line 14* is highlighted in red. This is because you haven't declared **`item_cat_photo`** in **`strings.xml`** under the **`res/values`** folder yet. Do so now by placing the text cursor over **`item_cat_photo`** and pressing *Alt* + *Enter* (*Option* + *Enter* on Mac), then select **`Create string value resource 'item_cat_photo'`**:

Figure 6.7: A string resource is not yet defined

15. Under **`Resource value`**, fill in **`Photo`**. Press **`OK`**.
16. You will need a copy of **`ImageLoader.kt`**, introduced in *Chapter 5*, *Essential Libraries: Retrofit, Moshi, and Glide*, so right-click on the package name of your app, navigate to **`New`** | **`Kotlin File/Class`**, and then set the name to **`ImageLoader`** and **`kind`** to **`Interface`**, and click **`OK`**.
17. Similar to *Chapter 5*, *Essential Libraries: Retrofit, Moshi, and Glide*, you only need to add one function here:

```
interface ImageLoader {
    fun loadImage(imageUrl: String, imageView:
ImageView)
}
```

Make sure to import **ImageView**.

18. Right-click on the package name of your app again, then select **New | Kotlin File/Class**.
19. Call the new file **CatViewHolder**. Click **OK**.
20. To implement **CatViewHolder**, which will bind the cat agent data to your views, replace the contents of the **CatViewHolder.kt** file with the following:

```
private val FEMALE_SYMBOL by lazy {
    HtmlCompat.fromHtml("&#9793;",
HtmlCompat.FROM_HTML_MODE_LEGACY)
}
private val MALE_SYMBOL by lazy {
    HtmlCompat.fromHtml("&#9794;",
HtmlCompat.FROM_HTML_MODE_LEGACY)
}
private const val UNKNOWN_SYMBOL = "?"
class CatViewHolder(
    containerView: View,
    private val imageLoader: ImageLoader
) : ViewHolder(containerView) {
    private val catBiographyView: TextView
        by lazy {
containerView.findViewById(R.id.item_cat_biography) }
    private val catBreedView: TextView
        by lazy {
containerView.findViewById(R.id.item_cat_breed) }
    private val catGenderView: TextView
        by lazy {
containerView.findViewById(R.id.item_cat_gender) }
    private val catNameView: TextView
```

```
        by lazy {
containerView.findViewById(R.id.item_cat_name) }
    private val catPhotoView: ImageView
        by lazy {
containerView.findViewById(R.id.item_cat_photo) }
    fun bindData(catData: CatUiModel) {
        imageLoader.loadImage(catData.imageUrl,
catPhotoView)
        catNameView.text = catData.name
        catBreedView.text = when (catData.breed) {
            CatBreed.AmericanCurl -> "American Curl"
            CatBreed.BalineseJavanese -> "Balinese-
Javanese"
            CatBreed.ExoticShorthair -> "Exotic
Shorthair"
        }
        catBiographyView.text = catData.biography
        catGenderView.text = when (catData.gender) {
            Gender.Female -> FEMALE_SYMBOL
            Gender.Male -> MALE_SYMBOL
            else -> UNKNOWN_SYMBOL
        }
    }
}
```

21. Still under our app package name, create a new Kotlin file named **`CatsAdapter`**.
22. To implement **`CatsAdapter`**, which is responsible for storing the data for **`RecyclerView`**, as well as creating instances of your view holder and using them to bind data to views, replace the contents of the **`CatsAdapter.kt`** file with this:

```
package com.example.myrecyclerviewapp
import android.view.LayoutInflater
import android.view.ViewGroup
import androidx.recyclerview.widget.RecyclerView
import com.example.myrecyclerviewapp.model.CatUiModel
class CatsAdapter(
    private val layoutInflater: LayoutInflater,
    private val imageLoader: ImageLoader
) : RecyclerView.Adapter<CatViewHolder>() {
    private val catsData = mutableListOf<CatUiModel>
()
    fun setData(catsData: List<CatUiModel>) {
        this.catsData.clear()
        this.catsData.addAll(catsData)
        notifyDataSetChanged()
    }
    override fun onCreateViewHolder(parent:
ViewGroup,
      viewType: Int): CatViewHolder {
        val view =
layoutInflater.inflate(R.layout.item_cat,
      parent, false)
        return CatViewHolder(view, imageLoader)
    }
    override fun getItemCount() = catsData.size
    override fun onBindViewHolder(holder:
CatViewHolder,
      position: Int) {
        holder.bindData(catsData[position])
    }
}
```

23. At this point, you need to include Glide in your project. Start by adding the following line of code to the **`dependencies`** block inside your app's **`gradle.build`** file:

```
implementation
'com.github.bumptech.glide:glide:4.11.0'
```

24. Create a **`GlideImageLoader`** class in your app package path, containing the following:

```
package com.example.myrecyclerviewapp
import android.content.Context
import android.widget.ImageView
import com.bumptech.glide.Glide
class GlideImageLoader(private val context: Context)
: ImageLoader {
    override fun loadImage(imageUrl: String,
imageView: ImageView) {
        Glide.with(context)
            .load(imageUrl)
            .centerCrop()
            .into(imageView)
    }
}
```

This is a simple implementation assuming the loaded image should always be center-cropped.

25. Update your **`MainActivity`** file:

```
class MainActivity : AppCompatActivity() {
    private val recyclerView: RecyclerView
        by lazy { findViewById(R.id.recycler_view) }
    private val catsAdapter by lazy {
CatsAdapter(layoutInflater, GlideImageLoader(this)) }
    override fun onCreate(savedInstanceState:
Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)
        recyclerView.adapter = catsAdapter
        recyclerView.layoutManager =
LinearLayoutManager(this,
          LinearLayoutManager.VERTICAL, false)
        catsAdapter.setData(
            listOf(
                CatUiModel(
                    Gender.Male,
                    CatBreed.BalineseJavanese,
                    "Fred",
                    "Silent and deadly",
                    "https://cdn2.thecatapi.com/image
s/DBmIBhhyv.jpg"
                ),
                CatUiModel(
                    Gender.Female,
                    CatBreed.ExoticShorthair,
                    "Wilma",
                    "Cuddly assassin",
                    "https://cdn2.thecatapi.com/image
s/KJF8fB_20.jpg"
                ),
                CatUiModel(
                    Gender.Unknown,
                    CatBreed.AmericanCurl,
                    "Curious George",
                    "Award winning investigator",
                    "https://cdn2.thecatapi.com/image
s/vJB8rwfdX.jpg"
                )
            )
        )
    }
}
```

This will define your adapter, attach it to **RecyclerView**, and populate it with some hardcoded data.

26. In your **AndroidManifest.xml** file, add the following in the **manifest** tag before the application tag:

```
<uses-permission
android:name="android.permission.INTERNET" />
```

This will allow your app to download images off the internet.

27. For some final touches, such as giving our title view a proper name and text, update your **activity_main.xml** file like so:

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.ConstraintLayout
xmlns:android="http://schemas.android.com/apk/res/and
roid"
    xmlns:app="http://schemas.android.com/apk/res-
auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".MainActivity">
    <TextView
        android:id="@+id/main_label"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/main_title"
        android:textSize="24sp"
        app:layout_constraintLeft_toLeftOf="parent"
        app:layout_constraintRight_toRightOf="parent"
        app:layout_constraintTop_toTopOf="parent" />
    <androidx.recyclerview.widget.RecyclerView
        android:id="@+id/recycler_view"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        app:layout_constraintTop_toBottomOf="@+id/mai
n_label" />
</androidx.constraintlayout.widget.ConstraintLayout>
```

28. Also, update your **`strings.xml`** file to give your app a proper name and title:

```
<resources>
    <string name="app_name">SCA - Secret Cat
Agents</string>
    <string name="item_cat_photo">Cat photo</string>
    <string name="main_title">Our Agents</string>
</resources>
```

29. Run your app. It should look like this:

Figure 6.8: RecyclerView with hardcoded secret cat agents

As you can see, **`RecyclerView`** now has content, and your app is starting to take shape. Note how the same layout is used to present different items based on the data bound to each instance. As you would expect, if you added enough items for them to go off-screen, scrolling works. Next, we'll look into allowing the user to interact with the items inside our **`RecyclerView`**.

## Responding to Clicks in RecyclerView

What if we want to let our users select an item from the presented list? To achieve that, we need to communicate clicks back to our app.

The first step in implementing click interaction is to capture clicks on items at the **`ViewHolder`** level.

To maintain separation between our view holder and the adapter, we define a nested **`OnClickListener`** interface in our view holder. We choose to define the interface within the view holder because they are tightly coupled. The interface will, in our case, have only one function. The purpose of this function is to inform the owner of the view holder about the clicks. The owner of a view holder is usually a Fragment or an Activity. Since we know that a view holder can be reused, we know that it can be challenging to define it at construction time in a way that would tell us which item was clicked (since that item will change over time with reuse). We work around that by passing the currently presented item back to the owner of the view holder on clicking. This means our interface would look like this:

```
interface OnClickListener {
    fun onClick(catData: CatUiModel)
}
```

We will also add this listener as a parameter to our **`ViewHolder`** constructor:

```
class CatViewHolder(
    containerView: View,
    private val imageLoader: ImageLoader,
    private val onClickListener: OnClickListener
) : ViewHolder(containerView) {
    .
    .
    .
}
```

It will be used like this:

```
containerView.setOnClickListener {
onClickListener.onClick(catData) }
```

Now, we want our adapter to pass in a listener. In turn, that listener will be responsible for informing the owner of the adapter of the click. This means our adapter, too, would need a nested listener interface, quite similar to the one we implemented in our view holder.

While this seems like duplication that can be avoided by reusing the same listener, that is not a great idea, as it leads to tight coupling between the view holder and the adapter through the listener. What happens when you want your adapter to also report other events through the listener? You would have to handle those events coming from the view holder, even though they would not actually be implemented in the view holder.

Finally, to handle the click event and show a dialog, we define a listener in our activity and pass it to our adapter. We set that listener to show a dialog on clicking. In an MVVM implementation, you would be notifying **`ViewModel`** of the click at this point instead. **`ViewModel`** would then update its state, telling the view (our activity) that it should display the dialog.

## Exercise 6.03: Responding to Clicks

Your app already shows the user a list of secret cat agents. It is time to allow your user to choose a secret cat agent by clicking on its view. Click events are delegates from the view holder to the adapter to the activity, as shown in *Figure 6.9*:

Figure 6.9: The flow of click events

The following are the steps that you need to follow to complete this exercise:

1. Open your **`CatViewHolder.kt`** file. Add a nested interface to it right before the final closing curly bracket:

   ```
         interface OnClickListener {
             fun onClick(catData: CatUiModel)
         }
   ```

   This will be the interface a listener would have to implement in order to register for click events on individual cat items.
2. Update the **`CatViewHolder`** constructor to accept **`OnClickListener`** and make containerView accessible:

   ```
   class CatViewHolder(
       private val containerView: View,
       private val imageLoader: ImageLoader,
       private val onClickListener: OnClickListener
   ) : ViewHolder(containerView) {
   ```

   Now, when constructing a **`CatViewHolder`** constructor, you also register for clicks on item views.
3. At the top of your **`bindData(CatUiModel)`** function, add the following to intercept clicks and report them to the provided listener:

   ```
   containerView.setOnClickListener {
   onClickListener.onClick(catData) }
   ```

4. Now, open your **`CatsAdapter.kt`** file. Add this nested interface right before the final closing curly bracket:

   ```
   interface OnClickListener {
       fun onItemClick(catData: CatUiModel)
   }
   ```

   This defines the interface that listeners would have to implement to receive item click events from the adapter.
5. Update the **`CatsAdapter`** constructor to accept a call implementing the **`OnClickListener`** adapter you just defined:

   ```
   class CatsAdapter(
       private val layoutInflater: LayoutInflater,
   ```

```
    private val imageLoader: ImageLoader,
    private val onClickListener: OnClickListener
) : RecyclerView.Adapter<CatViewHolder>() {
```

6. In **`onCreateViewHolder(ViewGroup, Int)`**, update the creation of the view holder, as follows:

```
return CatViewHolder(view, imageLoader, object :
CatViewHolder.OnClickListener {
override fun onClick(catData: CatUiModel) =
onClickListener.onItemClick(catData)
})
```

   This will add an anonymous class that delegates **`ViewHolder`** click events to the adapter listener.

7. Finally, open your **`MainActivity.kt`** file. Update your **`catsAdapter`** construction as follows to provide the required dependencies to the adapter in the form of an anonymous listener handling click events by showing a dialog:

```
    private val catsAdapter by lazy {
        CatsAdapter(
            layoutInflater,
            GlideImageLoader(this),
            object : CatsAdapter.OnClickListener {
            override fun onClick(catData: CatUiModel)
= onClickListener.onItemClick(catData)
            }
        )
    }
```

8. Add the following function right before the final closing curly bracket:

```
    private fun showSelectionDialog(catData:
CatUiModel) {
        AlertDialog.Builder(this)
            .setTitle("Agent Selected")
            .setMessage("You have selected agent
${catData.name}")
```

```
            .setPositiveButton("OK") { _, _ -> }
            .show()
    }
```

This function will show a dialog with the name of the cat whose data was passed in.

9. Make sure to import the right version of **`AlertDialog`**, which is **`androidx.appcompat.app.AlertDialog`**, not **`android.app.AlertDialog`**. This is usually a better choice to support backward compatibility.
10. Run your app. Clicking on one of the cats should now show a dialog:

Figure 6.10: A dialog showing an agent was selected

Try clicking the different items and note the different messages presented. You now know how to respond to users clicking on items inside your **`RecyclerView`**. Next, we will look at how we can support different item types in our lists.

## Supporting Different Item Types

In the previous sections, we learned how to handle a list of items of a single type (in our case, all our items were **`CatUiModel`**). What happens if you want to support more than one type of item? A good example of this would be having group titles within our list.

Let's say that instead of getting a list of cats, we instead get a list containing happy cats and sad cats. Each of the two groups of cats is preceded by a title of the corresponding group. Instead of a list of **`CatUiModel`** instances, our list would now contain **`ListItem`** instances. **`ListItem`** might look like this:

```
sealed class ListItem {
    data class Group(val name: String) : ListItem()
    data class Cat(val data: CatUiModel) : ListItem()
}
```

Our list of items may look like this:

```
listOf(
    ListItem.Group("Happy Cats"),
    ListItem.Cat(
        CatUiModel(
            Gender.Female,
            CatBreed.AmericanCurl,
            "Kitty",
            "Kitty is warm and fuzzy.",
            "https://cdn2.thecatapi.com/images/..."
        )
    ),
    ListItem.Cat(
        CatUiModel(
            Gender.Male,
            CatBreed.ExoticShorthair,
            "Joey",
            "Loves to cuddle.",
            "https://cdn2.thecatapi.com/images/..."
        )
    ),
    ListItem.Group("Sad Cats"),
    ListItem.Cat(
        CatUiModel(
            Gender.Unknown,
            CatBreed.AmericanCurl,
            "Ginger",
            "Just not in the mood.",
            "https://cdn2.thecatapi.com/images/..."
        )
    ),
    ListItem.Cat(
        CatUiModel(
            Gender.Female,
            CatBreed.ExoticShorthair,
            "Butters",
            "Sleeps most of the time.",
            "https://cdn2.thecatapi.com/images/..."
        )
    )
)
```

In this case, having just one layout type will not do. Luckily, as you may have noticed in our earlier exercises, **`RecyclerView.Adapter`** provides us with a mechanism to handle this (remember the **`viewType`** parameter used in the **`onCreateViewHolder(ViewGroup, Int)`** function?).

To help the adapter determine which view type is needed for each item, we override its **`getItemViewType(Int)`** function. An example of an implementation that would do the trick for us is the following:

```
override fun getItemViewType(position: Int) = when
(listData[position]) {
    is ListItem.Group -> VIEW_TYPE_GROUP
    is ListItem.Cat -> VIEW_TYPE_CAT
}
```

Here, **`VIEW_TYPE_GROUP`** and **`VIEW_TYPE_CAT`** are defined as follows:

```
private const val VIEW_TYPE_GROUP = 0
private const val VIEW_TYPE_CAT = 1
```

This implementation maps the data type at a given position to a constant value representing one of our known layout types. In our case, we know of titles and cats, thus the two types. The values we use can be any integer values, as they're passed back to us as is in the **`onCreateViewHolder(ViewGroup, Int)`** function. All we need to do is make sure not to repeat the same value more than once.

Now that we have told the adapter which view types are needed and where, we also need to tell it which view holder to use for each view type. This is done by implementing the **`onCreateViewHolder(ViewGroup, Int)`** function:

```
override fun onCreateViewHolder(parent: ViewGroup,
viewType: Int) = when (viewType) {
    VIEW_TYPE_GROUP -> {
        val view =
layoutInflater.inflate(R.layout.item_title,
          parent, false)
        GroupViewHolder(view)
    }
    VIEW_TYPE_CAT -> {
        val view =
layoutInflater.inflate(R.layout.item_cat, parent,
false)
        CatViewHolder(view, imageLoader, object :
          CatViewHolder.OnClickListener {
            override fun onClick(catData: CatUiModel) =
onClickListener.onItemClick(catData)
        })
    }
    else -> throw IllegalArgumentException("Unknown
view type requested: $viewType")
}
```

Unlike the earlier implementations of this function, we now take the value of **`viewType`** into account.

As we now know, **`viewType`** is expected to be one of the values we returned from **`getItemViewType(Int)`**.

For each of these values (**`VIEW_TYPE_GROUP`** and **`VIEW_TYPE_CAT`**), we inflate the corresponding layout and construct a suitable view holder. Note that we never expect to receive any other value, and so throw an exception if such a value is encountered. Depending on your needs, you could instead return a default view holder with a layout showing an error or nothing at all. It may also be a good idea to log such values to allow you to investigate why you received them and decide how to handle them.

For our title layout, a simple **`TextView`** may be sufficient. The **`item_cat.xml`** layout can remain as is.

Now onto the view holder. We need to create a view holder for the title. This means we will now have two different view holders. However, our adapter only supports one adapter type. The easiest solution is to define a common view holder that both **`GroupViewHolder`** and **`CatViewHolder`** will extend. Let's call it **`ListItemViewHolder`**. The **`ListItemViewHolder`** class can be abstract, as we never intend to use it directly. To make it easy to bind data, we can also introduce a function in our abstract view holder—**`abstract fun bindData(listItem: ListItemUiModel)`**. Our concrete implementations can expect to receive a specific type, and so we can add the following lines to both **`GroupViewHolder`** and **`CatViewHolder`**, respectively:

```
require(listItem is ListItemUiModel.Cat) {
    "Expected ListItemUiModel.Cat"
}
```

We can also add the following:

```
require(listItem is ListItemUiModel.Cat) { "Expected
ListItemUiModel.Cat" }
```

Specifically, in **`CatViewHolder`**, thanks to some Kotlin magic, we can then use **`define val catData = listItem.data`** and leave the rest of the class unchanged.

Having made those changes, we can now expect to see the **`Happy Cats`** and **`Sad Cats`** group titles, each followed by the relevant cats.

## Exercise 6.04: Adding Titles to RecyclerView

We now want to be able to present our secret cat agents in two groups: active agents that are available for us to deploy to the field, and sleeper agents, which cannot currently be deployed. We will do that by adding a title above the active agents and another above the sleeper agents:

1. Under **`com.example.myrecyclerviewapp.model`**, create a new Kotlin file called **`ListItemUiModel`**.
2. Add the following to the **`ListItemUiModel.kt`** file, defining our two data types—titles and cats:

   ```
   sealed class ListItemUiModel {
       data class Title(val title: String) :
   ListItemUiModel()
       data class Cat(val data: CatUiModel) :
   ListItemUiModel()
   }
   ```
3. Create a new Kotlin file in **`com.example.myrecyclerviewapp`** named **`ListItemViewHolder`**. This will be our base view holder.
4. Populate the **`ListItemViewHolder.kt`** file with the following:

   ```
   abstract class ListItemViewHolder(
       containerView: View
   ```

```
) : RecyclerView.ViewHolder(containerView) {
    abstract fun bindData(listItem: ListItemUiModel)
}
```

5. Open the **CatViewHolder.kt** file.
6. Make **CatViewHolder** extend **ListItemViewHolder**:

```
class CatViewHolder(
    ...
) : ListItemViewHolder(containerView) {
```

7. Replace the **bindData(CatUiModel)** parameter with **ListItemUiModel** and make it override the **ListItemViewHolder** abstract function:

```
    override fun bindData(listItem: ListItemUiModel)
```

8. Add the following two lines to the top of the **bindData(ListItemUiModel)** function to enforce casting **ListItemUiModel** to **ListItemUiModel.Cat** and to fetch the cat data from it:

```
require(listItem is ListItemUiModel.Cat) {
  "Expected ListItemUiModel.Cat" }
val catData = listItem.data
```

Leave the rest of the file untouched.

9. Create a new layout file. Name your layout **item_title**.
10. Replace the default content of the newly created **item_title.xml** file with the following:

```
<?xml version="1.0" encoding="utf-8"?>
<TextView
xmlns:android="http://schemas.android.com/apk/res/and
roid"
    xmlns:app="http://schemas.android.com/apk/res-
auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:id="@+id/item_title_title"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:padding="8dp"
    android:textSize="16sp"
    android:textStyle="bold"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toTopOf="parent"
    tools:text="Sleeper Agents" />
```

This new layout, containing only a **`TextView`** with a 16sp-sized bold font, will host our titles:

Figure 6.11: Preview of the item_title.xml layout

11. Implement **`TitleViewHolder`** in a new file with the same name under **`com.example.myrecyclerviewapp`**:

```
class TitleViewHolder(
    containerView: View
) : ListItemViewHolder(containerView) {
    private val titleView: TextView
        by lazy { containerView
.findViewById(R.id.item_title_title) }
    override fun bindData(listItem: ListItemUiModel)
{
        require(listItem is ListItemUiModel.Title) {
            "Expected ListItemUiModel.Title"
        }
        titleView.text = listItem.title
    }
}
```

This is very similar to **CatViewHolder**, but since we only set the text on **TextView**, it is also much simpler.

12. Now, to make things tidier, select **CatViewHolder**, **ListItemViewHolder**, and **TitleViewHolder**.
13. Move all the files to a new namespace: right-click on one of the files, and then select **Refactor** | **Move** (or press *F6*).
14. Append **/viewholder** to the prefilled **To directory** field. Leave **Search references** and **Update package directive (Kotlin files)** checked and **Open moved files in editor** unchecked. Click **OK**.
15. Open the **CatsAdapter.kt** file.
16. Now, rename **CatsAdapter** to **ListItemsAdapter**. It is important to maintain the naming of variables, functions, and classes to reflect their actual usage to avoid future confusion. Right-click on the **CatsAdapter** class name in the code window, then select **Refactor** | **Rename** (or *Shift* + *F6*).
17. When **CatsAdapter** is highlighted, type **ListItemsAdapter** and press *Enter*.
18. Change the adapter generic type to **ListItemViewHolder**:

```
class ListItemsAdapter(
    ...
) : RecyclerView.Adapter<ListItemViewHolder>() {
```

19. Update **listData** and **setData(List<CatUiModel>)** to handle **ListItemUiModel** instead:

```
    private val listData =
mutableListOf<ListItemUiModel>()
    fun setData(listData: List<ListItemUiModel>) {
        this.listData.clear()
        this.listData.addAll(listData)
        notifyDataSetChanged()
    }
```

20. Update **onBindViewHolder(CatViewHolder)** to comply with the adapter contract change:

```
    override fun onBindViewHolder(holder:
ListItemViewHolder, position: Int) {
        holder.bindData(listData[position])
    }
```

21. At the top of the file, after the imports and before the class definition, add the view type constants:

```
private const val VIEW_TYPE_TITLE = 0
private const val VIEW_TYPE_CAT = 1
```

22. Implement **`getItemViewType(Int)`** like so:

```
    override fun getItemViewType(position: Int) =
when (listData[position]) {
        is ListItemUiModel.Title -> VIEW_TYPE_TITLE
        is ListItemUiModel.Cat -> VIEW_TYPE_CAT
    }
```

23. Lastly, change your **`onCreateViewHolder(ViewGroup, Int)`** implementation, as follows:

```
    override fun onCreateViewHolder(parent:
ViewGroup, viewType: Int) = when (viewType) {
        VIEW_TYPE_TITLE -> {
            val view =
layoutInflater.inflate(R.layout.item_title, parent,
false)
            TitleViewHolder(view)
        }
        VIEW_TYPE_CAT -> {
            val view =
layoutInflater.inflate(R.layout.item_cat, parent,
false)
            CatViewHolder(
                view,
                imageLoader,
                object :
CatViewHolder.OnClickListener {
```

```
                    override fun onClick(catData: CatUiModel) =
                        onClickListener.onItemClick(catData)
                })
        }
        else -> throw IllegalArgumentException("Unknown view type requested: $viewType")
    }
```

24. Update **`MainActivity`** to populate the adapter with appropriate data, replacing the previous **`catsAdapter.setData(List<CatUiModel>)`** call. (Please note that the code below has been truncated for space. Refer to the link below to access the full code that you need to add.)

MainActivity.kt

```
        listItemsAdapter.setData(
            listOf(
                ListItemUiModel.Title("Sleeper Agents"),
                ListItemUiModel.Cat(
                    CatUiModel(
                        Gender.Male,
                        CatBreed.ExoticShorthair,
                        "Garvey",
                        "Garvey is as a lazy, fat, and cynical orange cat.",
                        "https://cdn2.thecatapi.com/images/FZpeiLi4n.jpg"
                    )
                ),
                ListItemUiModel.Cat(
                    CatUiModel(
                        Gender.Unknown,
                        CatBreed.AmericanCurl,
                        "Curious George",
                        "Award winning investigator",
                        "https://cdn2.thecatapi.com/images/vJB8rwfdX.jpg"
                    )
                ),
                ListItemUiModel.Title("Active Agents"),
```

*The complete code for this step can be found at* **http://packt.live/3icCrSt**.

25. Since **`catsAdapter`** is no longer holding **`CatsAdapter`** but **`ListItemsAdapter`**, rename it accordingly. Name it **`listItemsAdapter`**.
26. Run the app. You should see something similar to the following:

Figure 6.12: RecyclerView with the Sleeper Agents/Active Agents header views

As you can see, we now have titles above our two agent groups. Unlike the **`Our Agents`** title, these titles will scroll with our content. Next, we will learn how to swipe an item to remove it from **`RecyclerView`**.

## Swiping to Remove Items

In the previous sections, we learned how to present different view types. However, up until now, we have worked with a fixed list of items. What if you want to be able to remove items from the list? There are a few common mechanisms to achieve that—fixed delete buttons, swipe to delete, and long-click to select then a "click to delete" button, to name a few. In this section, we will focus on the "swipe to delete" approach.

Let's start by adding the deletion functionality to our adapter. To tell the adapter to remove an item, we need to indicate which item we want to remove. The simplest way to achieve this is by providing the position of the item. In our implementation, this will directly correlate to the position of the item in our **`listData`** list. So, our **`removeItem(Int)`** function should look like this:

```
fun removeItem(position: Int) {
    listData.removeAt(position)
    notifyItemRemoved(position)
}
```

*Note*

*Just like when setting data, we need to notify* **`RecyclerView`** *that the dataset has changed—in this case, an item was removed.*

Next, we need to define the swipe gesture detection. This is done by utilizing **`ItemTouchHelper`**. Now, **`ItemTouchHelper`** handles certain touch events, namely dragging and swiping, by reporting them to us via a callback. We handle these callbacks by implementing

**`ItemTouchHelper.Callback`**. Also, **`RecyclerView`** provides **`ItemTouchHelper.SimpleCallback`**, which takes away the writing of a lot of boilerplate code.

We want to respond to swipe gestures but ignore move gestures. More specifically, we want to respond to swipes to the right. Moving is used to reorder items, which is beyond the scope of this chapter. So, our implementation of **`SwipToDeleteCallback`** will look as follows:

```
inner class SwipeToDeleteCallback :
    ItemTouchHelper.SimpleCallback(0,
ItemTouchHelper.RIGHT) {
    override fun onMove(
        recyclerView: RecyclerView,
        viewHolder: RecyclerView.ViewHolder,
        target: RecyclerView.ViewHolder
    ): Boolean = false
    override fun getMovementFlags(
        recyclerView: RecyclerView,
        viewHolder: RecyclerView.ViewHolder
    ) = if (viewHolder is CatViewHolder) {
        makeMovementFlags(
            ItemTouchHelper.ACTION_STATE_IDLE,
            ItemTouchHelper.RIGHT
        )or makeMovementFlags(
            ItemTouchHelper.ACTION_STATE_SWIPE,
            ItemTouchHelper.RIGHT
        )
    } else {
        0
    }
    override fun onSwiped(viewHolder:
RecyclerView.ViewHolder,
      direction: Int) {
        val position = viewHolder.adapterPosition
```

```
            removeItem(position)
        }
    }
```

Because our implementation is tightly coupled to our adapter and its view types, we can comfortably define it as an inner class. The benefit we gain is the ability to directly call methods on the adapter.

As you can see, we return **`false`** from the **`onMove(RecyclerView, ViewHolder, ViewHolder)`** function. This means we ignore move events.

Next, we need to tell **`ItemTouchHelper`** which items can be swiped. We achieve this by overriding **`getMovementFlags(RecyclerView, ViewHolder)`**. This function is called when a user is about to start a drag or swipe gesture. **`ItemTouchHelper`** expects us to return the valid gestures for the provided view holder. We check the **`ViewHolder`** class, and if it is **`CatViewHolder`**, we want to allow swiping—otherwise, we do not. We use **`makeMovementFlags(Int, Int)`**, which is a helper function used to construct flags in a way that **`ItemTouchHelper`** can decipher them. Note that we define rules for **`ACTION_STATE_IDLE`**, which is the starting state of a gesture, thus allowing a gesture to start from the left or the right. We then combine it (using **`or`**) with the **`ACTION_STATE_SWIPE`** flags, allowing the ongoing gesture to swipe left or right. Returning **`0`** means neither swiping nor moving will occur for the provided view holder.

Once a swipe action is completed, **`onSwiped(ViewHolder, Int)`** is called. We then obtain the position from the passed-in view holder by calling **`adapterPosition`**. Now, **`adapterPosition`** is important because it is the only reliable way to obtain the real position of the item presented by the view holder.

With the correct position, we can remove the item by calling **`removeItem(Int)`** on the adapter.

To expose our newly created **SwipeToDeleteCallback** implementation, we define a read-only variable within our adapter, namely **swipeToDeleteCallback**, and set it to a new instance of **SwipeToDeleteCallback**.

Finally, to plug our **callback** mechanism to **RecyclerView**, we need to construct a new **ItemTouchHelper** and attach it to our **RecyclerView**. We should do this when setting up our **RecyclerView**, which we do in the **onCreate(Bundle?)** function of our main activity. This is how the creation and attaching will look:

```
val itemTouchHelper =
ItemTouchHelper(listItemsAdapter.swipeToDeleteCallback)
itemTouchHelper.attachToRecyclerView(recyclerView)
```

We can now swipe items to remove them from the list. Note how our titles cannot be swiped, just as we intended.

You may have noticed a small glitch: the last item is cut off as it animates up. This is happening because **RecyclerView** shrinks to accommodate the new (smaller) number of items before the animation starts. A quick fix to this would be to fix the height of our **RecyclerView** by confining its bottom to the bottom of its parent.

## Exercise 6.05: Adding Swipe to Delete Functionality

We previously added **RecyclerView** to our app and then added items of different types to it. We will now allow users to delete some items (we want to let the users remove secret cat agents, but not titles) by swiping them left or right:

1. To add item removal functionality to our adapter, add the following function to **ListItemsAdapter** right after the **setData(List<ListItemUiModel>)** function:

```
    fun removeItem(position: Int) {
        listData.removeAt(position)
```

```
            notifyItemRemoved(position)
        }
```

2. Next, right before the closing curly bracket of your **`ListItemsAdapter`** class, add the following **`callback`** implementation to handle the user swiping a cat agent left or right:

```
        inner class SwipeToDeleteCallback :
            ItemTouchHelper.SimpleCallback(0,
    ItemTouchHelper.LEFT or ItemTouchHelper.RIGHT) {
            override fun onMove(
                recyclerView: RecyclerView,
                viewHolder: RecyclerView.ViewHolder,
                target: RecyclerView.ViewHolder
            ): Boolean = false
            override fun getMovementFlags(
                recyclerView: RecyclerView,
                viewHolder: RecyclerView.ViewHolder
            ) = if (viewHolder is CatViewHolder) {
                makeMovementFlags(
                    ItemTouchHelper.ACTION_STATE_IDLE,
                    ItemTouchHelper.LEFT or
    ItemTouchHelper.RIGHT
                ) or makeMovementFlags(
                    ItemTouchHelper.ACTION_STATE_SWIPE,
                    ItemTouchHelper.LEFT or
    ItemTouchHelper.RIGHT
                )
            } else {
                0
            }
            override fun onSwiped(viewHolder:
    RecyclerView.ViewHolder, direction: Int) {
                val position = viewHolder.adapterPosition
                removeItem(position)
            }
        }
```

We have implemented an **`ItemTouchHelper.SimpleCallback`** instance, passing in the directions we were interested in—**`LEFT`** and **`RIGHT`**. Joining the values is achieved by using the **`or`** Boolean operator.
We have overridden the **`getMovementFlags`** function to make sure we have only handled swiping on a cat agent view, not a title. Creating flags for both **`ItemTouchHelper.ACTION_STATE_SWIPE`** and **`ItemTouchHelper.ACTION_STATE_IDLE`** allows us to intercept both swipe and release events, respectively.
Once a swipe is completed (the user has lifted their finger from the screen), **`onSwiped`** will be called, and in response, we remove the item at the position provided by the dragged view holder.

3. At the top of your adapter, expose an instance of the **`SwipeToDeleteCallback`** class you just created:

```
class ListItemsAdapter(
    ...
) : RecyclerView.Adapter<ListItemViewHolder>() {
    val swipeToDeleteCallback =
SwipeToDeleteCallback()
```

4. Lastly, tie it all together by implementing **`ItemViewHelper`** and attaching it to our **`RecyclerView`**. Add the following code to the **`onCreate(Bundle?)`** function of your **`MainActivity`** file right after assigning the layout manager to your adapter:

```
    recyclerView.layoutManager = ...
    val itemTouchHelper =
ItemTouchHelper(listItemsAdapter
.swipeToDeleteCallback)
    itemTouchHelper.attachToRecyclerView(recyclerView
)
```

5. To address the small visual glitch you would get when items are removed, scale **`RecyclerView`** to fit the screen by updating the code in **`activity_main.xml`**, as follows. The changes are in **`RecyclerView`** tag, right before the **`app:layout_constraintTop_toBottomOf`** attribute:

```
        android:layout_height="0dp"
        app:layout_constraintBottom_toBottomOf="paren
t"
        app:layout_constraintTop_toBottomOf="@+id/mai
n_label" />
```

Note that there are two changes: we added a constraint of the bottom of the view to the bottom of the parent, and we set the layout height to **`0dp`**. The latter change tells our app to calculate the height of **`RecyclerView`** based on its constraints:

Figure 6.13: RecyclerView taking the full height of the layout

6. Run your app. You should now be able to swipe secret cat agents left or right to remove them from the list. Note that **`RecyclerView`** handles the collapsing animation for us:

Figure 6.14: A cat being swiped to the right

Note how even though titles are item views, they cannot be swiped. You have implemented a callback for swiping gestures that distinguishes between different item types and responds to a swipe by deleting the swiped item. Now we know how to remove items interactively. Next, we will learn how to add new items as well.

## Adding Items Interactively

We have just learned how to remove items interactively. What about adding new items? Let's look into it.

Similar to the way we implemented the removal of items, we start by adding a function to our adapter:

```
fun addItem(position: Int, item: ListItemUiModel) {
    listData.add(position, item)
    notifyItemInserted(position)
}
```

You will notice that the implementation is very similar to the **`removeItem(Int)`** function we implemented earlier. This time, we also receive an item to add and a position to add it. We then add it to our **`listData`** list and notify **`RecyclerView`** that we added an item in the requested position.

To trigger a call to **`addItem(Int, ListItemUiModel)`**, we could add a button to our main activity layout. This button could be as follows:

```
<Button
    android:id="@+id/main_add_item_button"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:text="Add A Cat"
    app:layout_constraintBottom_toBottomOf="parent" />
```

The app will now look like this:

Figure 6.15: The main layout with a button to add a cat

Don't forget to update your **`RecyclerView`** so that its bottom will be constrained to the top of this button. Otherwise, the button and **`RecyclerView`** will overlap.

In a production app, you could add a rationale around what a new item would be. For example, you could have a form for the user to fill in different details. For the sake of simplicity, in our example, we will always add the same dummy item—an anonymous female secret cat agent.

To add the item, we set **`OnClickListener`** on our button:

```
addItemButton.setOnClickListener {
    listItemsAdapter.addItem(
        1,
        ListItemUiModel.Cat(
            CatUiModel(
                Gender.Female,
                CatBreed.BalineseJavanese,
                "Anonymous",
                "Unknown",
                "https://cdn2.thecatapi.com/images/zJke
Hza2K.jpg"
            )
        )
    )
}
```

And that is it. We add the item at position 1 so that it is added right below our first title, which is the item at position 0. In a production app, you could have logic to determine the correct place to insert an item. It could be below the relevant title or always be added at the top, bottom, or in the correct place to preserve some existing order.

We can now run the app. We will now have a new **`Add A Cat`** button. Every time we click the button, an anonymous secret cat agent will be added to **RecyclerView**. The newly added cats can be swiped away to be removed, just like the hardcoded cats before them.

## Exercise 6.06: Implementing an "Add A Cat" Button

Having implemented a mechanism to remove items, it is time we implemented a mechanism to add items:

1. Add a function to **ListItemsAdapter** to support adding items. Add it below the **removeItem(Int)** function:
```
    fun addItem(position: Int, item: ListItemUiModel)
{
        listData.add(position, item)
        notifyItemInserted(position)
    }
```
2. Add a button to **activity_main.xml**, right after **RecyclerView** tag:
```
    <Button
        android:id="@+id/main_add_item_button"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="Add A Cat"
        app:layout_constraintBottom_toBottomOf="paren
t" />
```
3. You will notice that **android:text="Add A Cat"** is highlighted. If you hover your mouse over it, you will see that this is because of the hardcoded string. Click on the **Add** word to place the editor cursor over it.
4. Press *Option* + *Enter* (iOS) or *Alt* + *Enter* (Windows) to show the context menu, then *Enter* again to show the **Extract Resource** dialog.
5. Name the resource **add_button_label**. Press **OK**.
6. To change the bottom constraint on **RecyclerView** so that the button and **RecyclerView** do not overlap, within your **RecyclerView** tag, locate the following:

```
      app:layout_constraintBottom_toBottomOf="parent"
```

Replace it with the following line of code:

```
      app:layout_constraintBottom_toTopOf="@+id/main_ad
  d_item_button"
```

7. Add a lazy field holding a reference to the button at the top of the class, right after the definition of **recyclerView**:

```
      private val addItemButton: View
          by lazy {
  findViewById(R.id.main_add_item_button) }
```

   Notice **addItemButton** is defined as a View. This is because in our code we don't need to know the type of View to add a click listener to it. Choosing the more abstract type allows us to later change the type of the view in the layout without having to modify this code.

8. Lastly, update **MainActivity** to handle the click. Find the line that says the following:

```
          itemTouchHelper.attachToRecyclerView(recycler
  View)
```

   Right after it, add the following:

```
          addItemButton.setOnClickListener {
            listItemsAdapter.addItem(
              1,
              ListItemUiModel.Cat(
                CatUiModel(
                  Gender.Female,
                  CatBreed.BalineseJavanese,
                  "Anonymous",
                  "Unknown",
                  "https://cdn2.thecatapi.com/images/zJ
  keHza2K.jpg"
                )
              )
            )
```

   This will add a new item to **RecyclerView** every time the button is clicked.

9. Run the app. You should see a new button at the bottom of your app:

   Figure 6.16: An anonymous cat is added with the click of a button

10. Try clicking it a few times. Every time you click it, a new anonymous secret cat agent is added to your **`RecyclerView`**. You can swipe away the newly added cats just like you could the hardcoded ones.

In this exercise, you added new items to **`RecyclerView`** in response to user interaction. You now know how to change the contents of **`RecyclerView`** at runtime. It is useful to know how to update lists at runtime because quite often, the data you are presenting to your users changes while the app is running, and you want to present your users with a fresh, up-to-date state.

## Activity 6.01: Managing a List of Items

Imagine you want to develop a recipe management app. Your app would support sweet and savory recipes. Users of your app could add new sweet or savory recipes, scroll through the list of added recipes—grouped by flavor (sweet or savory)—click a recipe to get information about it, and finally, they could delete recipes by swiping them aside.

The aim of this activity is to create an app with **`RecyclerView`** that lists the title of recipes, grouped by flavor. **`RecyclerView`** will support user interaction. Each recipe will have a title, a description, and a flavor. Interactions will include clicks and swipes. A click will present the user with a dialog showing the description of the recipe. A swipe will remove the swiped recipe from the app. Finally, with two **`EditText`** fields (see *Chapter 3, Screens and UI*) and two buttons, the user can add a new sweet or savory recipe, respectively, with the title and description set to the values set in the **`EditText`** fields.

The steps to complete are as follows:

1. Create a new empty activity app.
2. Add **`RecyclerView`** support to the app's **`build.gradle`** file.
3. Add **`RecyclerView`**, two **`EditText`** fields, and two buttons to the main layout. Your layout should look something like this:

Figure 6.17: Layout with RecyclerView, two EditText fields, and two buttons

4. Add models for flavor titles and recipes, and an enum for flavor.
5. Add a layout for flavor titles.
6. Add a layout for recipe titles.
7. Add view holders for flavor titles and recipe titles, as well as an adapter.
8. Add click listeners to show a dialog with recipe descriptions.
9. Update **`MainActivity`** to construct the new adapter and hook up the buttons for adding new savory and sweet recipes. Make sure the form is cleared after the recipe is added.
10. Add a swipe helper to remove items.

The final output will be as follows:

Figure 6.18: The Recipe Book app

*Note*

*The solution to this activity can be found at:* **http://packt.live/3sKj1cp**

## Summary

In this chapter, we learned how to add **`RecyclerView`** to our project. We also learned how to add it to our layout and how to populate it with items. We went through adding different item types, which is particularly useful for titles. We covered interaction with **`RecyclerView`**: responding to clicks on individual items and responding to swipe gestures. Lastly, we learned how to dynamically add and remove items to and from **`RecyclerView`**. The world of **`RecyclerView`** is very rich, and we have only scratched the surface. Going further would be beyond the scope of this book. However, it is strongly recommended that you investigate it on your own so that you can have carousels, designed dividers, and fancier swipe effects in your apps. You could start your exploration here: **https://awesomeopensource.com/projects/recyclerview-adapter**.

In the next chapter, we will look into requesting special permissions on behalf of our app to enable performing certain tasks, such as accessing the user's contacts list or their microphone. We will also look into using Google's Maps API and accessing the user's physical location.