Updating the RecyclerView

We'll cover the following

- Creating an adapter
- Creating new views
- Updating the list

Creating an adapter

The `RecyclerView` uses an adapter to display each row of data. We'll use a new `AirportAdapter` class for this purpose, to display the status of each airport.

Create a new Kotlin class file named `AirportAdapter.kt` that will hold the class `AirportAdapter`. Let's start with this initial code for the class in this file:

```
package com.agiledeveloper.airports

import android.support.v7.widget.RecyclerView
import android.view.LayoutInflater
import android.view.View
import android.view.ViewGroup
import kotlinx.android.synthetic.main.airport_info.view.*

class AirportAdapter : RecyclerView.Adapter<AirportViewHolder>() {
}
class AirportViewHolder(itemView: View) : RecyclerView.ViewHolder(itemView) {
}
```

AirportAdapter.kt

An adapter inherits from `RecyclerView.Adapter<T>`, where the parametric type `T` represents a holder of a view for the data to be displayed. In our implementation, we specialize the parametric type to an `AirportViewHolder`. The `AirportViewHolder` class, in turn, inherits from `RecyclerView.ViewHolder`, which expects an instance of View into which the data will be displayed. The central approach is `RecyclerView` will call upon an adapter to create a view holder for each row. The view holder will take on the responsibility to appropriately display the data for each row. Let's focus on the implementation of the adapter and then take a look at the view

holder.

In the `AirportAdapter` class, let's first define a field to store the list of airports:

```
// AirportAdapter.kt
private val airports = mutableListOf<Airport>()
```

The `airports` field is initialized to an empty mutable list of `Airports`. We'll soon implement the `updateAirportsStatus()` function to modify the values in this list. The `RecyclerView` needs to know how many rows it should create. For this, we'll override the `getItemCount()` function of the base class:

```
// AirportAdapter.kt
override fun getItemCount() = airports.size + 1
```

In addition to displaying the status of each airport, we also want to display a header row. For this reason, we return the size of the collection of `Airports` plus one. Next, the adapter needs to produce a view holder for each row. This is done by overriding the `onCreateViewHolder()` function:

```
override fun onCreateViewHolder(
    parent: ViewGroup, position: Int): AirportViewHolder {

    val view = LayoutInflater.from(parent.context)
        .inflate(R.layout.airport_info, parent, false)

    return AirportViewHolder(view)
}
```

AirportAdapter.kt

Creating new views

We create a view using the layout we created earlier in the file `airport_info.xml` — the `TableRow` with `TextViews` for code, name, and so on. We then attach that view to a new instance of `AirportViewHolder` and return that instance.

The `RecyclerView` will use the view holder created by the `onCreateViewHolder()` function to display the data for each row. But it has to map or bind the data to be displayed with the view holder. We achieve this by overriding the `onBindViewHolder()` function:

```
// AirportAdapter.kt
```

```
override fun onBindViewHolder(viewHolder: AirportViewHolder, position: Int) {


  if (position > 0) viewHolder.bind(airports[position - 1])
}
```

If the value of position is equal to 0, then the default text we hard-coded in the layout—`Code`, `Name`, `Temp`, and `Delay`—should be displayed as the header. Otherwise, we need to bind the data in an `Airport` instance at that position in the list to the view holder. We delegate that responsibility to the `bind()` function of the view holder.

Updating the list

The last function we need in the `AirportAdapter` class is `updateAirportsStatus()`, which is responsible for taking the updated/new airport statuses and modifying the mutable list stored as a field within the adapter.

```
fun updateAirportsStatus(updatedAirports: List<Airport>) {
  airports.apply {
    clear()
    addAll(updatedAirports)
  }

  notifyDataSetChanged()
}
```

AirportAdapter.kt

In the `updateAirportsStatus()` function, we clear the existing list of `Airports`, add all the airports from the provided `updatedAirports`, and trigger a refresh of the `RecyclerView` by calling the `notifyDataSetChanged()` function. This function will result in the `RecyclerView` calling the `getItemCount()` to find the number of airports, then calling `onCreateViewHolder()` that many times to create as many view holders, then bind the view to the data using `onBindViewHolder()` for each row of data.

The only piece of code left unfinished is the `bind()` function of the view holder. Let's implement that now.

```
class AirportViewHolder(itemView: View) : RecyclerView.ViewHolder(itemView) {
  fun bind(airport: Airport) {
    val (code, name, delay, weather) = airport
    val clock = if (delay) "\uD83D\uDD52" else ""

    itemView.apply {
```

```
            airportCode.text = code
            airportName.text = name
            airportTemperature.text = weather.temperature.firstOrNull()
            airportDelay.text = clock
        }
    }
}
```

AirportAdapter.kt

The `bind()` function uses the destructuring syntax to fetch the four properties from the given instance of `Airport`. If the `delay` is true, then the variable `clock` is set to an ASCII value that represents a clock; otherwise, it's set to an empty string. Finally, we update each widget in the view, to display the code, name, temperature value, and delay, respectively.

That completes all the code necessary for the view. Compile the code and make sure there are no errors. If you run into any errors, refer to the code from the course's source code to find the differences and resolve.

In the next lesson, we'll see how this application works.