Implementing the Activity

We'll cover the following

Creating `MainActivity`

It's time to implement the code for the UI, starting with the landing page, which is the main activity. We'll first write the code to handle the airport code that the user enters into the text box and the button click. Then we'll write the code to populate the `RecyclerView` with data obtained from the web service. That appears like a lot of work, but, surprisingly, we won't need too much code to implement these actions. Let's get started, one step at a time.

Coroutines in the UI

As the first step, let's address head-on the need for coroutines in the UI. The UI will have to fetch the airport status from the web service, but we don't want the UI to block and become inactive when the application is in the middle of network calls. To address this, we've already designed the `getAirportStatus()` top-level function in `AirportStatus` to be non-blocking, using the `suspend` keyword. From Chapter 16, Exploring Coroutines, we know that functions marked with `suspend` can't be called from within arbitrary functions—they have to be called from within coroutines. There's a catch, however—the UI code created by the Android Studio doesn't use coroutines. So if we invoke `getAirportStatus()` from an event handler, the code will fail compilation. In short, we have to run the UI code in a coroutine context so that the event handlers can call functions that are marked with suspend.

We can easily run the UI code as a coroutine by making a small change to the generated `MainActivity` class—we'll implement Kotlin's `CoroutineScope` and override the `coroutineContext` property. With this change, the activity class can provide a coroutine scope for the execution of coroutines. Let's make this change in the `MainActivity.kt` file:

```
package com.agiledeveloper.airports
class MainActivity : AppCompatActivity(), CoroutineScope {
    override val coroutineContext: CoroutineContext
        get() = Dispatchers.Main

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)
    }
}
```

MainActivity.kt

The `onCreate()` function shown in the code was generated by the IDE when the project was created. The only changes we made to the class are the addition of the `CoroutineScope` implementation and the `coroutineContext` property. Note the fact that the `coroutineContext` property's `get()` is returning a reference to `Dispatchers.Main`—we'll discuss this further when we're ready to make an asynchronous call.

Implementing a text watcher

As you key in the code, the IDE will prompt you for the necessary imports. Accept the appropriate choice the IDE gives, and move forward.

Next, we'll define a field in the `MainActivity` class:

```
// MainActivity.kt
private val airportCodes = mutableListOf<String>()
```

The `airportCodes` field will hold a list of airport IATA codes the user will provide at runtime.

Next, we'll edit the `onCreate()` function, which currently has two lines of code.

As a first step, we want to disable the `Add` button so the user won't be able to click it until an airport code is entered. To disable the button, we need a reference to the button widget from the layout. The Kotlin Android integration makes this effortless by synthesizing the UI widgets as properties with the activity class. To enable this

by synthesizing the UI widgets as properties with the activity class. To enable this facility, in the top of the `MainActivity.kt` file, import the following:

```
// MainActivity.kt
import kotlinx.android.synthetic.main.activity_main.*
```

Now, in the `onCreate()` function, we can refer to the button from the layout using the `id` we provided for the button, namely `addAirportCode`, like so:

```
// MainActivity.kt
addAirportCode.isEnabled = false
```

We set the `isEnabled` property of the `addAirportCode` button to true if the `airportCode` text box has some text; otherwise we set it to false, disabling the button.

As soon as the user starts typing in the text box, we'll want to enable the button. To achieve this, we'll handle the text change event on the `EditText` by calling the `addTextChangedListener()` function.

```
airportCode.addTextChangedListener(object: TextWatcher {
    override fun afterTextChanged(s: Editable) {
    addAirportCode.isEnabled = airportCode.text.isNotBlank()
    }

    override fun beforeTextChanged(
    s: CharSequence, start: Int, count: Int, after: Int) { /* no-op */ }

    override fun onTextChanged(
    s: CharSequence, start: Int, before: Int, count: Int) { /* no-op */ }
})
```

MainActivity.kt

The `addTextChangedListener()` function takes an object of `TextWatcher` as argument. We use the concise Kotlin syntax to create an anonymous instance of `TextWatcher`. In this object's `afterTextChanged()` callback event we enable the button, using the handle in the `addAirportCode` field, if the `airportCode` `EditText`'s text box is non-empty. The other two callback methods of `TextWatcher` are intentionally left blank since we have no use for them.

Implementing a listener for the button

As the next step within `onCreate()`, we need to register an event handler for the button click event. In the callback for the button click, we'll take the airport code

entered by the user and add it to the list `airportCodes`, which resides in the first

field we created in the `MainActivity` class. Let's add the following code to the bottom of the `onCreate()` function.

```
addAirportCode.setOnClickListener {
    airportCodes.add(airportCode.text.toString())
    airportCode.setText("")

    launch {
    updateAirportStatus()
    }
}
```

MainActivity.kt

After adding the airport code to the list, we clear the `EditText`'s text box. If the newly added airport code is the first, we need to populate the `RecyclerView` with status for that airport. If airport codes already are in the list, then at this time, we can update the status for all the airports the user has requested information for. We'll use a yet-to-be-written function, `updateAirportStatus()`, to get the data from the `AirportStatus`'s `getAirportStatus()` function. Since the call to that function will be asynchronous, we'll have to mark `updateAirportStatus()` with suspend. That, in turn, will demand that the call to `updateAirportStatus()` be made from a coroutine from within that callback. But, there's a catch—let's dig in.

Asynchronous calls

If you take a look at the `getAirportStatus()` function in `AirportStatus.kt`, that function runs the asynchronous calls in the `Dispatchers.IO` thread. This makes good sense since calls to the web service are IO operations. However, when the response comes back from the calls, the UI code can't directly add the result to the UI components. The reason for this is the UI components aren't thread-safe, and accessing UI components from within arbitrary threads will result in an exception. In short, whereas the asynchronous calls run in the IO thread pool, we should run the UI updates in the `Main` thread. Thankfully, that's really easy to achieve, and in fact, we've already taken care of it.

When we call `launch()` in the callback, that coroutine will run in the `CoroutineScope` defined in `MainActivity`. Take a look at the definition of the `MainActivity` class—it implements `CoroutineScope`. The `coroutineContext` property is returning a `Dispatchers.Main` as the coroutine context. This option `Main` is only

available in the Android API—we discussed this in Explicitly Setting a Context. As a result, the code directly invoked within `launch()` will run concurrently in the UI's main thread—see Parallel vs. Concurrent. The calls to the web service, though, will run in parallel in the IO threads. Thus, the data fetch will happen in parallel from the IO threads, but the update of the UI will happen in the main threads concurrently with the user's interaction.

Configuring the `RecyclerView`

As a final step within the `onCreate()` function, we need to get the `RecyclerView` ready to display the statuses of the airports:

```
airportStatus.apply {
    setHasFixedSize(true)
    layoutManager = LinearLayoutManager(this@MainActivity)
    adapter = AirportAdapter()
}
```

MainActivity.kt

We configure the `RecyclerView` widget (connected to the id `airportStatus`) to have a fixed size, assign a layout manager to manage the child widgets, and connect an adapter (a yet-to-be-written `AirportAdapter`) to manage the display of the airport statuses.

We have one final task to complete in the `MainActivity` class—we need to implement the `updateAirportStatus()` function, which will trigger the redisplay of the `RecyclerView` when new airport statuses are received from `getAirportStatus()`.

We can obtain the airport status information from the `AirportStatus`'s top-level `getAirportStatus()` function. Once the data arrives within the `updateAirportStatus()` function, we can pass that to the adapter of the `RecyclerView` so it can take up the task of displaying the airport statuses. Here's the code to accomplish that:

```
private suspend fun updateAirportStatus() {
    val airports = getAirportStatus(airportCodes)
    val airportAdapter = airportStatus.adapter as AirportAdapter
    airportAdapter.updateAirportsStatus(airports)
}
```

MainActivity.kt

We obtain a sorted list of `Airports` by calling the `getAirportStatus()` top-level function from the `com.agiledeveloper.airport` package—this is the function we wrote in the `AirportStatus` file. The list of `Airports` is passed to the adapter using the `updateAirportStatus()`.

Next, it's time to implement the adapter.