CP3406_CP5307 Codelab 3.3: RecyclerView for scrollable lists

Contents

1. Overview

If you think about the apps you use most often on your phone, almost every app has at least one list. The call history screen, the Contacts app, your favorite social media app - they all display a list of data. Some of these apps show a simple list of words or phrases, while others may show more complex items, such as text and image cards. Regardless of what the content is, displaying a list of data is one of Android's most common UI tasks.

To help you create apps with lists, Android provides `RecyclerView`. `RecyclerView` is designed to be very efficient, even with large lists, as it can reuse or recycle views that move off the screen. When a list item is moved off the screen, `RecyclerView` reuses the view of the next item to be displayed. This means that the element is filled with new content that moves towards the screen. From the perspective of the user, it looks as though the list is scrolling through many items, but in behind-the-scenes Android is simply reusing the same set of Views to display the list of data. This behavior of `RecyclerView` saves a lot of processing time and helps lists move more smoothly.

In this codelab, you will compile the Affirmations app. It is a simple app that shows ten positive affirmations as text on a scrollable list. Then, as a challenge, you should attempt to modify the app so that it can display some images or other data alongside each affirmation.

2. Create the project

1. Start a new project in Android Studio with the **Empty Views Activity** template.
2. Enter **Affirmations** as the app's name, **au.edu.jcu.affirmations** as the package name, choose **Kotlin** as the language and choose **API Level 24** as the minimum SDK.
3. Click **Finish** to create the project.

3. How to set up the data list

The next step to create the Affirmations app is to add resources. You will add the following to the project:

- String resources to represent the affirmations
- A data source to provide the list of affirmations to the app

Note: In most production Android apps, data such as this would be retrieved from a database or server. We will cover persistent data storage and connectivity in later classes, so to keep things simple we will store the affirmation strings within the app.

How to add the Affirmation strings

1. In `strings.xml` (in the res/values folder), add the following affirmations as individual strings resources. Assign each one the name `affirmation1`, `affirmation2` and so on (there are 10 in total).

Affirmations:

```
I am strong.
I believe in myself.
Each day is a new opportunity to grow and be a better version of myself.
Every challenge in my life is an opportunity to learn from.
I have so much to be grateful for.
Good things are always coming into my life.
New opportunities await me at every turn.
I have the courage to follow my heart.
Things will unfold at precisely the right time.
I will be present in all the moments that this day brings.
```

When you're done, the file `strings.xml` should look like this:.

```
<resources>
    <string name="app_name">Affirmations</string>
    <string name="affirmation1">I am strong.</string>
    <string name="affirmation2">I believe in myself.</string>
    <string name="affirmation3">Each day is a new opportunity to grow and
be a better version of myself.</string>
    <string name="affirmation4">Every challenge in my life is an
opportunity to learn from.</string>
    <string name="affirmation5">I have so much to be grateful for.</string>
    <string name="affirmation6">Good things are always coming into my life.
</string>
    <string name="affirmation7">New opportunities await me at every turn.
</string>
    <string name="affirmation8">I have the courage to follow my heart.
</string>
    <string name="affirmation9">Things will unfold at precisely the right
time.</string>
    <string name="affirmation10">I will be present in all the moments that
this day brings.</string>
</resources>
```

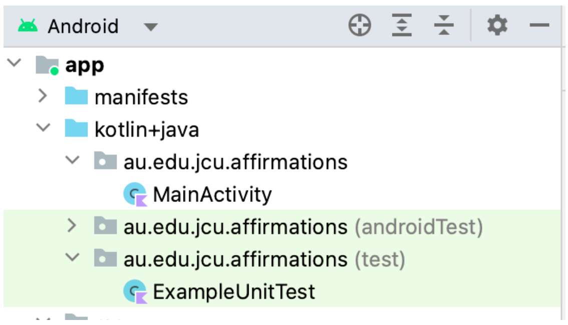
Remember, now that you have added string resources, you can refer to them in your code using their ID, e.g., `R.string.affirmation1, R.string.affirmation2` etc.

Create a new package

It is a good idea to organize your code logically so that you and other developers can easily understand, maintain and distribute it. Just as you can organize the papers in files and folders, you can organize your code in files and packages.

What is a package?

1. In Android Studio, in the **Project** window (**Android**), check the files of your new project in **app > kotlin+java** for the Affirmations app. They should be similar to the screenshot shown below, where three packages appear, one for your code (**au.edu.jcu.affirmations**) and two for the test files (**au.edu.jcu.affirmations (androidTest)** and **au.edu.jcu.affirmations (test)**).



```
Android
app
  manifests
  kotlin+java
    au.edu.jcu.affirmations
      MainActivity
    au.edu.jcu.affirmations (androidTest)
    au.edu.jcu.affirmations (test)
      ExampleUnitTest
```

2. Note that the name of the package consists of several words separated by one point.
(If your package is not displayed on one line, and individual nested folders are showing, click settings (the cog icon at top of this pane) and select *Tree Appearance* > *Compact Middle Packages*.

There are two ways to use packages:

- Create different packages for different parts of your code. For example, developers often separate classes that work with the data and classes that compile UI into different packages.
- Use code from other packages in your code. To use the classes of other packages, you must define them in the dependencies of your compilation system. It is also recommended to use `import` to import them into your code so you can use the short names (e.g., `TextView`) instead of fully qualified names (e.g. `android.widget.TextView`). For example, you've already used `import` for classes like `View` (`import android.view.View`).

In the Affirmations app, in addition to importing Android and Kotlin classes, you will organize the app in several packages. Even if you don't have many classes for your app, we recommend using packages to group classes by functionality. It is a good habit, and will help you to organize your project when you begin to work on more complex apps.

How to name packages

You can assign any name to a package, provided it is unique globally. No other package can have the same name. Given these constraints, programmers use conventions to make it easier to create and understand package names.

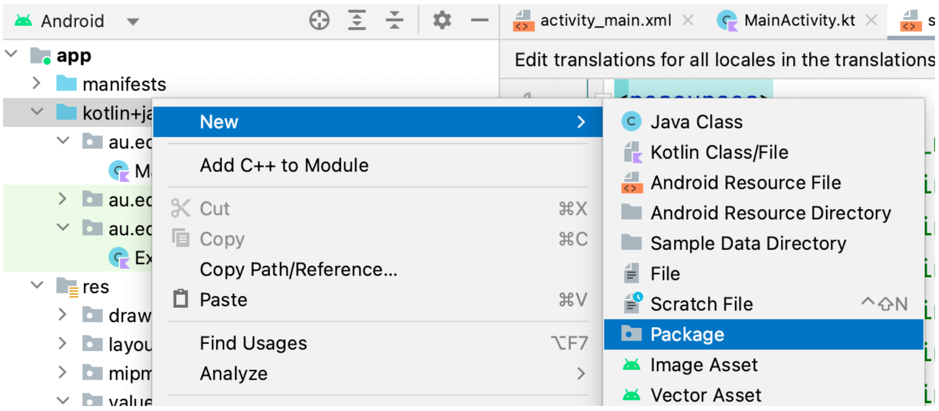
- The name of the package is usually structured from general to specific, and each part of the name is in lowercase letters and separated by one point. Important: The point is just part of the name. It does not indicate a hierarchy in the code or the structure of a folder.
- Because Internet domains are unique globally, it is convention that you use a domain, in general, yours or your company, as the first part of the name (e.g., au.jcu.edu).
- You can choose the names of the packages to indicate what's inside and how they relate to each other.

Below are some examples of predefined package names and their content:

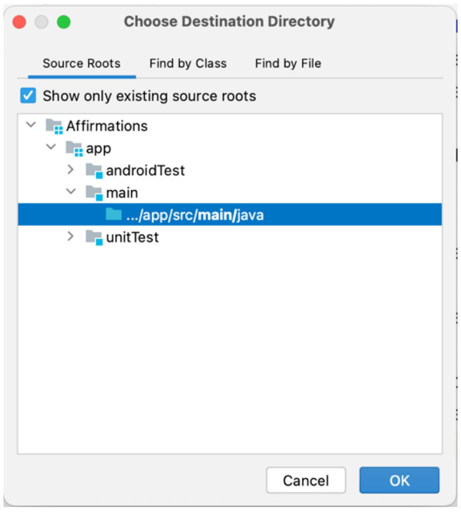
- `kotlin.math`: Functions and constant mathematics.
- `android.widget`: Views, as `TextView`.

How to create a package

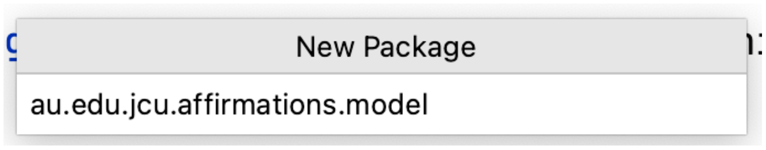
1. On Android Studio, on the **Project** panel, right-click on **app > kotlin+java** and select **New Package**.



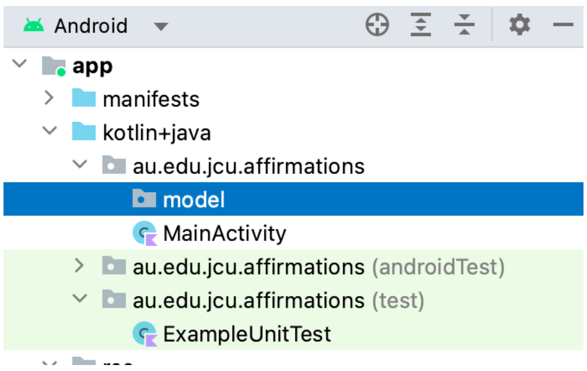
2. In the **Choose Destination Directory** pop-up window, leave the default option select and click OK.



3. A little popup window will appear for you to type in the new package name. Developers often use **model** as the package name for classes that model (or represent) the data. So enter **au.edu.jcu.affirmations.model** and then press Enter.



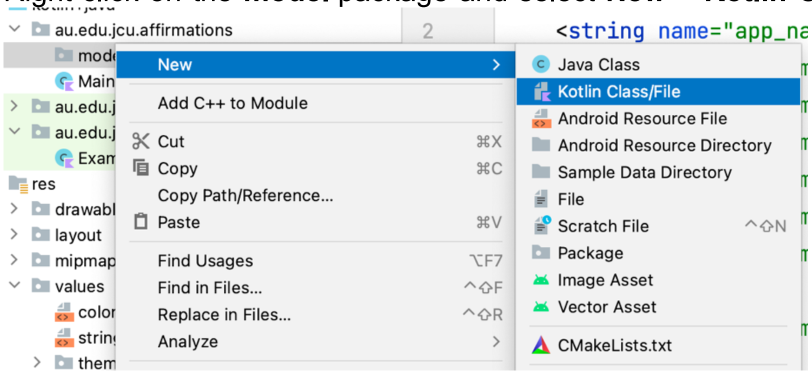
4. A new package will be created in the **au.edu.jcu.affirmations** package. This new package will contain any class related to the data defined in your app.



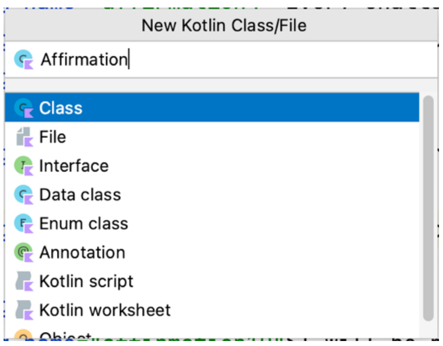
How to create the Affirmation data class

In this task, you'll create a new class called `Affirmation`. An object instance of `Affirmation` represent one of our affirmations and contains the string's resource ID along with the text.

1. Right-click on the **model** package and select **New > Kotlin Class/File**.



2. In the pop-up window, select **Class** and then enter `Affirmation` as the class name. Press Enter. This creates a new file called `Affirmation.kt` in the package `model`.



3. To indicate that `Affirmation` represents some data, add the keyword `data` before the class definition. This causes an error, as data classes must have at least one defined property.

```
package au.edu.jcu.affirmations.model

data class Affirmation {
}
```

When you create an instance of `Affirmation`, you must pass the resource ID for the affirmation. The resource ID is an integer.

4. Add an integer parameter `val stringResourceId` to the constructor of the class `Affirmation`.

This fixes the error.

```
package au.edu.jcu.affirmations.model

data class Affirmation(val stringResourceId: Int)
```

How to create a Data Source class

The data shown in an app could come from different sources (e.g., within your app project or from an external source that requires Internet connection to download data). As a result, the data may not be in the exact format you need. The rest of the app should not have to worry about the location of the data or the original format. You can and you should hide this data preparation in a separate class `Datasource` that prepares the data for the app. As data preparation is a different concern, we should place the class `Datasource` in a separate **data** package.

1. Similar to our **model** package, create a new package **au.edu.jcu.affirmations.data**.
2. Right-click on the package `data` and select **New Kotlin Class/File**.
3. Enter `Datasource` as the class name.
4. Inside the class `Datasource` creates a function called `loadAffirmations()`.

The function `loadAffirmations()` must return our list of `Affirmations`. To do this, create a list with an instance `Affirmation` for each of our string resources.

5. Declare `List<Affirmation>` as the return type of the method `loadAffirmations()`.
6. In the body of `loadAffirmations()`, adds a `return` statement.
7. After the keyword `return`, call `listOf<>()` to create a `List`.
8. Inside the angular parentheses <>, specify the type of list items such as `Affirmation`.
9. Inside the parentheses, create an `Affirmation` and pass `R.string.affirmation1` as the resource ID, as shown below.

```
Affirmation(R.string.affirmation1)
```

10. Add all ten `Affirmation` objects to the list. The finished code should look as follows.

```
package au.edu.jcu.affirmations.data

import au.edu.jcu.affirmations.R
import au.edu.jcu.affirmations.model.Affirmation
class Datasource {
       fun loadAffirmations(): List<Affirmation> {
           return listOf<Affirmation>(
               Affirmation(R.string.affirmation1),
               Affirmation(R.string.affirmation2),
               Affirmation(R.string.affirmation3),
               Affirmation(R.string.affirmation4),
               Affirmation(R.string.affirmation5),
               Affirmation(R.string.affirmation6),
               Affirmation(R.string.affirmation7),
               Affirmation(R.string.affirmation8),
               Affirmation(R.string.affirmation9),
               Affirmation(R.string.affirmation10)
       )
   }
}
```

4. How to add a RecyclerView to your app

In this section, you will set up a `RecyclerView` to show the list of `Affirmations`.

There are several steps involved in the creation and use of `RecyclerView`. The following are the key components, and more information will be provided about each component as you implement it.

- **item:** A data item from the list to display. Represents an `Affirmation` object in your app.
- **Adapter:** Takes data and prepares it for `RecyclerView` to display.
- **ViewHolders:** A set of views for your `RecyclerView` to use and reuse it in order to display data.
- **RecyclerView:** A view on the screen.

How to add a RecyclerView to the design

The Affirmations app consists of a single activity called `MainActivity`, and a layout/design file called `activity_main.xml`. First, you should add the `RecyclerView` to the design of `MainActivity`.

1. Open up. `activity_main.xml`.
2. Since you will be working between the XML and the Design View, open the **Split view** option.
3. Delete the `TextView`.

The current design uses `ConstraintLayout`. `ConstraintLayout` is ideal and flexible when you want to position several secondary views in a design. Since your design will have only one secondary view, `RecyclerView`, you can switch to a simpler `ViewGroup`, in this case `FrameLayout`, which can be used to maintain a single secondary view.

4. In the XML file, replace `ConstraintLayout` with `FrameLayout`. The completed XML is shown below.

```
<?xml version="1.0" encoding="utf-8"?>
<FrameLayout xmlns:android="http://schemas.android.com/apk/res/android"
    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">
</FrameLayout>
```

5. Change to **Design** view.
6. From the **Palette**, select **Containers** and search for **RecyclerView**.
7. Drag a **RecyclerView** to the design.
8. If it appears, read the **Add Project Dependency** pop-up window and click **OK**. (If the pop-up window doesn't appear, no action is required.)
9. If necessary, change attributes `layout_width` and `layout_height` of `RecyclerView` to `match_parent` so that it can fill the whole screen.
10. Set the id attribute of `RecyclerView` to be `recycler_view`.

`RecyclerView` supports displaying elements in different ways, such as a linear list or grid. The organisation of the elements is controlled by a `LayoutManager`. The Android framework provides some basic layout managers we can use. The Affirmations app will display elements as a vertical list, for which we can use `LinearLayoutManager`.

11. Go back to **Code** view. In the XML code, within the element `RecyclerView`, add `app:layoutManager=”LinearLayoutManager”` as an attribute of the `RecyclerView` as shown below.

```
app:layoutManager="LinearLayoutManager"
```

For the user to be able to scroll through a list of items that extends beyond the screen, you must add a vertical scroll bar.

12. In `RecyclerView`, add an attribute `android:scrollbars` set to `vertical`.

```
android:scrollbars="vertical"
```

The XML in **activity_main.xml** should look as follows:

```
<FrameLayout xmlns:android="http://schemas.android.com/apk/res/android"
    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">
    <androidx.recyclerview.widget.RecyclerView
        android:id="@+id/recycler_view"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:scrollbars="vertical"
        app:layoutManager="LinearLayoutManager" />
</FrameLayout>
```

13. Run your app.

The project should compile and run smoothly. However, only a white background is displayed in the app because a crucial code fragment is missing. You have the source of the data and a `RecyclerView` added to your design, but `RecyclerView` is missing the instructions on how to show the `Affirmations`.

How to implement an Adapter for RecyclerView

Your app needs a way to take the data from `Datasource` and format them so that each `Affirmation` can be displayed as an element in `RecyclerView`.

Adapter is a design pattern that adapts data to something that you can use in a `RecyclerView`. In this case, you need an adapter that takes an instance of `Affirmation` from the list returned by `loadAffirmations()` and turn it into a list item view so that it can appear in the `RecyclerView`.

When you run the app, `RecyclerView` uses the adapter to decipher how to display your data on the screen. `RecyclerView` will ask the adapter to create a new view for the first data item in your list. Once it has created the view, it will ask to the adapter to provide the data to draw the item. This process is repeated until `RecyclerView` has covered the screen with items. If only three items fit the screen at once, `RecyclerView` only asks the adapter to prepare three views to displays those list items (instead of views for all 10 list items).

In this step, you will create an adapter for the `Affirmation` object so that it can be shown in the `RecyclerView`.

An adapter has several parts, and you'll need to write some “complex” code to achieve it. Don’t worry if you don't understand all the details at first. After completing the entire app with a `RecyclerView`, you will have a better understanding of how all the parts fit together. You can then use this code as a basis for future apps you create with a `RecyclerView`.

How to create a design for elements

Each element displayed in a `RecyclerView` has its own design, which is defined in a separate design file. Since only one string will be displayed in our list, you can use a `TextView` for the design of your element.

1. In **res/layout**, create a new empty **layout file** called `list_item.xml` and open in **Code** view.
2. Add a `TextView` with `id` of `item_title`.
3. Add `wrap_content` for `layout_width` and `layout_height`, as shown in the following code.

Note that you don't need a `ViewGroup` around your design here, as this list element design will be added as a secondary element of the parent element `RecyclerView`.

```
<?xml version="1.0" encoding="utf-8"?>
<TextView xmlns:android="http://schemas.android.com/apk/res/android"
    android:id="@+id/item_title"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content" />
```

How to Create the Adapter

1. As we did earlier, create a new package called **au.edu.jcu.affirmations.adapter**.
2. Right-click on the package `adapter` and select **New Kotlin Class/File**.
3. Name it `ItemAdapter`, click Enter, and open thew created file `ItemAdapter.kt`.

   You must add a parameter to the constructor of `ItemAdapter` so that you can pass the list of affirmations to the adapter.

4. Add a parameter to the constructor of `ItemAdapter`, that is a `val` named `dataset` of type `List<Affirmation>`.
5. Since `dataset` will only be used within this class, make it `private`.

```
import au.edu.jcu.affirmations.model.Affirmation
class ItemAdapter(private val dataset: List<Affirmation>) {
}
```

`ItemAdapter` need information on how to access strings resources. This and other information about the app is stored in an instance of the object `Context`, that you can provide to an instance of `ItemAdapter`.

6. Add a parameter to the constructor of `ItemAdapter`, that's a `val` named `context` of type `Context`. Set it as the first parameter in the constructor.

```
class ItemAdapter(private val context: Context, private val dataset:
List<Affirmation>) {
}
```

How to create a ViewHolder

`RecyclerView` does not interact directly with the views of elements, but with `ViewHolders`. A `ViewHolder` object represents a single view of a list item in `RecyclerView` and can be reused when possible. An instance `ViewHolder` contains references to individual views within a list element design. This makes it easy to update the list element view with new data.

1. Inside the class `ItemAdapter`, before the closing brace, create an inner class `ItemViewHolder`.

```
class ItemAdapter(private val context: Context, private val dataset:
List<Affirmation>) {
    class ItemViewHolder()
}
```

2. Add a private parameter named `view` of type `View` to the class constructor `ItemViewHolder`.
3. Make `ItemViewHolder` a subclass of `RecyclerView.ViewHolder` and passes the parameter `view` to the constructor of the superclass.
4. Inside `ItemViewHolder`, defines a property named `textView` of type `TextView`. Use `findViewById` with ID `item_title` that you defined in `list_item.xml`.

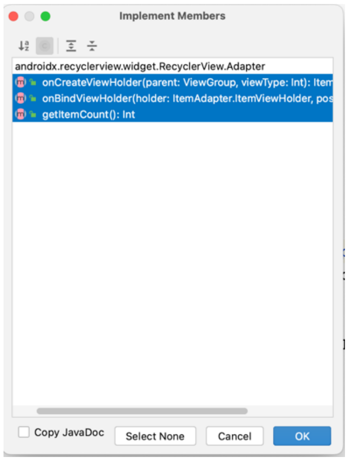
```
class ItemAdapter(private val context: Context, private val dataset:
List<Affirmation>) {
    class ItemViewHolder(private val view: View) :
RecyclerView.ViewHolder(view) {
        val textView: TextView = view.findViewById(R.id.item_title)
    }
}
```

How to implement adapter methods

1. Add the following line to specify `ItemAdapter.ItemViewHolder` as the type of container of views in angular brackets.

```
class ItemAdapter(
    private val context: Context,
    private val dataset: List<Affirmation>
) : RecyclerView.Adapter<ItemAdapter.ItemViewHolder>() {
    class ItemViewHolder(private val view: View) :
RecyclerView.ViewHolder(view) {
        val textView: TextView = view.findViewById(R.id.item_title)
    }
}
```

You'll see an error because you have to implement some abstract methods of `RecyclerView.Adapter`.


Implement Members

androidx.recyclerview.widget.RecyclerView.Adapter
onCreateViewHolder(parent: ViewGroup, viewType: Int): Item
onBindViewHolder(holder: ItemAdapter.ItemViewHolder, pos
getItemCount(): Int

2. Place the cursor in `ItemAdapter` and press **Command + I** (**Control + I** on Windows). This shows you the list of methods you should implement: `getItemCount()`, `onCreateViewHolder()` and `onBindViewHolder()`.

3. Select the three methods with and click **OK**.

This creates stubs with the parameters for the three methods, as shown below.

```
override fun onCreateViewHolder(parent: ViewGroup, viewType: Int):
ItemViewHolder {
    TODO("Not yet implemented")
}
override fun getItemCount(): Int {
    TODO("Not yet implemented")
}

override fun onBindViewHolder(holder: ItemViewHolder, position: Int) {
    TODO("Not yet implemented")
}
```

You shouldn't see any more errors. Now you must implement those methods so that they perform the right actions for your app.

How to implement getItemCount()

The method `getItemCount()` must return the size of your data set. Your app's data is stored in the property `dataset` that you pass to the constructor of `ItemAdapter` and you can get its size from `size`.

1. Replace `getItemCount()` with the following:

```
override fun getItemCount() = dataset.size
```

It's a more concise way of writing the following:

```
override fun getItemCount(): Int {
    return dataset.size
}
```

How to implement onCreateViewHolder()

The design manager calls the method `onCreateViewHolder()` to create new view interfaces for `RecyclerView` (e.g., when there are no existing view containers that can be reused). Remember that a view holder represents a single view for a list item.

The method `onCreateViewHolder()` takes two parameters and returns a new `ViewHolder`.

- The first parameter is `parent`, which is the group of views to which the new item view will be attached. The parent it attaches to is the `RecyclerView`.
- The second parameter is `viewType`, which becomes important when there are several types of views being used in a `RecyclerView`. If you have different list element designs in a single `RecyclerView`, you can only recycle views with the same type of view. In your case, there is only one list item design/type, so you don't have to worry about this parameter for now.

1. In the method `onCreateViewHolder()`, get an instance of `LayoutInflater` from the context provided. The will help us to "inflate" our layout for each list item.

```
val adapterLayout = LayoutInflater.from(parent.context)
```

2. When you have an object instance `LayoutInflater`, adds a point followed by another method call to inflate the list item view. Pass the XML design resource ID `R.layout.list_item` and the view group `parent`. The third Boolean argument is `attachToRoot`. This argument can be `false`, because `RecyclerView` will add this element to the view hierarchy when it is ready.

3. Now we can return a new instance of `ItemViewHolder` in which the root view is `adapterLayout`.

```
return ItemViewHolder(adapterLayout)
```

This is the code for `onCreateViewHolder()` so far:

```
override fun onCreateViewHolder(parent: ViewGroup, viewType: Int):
ItemViewHolder {
    // create a new view
    val adapterLayout = LayoutInflater.from(parent.context)
        .inflate(R.layout.list_item, parent, false)
    return ItemViewHolder(adapterLayout)
}
```

How to implement onBindViewHolder()

The last method you have to implement is `onBindViewHolder()`. The design manager calls this method to replace the contents of a list element view if the data is updated or it is being reused.

The method `onBindViewHolder()` has two parameters: `ItemViewHolder`, previously created by the method `onCreateViewHolder()` and an `int` represent the current element's `position` in the list. In this method, you will retrieve the `Affirmation` object from the data set according to position.

1. Inside `onBindViewHolder()` create a `val item` and get the item from the dataset at `position`.

```
val item = dataset[position]
```

Finally, you should update all the views to reflect the correct data of this element. In this case, there is only one view to update, which is the `TextView` inside `ItemViewHolder`. Set the text of the `TextView` to display the `Affirmation` at this position.

2. With an object instance of `Affirmation`, you can search for the corresponding string's resource ID by calling `item.stringResourceId`. However, this is represented as an integer and you need to get its corresponding string value.

In the Android framework, you can call `getString()` with a string resource ID and it will show return the associated string value. `getString()` is a method of the class `Resources`, and we can get an instance of `Resources` through `context`.

This means you can call `context.resources.getString()` and pass a string resource ID. The resulting string can be used as the `text` of the `textView` in `holderItemViewHolder`. In short, this code updates the viewto display the affirmation string.

```
holder.textView.text = context.resources.getString(item.stringResourceId)
```

The competed method `onBindViewHolder()` is as follows:

```
override fun onBindViewHolder(holder: ItemViewHolder, position: Int) {
    val item = dataset[position]
    holder.textView.text =
 context.resources.getString(item.stringResourceId)
}
```

Here's the finished adapter code.

```
package au.edu.jcu.affirmations.adapter


import android.content.Context
import android.view.LayoutInflater
import android.view.View
import android.view.ViewGroup
import android.widget.TextView
import androidx.recyclerview.widget.RecyclerView
import au.edu.jcu.affirmations.R
import au.edu.jcu.affirmations.model.Affirmation
/**
 * Adapter for the [RecyclerView] in [MainActivity]. Displays [Affirmation]
data object.
 */
class ItemAdapter(
    private val context: Context,
    private val dataset: List<Affirmation>
) : RecyclerView.Adapter<ItemAdapter.ItemViewHolder>() {
    // Provide a reference to the views for each data item
    // Complex data items may need more than one view per item, and
    // you provide access to all the views for a data item in a view
holder.
    // Each data item is just an Affirmation object.
    class ItemViewHolder(private val view: View) :
RecyclerView.ViewHolder(view)  {
        val textView: TextView = view.findViewById(R.id.item_title)
    }
    /**
     * Create new views (invoked by the layout manager)
     */
    override fun onCreateViewHolder(parent: ViewGroup, viewType: Int):
ItemViewHolder {
        // create a new view
        val adapterLayout = LayoutInflater.from(parent.context)
            .inflate(R.layout.list_item, parent, false)
        return ItemViewHolder(adapterLayout)
    }
    /**
     * Replace the contents of a view (invoked by the layout manager)
     */
    override fun onBindViewHolder(holder: ItemViewHolder, position: Int) {
        val item = dataset[position]
        holder.textView.text =
context.resources.getString(item.stringResourceId)
    }
    /**
     * Return the size of your dataset (invoked by the layout manager)
     */
    override fun getItemCount() = dataset.size
}
```

Now that you have implemented the `ItemAdapter`, you must instruct the `RecyclerView` to use this adapter.

5. Modify MainActivity to use the RecyclerView and Adapter

Finally, we must use you're the classes `Datasource` and `ItemAdapter` in order to create and display elements in `RecyclerView`. We do this in `MainActivity`.

1. Open up. `MainActivity.kt`.
2. In `MainActivity,` go to the method `onCreate()`. Insert the new code described below after the call to `setContentView(R.layout.activity_main)..`
3. Create an instance of `Datasource` and call the method `loadAffirmations()`. We will store the list of affirmations in a `val` called `myDataset`.

```
val myDataset = Datasource().loadAffirmations()
```

4. Create a variable called `recyclerView` and use `findViewById()` to seek a reference to `RecyclerView` from the design.

```
val recyclerView = findViewById<RecyclerView>(R.id.recycler_view)
```

5. To ensure that `recyclerView` uses the class `ItemAdapter` that you created, create an instance of `ItemAdapter`. `ItemAdapter` expects two parameters: context (`this`) of this activity and the affirmations stored in `myDataset`.
6. Assign the object `ItemAdapter` to the property `adapter` of `recyclerView`.

```
recyclerView.adapter = ItemAdapter(this, myDataset)
```

7. Since the size of your data to be displayed in `RecyclerView` is fixed (10 in total), you can set the parameter `setHasFixedSize` of `RecyclerView` in `true`. This parameter is necessary to improve performance, since our app will know that changes in content will not change the size of the `RecyclerView`.

```
recyclerView.setHasFixedSize(true)
```

8. When you're done, the code for `MainActivity` will look as follows.

```
package au.edu.jcu.affirmations


import android.os.Bundle
import androidx.appcompat.app.AppCompatActivity
import androidx.recyclerview.widget.RecyclerView
import au.edu.jcu.affirmations.adapter.ItemAdapter
import au.edu..affirmations.data.Datasource class
MainActivity : AppCompatActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)
        // Initialize data.
        val myDataset = Datasource().loadAffirmations()
        val recyclerView = findViewById<RecyclerView>(R.id.recycler_view)
        recyclerView.adapter = ItemAdapter(this, myDataset)
        // Use this setting to improve performance if you know that changes
        // in content do not change the layout size of the RecyclerView
        recyclerView.setHasFixedSize(true)
    }
}
```

9. Run the app. You should see a list of affirmation strings on the screen.

Congratulations. You just created an app that shows a list of data with `RecyclerView` and a custom adapter. Spend some time analyzing the code you created and understanding how the different pieces work together.

6. Summary

- The `RecyclerView` helps you show a list of data.
- `RecyclerView` uses the adapter pattern to adapt and display the data.
- `ViewHolder` creates and contains the views for `RecyclerView`.
- `RecyclerView` includes `LayoutManagers`. `RecyclerView` delegates the distribution of the elements to `LayoutManagers`.
- Create a new class for adapter, for example, `ItemAdapter`.
- Create a class `ViewHolder` that represents a single view of list items, extending from class `RecyclerView.ViewHolder`.
- Implement these methods within the adapter: `getItemsCount()`, `onCreateViewHolder()` and `onBindViewHolder()`.