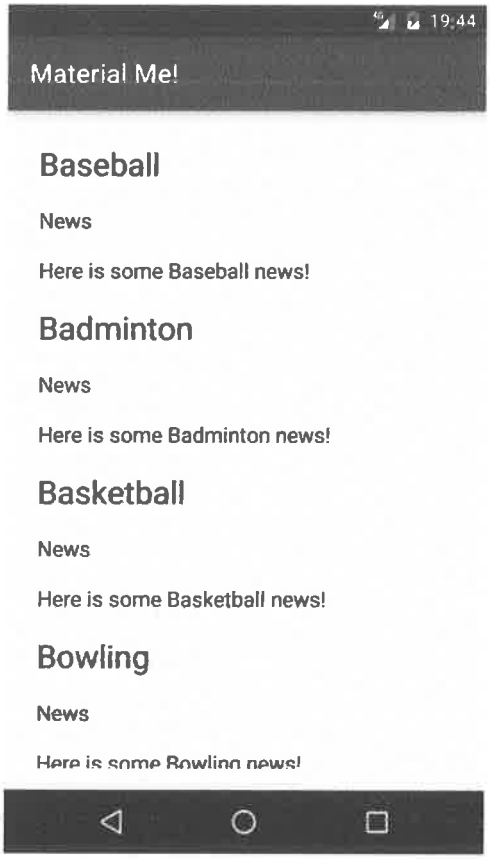
Practical 10: Cards and colors

Google's Material Design guidelines are a series of best practices for creating visually appealing and intuitive applications. In this practical you learn how to add `CardView` and `FloatingActionButton` widgets to your app, how to use images efficiently, and how to employ Material Design best practices to make your user's experience delightful.

- Modify an app to follow Material Design guidelines.
- Add images and styling to a RecyclerView list.
- Implement an ItemTouchHelper to add drag-and-drop functionality to your app.

App overview

The MaterialMe app is a mock sports-news app with very poor design implementation. You will fix it up to meet the design guidelines to create a delightful user experience! Below are screenshots of the app before and after the Material Design improvements.



Task 1: Download the starter code

The complete starter app project for this practical is available at MaterialMe-Starter. In this task you will load the project into Android Studio and explore some of the app's key features.

1.1 Open and run the MaterialMe project

1. Download the MaterialMe-Starter code.
2. Open the app in Android Studio.
3. Run the app.

The app shows a list of sports names with some placeholder news text for each sport. The current layout and style of the app makes it nearly unusable: each row of data is not clearly separated and there is no imagery or color to engage the user.

1.2 Explore the app

Before making modifications to the app, explore its current structure. It contains the following elements:

Sport.java

This class represents the **data model for each row of data** in the **RecyclerView**. Right now it contains a **field for the title of the sport** and a **field for some information about the sport**.

SportsAdapter.java

This is the **adapter** for the RecyclerView. It uses an **ArrayList of Sport objects as its data** and populates each row with this data.

MainActivity.java

The **MainActivity initializes the RecyclerView and adapter**, and **creates the data from resource files.**

strings.xml

This **resource file contains all of the data** for the app, including the **titles** and **information for each sport**.

list_item.xml

This **layout file defines each row of the RecyclerView**. It consists of **three TextView elements, one for each piece of data** (the **title** and the **info for each sport**) and **one used as a label**.

Task 2: Add a CardView and images

One of the fundamental principles of Material Design is the use of **bold imagery** to enhance the user experience. **Adding images to the RecyclerView list items** is a good start for creating a **dynamic and captivating** user experience.

When presenting information that has **mixed media** (like **images** and **text**), the Material Design guidelines **recommend** using a CardView, which is a FrameLayout **with some extra features** (such as **elevation** and **rounded corners**) that give it a **consistent look and feel across many different applications and platforms**. **CardView** is a UI component found in the Android Support Libraries.

In this section, you will move each list item into a **CardView** and add an **Image** to make the app comply with Material guidelines.

2.1 Add the CardView

CardView is **not included in the default Android SDK**, so you must **add it** as a **build.gradle dependency**. Do the following:

1. Open the **build.gradle (Module: app)** file, and **add** the following line to the **dependencies section**:

```
implementation 'com.android.support:cardview-v7:26.1.0'
```

The **version of the support library may have changed** since the writing of this practical. Update the above to the **version suggested by Android Studio**, and click **Sync** to sync your **build.gradle files**.

2. Open the **list_item.xml** file, and surround the root **LinearLayout** with **android.support.v7.widget.CardView**. Move the schema declaration **(xmlns:android="http://schemas.android.com/apk/res/android)** to the **CardView,** and add the following attributes:

Attribute	Value
android:layout_width	"match_parent"
android:layout_height	"wrap_content"
android:layout_margin	"8dp"

The **schema declaration needs to move to the CardView** because the **CardView is now the top-level view in your layout file**.

3. Choose **Code > Reformat Code** to reformat the **XML code**, which should **now look like this** at the **beginning** and **end** of the file:

```
<android.support.v7.widget.CardView
    xmlns:android="http://schemas.android.com/apk/res/android"
   android:layout_width="match_parent"
   android:layout_height="wrap_content"
   android:layout_margin="8dp">

   <LinearLayout
      android:layout_width="match_parent"
      android:layout_height="wrap_content"
      android:orientation="vertical">
      <!-- Rest of LinearLayout -->
      <!-- TextView elements -->
   </LinearLayout>
</android.support.v7.widget.CardView>
```

4. Run the app. Now **each row item is contained inside a CardView**, which is elevated above the bottom layer and casts a shadow.

2.2 Download the images

The **CardView** is not intended to be used exclusively with plain text: it is best for **displaying a mixture of content**. You have a good opportunity to make this app more exciting by **adding banner images** to every row!

Using images is resource intensive for your app: the Android framework has to load the entire image into memory at full resolution, even if the app only displays a small thumbnail of the image.

In this section you learn how to **use the Glide library to load large images efficiently**, without draining your resources or even crashing your app due to 'Out of Memory' exceptions.

1. Download the banner images zip file.
2. Open the **MaterialMe > app > src > main > res** directory in your operating system's file explorer, and create a **drawable** directory, and **copy** the individual graphics files into the **drawable** directory.
3. You **need an array with the path to each image** so that you can include it in the Sports object. To do this, **define an array that contains all of the paths to the drawables as items in your string.xml file.** Be sure to that they are in the **same order as the sports_titles array** also defined in the same file:

```
<array name="sports_images">
  <item>@drawable/img_baseball</item>
  <item>@drawable/img_badminton</item>
  <item>@drawable/img_basketball</item>
  <item>@drawable/img_bowling</item>
  <item>@drawable/img_cycling</item>
  <item>@drawable/img_golf</item>
  <item>@drawable/img_running</item>
  <item>@drawable/img_soccer</item>
  <item>@drawable/img_swimming</item>
  <item>@drawable/img_tabletennis</item>
  <item>@drawable/img_tennis</item>
</array>
```

2.3 Modify the Sport object

The **Sport object** will need to include the **Drawable resource** that corresponds to the sport. To achieve that:

1. **Add** an **integer member variable** to the **Sport object** that will **contain the Drawable resource**:

```
private final int imageResource;
```

2. **Modify the constructor** so that it takes an integer as a parameter and assigns it to the member variable:

```
public Sport(String title, String info, int imageResource) {
  this.title = title;
  this.info = info;
  this.imageResource = imageResource;
}
```

3. **Create a getter** for the resource integer:

```
public int getImageResource() {
  return imageResource;
}
```

2.4 Fix the initializeData() method

In **MainActivity**, the **initializeData() method** is now broken, because the constructor for the **Sport object demands the image resource** as the **third parameter**.

- A convenient data structure to use would be a **TypedArray**. A **TypedArray** allows you to **store an array of other XML resources**.
- Using a **TypedArray**, you can obtain the image resources as well as the sports title and information by using indexing in the same loop.

1. In the **initializeData() method**, **get** the **TypedArray of resource IDs** by calling **getResources().obtainTypedArray()**, passing in the name of the array of **Drawable** resources you defined in your **strings.xml** file:

```
TypedArray sportsImageResources =
    getResources().obtainTypedArray(R.array.sports_images);
```

You can **access** an element at **index i** in the **TypedArray** by using the appropriate **"get"** method, depending on the type of resource in the array. In this specific case, it contains resource IDs, so you use the **getResourceId() method**.

2. **Fix the code in the loop** that creates the Sport objects, **adding** the appropriate **Drawable resource ID** as the third parameter by **calling getResourceId()** on the TypedArray:

```
for(int i=0;i<sportsList.length;i++){
  mSportsData.add(new Sport(sportsList[i],sportsInfo[i],
    sportsImageResources.getResourceId(i,0)));
}
```

3. **Clean up the data** in the typed array once you have created the Sport data ArrayList:

```
sportsImageResources.recycle();
```

2.5 Add an ImageView to the list items

1. **Change** the **LinearLayout** inside the **list_item.xml** file to a **RelativeLayout**, and **delete** the **android:orientation attribute**.
2. **Add** an **ImageView** as the **first element** within the **RelativeLayout** with the following attributes:

Attribute	Value
android:layout_width	"match_parent"
android:layout_height	"wrap_content"
android:id	**"@+id/sportsImage"**
android:adjustViewBounds	"true"

The **adjustViewBounds** attribute makes the **ImageView** adjust its boundaries to preserve the aspect ratio of the image.

3. **Add** the following attributes to the **title TextView element**:

Attribute	Value
android:layout_alignBottom	**"@id/sportsImage"**
android:theme	**"@style/ThemeOverlay.AppCompat.Dark"**

4. **Add** the following attributes to the **newsTitle TextView element**:

Attribute	Value
android:layout_below	**"@id/sportsImage"**
android:textColor	"?android:textColorSecondary"

5. **Add** the following attributes to the **subTitle TextView element**:

Attribute	Value
android:layout_below	**"@id/newsTitle"**

The question mark in the above textColor attribute (**"?android:textColorSecondary"**) means that the **framework will apply the value from the currently applied theme**. In this case, this attribute is inherited from the **"Theme.AppCompat.Light.DarkActionBar"** theme, which defines it as a light **gray** color, often used for subheadings.

2.6 Load the images using Glide

After downloading the images and setting up the **ImageView**, the next step is to **modify the SportsAdapter to load an image into the ImageView in onBindViewHolder()**.

- If you take this approach, you will find that **your app crashes due to "Out of Memory"** errors.
- The Android framework has to **load the image into memory each time at full resolution**, no matter what the display size of the ImageView is.

There are a number of ways to **reduce the memory consumption** when loading images, but one of the easiest approaches is to **use an Image Loading Library like Glide**, which you will do in this step.

- Glide uses **background processing**, as well some other complex processing, to **reduce the memory requirements of loading images**.
- It also includes some **useful features** like **showing placeholder images** while the desired images are loaded.

Note: To learn more about reducing memory consumption in your app, see Loading Large Bitmaps Efficiently.

1. **Open** the **build.gradle (Module: app) file**, and **add** the following **dependency for Glide** in the **dependencies section**:

   implementation 'com.github.bumptech.**glide:glide:3.7.0**'

2. **Open SportsAdapter**, and **add a variable** in the **ViewHolder** class for the **ImageView**:

   private **ImageView** mSportsImage;

3. **Initialize the variable** in the **ViewHolder constructor** for the **ViewHolder class**:

   mSportsImage = itemView.**findViewById**(R.id.**sportsImage**);

4. **Add** the following line of **code to the bindTo() method** in the **ViewHolder class** to **get** the **image resource** from the **Sport object** and **load it** into the **ImageView using Glide**:

   Glide.with(mContext).**load**(currentSport.**getImageResource()**).**into**(mSportsImage);

5. Run the app, your list items should now have bold graphics that make the user experience dynamic and exciting!

That's all takes to load an image with Glide. Glide also has several additional features that let you resize, transform and load images in a variety of ways. Head over to the Glide GitHub page to learn more.

Task 3: Make your CardView swipeable, movable, and clickable

When users see cards in an app, they have expectations about the way the cards behave. Material Design offers the following guidelines:

- A card can be **dismissed**, usually by **swiping** it away.
- A list of cards can be reordered by **holding down** and **dragging the cards**.
- **Tapping** on card provides further **details**.

You now implement these behaviors in your app.

3.1 Implement swipe to dismiss

The Android SDK includes a class called **ItemTouchHelper** that is used to define what happens to **RecyclerView list items** when the user performs **various touch actions**, such as swipe, or drag and drop. Some of the common use cases are already implemented in a set of methods in **ItemTouchHelper.SimpleCallback**.

ItemTouchHelper.SimpleCallback lets you define which **directions** are supported for swiping and moving list items, and implement the swiping and moving behavior.

Do the following:

1. **Open MainActivity** and **create** a new **ItemTouchHelper object** in the **onCreate() method** at the end, below the **initializeData() method**.
   - For its argument, you will **create a new instance** of **ItemTouchHelper.SimpleCallback**.
   - As you enter **new ItemTouchHelper**, suggestions appear. **Select ItemTouchHelper.SimpleCallback{...}** from the suggestion menu.
   - Android Studio fills in the required methods: **onMove()** and **onSwiped()** as shown below.

```
ItemTouchHelper helper = new ItemTouchHelper(new
                    ItemTouchHelper.SimpleCallback() {
    @Override
    public boolean onMove(RecyclerView recyclerView,
                          RecyclerView.ViewHolder viewHolder,
                          RecyclerView.ViewHolder target) {
        return false;
    }
    @Override
    public void onSwiped(RecyclerView.ViewHolder viewHolder,
                         int direction) {
    }
});
```

If the required methods were not automatically added, **click** on the **red bulb** in the left margin, and select **Implement methods**.

The **SimpleCallback constructor** will be **underlined in red because you have not yet provided the required parameters**: the **direction** that you plan to support for **moving and swiping** list items, respectively.

2. Because **we are only implementing swipe to dismiss at the moment**, you should **pass** in **0** for the **supported move directions** and **ItemTouchHelper.LEFT | ItemTouchHelper.RIGHT** for the supported swipe directions:

```
ItemTouchHelper helper = new ItemTouchHelper(new ItemTouchHelper
        .SimpleCallback(0, ItemTouchHelper.LEFT |
        ItemTouchHelper.RIGHT) {
```

3. You must now **implement** the desired behavior in **onSwiped()**. In this case, **swiping** the card **left** or **right** should **delete** it from the list. **Call remove() on the data set**, **passing in the appropriate index** by **getting the position** from the **ViewHolder**:

```
mSportsData.remove(viewHolder.getAdapterPosition());
```

4. To **allow** the **RecyclerView** to animate the **deletion** properly, you must also **call notifyItemRemoved()**, again **passing in the appropriate index** by **getting the position** from the **ViewHolder**:

```
mAdapter.notifyItemRemoved(viewHolder.getAdapterPosition());
```

5. Below the new **ItemTouchHelper object** in the **onCreate() method** for **MainActivity**, **call attachToRecyclerView()** on the **ItemTouchHelper instance** to **add** it to your **RecyclerView**:

```
helper.attachToRecyclerView(mRecyclerView);
```

6. Run your app, you can now swipe list items left and right to delete them!

3.2 Implement drag and drop

You can also **implement drag and drop functionality** using the same **SimpleCallback**. The **first argument** of the **SimpleCallback** determines which **directions** the **ItemTouchHelper** supports for moving the objects around. **Do the following**:

1. **Change** the **first argument** of the **SimpleCallback** from **0 to include every direction**, since we want to be able to drag and drop anywhere:

```
ItemTouchHelper helper = new ItemTouchHelper(new ItemTouchHelper
.SimpleCallback(ItemTouchHelper.LEFT | ItemTouchHelper.RIGHT |
ItemTouchHelper.DOWN | ItemTouchHelper.UP, ItemTouchHelper.LEFT |
ItemTouchHelper.RIGHT) {
```

2. In the **onMove()** **method**, **get** the **original** and **target index** from the **second** and **third argument passed in** (corresponding to the original and target view holders).

```
int from = viewHolder.getAdapterPosition();
int to = target.getAdapterPosition();
```

3. **Swap** the **items in the dataset** by calling **Collections.swap()** and **pass in the dataset**, and the **initial and final indexes**:

```
Collections.swap(mSportsData, from, to);
```

4. **Notify** the **adapter** that the item was moved, passing in the old and new indexes, and change the **return** statement to **true**:

```
mAdapter.notifyItemMoved(from, to);
return true;
```

5. Run your app. You can now delete your list items by swiping them left or right, or reorder them using a long press to activate Drag and Drop mode.

3.3 Implement the DetailActivity layout

According to Material Design guidelines, **a card is used to provide an entry point to more detailed information**. You may find yourself **tapping on the cards to see more information** about the sports, because that is how you expect cards to behave.

In this section, you will **add a detail Activity** that will be launched when any list item is pressed. For this practical, the detail **Activity** will contain the **name** and **image** of the **list item** you clicked, but will contain only **generic placeholder detail text**, so you **don't have to create custom detail for each list item**.

1. **Create** a new **Activity** by going to **File > New > Activity > Empty Activity**.
2. Call it **DetailActivity**, and accept all of the defaults.

3. **Open** the newly created **activity_detail.xml layout file** and **change** the **root ViewGroup** to **RelativeLayout,** as you've done in previous exercises.
4. **Remove** the **xmlns:app="http://schemas.android.com/apk/res-auto"** statement from the **RelativeLayout**.
5. **Copy** all of the **TextView** and **ImageView elements** from the **list_item.xml** file to the **activity_detail.xml file**.
6. **Add** the word **"Detail"** to the reference in each **android:id attribute** in order to **differentiate it from list_item.xml IDs**. For example, change the **ImageView** ID from **sportsImage** to **sportsImageDetail**.
7. In all **TextView** and **ImageView** elements, **change all references** to the IDs for relative placement such **layout_below** to **use** the **"Detail" ID**.
8. For the **subTitleDetail TextView**, **remove** the **placeholder text string** and **paste** a **paragraph** of **generic text** to **substitute detail text** (For example, a few paragraphs of Lorem Ipsum). **Extract the text to a string resource**.
9. **Change the padding** on the **TextView elements** to **16dp**.
10. **Wrap** the entire **RelativeLayout** with a **ScrollView**.
    - **Add** the required **layout_height** and **layout_width** attributes, and
    - **Append** the **xmlns:android="http://schemas.android.com/apk/res/android" attribute** to the end of the **ScrollView**.
11. **Change** the **layout_height attribute** of the **RelativeLayout** to **"wrap_content"**.

    The first two elements of the **activity_detail.xml layout** should now look as follows:

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

  <RelativeLayout xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    tools:context="com.example.android.materialme.DetailActivity">
```

3.4 Implement the detail view and click listener

Follow these steps to **implement the detail view and click listener**:

1. **Open SportsAdapter** and **change** the **ViewHolder inner class**, which **already extends RecyclerView.ViewHolder**, to also **implement View.OnClickListener**, and **implement** the **required method (onClick())**.

```
class ViewHolder extends RecyclerView.ViewHolder
                    implements View.OnClickListener{
  // Rest of ViewHolder code.
  //
  @Override
  public void onClick(View view) {

  }
}
```

2. **Set** the **OnClickListener** to the **itemView** in the **ViewHolder constructor**. The entire constructor should now look like this:

```
ViewHolder(View itemView) {
  super(itemView);

  //Initialize the views
  mTitleText = itemView.findViewById(R.id.title);
  mInfoText = itemView.findViewById(R.id.subTitle);
  mSportsImage = itemView.findViewById(R.id.sportsImage);

  // Set the OnClickListener to the entire view.
  itemView.setOnClickListener(this);
}
```

3. In the **onClick() method**, **get** the **Sport object** for the item that was clicked **using getAdapterPosition()**:

```
Sport currentSport = mSportsData.get(getAdapterPosition());
```

4. In the same method, **add** an **Intent** that **launches DetailActivity**, **put** the **title** and **image_resource** as **extras** in the **Intent**, and **call startActivity()** on the **mContext variable**, **passing** in the new **Intent**.

```
Intent detailIntent = new Intent(mContext, DetailActivity.class);
detailIntent.putExtra("title", currentSport.getTitle());
detailIntent.putExtra("image_resource",
```

```
                    currentSport.getImageResource());
mContext.startActivity(detailIntent);
```

5. **Open** DetailActivity and **initialize** the ImageView and title TextView in onCreate():

```
TextView sportsTitle = findViewById(R.id.titleDetail);
ImageView sportsImage = findViewById(R.id.sportsImageDetail);
```

6. **Get** the title from the **incoming** Intent and **set** it to the TextView:

```
sportsTitle.setText(getIntent().getStringExtra("title"));
```

7. **Use Glide to load the image** into the ImageView:

```
Glide.with(this).load(getIntent().getIntExtra("image_resource",0))
.into(sportsImage);
```

8. Run the app. Tapping on a list item now launches DetailActivity.

Task 4 Add the FAB and choose a Material Design color palette

One of the principles behind Material Design is **using consistent elements across applications and platforms so that users recognize patterns and know how to use them.**

You have already used one such element: the Floating Action Button **(FAB)**. The FAB is a circular **button** that **floats** above the rest of the UI. It is used **to promote a particular action** to the user, one that is very likely to be used in a given activity. In this task, you will create a FAB that resets the dataset to its original state.

4.1 Add the FAB

The Floating Action Button is part of the Design Support Library.

1. **Open** the **build.gradle (Module: app) file** and **add** the following line of **code** for the design support library in the **dependencies** section:

   implementation 'com.android.support:**design:26.1.0**'

2. **Add** an **icon** for the **FAB by right-clicking** (or Control-clicking) the **res folder** in the Project > Android pane, and choosing **New > Vector Asset**. The **FAB** will **reset** the contents of the **RecyclerView**, so the **refresh** icon should do: ↻. **Change** the **name** to **ic_reset**, click **Next**, and click **Finish**.

3. **Open activity_main.xml** and **add** a **FloatingActionButton** with the following attributes:

Attribute	Value
android:layout_width	"wrap_content"
android:layout_height	"wrap_content"
android:layout_alignParentBottom	"true"
android:layout_alignParentRight	"true
android:layout_alignParentEnd	"true
android:layout_margin	"16dp"
android:src	"@drawable/ic_reset"
android:tint	"@android:color/white"
android:onClick	resetSports

4. **Open MainActivity** and **add** the **resetSports() method** with a statement to **call initializeData()** to **reset** the **data**.
5. Run the app. You can now **reset** the **data** by **tapping** the **FAB**.
   - Because the **Activity** is destroyed and recreated when the configuration changes, rotating the device resets the data in this implementation.
   - In order for the **changes** to **be persistent** (as in the case of **reordering** or **removing data**), you would have to **implement onSaveInstanceState()** or write the changes to a persistent source (like a database or SharedPreferences, which are described in other lessons).

4.2 Choose a Material Design palette

Material Design recommends picking at least these **three colors** for your app:

- **A primary color**. This one is automatically used to color your app bar (the bar that contains the title of your app).
- **A primary dark color**. A darker shade of the same color. This is used for the status bar above the app bar, among other things.
- **An accent color**. A color that contrasts well with the primary color. This is used for various highlights, but it is also the default color of the FAB.

When you ran your app, you may have noticed that the FAB color and app bar color are already set.

In this task you will learn **where** these **colors** are **set**. You can use the Material Color Guide to pick some colors to experiment with.

1. In the **Project > Android** pane, **navigate** to your **styles.xml** file (located in the **values** directory). The **AppTheme style** defines **three colors** by default: **colorPrimary, colorPrimaryDark**, and **colorAccent**. These styles are **defined by values** from the **colors.xml file**.
2. **Pick a color** from the Material Color Guide to **use** as your **primary color**, such as **#607D8B** (in the **Blue Grey** color swatch). It should be within the **300-700 range** of the color swatch so that you can still pick a proper **accent and dark color**.
3. **Open** the **colors.xml** file, and **modify** the **colorPrimary hex value** to **match** the color you **picked**.

4. **Pick a darker shade of the same color** to use as your **primary dark color**, such as **#37474F**. Again, **modify** the **colors.xml hex value** for **colorPrimaryDark** to match.
5. **Pick an accent color** for your FAB from the colors whose values start with an **A**, and whose color contrasts well with the primary color (like **Deep Orange A200**). **Change** the **colorAccent** value in **colors.xml to match**.
6. Run the app. The app bar and FAB have now changed to reflect the new color palette!

If you want to **change the color of the FAB to something other than theme colors**, use the **app:backgroundTint attribute**. Note that this uses the app: namespace and Android Studio will prompt you to add a statement to define the namespace.

Task 9: Create an Adapter and adding the RecyclerView

You are going to display the data in a **RecyclerView**, which is a little nicer than just throwing the data in a **TextView**. This practical assumes that you know how **RecyclerView**, **RecyclerView.LayoutManager**, **RecyclerView.ViewHolder**, and **RecyclerView.Adapter** work.

9.1 Create the WordListAdapter class

- Add a class **WordListAdapter** that extends **RecyclerView.Adapter**. The adapter caches data and populates the **RecyclerView** with it. The inner class **WordViewHolder** holds and manages a view for one list item.

Here is the code:

```
public class WordListAdapter extends RecyclerView.Adapter<WordListAdapter.WordViewHolder>
{

  private final LayoutInflater mInflater;
  private List<Word> mWords; // Cached copy of words

  WordListAdapter(Context context) { mInflater = LayoutInflater.from(context); }

  @Override
  public WordViewHolder onCreateViewHolder(ViewGroup parent, int viewType) {
    View itemView = mInflater.inflate(R.layout.recyclerview_item, parent, false);
    return new WordViewHolder(itemView);
  }

  @Override
  public void onBindViewHolder(WordViewHolder holder, int position) {
    if (mWords != null) {
      Word current = mWords.get(position);
      holder.wordItemView.setText(current.getWord());
    } else {
      // Covers the case of data not being ready yet.
      holder.wordItemView.setText("No Word");
    }
  }
```

```
  void setWords(List<Word> words){
    mWords = words;
    notifyDataSetChanged();
  }

  // getItemCount() is called many times, and when it is first called,
  // mWords has not been updated (means initially, it's null, and we can't return null).
  @Override
  public int getItemCount() {
    if (mWords != null)
      return mWords.size();
    else return 0;
  }

  class WordViewHolder extends RecyclerView.ViewHolder {
    private final TextView wordItemView;

    private WordViewHolder(View itemView) {
      super(itemView);
      wordItemView = itemView.findViewById(R.id.textView);
    }
  }
}
```

Note: The **mWords** variable in the adapter caches the data. In the next task, you add the code that updates the data automatically.

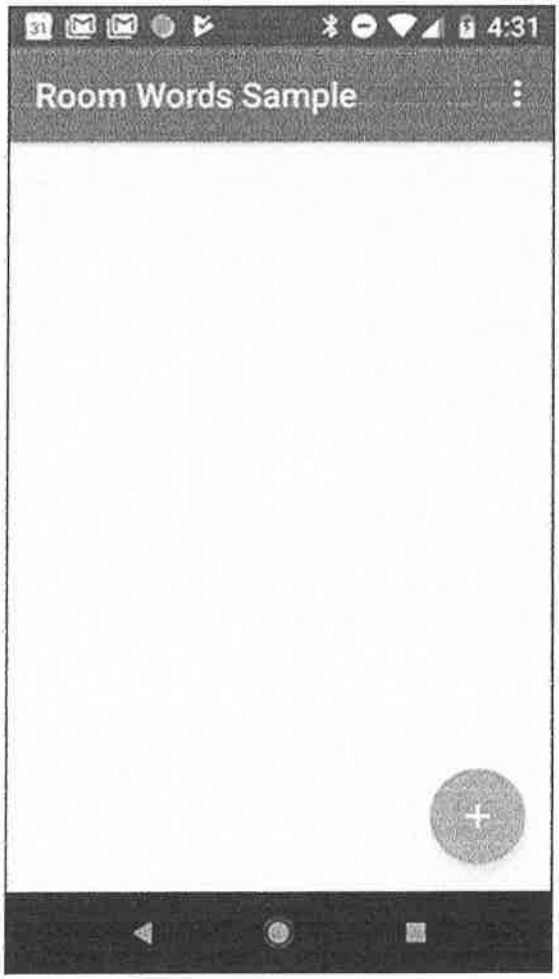
Note: The **getItemCount()** method needs to account gracefully for the possibility that the data is not yet ready and **mWords** is still null. In a more sophisticated app, you could display placeholder data or something else that would be meaningful to the user.

9.2 Add RecyclerView to MainActivity

1. Add the **RecyclerView** in the **onCreate()** method of **MainActivity**:

```
RecyclerView recyclerView = findViewById(R.id.recyclerview);
final WordListAdapter adapter = new WordListAdapter(this);
recyclerView.setAdapter(adapter);
recyclerView.setLayoutManager(new LinearLayoutManager(this));
```

2. Run your app to make sure the app compiles and runs. There are no items, because you have not hooked up the data yet. The app should display the empty recycler view.



Task 10: Populate the database

There is no data in the database yet. You will add data in two ways: Add some data when the database is opened, and add an **Activity** for adding words. Every time the database is opened, all content is deleted and repopulated. This is a reasonable solution for a sample app, where you usually want to restart on a clean slate.

10.1 Create the callback for populating the database

To delete all content and repopulate the database whenever the app is started, you create a **RoomDatabase.Callback** and override the **onOpen()** method. Because you cannot do Room database operations on the UI thread, **onOpen()** creates and executes an **AsyncTask** to add content to the database.

1. Add the **onOpen()** callback in the **WordRoomDatabase** class:

```
private static RoomDatabase.Callback sRoomDatabaseCallback =
    new RoomDatabase.Callback(){

    @Override
    public void onOpen (@NonNull SupportSQLiteDatabase db){
        super.onOpen(db);
        new PpopulateDbAsync(INSTANCE).execute();
    }
};
```

2. Create an inner class **PopulateDbAsync** that extends **AsycTask**. Implement the **doInBackground()** method to delete all words, then create new ones. Here is the code for the **AsyncTask** that deletes the contents of the database, then populates it with an initial list of words. Feel free to use your own words!

```
/**
* Populate the database in the background.
*/
private static class PopulateDbAsync extends AsyncTask<Void, Void, Void> {

  private final WordDao mDao;
  String[] words = {"dolphin", "crocodile", "cobra"};

  PopulateDbAsync(WordRoomDatabase db) {
    mDao = db.wordDao();
  }
```

```
    @Override
    protected Void doInBackground(final Void... params) {
        // Start the app with a clean database every time.
        // Not needed if you only populate the database
        // when it is first created
        mDao.deleteAll();

        for (int i = 0; i <= words.length - 1; i++) {
            Word word = new Word(words[i]);
            mDao.insert(word);
        }
        return null;
    }
}
```

3. Add the callback to the database build sequence in **WordRoomDatabase**, right before you call **.build()**:

```
        .addCallback(sRoomDatabaseCallback)
```

Task 11: Connect the UI with the data

Now that you have created the method to populate the database with the initial set of words, the next step is to add the code to display those words in the **RecyclerView**.

To display the current contents of the database, you add an observer that observes the **LiveData** in the **ViewModel**. Whenever the data changes (including when it is initialized), the **onChanged()** callback is invoked. In this case, the **onChanged()** callback calls the adapter's **setWord()** method to update the adapter's cached data and refresh the displayed list.

11.1 Display the words

1. In **MainActivity**, create a member variable for the **ViewModel**, because all the activity's interactions are with the **WordViewModel** only.

```
private WordViewModel mWordViewModel;
```

2. In the **onCreate()** method, get a **ViewModel** from the **ViewModelProviders** class.

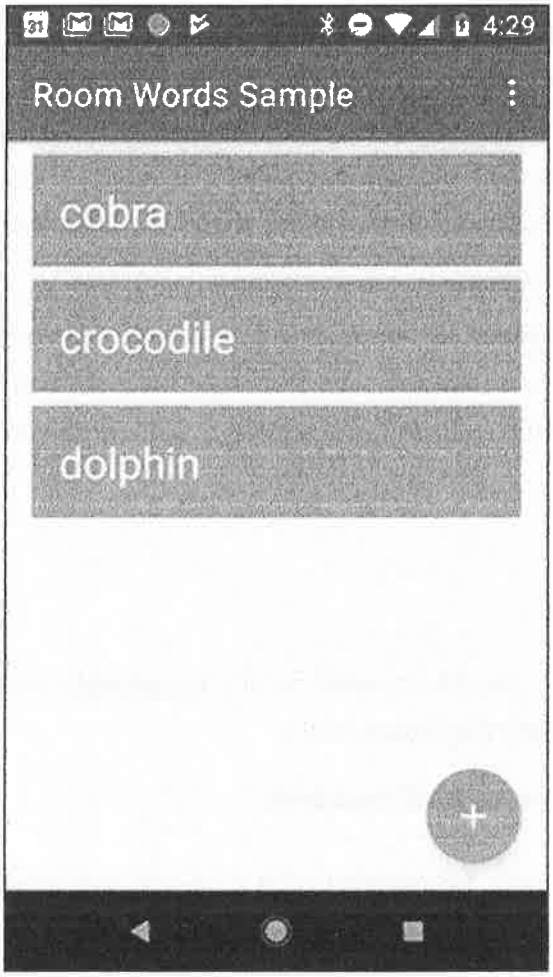
```
mWordViewModel = ViewModelProviders.of(this).get(WordViewModel.class);
```

Use **ViewModelProviders** to associate your **ViewModel** with your UI controller. When your app first starts, the **ViewModelProviders** class creates the **ViewModel**. When the activity is destroyed, for example through a configuration change, the **ViewModel** persists. When the activity is re-created, the **ViewModelProviders** return the existing **ViewModel**. See **ViewModel**.

3. Also in **onCreate()**, add an observer for the **LiveData** returned by **getAllWords()**. When the observed data changes while the activity is in the foreground, the **onChanged()** method is invoked and updates the data cached in the adapter. Note that in this case, when the app opens, the initial data is added, so **onChanged()** method is called.

```
mWordViewModel.getAllWords().observe(this, new Observer<List<Word>>() {
  @Override
  public void onChanged(@Nullable final List<Word> words) {
    // Update the cached copy of the words in the adapter.
    adapter.setWords(words);
  }
});
```

4. Run the app. The initial set of words appears in the **RecyclerView**.



Task 12: Create an Activity for adding words

Now you will add an Activity that lets the user use the FAB to enter new words. This is what the interface for the new activity will look like:

12.1 Create the NewWordActivity

1. Add these string resources in the values/strings.xml file:

```
<string name="hint_word">Word...</string>
<string name="button_save">Save</string>
<string name="empty_not_saved">Word not saved because it is empty.</string>
```

3. Add a style for buttons in value/styles.xml:

```
<style name="button_style" parent="android:style/Widget.Material.Button">
  <item name="android:layout_width">match_parent</item>
  <item name="android:layout_height">wrap_content</item>
  <item name="android:background">@color/colorPrimaryDark</item>
  <item name="android:textAppearance">@android:style/TextAppearance.Large</item>
  <item name="android:layout_marginTop">16dp</item>
  <item name="android:textColor">@color/colorTextPrimary</item>
</style>
```

5. Use the Empty Activity template to create a new activity, NewWordActivity. Verify that the activity has been added to the Android Manifest.

```
<activity android:name=".NewWordActivity"></activity>
```

6. Update the activity_new_word.xml file in the layout folder:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
  android:layout_width="match_parent"
  android:layout_height="match_parent"
  android:background="@color/colorScreenBackground"
  android:orientation="vertical"
  android:padding="24dp">

  <EditText
    android:id="@+id/edit_word"
    style="@style/text_view_style"
    android:hint="@string/hint_word"
    android:inputType="textAutoComplete" />

  <Button
    android:id="@+id/button_save"
    style="@style/button_style"
    android:text="@string/button_save" />
</LinearLayout>
```

7. Implement the NewWordActivity class. The goal is that when the user presses the **Save** button, the new word is put in an Intent to be sent back to the parent Activity.

Here is the code for the NewWordActivity activity:

```
public class NewWordActivity extends AppCompatActivity {
  public static final String EXTRA_REPLY =
        "com.example.android.roomwordssample.REPLY";

  private  EditText mEditWordView;

  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_new_word);
    mEditWordView = findViewById(R.id.edit_word);

    final Button button = findViewById(R.id.button_save);
    button.setOnClickListener(new View.OnClickListener() {
      public void onClick(View view) {
        Intent replyIntent = new Intent();
        if (TextUtils.isEmpty(mEditWordView.getText())) {
          setResult(RESULT_CANCELED, replyIntent);
        } else {
          String word = mEditWordView.getText().toString();
          replyIntent.putExtra(EXTRA_REPLY, word);
          setResult(RESULT_OK, replyIntent);
        }
        finish();
      }
    });
  }
}
```

12.2 Add code to insert a word into the database

1. In MainActivity, add the onActivityResult() callback for the NewWordActivity. If the activity returns with RESULT_OK, insert the returned word into the database by calling the insert() method of the WordViewModel.

```
public void onActivityResult(int requestCode, int resultCode, Intent data) {
  super.onActivityResult(requestCode, resultCode, data);

  if (requestCode == NEW_WORD_ACTIVITY_REQUEST_CODE && resultCode ==
RESULT_OK) {
    Word word = new Word(data.getStringExtra(NewWordActivity.EXTRA_REPLY));
    mWordViewModel.insert(word);
  } else {
    Toast.makeText(
        getApplicationContext(),
        R.string.empty_not_saved,
        Toast.LENGTH_LONG).show();
  }
}
```

2. Define the missing request code:

   ```
   public static final int NEW_WORD_ACTIVITY_REQUEST_CODE = 1;
   ```

3. In MainActivity,start NewWordActivity when the user taps the FAB. Replace the code in the FAB's onClick() click handler with the following code:

   ```
   Intent intent = new Intent(MainActivity.this, NewWordActivity.class);
   startActivityForResult(intent, NEW_WORD_ACTIVITY_REQUEST_CODE);
   ```

4. Run your app. When you add a word to the database in NewWordActivity, the UI automatically updates.

5. Add a word that already exists in the list. What happens? Does your app crash? Your app uses the word itself as the primary key, and each primary key **must** be unique. You can specify a conflict strategy to tell your app what to do when the user tries to add an existing word.

6. In the WordDao interface, change the annotation for the insert() method to:

   ```
   @Insert(onConflict = OnConflictStrategy.IGNORE)
   ```

   To learn about other conflict strategies, see the OnConflictStrategy reference.

7. Run your app again and try adding a word that already exists. What happens now?