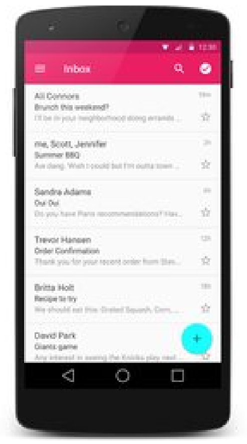
RecyclerView

A better way to display collections in Android

Overview

Many apps need to display collections of the same type (such as messages, contacts, images, or songs); often, this collection is too large to fit on the screen, so the collection is presented in a small window that can smoothly scroll through all items in the collection. **RecyclerView** is an Android widget that displays a collection of items in a list or a grid, enabling the user to scroll through the collection. The following is a screenshot of an example app that uses **RecyclerView** to display email inbox contents in a vertical scrolling list:



RecyclerView offers two compelling features:

- It has a flexible architecture that lets you modify its behavior by plugging in your preferred components.
- It is efficient with large collections because it reuses item views and requires the use of *view holders* to cache view references.

This guide explains how to use **RecyclerView** in Xamarin.Android applications; it explains how to add the **RecyclerView** package to your Xamarin.Android project, and it describes how **RecyclerView** functions in a typical application. Real code examples are provided to show you how to integrate **RecyclerView** into your application, how to implement item-view click, and how to refresh **RecyclerView** when its underlying data changes. This guide assumes that you are familiar with Xamarin.Android development.

Requirements

Although **RecyclerView** is often associated with Android 5.0 Lollipop, it is offered as a support library – **RecyclerView** works with apps that target API level 7 (Android 2.1) and later. The following is required to use **RecyclerView** in Xamarin-based applications:

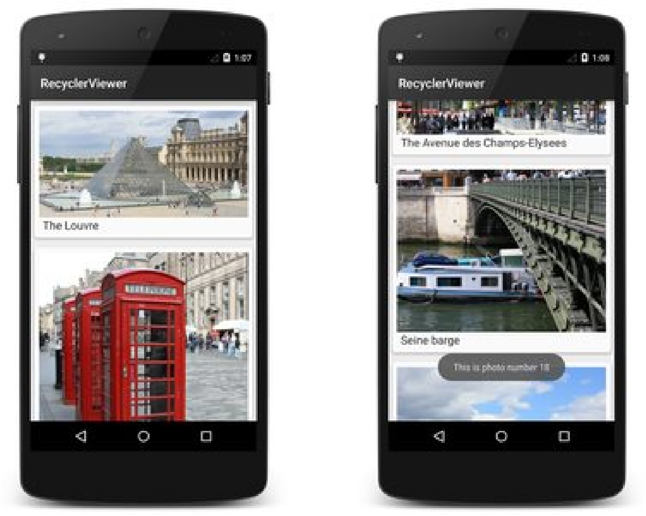
- **Xamarin.Android** – Xamarin.Android 4.20 or later must be installed and configured with either Visual Studio or Xamarin Studio. If you are using Xamarin Studio, version 5.5.4 or later is required.
- Your app project must include the **Xamarin.Android.Support.v7.RecyclerView** package. For more information about installing NuGet packages, see Walkthrough: Including a NuGet in your project.

Introducing RecyclerView

RecyclerView can be thought of as a replacement for the **ListView** and **GridView** widgets in Android. Like its predecessors, **RecyclerView** is designed to display a large data set in a small window, but **RecyclerView** offers more layout options and is better optimized for displaying large collections. If you are familiar with **ListView**, there are several important differences between **ListView** and **RecyclerView**:

- **RecyclerView** is slightly more complex to use: you have to write more code to use **RecyclerView** compared to **ListView**.
- **RecyclerView** does not provide a predefined adapter; you must implement the adapter code that accesses your data source. However, Android includes several predefined adapters that work with **ListView** and **GridView**.
- **RecyclerView** does not offer an item-click event when a user taps an item; instead, item-click events are handled by helper classes. By contrast, **ListView** offers an item-click event.
- **RecyclerView** enhances performance by recycling views and by enforcing the view-holder pattern, which eliminates unnecessary layout resource lookups. Use of the view-holder pattern is optional in **ListView**.
- **RecyclerView** is based on a modular design that makes it easier to customize. For example, you can plug in a different layout policy without significant code changes to your app. By contrast, **ListView** is relatively monolithic in structure.
- **RecyclerView** includes built-in animations for item add and remove. **ListView** animations require some additional effort on the part of the app developer.

To understand how **RecyclerView** works in a typical application, we'll explore the RecyclerViewer sample app, a simple code example that uses **RecyclerView** to display a large collection of photos:



This app uses CardView to implement each photograph item in the **RecyclerView** layout. Because of **RecyclerView**'s performance advantages, this sample app is able to quickly scroll through a large collection of photos smoothly and without noticeable delays.

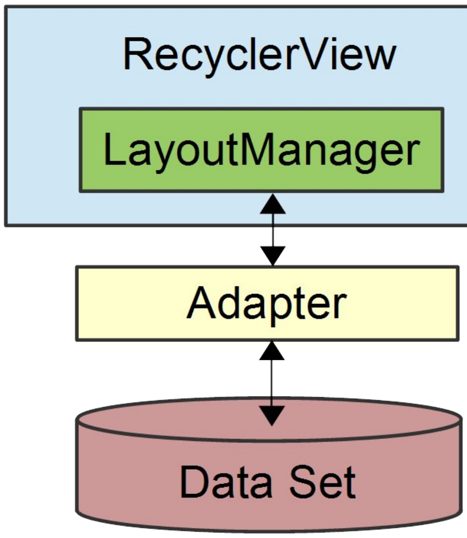
RecyclerView Helper Classes

RecyclerView handles some tasks internally (such as the scrolling and recycling of views), but it is essentially a manager that coordinates helper classes to display a collection. **RecyclerView** delegates tasks to the following helper classes:

- `Adapter` – Inflates item layouts (instantiates the contents of a layout file) and binds data to views that are displayed within a **RecyclerView**. The adapter also reports item-click events.
- `LayoutManager` – Measures and positions item views within a **RecyclerView** and manages the policy for view recycling.
- `ViewHolder` – Looks up and stores view references. The view holder also helps with detecting item-view clicks.
- `ItemDecoration` – Allows your app to add special drawing and layout offsets to specific views for drawing dividers between items, highlights, and visual grouping boundaries.

- **`ItemAnimator`** – Defines the animations that take place during item actions or as changes are made to the adapter.

The relationship between the `RecyclerView`, `LayoutManager`, and `Adapter` classes is depicted in the following diagram.



As this figure illustrates, the `LayoutManager` can be thought of as the intermediary between the `Adapter` and the `RecyclerView`. The `LayoutManager` makes calls into `Adapter` methods on behalf of the `RecyclerView`. For example, the `LayoutManager` calls an `Adapter` method when it is time to create a new view for a particular item position in the `RecyclerView`. The `Adapter` inflates the layout for that item and creates a `ViewHolder` instance (not shown) to cache references to the views at that position. When the `LayoutManager` calls the `Adapter` to bind a particular item to the data set, the `Adapter` locates the data for that item, retrieves it from the data set, and copies it to the associated item view.

When using **RecyclerView** in your app, creating derived types of the following classes is required:

- **`RecyclerView.Adapter`** – Provides a binding from your app's data set (which is specific to your app) to item views that are displayed within the **RecyclerView**. The adapter knows how to associate each item-view position in the **RecyclerView** to a specific location in the data source. In addition, the adapter handles the layout of the contents within each individual item view and creates the view holder for each view. The adapter also reports item-click events that are detected by the item view.

- **`RecyclerView.ViewHolder`** – Caches references to the views in your item layout file so that resource lookups are not repeated unnecessarily. The view holder also arranges for item-click events to be forwarded to the adapter when a user taps the view-holder's associated item view.

- **`RecyclerView.LayoutManager`** – Positions items within the **RecyclerView**. You can use one of several predefined layout managers or you can implement your own custom layout manager. **RecyclerView** delegates the layout policy to the layout manager, so you can plug in a different layout manager without having to make significant changes to your app.

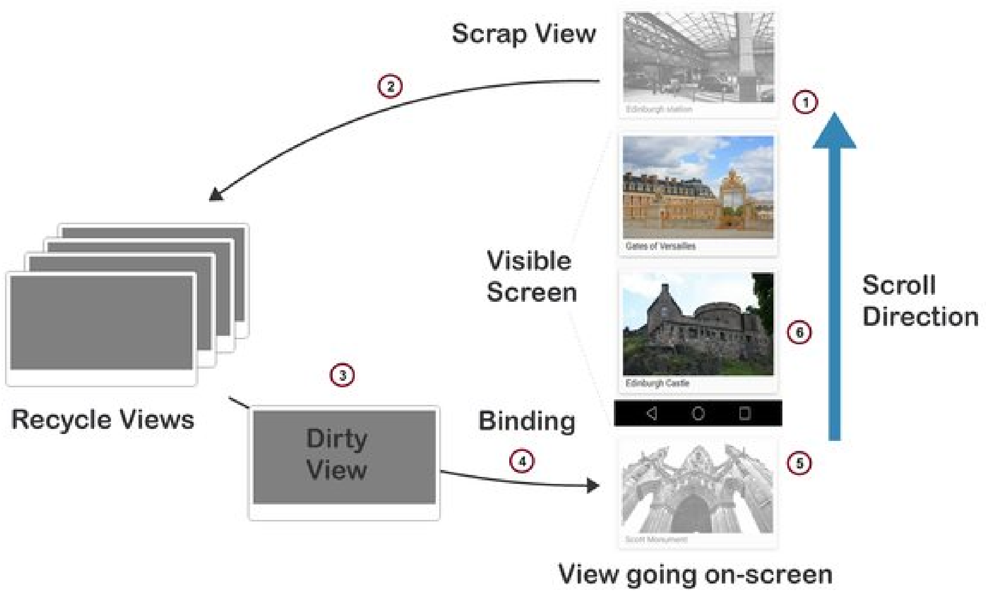
Also, you can optionally extend the following classes to change the look and feel of **RecyclerView** in your app:

- **`RecyclerView.ItemDecoration`**
- **`RecyclerView.ItemAnimator`**

If you do not extend `ItemDecoration` and `ItemAnimator`, **RecyclerView** uses default implementations. This guide does not explain how to create custom `ItemDecoration` and `ItemAnimator` classes; for more information about these classes, see RecyclerView.ItemDecoration and RecyclerView.ItemAnimator.

How View Recycling Works

RecyclerView does not allocate an item view for every item in your data source. Instead, it allocates only the number of item views that fit on the screen and it reuses those item layouts as the user scrolls. When the view first scrolls out of sight, it goes through the recycling process illustrated in the following figure:



1. When a view scrolls out of sight and is no longer displayed, it becomes a *scrap view*.

2. The scrap view is placed in a pool and becomes a *recycle view*. This pool is a cache of views that display the same type of data.

3. When a new item is to be displayed, a view is taken from the recycle pool for reuse. Because this view must be re-bound by the adapter before being displayed, it is called a *dirty view*.
4. The dirty view is recycled: the adapter locates the data for the next item to be displayed and copies this data to the views for this item. References for these views are retrieved from the view holder associated with the recycled view.
5. The recycled view is added to the list of items in the **RecyclerView** that are about to go on-screen.
6. The recycled view goes on-screen as the user scrolls the **RecyclerView** to the next item in the list. Meanwhile, another view scrolls out of sight and is recycled according to the above steps.

In addition to item-view reuse, **RecyclerView** also uses another efficiency optimization: view holders. A view holder is a simple class that caches view references. Each time the adapter inflates an item-layout file, it also creates a corresponding view holder. The view holder uses `FindViewById` to get references to the views inside the inflated item-layout file. These references are used to load new data into the views every time the layout is recycled to show new data.

A Basic RecyclerView Example

In the following code examples, we'll look at the implementation of the RecyclerViewer sample app to understand how **RecyclerView** works in a real application. We'll examine the adapter and view holder implementations to see how they support **RecyclerView**. In addition, we'll see how to specify the layout manager, and we'll modify the app to use a different layout manager.

An Example Data Source

In this example, we use a "photo album" data source (represented by the `PhotoAlbum` class) to supply **RecyclerView** with item content. `PhotoAlbum` is a collection of photos with captions; when you instantiate it, you get a ready-made collection of 32 photos:

```
PhotoAlbum mPhotoAlbum = new PhotoAlbum();
```

Each photo instance in `PhotoAlbum` exposes properties that allow you to read its image resource ID, `PhotoID`, and its caption string, `Caption`. The collection of photos is organized such that each photo can be accessed by an indexer. For example, the following lines of code access the image resource ID and caption for the tenth photo in the collection:

```
int imageId = mPhotoAlbum[9].ImageId;
```

```
string caption = mPhotoAlbum[9].Caption;
```

`PhotoAlbum` also provides a `RandomSwap` method that you can call to swap the first photo in the collection with a randomly-chosen photo elsewhere in the collection:

```
mPhotoAlbum.RandomSwap();
```

Because the implementation details of `PhotoAlbum` are not relevant to understanding **RecyclerView**, the `PhotoAlbum` source code is not presented here. The source code to `PhotoAlbum` is available in the file **PhotoAlbum.cs** in the RecyclerViewer sample app.

Initialization

Before we implement the layout manager, view holder, and adapter, we need some preliminary code to initialize the application. The layout file, **Main.axml**, consists of a single **RecyclerView** within a `LinearLayout`:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent">
    <android.support.v7.widget.RecyclerView
        android:id="@+id/recyclerView"
        android:scrollbars="vertical"
        android:layout_width="fill_parent"
        android:layout_height="fill_parent" />
</LinearLayout>
```

Note that you must use the fully-qualified name, **android.support.v7.widget.RecyclerView**, because **RecyclerView** is packaged in a support library. The `OnCreate` method of our `MainActivity` must initialize this layout, instantiate the adapter, and prepare the underlying data source:

```
public class MainActivity : Activity
{
    RecyclerView mRecyclerView;
    RecyclerView.LayoutManager mLayoutManager;
    PhotoAlbumAdapter mAdapter;
    PhotoAlbum mPhotoAlbum;
    protected override void OnCreate (Bundle bundle)
    {
        base.OnCreate (bundle);

        // Prepare the data source:
        mPhotoAlbum = new PhotoAlbum();

        // Instantiate the adapter and pass in its data source:
        mAdapter = new PhotoAlbumAdapter (mPhotoAlbum);

        // Set our view from the "main" layout resource:
        SetContentView (Resource.Layout.Main);

        // Get our RecyclerView layout:
        mRecyclerView = FindViewById<RecyclerView> (Resource.Id.recyclerView);

        // Plug the adapter into the RecyclerView:
        mRecyclerView.SetAdapter (mAdapter);
```

This code instantiates the `PhotoAlbum` data source and passes it to our adapter, `PhotoAlbumAdapter` (which is defined later in this guide). Note that it is considered a best practice to pass the data source as a parameter to the constructor of the adapter. To plug our adapter into the `RecyclerView` instance, we call the **RecyclerView** `SetAdapter` method as shown above.

The Layout Manager

The layout manager is responsible for positioning items in the **RecyclerView** display; it determines the presentation type (a list or a grid), the orientation (whether items are displayed vertically or horizontally), and which direction items should be displayed (in normal order or in reverse order). The layout manager is also responsible for calculating the size and position of each item in the **RecycleView** display.

The layout manager has an additional purpose: it determines the policy for when to recycle item views that are no longer visible to the user. Because the layout manager is aware of which views are visible (and which are not), it is in the best position to decide when a view can be recycled. To recycle a view, the layout manager typically makes calls to the adapter to replace the contents of a recycled view with different data, as described previously in How View Recycling Works.

We can extend `RecyclerView.LayoutManager` to create our own layout manager, or we can use a predefined layout manager. **RecyclerView** provides the following predefined layout managers:

- **`LinearLayoutManager`** – Arranges items in a column that can be scrolled vertically, or in a row that can be scrolled horizontally.
- **`GridLayoutManager`** – Displays items in a grid.
- **`StaggeredGridLayoutManager`** – Displays items in a staggered grid, where some items have different heights and widths.

To specify the layout manager, we instantiate our chosen layout manager and pass it to the `SetLayoutManager` method as shown in the next code example. Note that we *must* specify the layout manager – **RecyclerView** does not select a predefined layout manager by default. The following snippet is an example of how to instantiate the `LinearLayoutManager` and plug it into the `RecyclerView` instance:

```
mLayoutManager = new LinearLayoutManager (this);
mRecyclerView.SetLayoutManager (mLayoutManager);
```

This code resides in the main activity's `OnCreate` method. The constructor to the layout manager requires a *context*; typically, you supply the entire `MainActivity` by passing `this` as shown.

In the example photo-viewing app, the predefind `LinearLayoutManager` is used to lay out each `CardView` in a vertical scrolling arrangement. Alternatively, we could plug in a custom layout manager that displays two `CardView` items side-by-side, implementing a page-turning animation effect to traverse through the collection of photos. Later in this guide, we'll provide an example of how to modify the layout by swapping in a different layout manager.

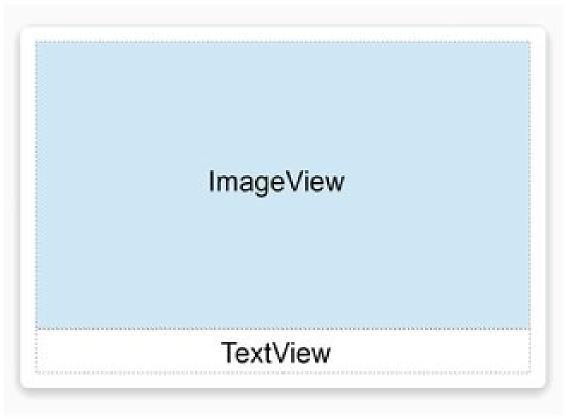
For more information about the layout manager, see the RecyclerView.LayoutManager class reference.

The View Holder

The view holder is a class that we define for caching our view references. The adapter uses these view references to bind each view to its content. Every item in the **RecyclerView** has an associated view holder instance that caches the view references for that item. To create a view holder, we use the following steps to define a class to hold our exact set of views per item:

1. Subclass `RecyclerView.ViewHolder`.
2. Implement a constructor that looks up and stores the view references.
3. Implement properties that the adapter can use to access these references.

For example, in the RecyclerViewer sample app, the view holder class is called `PhotoViewHolder`. Each `PhotoViewHolder` instance holds references to the `ImageView` and `TextView` of an associated row item, which is laid out in a `CardView` as diagrammed here:



`PhotoViewHolder` derives from `RecyclerView.ViewHolder` and contains properties to store references to the `ImageView` and `TextView` shown in the above layout. `PhotoViewHolder` consists of two properties and one constructor:

```
public class PhotoViewHolder : RecyclerView.ViewHolder
{
    public ImageView Image { get; private set; }
    public TextView Caption { get; private set; }

    public PhotoViewHolder (View itemView) : base (itemView)
    {
        // Locate and cache view references:
        Image = itemView.FindViewById<ImageView> (Resource.Id.imageView);
        Caption = itemView.FindViewById<TextView> (Resource.Id.textView);
    }
}
```

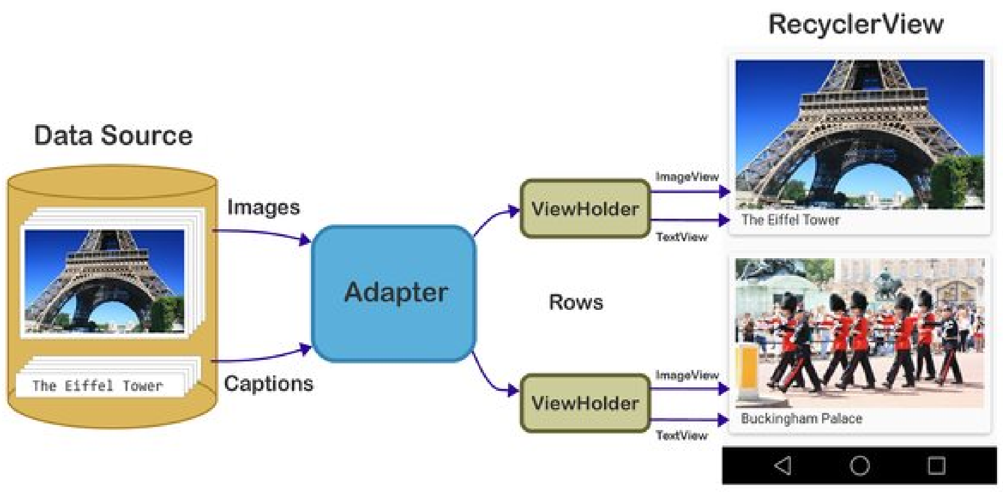
In this code example, the `PhotoViewHolder` constructor is passed a reference to the parent item view (the **CardView**) that `PhotoViewHolder` wraps. Note that we always forward the parent item view to the base constructor. The `PhotoViewHolder` constructor calls `FindViewById` on the parent item view to locate each of its child view references, `ImageView` and `TextView`, storing the results in the `Image` and `Caption` properties, respectively. The adapter later retrieves view references from these properties when it updates this **CardView**'s child views with new data.

For more information about `RecyclerView.ViewHolder`, see the [RecyclerView.ViewHolder class reference](#).

The Adapter

Most of the "heavy-lifting" of our **RecyclerView** integration code takes place in the adapter. **RecyclerView** requires that we provide an adapter derived from `RecyclerView.Adapter` to access our data source and populate each item with content from the data source. Because the data source is app-specific, we must implement adapter functionality that understands how to access our data. The adapter extracts information from the data source and loads it into each item in the **RecyclerView** collection.

The following drawing illustrates how the sample app adapter maps content in the data source (the photo album) through view holders to individual views within each **CardView** row item in the **RecyclerView**:



In the example photo-viewing app, the adapter loads each **RecyclerView** row with data for a particular photograph. For a given photograph at row position *P*, for example, the adapter locates the associated data at position *P* within the data source and copies this data to the row item at position *P* in the **RecyclerView** collection. The adapter uses the view holder to lookup the references for the `ImageView` and `TextView` at that position so it doesn't have to repeatedly call `FindViewById` for those views as the user scrolls through the photograph collection and reuses views.

In the example photo-viewer app, we derive from `RecyclerView.Adapter` to create `PhotoAlbumAdapter`:

```
public class PhotoAlbumAdapter : RecyclerView.Adapter
{
    public PhotoAlbum mPhotoAlbum;

    public PhotoAlbumAdapter (PhotoAlbum photoAlbum)
    {
        mPhotoAlbum = photoAlbum;
    }
    ...
}
```

The `mPhotoAlbum` member contains the data source (the photo album) that is passed into the constructor; the constructor copies the photo album into this member variable. Next, we must override the following `RecyclerView.Adapter` methods:

- **`OnCreateViewHolder`** – Instantiates the item layout file and view holder.
- **`OnBindViewHolder`** – Loads the data at the specified position into the views whose references are stored in the given view holder.
- **`ItemCount`** – Returns the number of items in the data source.

The layout manager calls these methods while it is positioning items within the **RecyclerView**. In the next few sections, we'll look at the actual implementation of these methods in the example photo-viewing app.

OnCreateViewHolder

The layout manager calls `OnCreateViewHolder` when the **RecyclerView** needs a new view holder to represent an item. `OnCreateViewHolder` inflates the item view from the view's layout file and wraps the view in a new `PhotoViewHolder` instance. The `PhotoViewHolder` constructor locates and stores references to child views in the layout as described previously in The View Holder.

In the example photo-viewing app, each row item is represented by a `CardView` that contains an `ImageView` (for the photo) and a `TextView` (for the caption). This layout resides in the file **PhotoCardView.axml**:

```
<?xml version="1.0" encoding="utf-8"?>
<FrameLayout xmlns:card_view="http://schemas.android.com/apk/res-auto"
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content">
    <android.support.v7.widget.CardView
```

```
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        card_view:cardElevation="4dp"
        card_view:cardUseCompatPadding="true"
        card_view:cardCornerRadius="5dp">
        <LinearLayout
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            android:orientation="vertical"
            android:padding="8dp">
            <ImageView
                android:layout_width="match_parent"
                android:layout_height="wrap_content"
                android:id="@+id/imageView"
                android:scaleType="centerCrop" />
            <TextView
                android:layout_width="match_parent"
                android:layout_height="wrap_content"
                android:textAppearance="?android:attr/textAppearanceMedium"
                android:textColor="#333333"
                android:text="Caption"
                android:id="@+id/textView"
                android:layout_gravity="center_horizontal"
                android:layout_marginLeft="4dp" />
        </LinearLayout>
    </android.support.v7.widget.CardView>
</FrameLayout>
```

This layout represents a single row item in the **RecyclerView**. `OnBindViewHolder` (described below) copies data from the data source into the `ImageView` and `TextView` of this layout. `OnCreateViewHolder` inflates this layout for a given photo location in the **RecyclerView** and instantiates a new `PhotoViewHolder` instance (which locates and caches references to the `ImageView` and `TextView` child views in the associated `CardView` layout):

```
public override RecyclerView.ViewHolder
    OnCreateViewHolder (ViewGroup parent, int viewType)
{
    // Inflate the CardView for the photo:
```

```
    View itemView = LayoutInflater.From (parent.Context).
                Inflate (Resource.Layout.PhotoCardView, parent, false);

    // Create a ViewHolder to hold view references inside the CardView:
    PhotoViewHolder vh = new PhotoViewHolder (itemView);
    return vh;
}
```

The resulting view holder instance, `vh`, is returned back to the caller (the layout manager).

OnBindViewHolder

When the layout manager is ready to display a particular view in the **RecyclerView**'s visible screen area, it calls the adapter's `OnBindViewHolder` method to fill the item at the specified row position with content from the data source. In the photo-viewing app, `OnBindViewHolder` gets the photo information for the specified row position (the photo's image resource and the string for the photo's caption) and copies this data to the associated views. Views are located via references stored in the view holder object (which is passed in through the `holder` parameter):

```
public override void
    OnBindViewHolder (RecyclerView.ViewHolder holder, int position)
{
    PhotoViewHolder vh = holder as PhotoViewHolder;

    // Load the photo image resource from the photo album:
    vh.Image.SetImageResource (mPhotoAlbum[position].PhotoID);

    // Load the photo caption from the photo album:
    vh.Caption.Text = mPhotoAlbum[position].Caption;
}
```

The passed-in view holder object must first be cast into the derived view holder type (in this case, `PhotoViewHolder`) before it is used. The adapter loads the image resource into the view referenced by the view holder's `Image` property, and it copies the caption text into the view referenced by the view holder's `Caption` property. This *binds* the associated view with its data.

Notice that `OnBindViewHolder` is the code that deals directly with the structure of the data. In this case, `OnBindViewHolder` understands how to map the **RecyclerView** item position to its associated data item in the data source. The mapping is trivial in this case because the position can be used as an array index into the photo album; however, more complex data sources may require extra code to establish such a mapping.

ItemCount

The `ItemCount` method returns the number of items in the data collection. In the example photo viewer app, the item count is the number of photos in the photo album:

```
public override int ItemCount
{
    get { return mPhotoAlbum.NumPhotos; }
}
```

For more information about `RecyclerView.Adapter`, see the RecyclerView.Adapter class reference.

Putting it All Together

The resulting **RecyclerView** implementation for the example photo app consists of `MainActivity` initialization code, the view holder, and the adapter. `MainActivity` creates the `mRecyclerView` instance, instantiates the data source and the adapter, and plugs in the layout manager and adapter:

```
public class MainActivity : Activity
{
    RecyclerView mRecyclerView;
    RecyclerView.LayoutManager mLayoutManager;
    PhotoAlbumAdapter mAdapter;
    PhotoAlbum mPhotoAlbum;

    protected override void OnCreate (Bundle bundle)
    {
        base.OnCreate (bundle);
        mPhotoAlbum = new PhotoAlbum();
        SetContentView (Resource.Layout.Main);
        mRecyclerView = FindViewById<RecyclerView> (Resource.Id.recyclerView);

        // Plug in the linear layout manager:
        mLayoutManager = new LinearLayoutManager (this);
        mRecyclerView.SetLayoutManager (mLayoutManager);
        // Plug in my adapter:
        mAdapter = new PhotoAlbumAdapter (mPhotoAlbum);
        mRecyclerView.SetAdapter (mAdapter);
    }
}
```

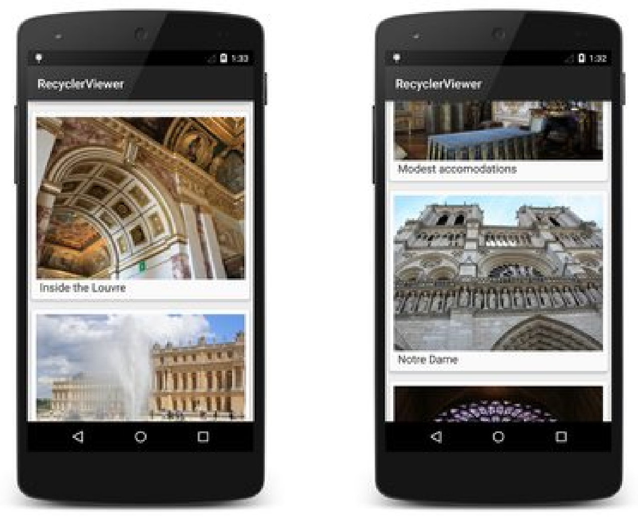
`PhotoViewHolder` locates and caches the view references:

```
public class PhotoViewHolder : RecyclerView.ViewHolder
{
    public ImageView Image { get; private set; }
    public TextView Caption { get; private set; }

    public PhotoViewHolder (View itemView) : base (itemView)
    {
        // Locate and cache view references:
        Image = itemView.FindViewById<ImageView> (Resource.Id.imageView);
        Caption = itemView.FindViewById<TextView> (Resource.Id.textView);
    }
}
```

`PhotoAlbumAdapter` implements the three required method overrides:

```
public class PhotoAlbumAdapter : RecyclerView.Adapter
{
    public PhotoAlbum mPhotoAlbum;
    public PhotoAlbumAdapter (PhotoAlbum photoAlbum)
    {
        mPhotoAlbum = photoAlbum;
    }

    public override RecyclerView.ViewHolder
        OnCreateViewHolder (ViewGroup parent, int viewType)
    {
        View itemView = LayoutInflater.From (parent.Context).
                    Inflate (Resource.Layout.PhotoCardView, parent, false);
        PhotoViewHolder vh = new PhotoViewHolder (itemView);
        return vh;
    }
    public override void
        OnBindViewHolder (RecyclerView.ViewHolder holder, int position)
    {
        PhotoViewHolder vh = holder as PhotoViewHolder;
        vh.Image.SetImageResource (mPhotoAlbum[position].PhotoID);
        vh.Caption.Text = mPhotoAlbum[position].Caption;
    }

    public override int ItemCount
    {
        get { return mPhotoAlbum.NumPhotos; }
    }
}
```

When this code is compiled and run, it creates the basic photo viewing app as shown in the following screenshots:



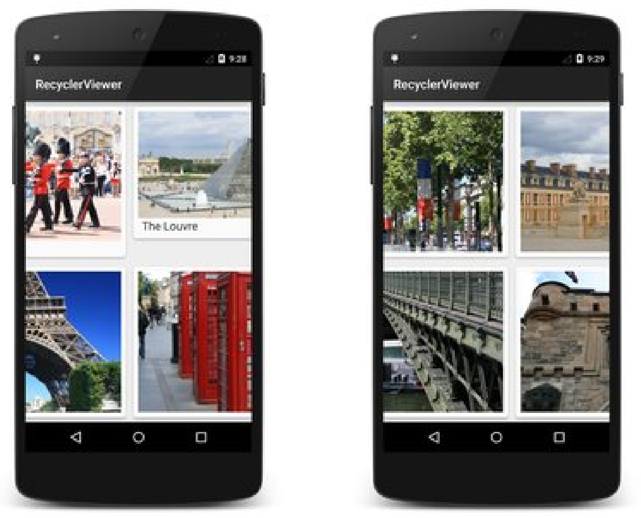
This basic app does not respond to item-touch events, nor does it handle changes in the underlying data. Later in this guide, we'll add this functionality to the sample app.

Changing the LayoutManager

Because of **RecyclerView**'s flexibility, it's easy to modify the app to use a different layout manager. In this example, we'll modify it to display with a grid layout that scrolls horizontally rather than with a vertical linear layout. To do this, we modify the layout manager instantiation to use the `GridLayoutManager` as follows:

```
mLayoutManager = new GridLayoutManager(this, 2, GridLayoutManager.Horizontal,
false);
```

This code change replaces the vertical `LinearLayoutManager` with a `GridLayoutManager` that presents a grid made up of two rows that scroll in the horizontal direction. When we compile and run the app again, we see that the photographs are displayed in a grid and that scrolling is horizontal rather than vertical:



By changing only one line of code, we were able to modify the photo-viewing app to use a different layout with different behavior. Notice that neither the adapter code nor the layout XML had to be modified in order to change the layout policy.

Extending the Basic Example

The basic example described in the previous sections actually doesn't do much – it simply scrolls and displays a fixed list of photograph items. In real-world applications, users expect to be able to interact with the app by tapping on items in the display. Also, the underlying data source often changes (or is changed by the app), and the contents of the display must be consistent with these changes. In the following sections, we'll learn how to handle item-click events and update **RecyclerView** when the underlying data source changes.

Handling Item-Click Events

When a user touches an item in the **RecyclerView**, an item-click event is generated to notify our app as to which item was touched. This event is not generated by **RecyclerView** – instead, the item view (which is wrapped in the view holder) detects touches and reports these touches as click events.

To illustrate how to handle item-click events, we'll modify the sample photo-viewing app to report which photograph had been touched by the user. When an item-click event occurs in the sample app, the following sequence takes place:

1. The photograph's **CardView** detects the item-click event and notifies the adapter.
2. The adapter forwards the event (with item position information) to the activity's item-click handler.
3. The activity's item-click handler responds to the item-click event.

First, let's add an event handler member called `ItemClick` to our `PhotoAlbumAdapter` class definition:

```
public event EventHandler<int> ItemClick;
```

Next, we'll create an item-click event handler method in our `MainActivity`. This handler briefly displays a toast that indicates which photograph item was touched:

```
void OnItemClick (object sender, int position)
{
    int photoNum = position + 1;
    Toast.MakeText(this, "This is photo number " + photoNum,
ToastLength.Short).Show();
}
```

Next, let's add a line of code to register our `OnItemClick` handler with `PhotoAlbumAdapter`. A good place to do this is immediately after `PhotoAlbumAdapter` is created (in the main activity's `OnCreate` method):

```
mAdapter = new PhotoAlbumAdapter (mPhotoAlbum);
mAdapter.ItemClick += OnItemClick;
```

`PhotoAlbumAdapter` will now call `OnItemClick` when it receives an item-click event. Next, let's create a handler in the adapter that raises this `ItemClick` event. We'll add the following method, `OnClick`, immediately after the adapter's `ItemCount` method:

```
void OnClick (int position)
```

```
{
    if (ItemClick != null)
        ItemClick (this, position);
}
```

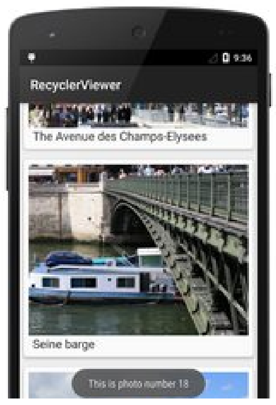
This `OnClick` method is the adapter's *listener* for item-click events from item views. Before we can register this listener with an item view (via the item view's view holder), we have to modify the `PhotoViewHolder` constructor to accept this method as an additional argument, and register `OnClick` with the item view `Click` event. Here's the modified `PhotoViewHolder` constructor:

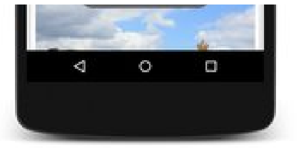
```
public PhotoViewHolder (View itemView, Action<int> listener)
    : base (itemView)
{
    Image = itemView.FindViewById<ImageView> (Resource.Id.imageView);
    Caption = itemView.FindViewById<TextView> (Resource.Id.textView);

    itemView.Click += (sender, e) => listener (base.Position);
}
```

The `itemView` parameter contains a reference to the **CardView** that was touched by the user. Note that the view holder base class knows the position of the item (**CardView**) that it represents (via the `Position` property), and this position is passed to the adapter's `OnClick` method when an item-click event takes place. Let's modify the adapter's `OnCreateViewHolder` method to pass the adapter's `OnClick` method to the view-holder's constructor:

```
PhotoViewHolder vh = new PhotoViewHolder (itemView, OnClick);
```

Now when we build and run the sample photo-viewing app, tapping a photo in the display will cause a toast to appear that reports which photograph was touched:





This example demonstrates just one approach for implementing event handlers with **RecyclerView**. Another approach that could be used here is to place events on the view holder and have the adapter subscribe to these events. If the sample photo app provided a photo editing capability, separate events would be required for the `ImageView` and the `TextView` within each **CardView**: touches on the `TextView` would launch an `EditView` dialog that lets the user edit the caption, and touches on the `ImageView` would launch a photo touchup tool that lets the user crop or rotate the photo. Depending on the needs of your app, you must design the best approach for handling and responding to touch events.

Notifying RecyclerView of Data Changes

RecyclerView does not automatically update its display when the contents of its data source changes; the adapter must notify **RecyclerView** when there is a change in the data set. The data set can change in many ways; for example, the contents within an item can change or the overall structure of the data may be altered. `RecyclerView.Adapter` provides a number of methods that you can call so that **RecyclerView** responds to data changes in the most efficient manner:

- **`NotifyItemChanged`** – Signals that the item at the specified position has changed.
- **`NotifyItemRangeChanged`** – Signals that the items in the specified range of positions have changed.
- **`NotifyItemInserted`** – Signals that the item in the specified position has been newly inserted.
- **`NotifyItemRangeInserted`** – Signals that the items in the specified range of positions have been newly inserted.
- **`NotifyItemRemoved`** – Signals that the item in the specified position has been removed.
- **`NotifyItemRangeRemoved`** – Signals that the items in the specified range of positions have been removed.
- **`NotifyDataSetChanged`** – Signals that the data set has changed (forces a full update).

If you know exactly how your data set has changed, you can call the appropriate methods above to refresh **RecyclerView** in the most efficient manner. If you do not know exactly how your data set has changed, you can call `NotifyDataSetChanged`, which is far less efficient because **RecyclerView** must refresh all the views that are visible to the user. For more information about these methods, see RecyclerView.Adapter.

To demonstrate how **RecyclerView** can be updated when the data set changes, we'll modify the sample photo-viewing app to randomly pick a photo in the data source and swap it with the first photo. First, let's add a Random Pick button to the example photo app's **Main.axml** layout:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent">
    <Button
        android:id="@+id/randPickButton"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:gravity="center_horizontal"
        android:textAppearance="?android:attr/textAppearanceLarge"
        android:text="Random Pick" />
    <android.support.v7.widget.RecyclerView
        android:id="@+id/recyclerView"
        android:scrollbars="vertical"
        android:layout_width="fill_parent"
        android:layout_height="fill_parent" />
</LinearLayout>
```

Next, let's add code at the end of the main activity's `OnCreate` method to locate the `Random Pick` button in the layout and attach a handler to it:

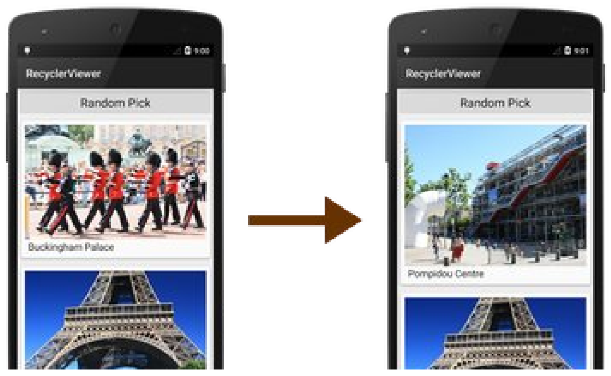
```
Button randomPickBtn = FindViewById<Button>(Resource.Id.randPickButton);

randomPickBtn.Click += delegate
{
    if (mPhotoAlbum != null)
    {
        // Randomly swap a photo with the first photo:
        int idx = mPhotoAlbum.RandomSwap();
    }
};
```

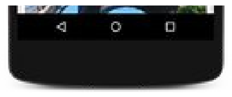
This handler calls the photo album's `RandomSwap` method when the Random Pick button is tapped. The `RandomSwap` method randomly swaps a photo with the first photo in the data source, then returns the index of the randomly-swapped photo. When we compile and run the sample app with this code, tapping the Random Pick button does not result in a display change because the **RecyclerView** is not aware of the change to the data source.

To keep **RecyclerView** updated after the data source changes, we must modify the Random Pick click handler to call the adapter's `NotifyItemChanged` method for each item in the collection that has changed (in this case, two items have changed: the first photo and the swapped photo). This causes **RecyclerView** to update its display so that it is consistent with the new state of the data source:

```
Button randomPickBtn = FindViewById<Button>(Resource.Id.randPickButton);

randomPickBtn.Click += delegate
{
    if (mPhotoAlbum != null)
    {
        int idx = mPhotoAlbum.RandomSwap();

        // First photo has changed:
        mAdapter.NotifyItemChanged(0);

        // Swapped photo has changed:
        mAdapter.NotifyItemChanged(idx);
    }
};
```

Now, when the Random Pick button is tapped, **RecyclerView** updates the display to show that a photo further down in the collection has been swapped with the first photo in the collection:





Of course, we could call `NotifyDataSetChanged` instead of making the two calls to `NotifyItemChanged`, but doing so would force `RecyclerView` to refresh the entire collection even though only two items in the collection had changed. Calling `NotifyItemChanged` is significantly more efficient than calling `NotifyDataSetChanged`.

Summary

This guide introduced the Android **RecyclerView** widget; it explained how to add the **RecyclerView** support library to Xamarin.Android projects, how **RecyclerView** recycles views, how it enforces the view-holder pattern for efficiency, and how the various helper classes that make up **RecyclerView** collaborate to display collections. It provided example code to demonstrate how **RecyclerView** is integrated into an application, it demonstrated how to tailor **RecyclerView**'s layout policy by plugging in different layout managers, and it explained how to handle item click events and notify **RecyclerView** of data source changes.

For more information about **RecyclerView**, see the RecyclerView class reference.