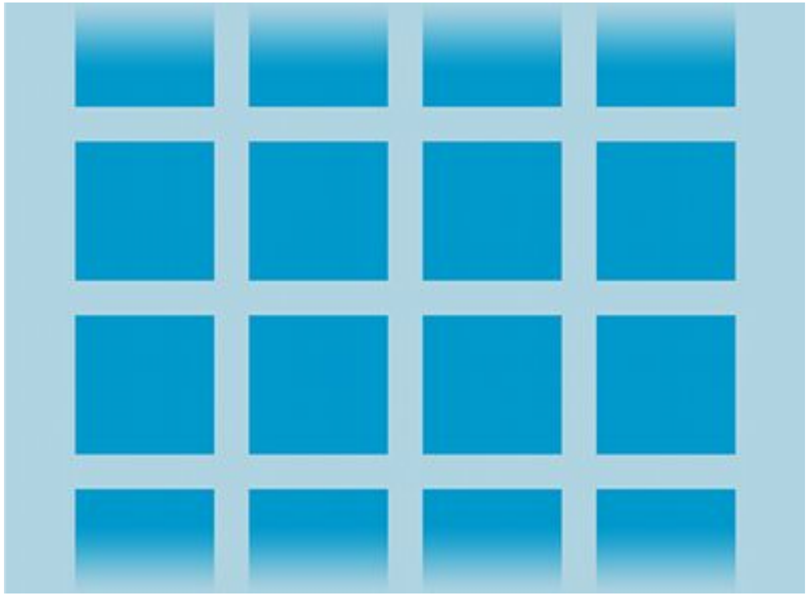
Mobile programming

Omid Jafarinezhad

Android 04

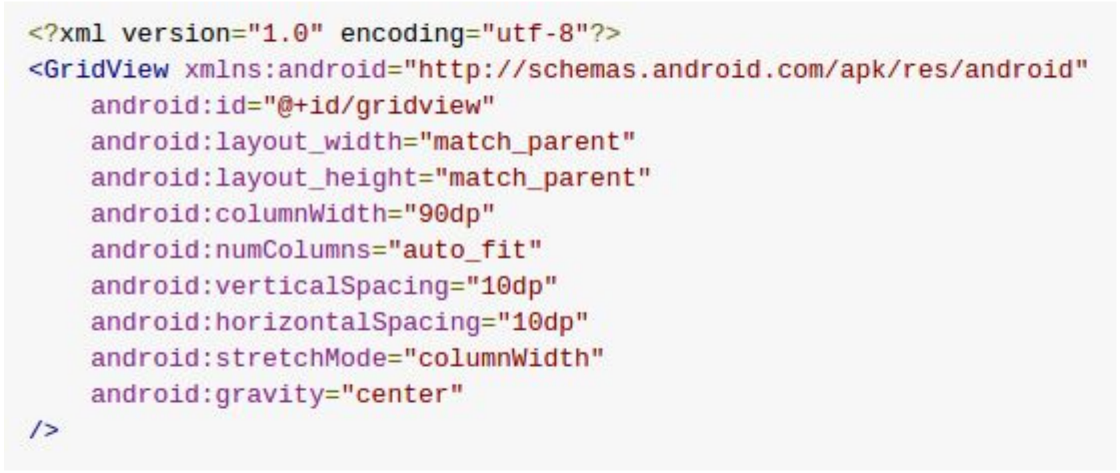
Grid View

GridView is a **ViewGroup** that displays items in a two-dimensional, scrollable grid. The grid items are automatically inserted to the layout using a **ListAdapter**



Create a grid of image thumbnails

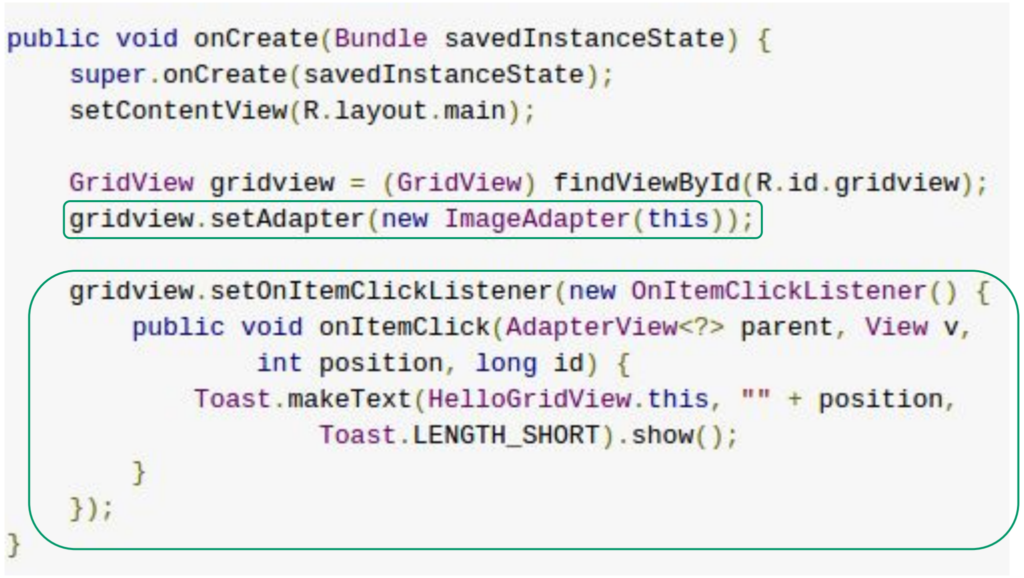
- Start a new project named HelloGridView
- Save the image files into the project's res/drawable/ directory
- Open the res/layout/main.xml file and insert the following
  This GridView will fill the entire screen



```xml
<?xml version="1.0" encoding="utf-8"?>
<GridView xmlns:android="http://schemas.android.com/apk/res/android"
    android:id="@+id/gridview"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:columnWidth="90dp"
    android:numColumns="auto_fit"
    android:verticalSpacing="10dp"
    android:horizontalSpacing="10dp"
    android:stretchMode="columnWidth"
    android:gravity="center"
/>
```

Create a grid of image thumbnails (2)

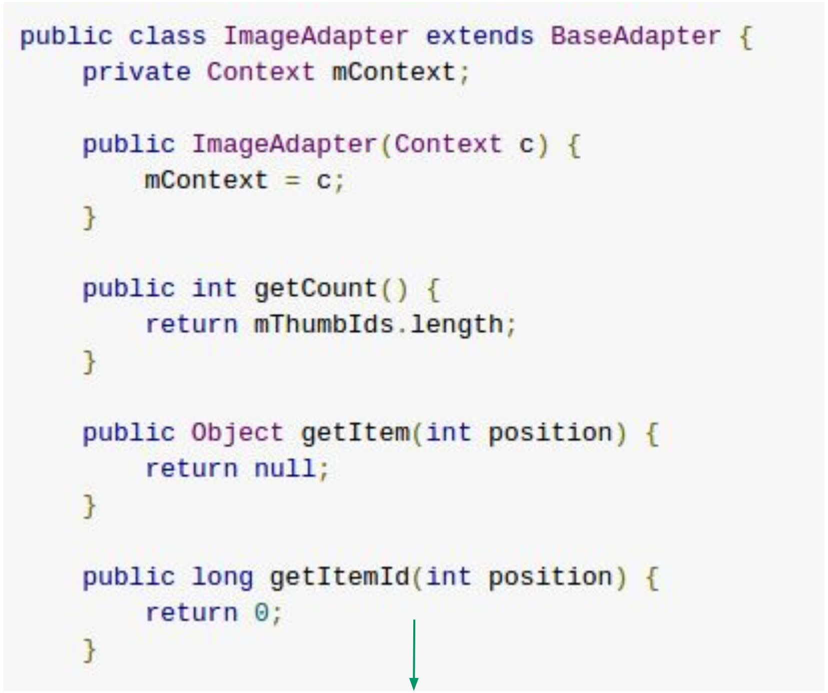
- Open HelloGridView.java and insert the following code for the **onCreate**() method



```java
public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main);

    GridView gridview = (GridView) findViewById(R.id.gridview);
    gridview.setAdapter(new ImageAdapter(this));

    gridview.setOnItemClickListener(new OnItemClickListener() {
        public void onItemClick(AdapterView<?> parent, View v,
                int position, long id) {
            Toast.makeText(HelloGridView.this, "" + position,
                    Toast.LENGTH_SHORT).show();
        }
    });
}
```

Create a grid of image thumbnails (3)

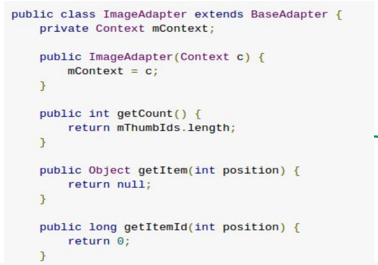
- Create a new class called ImageAdapter that extends BaseAdapter:



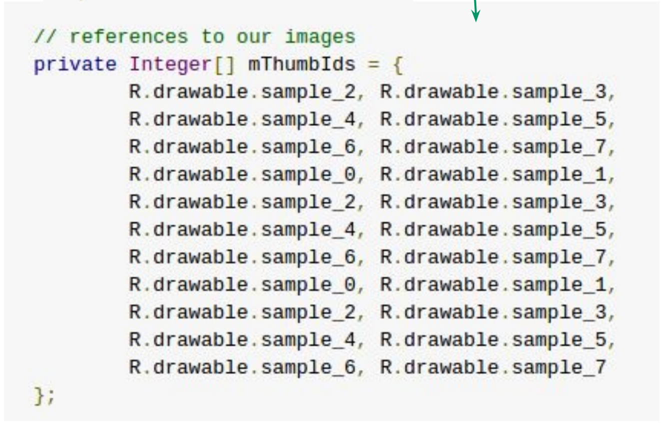
```java
public class ImageAdapter extends BaseAdapter {
    private Context mContext;

    public ImageAdapter(Context c) {
        mContext = c;
    }

    public int getCount() {
        return mThumbIds.length;
    }

    public Object getItem(int position) {
        return null;
    }

    public long getItemId(int position) {
        return 0;
    }
```

Create a grid of image thumbnails (4)

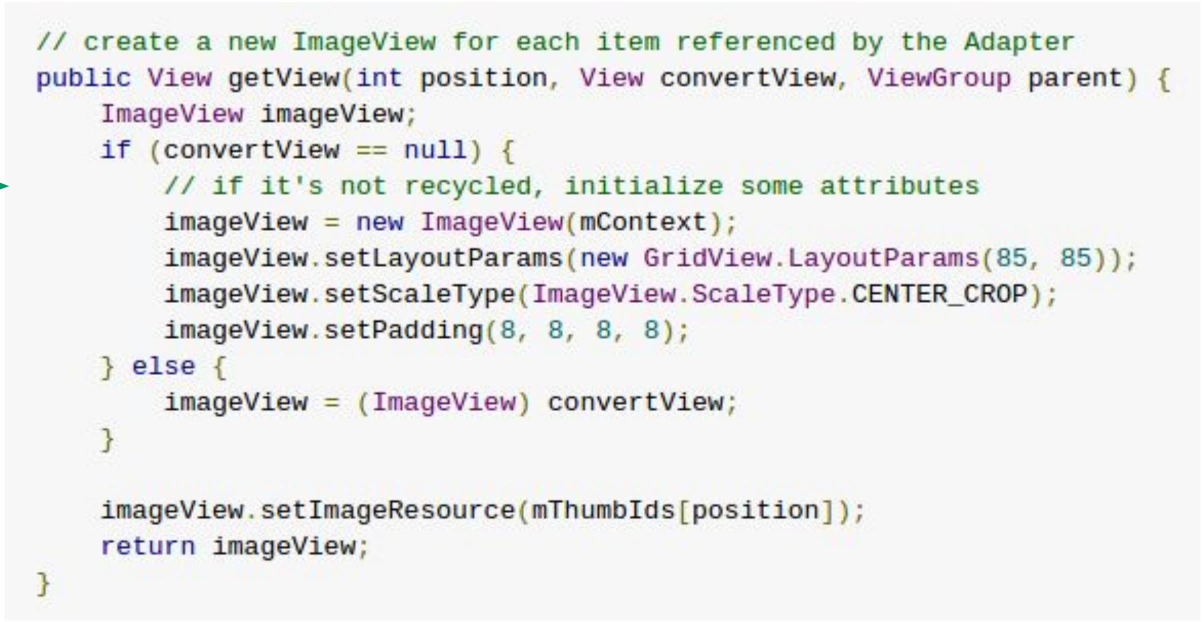
- Create a new class called ImageAdapter that extends BaseAdapter:



```java
public class ImageAdapter extends BaseAdapter {
    private Context mContext;

    public ImageAdapter(Context c) {
        mContext = c;
    }

    public int getCount() {
        return mThumbIds.length;
    }

    public Object getItem(int position) {
        return null;
    }

    public long getItemId(int position) {
        return 0;
    }
```



```java
// references to our images
private Integer[] mThumbIds = {
        R.drawable.sample_2, R.drawable.sample_3,
        R.drawable.sample_4, R.drawable.sample_5,
        R.drawable.sample_6, R.drawable.sample_7,
        R.drawable.sample_0, R.drawable.sample_1,
        R.drawable.sample_2, R.drawable.sample_3,
        R.drawable.sample_4, R.drawable.sample_5,
        R.drawable.sample_6, R.drawable.sample_7,
        R.drawable.sample_0, R.drawable.sample_1,
        R.drawable.sample_2, R.drawable.sample_3,
        R.drawable.sample_4, R.drawable.sample_5,
        R.drawable.sample_6, R.drawable.sample_7
};
```



```java
// create a new ImageView for each item referenced by the Adapter
public View getView(int position, View convertView, ViewGroup parent) {
    ImageView imageView;
    if (convertView == null) {
        // if it's not recycled, initialize some attributes
        imageView = new ImageView(mContext);
        imageView.setLayoutParams(new GridView.LayoutParams(85, 85));
        imageView.setScaleType(ImageView.ScaleType.CENTER_CROP);
        imageView.setPadding(8, 8, 8, 8);
    } else {
        imageView = (ImageView) convertView;
    }

    imageView.setImageResource(mThumbIds[position]);
    return imageView;
}
```

Hold View Objects in a View Holder

Your code might call **findViewById() frequently during the scrolling** of ListView/GridView, which can slow down performance

A way around repeated use of findViewById() is to use the "***view holder***" design pattern

Hold View Objects in a View Holder

A ViewHolder object stores each of the component views inside the tag field of the Layout, so you can immediately access them without the need to look them up repeatedly. First, you need to create a class to hold your exact set of views. For example:

```
static class ViewHolder {
  TextView text;
  TextView timestamp;
  ImageView icon;
  ProgressBar progress;
  int position;
}
```

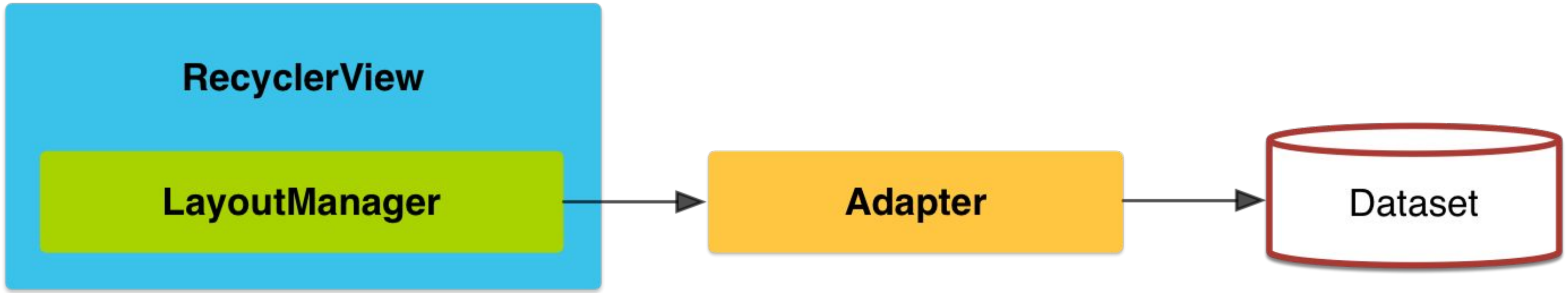
```
ViewHolder holder = new ViewHolder();
holder.icon = (ImageView) convertView.findViewById(R.id.listitem_image);
holder.text = (TextView) convertView.findViewById(R.id.listitem_text);
holder.timestamp = (TextView) convertView.findViewById(R.id.listitem_timestamp);
holder.progress = (ProgressBar) convertView.findViewById(R.id.progress_spinner);
convertView.setTag(holder);
```

```java
public View getView(int position, View convertView, ViewGroup parent) {
        // Avoid unneccessary calls to findViewById() on each row, which is expensive!
        ViewHolder holder;

        /*
         * If convertView is not null, we can reuse it directly, no inflation required!
         * We only inflate a new View when the convertView is null.
         */
        if (convertView == null) {
            convertView = inflater.inflate(R.layout.list_item_pocket, null);

            // Create a ViewHolder and store references to the two children views
            holder = new ViewHolder();
            holder.title = (TextView) convertView.findViewById(R.id.title);
            holder.subtitle = (TextView) convertView.findViewById(R.id.subtitle);
            holder.icon = (ImageView) convertView.findViewById(R.id.icon);

            // The tag can be any Object, this just happens to be the ViewHolder
            convertView.setTag(holder);
        } else {
            // Get the ViewHolder back to get fast access to the TextView
            // and the ImageView.
            holder = (ViewHolder) convertView.getTag();
        }

        // Bind that data efficiently!
        holder.title.setText(someTitle[position]);
        holder.subtitle.setText(someSubtitle[position]);
        holder.icon.setImageBitmap((position % 2) == 0 ? bitmap1 : bitmap2);

        return convertView;
```

RecyclerView

- The **RecyclerView** widget is a more **advanced and flexible version of ListView**. This widget is a container **for displaying large data sets** that can be scrolled **very efficiently by maintaining a limited number of views**. Use the RecyclerView widget when you have data collections whose elements change at runtime based on user action or network events
- The RecyclerView class simplifies the display and handling of large data sets by providing:
  - **Layout managers** for positioning items
  - Default **animations for common item operations**, such as removal or addition of items

RecyclerView



RecyclerView - layout manager

A layout manager positions item views inside a RecyclerView and **determines when to reuse item views that are no longer visible to the user**. To reuse (or recycle) a view, a layout manager may ask the adapter to replace the contents of the view with a different element from the dataset. Recycling views in this manner improves performance by avoiding the creation of unnecessary views or performing expensive findViewById() lookups

RecyclerView - layout manager (2)

RecyclerView provides these built-in layout managers:

- **LinearLayoutManager** shows items in a vertical or horizontal scrolling list
- **GridLayoutManager** shows items in a grid
- **StaggeredGridLayoutManager** shows items in a staggered grid

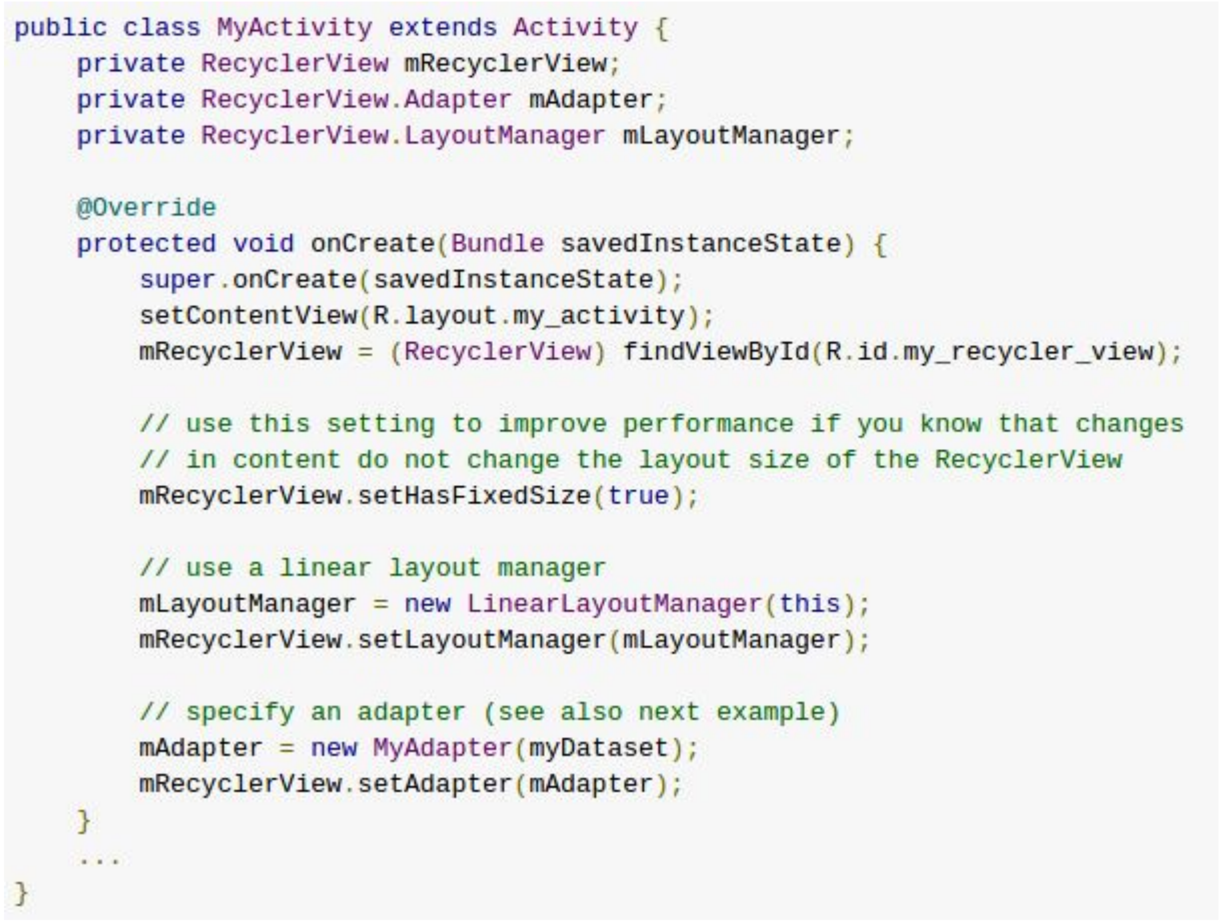
RecyclerView - Animations

Animations for adding and removing items are enabled by default in RecyclerView. **To customize these animations, extend the RecyclerView.ItemAnimator class and use the RecyclerView.setItemAnimator() method**

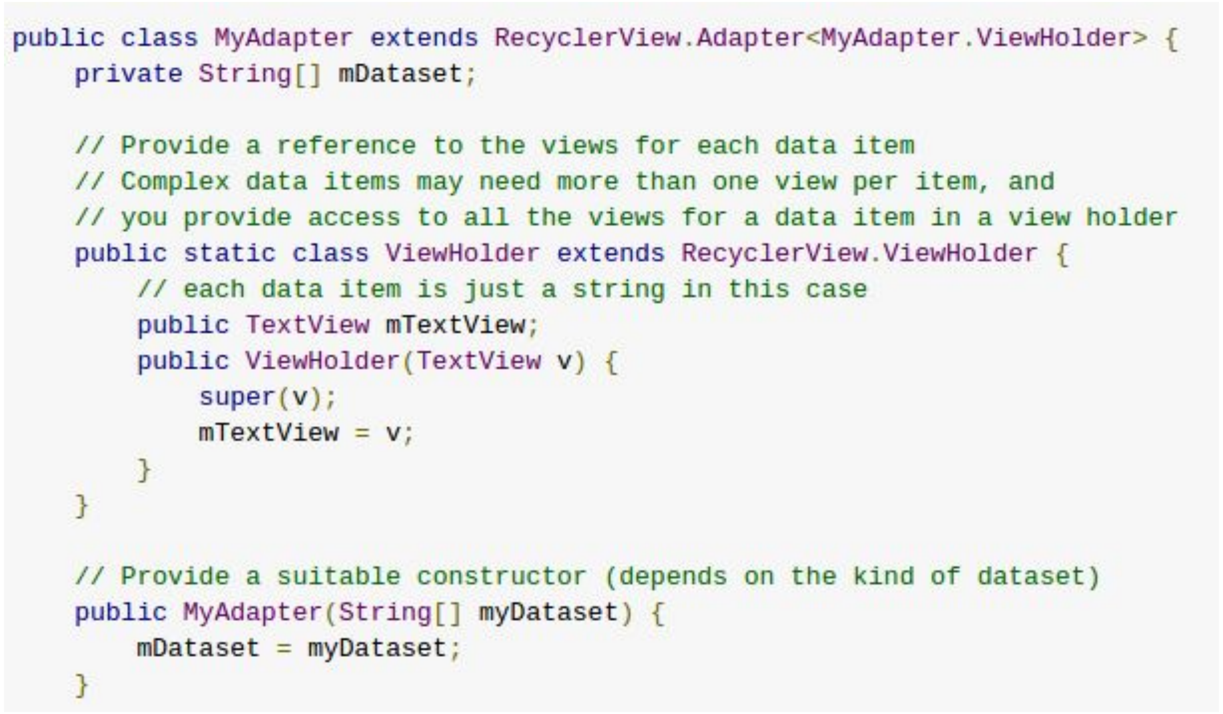
Example

```xml
<!-- A RecyclerView with some commonly used attributes -->
<android.support.v7.widget.RecyclerView
    android:id="@+id/my_recycler_view"
    android:scrollbars="vertical"
    android:layout_width="match_parent"
    android:layout_height="match_parent"/>
```

Example



```java
public class MyActivity extends Activity {
    private RecyclerView mRecyclerView;
    private RecyclerView.Adapter mAdapter;
    private RecyclerView.LayoutManager mLayoutManager;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.my_activity);
        mRecyclerView = (RecyclerView) findViewById(R.id.my_recycler_view);

        // use this setting to improve performance if you know that changes
        // in content do not change the layout size of the RecyclerView
        mRecyclerView.setHasFixedSize(true);

        // use a linear layout manager
        mLayoutManager = new LinearLayoutManager(this);
        mRecyclerView.setLayoutManager(mLayoutManager);

        // specify an adapter (see also next example)
        mAdapter = new MyAdapter(myDataset);
        mRecyclerView.setAdapter(mAdapter);
    }
    ...
}
```



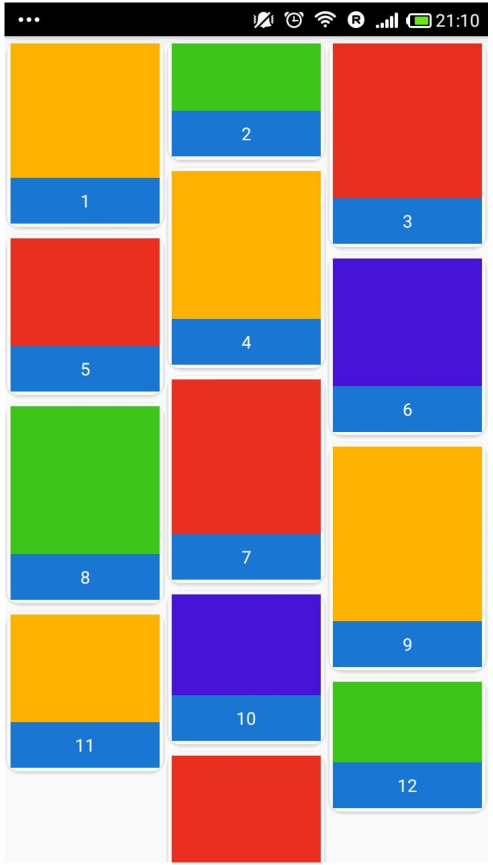
```
public class MyAdapter extends RecyclerView.Adapter<MyAdapter.ViewHolder> {
    private String[] mDataset;

    // 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
    public static class ViewHolder extends RecyclerView.ViewHolder {
        // each data item is just a string in this case
        public TextView mTextView;
        public ViewHolder(TextView v) {
            super(v);
            mTextView = v;
        }
    }

    // Provide a suitable constructor (depends on the kind of dataset)
    public MyAdapter(String[] myDataset) {
        mDataset = myDataset;
    }
```

```java
public class MyAdapter extends RecyclerView.Adapter<MyAdapter.ViewHolder> {
    private String[] mDataset;

    // 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
    public static class ViewHolder extends RecyclerView.ViewHolder {
        // each data item is just a string in this case
        public TextView mTextView;
        public ViewHolder(TextView v) {
            super(v);
            mTextView = v;
        }
    }

    // Provide a suitable constructor (depends on the kind of dataset)
    public MyAdapter(String[] myDataset) {
        mDataset = myDataset;
    }
```

```java
// Create new views (invoked by the layout manager)
@Override
public MyAdapter.ViewHolder onCreateViewHolder(ViewGroup parent,
                                               int viewType) {
    // create a new view
    View v = LayoutInflater.from(parent.getContext())
                           .inflate(R.layout.my_text_view, parent, false);
    // set the view's size, margins, paddings and layout parameters
    ...
    ViewHolder vh = new ViewHolder(v);
    return vh;
}

// Replace the contents of a view (invoked by the layout manager)
@Override
public void onBindViewHolder(ViewHolder holder, int position) {
    // - get element from your dataset at this position
    // - replace the contents of the view with that element
    holder.mTextView.setText(mDataset[position]);

}

// Return the size of your dataset (invoked by the layout manager)
@Override
public int getItemCount() {
    return mDataset.length;
}
```

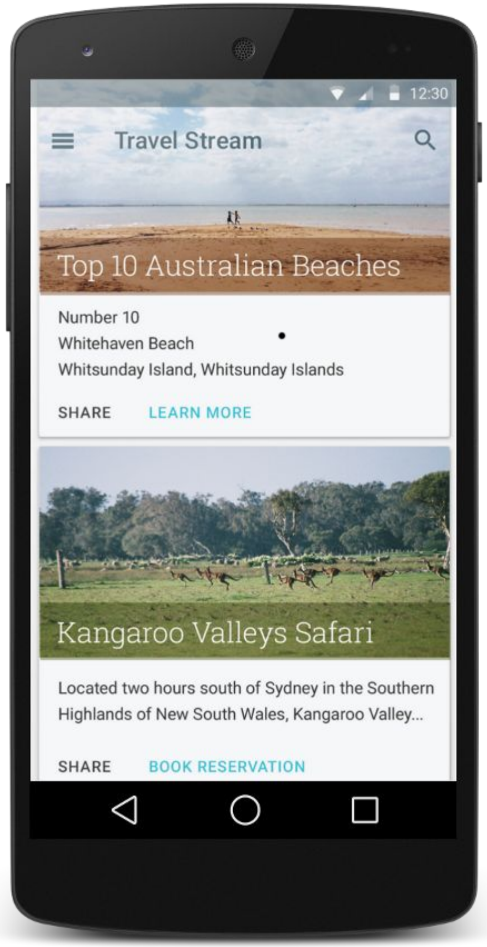
StaggeredGridLayoutManager

A LayoutManager that lays out children in a **staggered grid formation**. It supports **horizontal & vertical** layout as well as an ability to layout children in reverse.



Create Cards

CardView extends the FrameLayout class and lets you show information inside **cards that have a consistent look across the platform**. CardView widgets can have shadows and rounded corners

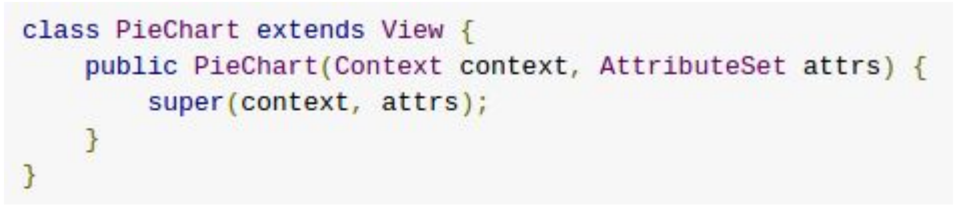

12:30
Travel Stream
Top 10 Australian Beaches
Number 10
Whitehaven Beach
Whitsunday Island, Whitsunday Islands
SHARE
LEARN MORE
Kangaroo Valleys Safari
Located two hours south of Sydney in the Southern Highlands of New South Wales, Kangaroo Valley...
SHARE
BOOK RESERVATION

Cards

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    xmlns:card_view="http://schemas.android.com/apk/res-auto"
    ... >
    <!-- A CardView that contains a TextView -->
    <android.support.v7.widget.CardView
        xmlns:card_view="http://schemas.android.com/apk/res-auto"
        android:id="@+id/card_view"
        android:layout_gravity="center"
        android:layout_width="200dp"
        android:layout_height="200dp"
        card_view:cardCornerRadius="4dp">

        <TextView
            android:id="@+id/info_text"
            android:layout_width="match_parent"
            android:layout_height="match_parent" />
    </android.support.v7.widget.CardView>
</LinearLayout>
```

Custom view

- The Android framework has a large set of View classes for interacting with the user and displaying various types of data. **But sometimes your app has unique needs that aren't covered by the built-in view**
- All of the view classes defined in the Android framework **extend View**
- you can save time by extending one of the existing view **subclasses**, such as **Button**



```
class PieChart extends View {
    public PieChart(Context context, AttributeSet attrs) {
        super(context, attrs);
    }
}
```

Define Custom Attributes

To define custom attributes, add <declare-styleable> resources to your project. It's customary to put these resources into a **res/values/attrs.xml** file. Here's an example of an attrs.xml file:

```xml
<resources>
   <declare-styleable name="PieChart">
       <attr name="showText" format="boolean" />
       <attr name="labelPosition" format="enum">
           <enum name="left" value="0"/>
           <enum name="right" value="1"/>
       </attr>
   </declare-styleable>
</resources>
```

```xml
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
   xmlns:custom="http://schemas.android.com/apk/res/com.example.customviews">
 <com.example.customviews.charting.PieChart
     custom:showText="true"
     custom:labelPosition="left" />
</LinearLayout>
```

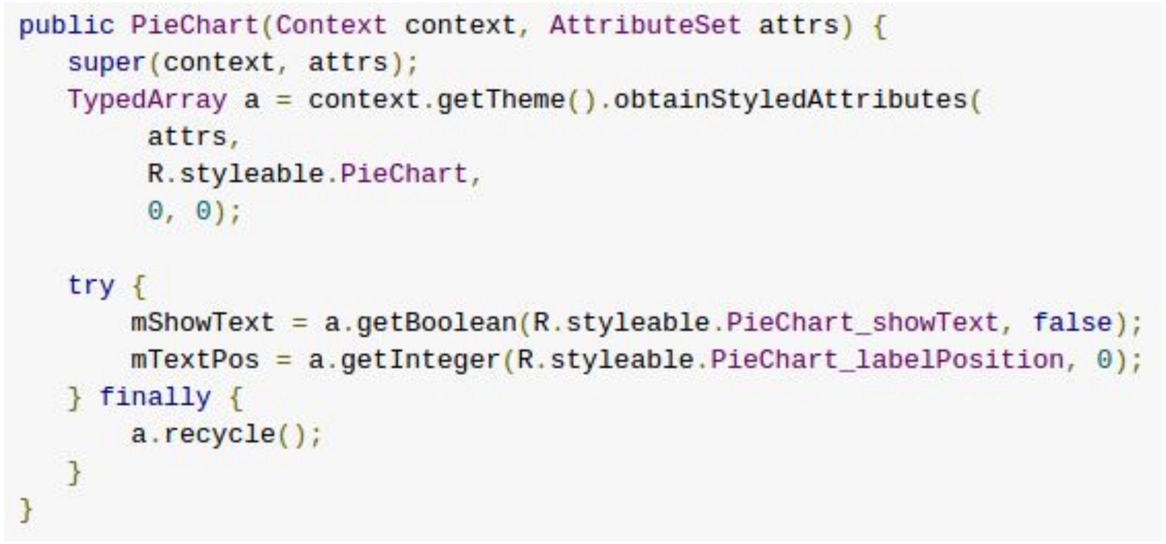
Apply Custom Attributes

When a view is created from an XML layout, **all of the attributes in the XML tag** are read **from the resource bundle and passed into the view's constructor as an AttributeSet**. Although it's possible to read values from the AttributeSet directly, doing so has some disadvantages:

- Resource references within attribute values are not resolved
- **Styles are not applied**

Instead, pass the AttributeSet to **obtainStyledAttributes()**. This method passes back a TypedArray array of values that have already been dereferenced and styled.

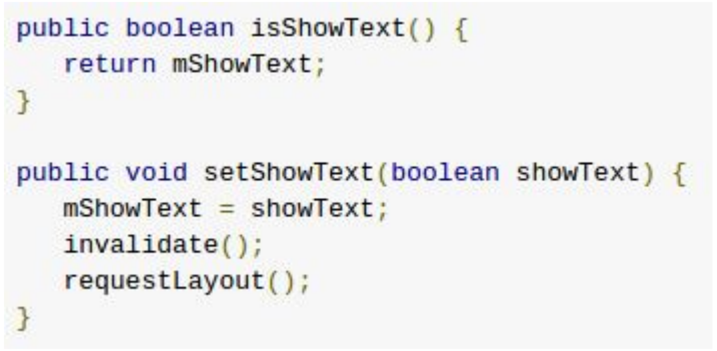
Apply Custom Attributes (2)



```java
public PieChart(Context context, AttributeSet attrs) {
   super(context, attrs);
   TypedArray a = context.getTheme().obtainStyledAttributes(
        attrs,
        R.styleable.PieChart,
        0, 0);

   try {
       mShowText = a.getBoolean(R.styleable.PieChart_showText, false);
       mTextPos = a.getInteger(R.styleable.PieChart_labelPosition, 0);
   } finally {
       a.recycle();
   }
}
```

Add Properties and Events

Attributes are a powerful way of controlling the behavior and appearance of views, but they can only be read when the view is initialized. To provide dynamic behavior, expose a property getter and setter pair for each custom attribute



```
public boolean isShowText() {
   return mShowText;
}

public void setShowText(boolean showText) {
   mShowText = showText;
   invalidate();
   requestLayout();
}
```

Custom Drawing

The most important part of a custom view is its **appearance**. Custom drawing can be easy or complex according to your application's needs

The most important step in drawing a custom view is to **override the onDraw()** method. **The parameter to onDraw() is a Canvas object** that the view can use to draw itself. The Canvas class defines methods **for drawing text, lines, bitmaps, and many other graphics primitives**. You can use these methods in onDraw() to create your custom user interface (UI)

Create Drawing Objects

The **android.graphics** framework divides drawing into two areas:

- **What to draw**, handled by **Canvas**
- **How to draw**, handled by **Paint**

Canvas defines shapes that you can draw on the screen, while Paint defines the color, style, font, and so forth of each shape you draw

Example

before you draw anything, you need to create one or more Paint objects. The **PieChart example** does this in a method called init, which is called from the constructor:

```
private void init() {
   mTextPaint = new Paint(Paint.ANTI_ALIAS_FLAG);
   mTextPaint.setColor(mTextColor);
   if (mTextHeight == 0) {
       mTextHeight = mTextPaint.getTextSize();
   } else {
       mTextPaint.setTextSize(mTextHeight);
   }

   mPiePaint = new Paint(Paint.ANTI_ALIAS_FLAG);
   mPiePaint.setStyle(Paint.Style.FILL);
   mPiePaint.setTextSize(mTextHeight);

   mShadowPaint = new Paint(0);
   mShadowPaint.setColor(0xff101010);
   mShadowPaint.setMaskFilter(new BlurMaskFilter(8, BlurMaskFilter.Blur.NORMAL));

   ...
```

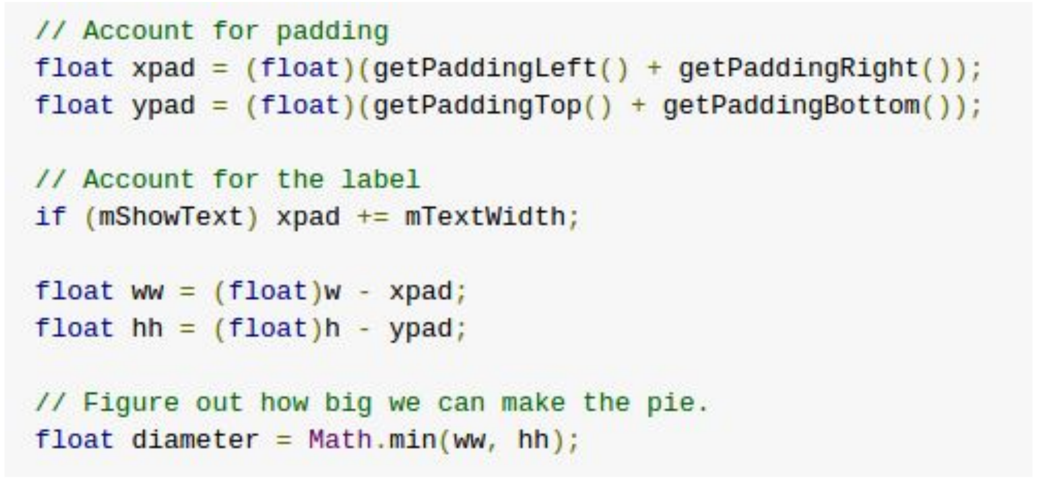
Handle Layout Events

In order to properly draw your custom view, you need to know what size it is. Complex custom views often need to perform multiple layout calculations depending on the size and shape of their area on screen. You should never make assumptions about the size of your view on the screen

onSizeChanged() is called when your view is first assigned a size, and again if the size of your view changes for any reason. Calculate positions, dimensions, and any other values related to your view's size in onSizeChanged(), **instead of recalculating them every time you draw**

Handle Layout Events

When your view is assigned a size, the layout manager assumes that the size includes all of the view's padding. You must handle the padding values when you calculate your view's size. Here's a snippet from PieChart.**onSizeChanged()** that shows how to do this



```
// Account for padding
float xpad = (float)(getPaddingLeft() + getPaddingRight());
float ypad = (float)(getPaddingTop() + getPaddingBottom());

// Account for the label
if (mShowText) xpad += mTextWidth;

float ww = (float)w - xpad;
float hh = (float)h - ypad;

// Figure out how big we can make the pie.
float diameter = Math.min(ww, hh);
```

onMeasure

If you need finer control over your view's layout parameters, implement onMeasure()

This method's parameters are View.MeasureSpec values that tell **you how big your view's parent wants your view to be**, and whether that size is a hard maximum or just a suggestion. As an optimization, these values are stored as packed integers, and you use the static methods of View.MeasureSpec to unpack the information stored in each integer

onMeasure

```
@Override
protected void onMeasure(int widthMeasureSpec, int heightMeasureSpec) {
   // Try for a width based on our minimum
   int minw = getPaddingLeft() + getPaddingRight() +
getSuggestedMinimumWidth();
   int w = resolveSizeAndState(minw, widthMeasureSpec, 1);

   // Whatever the width ends up being, ask for a height that would let the pie
   // get as big as it can
   int minh = MeasureSpec.getSize(w) - (int)mTextWidth + getPaddingBottom()
+ getPaddingTop();
   int h = resolveSizeAndState(MeasureSpec.getSize(w) - (int)mTextWidth,
heightMeasureSpec, 0);

   setMeasuredDimension(w, h);
}
```

View.MeasureSpec

A MeasureSpec encapsulates the layout requirements passed from parent to child. Each MeasureSpec represents a requirement for either the width or the height. **A MeasureSpec is comprised of a size and a mode**. There are three possible modes:

- **UNSPECIFIED**: The parent has not imposed any constraint on the child. It can be whatever size it wants.
- **EXACTLY**: The parent has determined an exact size for the child. The child is going to be given those bounds regardless of how big it wants to be
- **AT_MOST**: The child can be as large as it wants up to the specified size

Draw!

```
protected void onDraw(Canvas canvas) {
   super.onDraw(canvas);

   // Draw the shadow
   canvas.drawOval(
           mShadowBounds,
           mShadowPaint
   );

   // Draw the label text
   canvas.drawText(mData.get(mCurrentItem).mLabel, mTextX, mTextY, mTextPaint);

   // Draw the pie slices
   for (int i = 0; i < mData.size(); ++i) {
       Item it = mData.get(i);
       mPiePaint.setShader(it.mShader);
       canvas.drawArc(mBounds,
               360 - it.mEndAngle,
               it.mEndAngle - it.mStartAngle,
               true, mPiePaint);
   }

   // Draw the pointer
   canvas.drawLine(mTextX, mPointerY, mPointerX, mPointerY, mTextPaint);
   canvas.drawCircle(mPointerX, mPointerY, mPointerSize, mTextPaint);
}
```

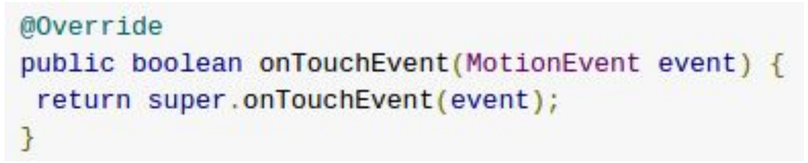
Making the View Interactive

Drawing a UI is only one part of creating a custom view. **You also need to make your view respond to user input in a way that closely resembles the real-world action** you're mimicking

Like many other UI frameworks, Android supports an input event model. **User actions are turned into events that trigger callbacks**, and you can override the callbacks to customize how your application responds to the user.

Handle Input Gestures

The most common input event in the Android system is touch, which triggers **onTouchEvent(android.view.MotionEvent)**



```
@Override
public boolean onTouchEvent(MotionEvent event) {
 return super.onTouchEvent(event);
}
```

Touch events by themselves are not particularly useful**. Modern touch UIs define interactions in terms of gestures such as tapping, pulling, pushing, flinging, and zooming.** To convert raw touch events into gestures, Android provides **GestureDetector**

GestureDetector

Construct a GestureDetector by passing in **an instance of a class that implements GestureDetector.OnGestureListener**. If you only want to process a few gestures, you can extend **GestureDetector.SimpleOnGestureListener** instead of implementing the GestureDetector.OnGestureListener interface

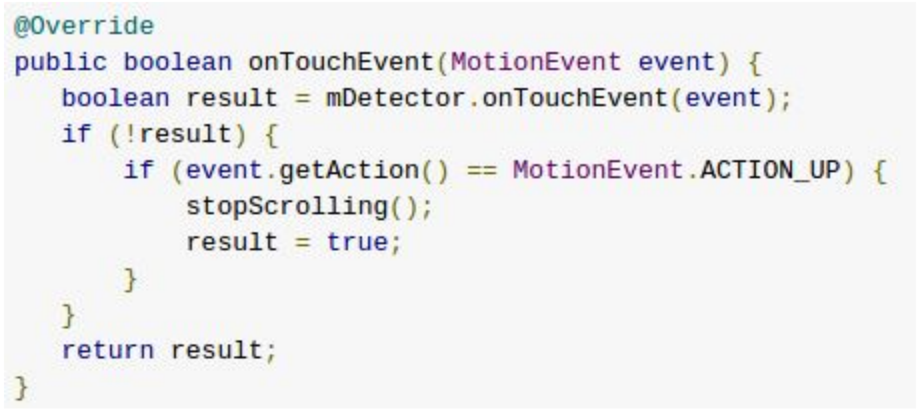
```
class mListener extends GestureDetector.SimpleOnGestureListener {
   @Override
   public boolean onDown(MotionEvent e) {
       return true;
   }
}
mDetector = new GestureDetector(PieChart.this.getContext(), new mListener());
```

GestureDetector

Whether or not you use GestureDetector.SimpleOnGestureListener, you must always implement an onDown() method that returns true. This step is necessary because all gestures begin with an onDown() message. If you return false from onDown(), as GestureDetector.SimpleOnGestureListener does, the system assumes that you want to ignore the rest of the gesture, and the other methods of GestureDetector.OnGestureListener never get called. The only time you should return false from onDown() is if you truly want to ignore an entire gesture

GestureDetector

When you pass onTouchEvent() a touch event that it doesn't recognize as part of a gesture, it returns false. You can then run your own custom gesture-detection code



```
@Override
public boolean onTouchEvent(MotionEvent event) {
   boolean result = mDetector.onTouchEvent(event);
   if (!result) {
       if (event.getAction() == MotionEvent.ACTION_UP) {
           stopScrolling();
           result = true;
       }
   }
   return result;
}
```

Fling

fling gesture, where the user quickly moves a finger across the screen and then lifts it. **This gesture makes sense if the UI responds by moving quickly in the direction of the fling, then slowing down, as if the user had pushed on a flywheel and set it spinning**

However, simulating the feel of a flywheel isn't trivial. A lot of physics and math are required to get a flywheel model working correctly. **Fortunately, Android provides helper classes to simulate this and other behaviors. The Scroller class is the basis for handling flywheel-style fling gestures**

Fling

Note: Although the velocity calculated by GestureDetector is physically accurate, many developers feel that using this value makes the fling animation too fast. It's common to divide the x and y velocity by a factor of 4 to 8.

```
@Override
public boolean onFling(MotionEvent e1, MotionEvent e2, float velocityX, float velocityY) {
   mScroller.fling(currentX, currentY, velocityX / SCALE, velocityY / SCALE, minX, minY,
maxX, maxY);
   postInvalidate();
}
```

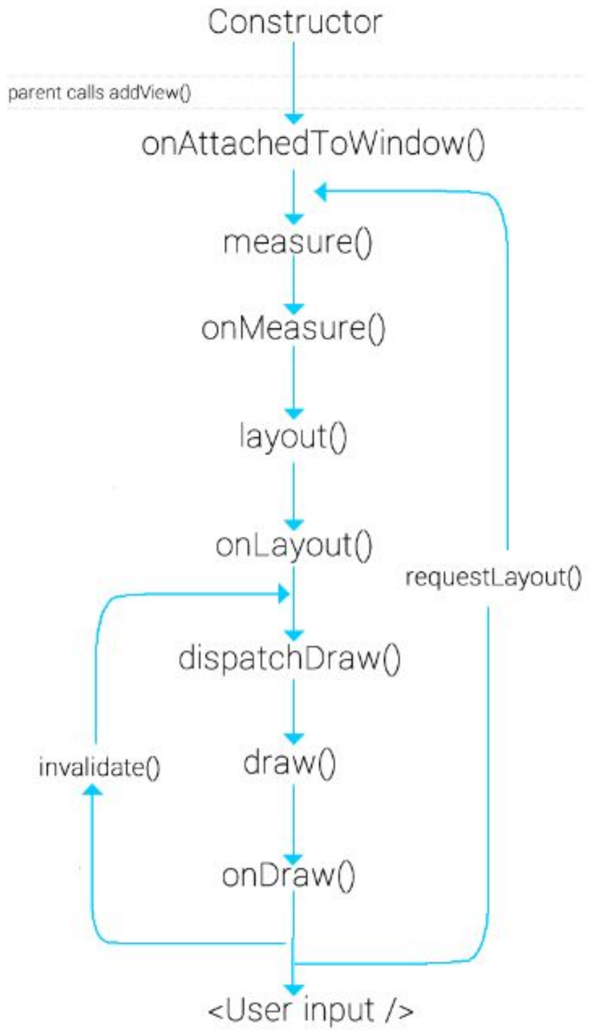
Scroller

The Scroller class computes scroll positions for you, but it does not automatically apply those positions to your view. It's your responsibility to make sure you get and apply new coordinates often enough to make the scrolling animation look smooth. There are two ways to do this:

- Call **postInvalidate()** after calling fling(), in order to force a redraw. This technique requires that you compute scroll offsets in onDraw() and call postInvalidate() every time the scroll offset changes.
- **Set up a ValueAnimator** to animate for the duration of the fling, and add a listener to process animation updates by calling addUpdateListener()

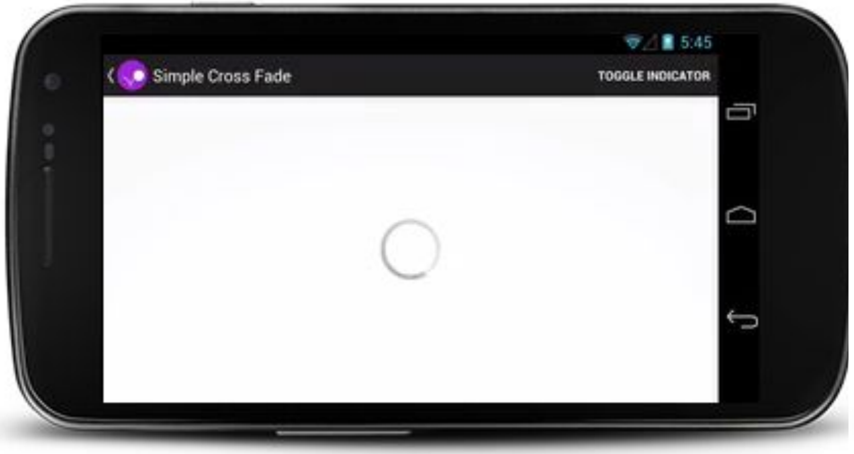
Scroller(2)

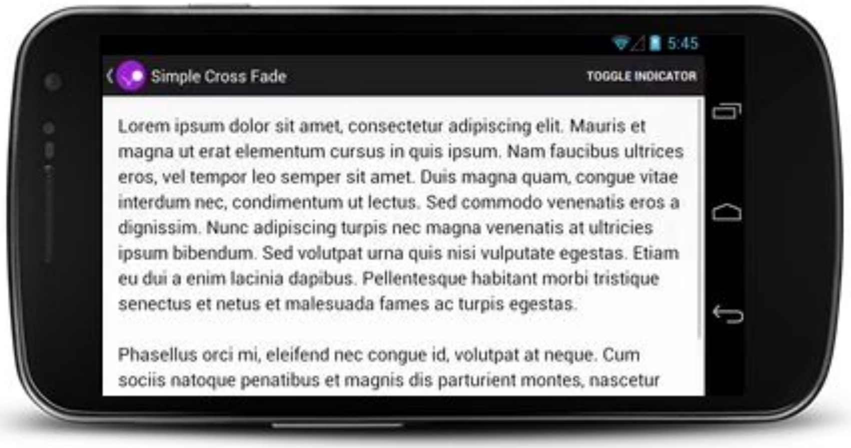
```java
mScroller = new Scroller(getContext(), null, true);
mScrollAnimator = ValueAnimator.ofFloat(0,1);
mScrollAnimator.addUpdateListener(new ValueAnimator.AnimatorUpdateListener() {
    @Override
    public void onAnimationUpdate(ValueAnimator valueAnimator) {
        if (!mScroller.isFinished()) {
            mScroller.computeScrollOffset();
            setPieRotation(mScroller.getCurrY());
        } else {
            mScrollAnimator.cancel();
            onScrollFinished();
        }
    }
});
```


Constructor
parent calls addView()
onAttachedToWindow()
measure()
onMeasure()
layout()
onLayout()
requestLayout()
dispatchDraw()
invalidate()
draw()
onDraw()
<User input />

Crossfading Two Views

Crossfade animations (also know as dissolve) gradually fade out one UI component while simultaneously fading in another. This animation is useful for situations where you want to switch content or views in your app





Lorem ipsum dolor sit amet, consectetur adipiscing elit. Mauris et magna ut erat elementum cursus in quis ipsum. Nam faucibus ultrices eros, vel tempor leo semper sit amet. Duis magna quam, congue vitae interdum nec, condimentum ut lectus. Sed commodo venenatis eros a dignissim. Nunc adipiscing turpis nec magna venenatis at ultricies ipsum bibendum. Sed volutpat urna quis nisi vulputate egestas. Etiam eu dui a enim lacinia dapibus. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas.

Phasellus orci mi, eleifend nec congue id, volutpat at neque. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur

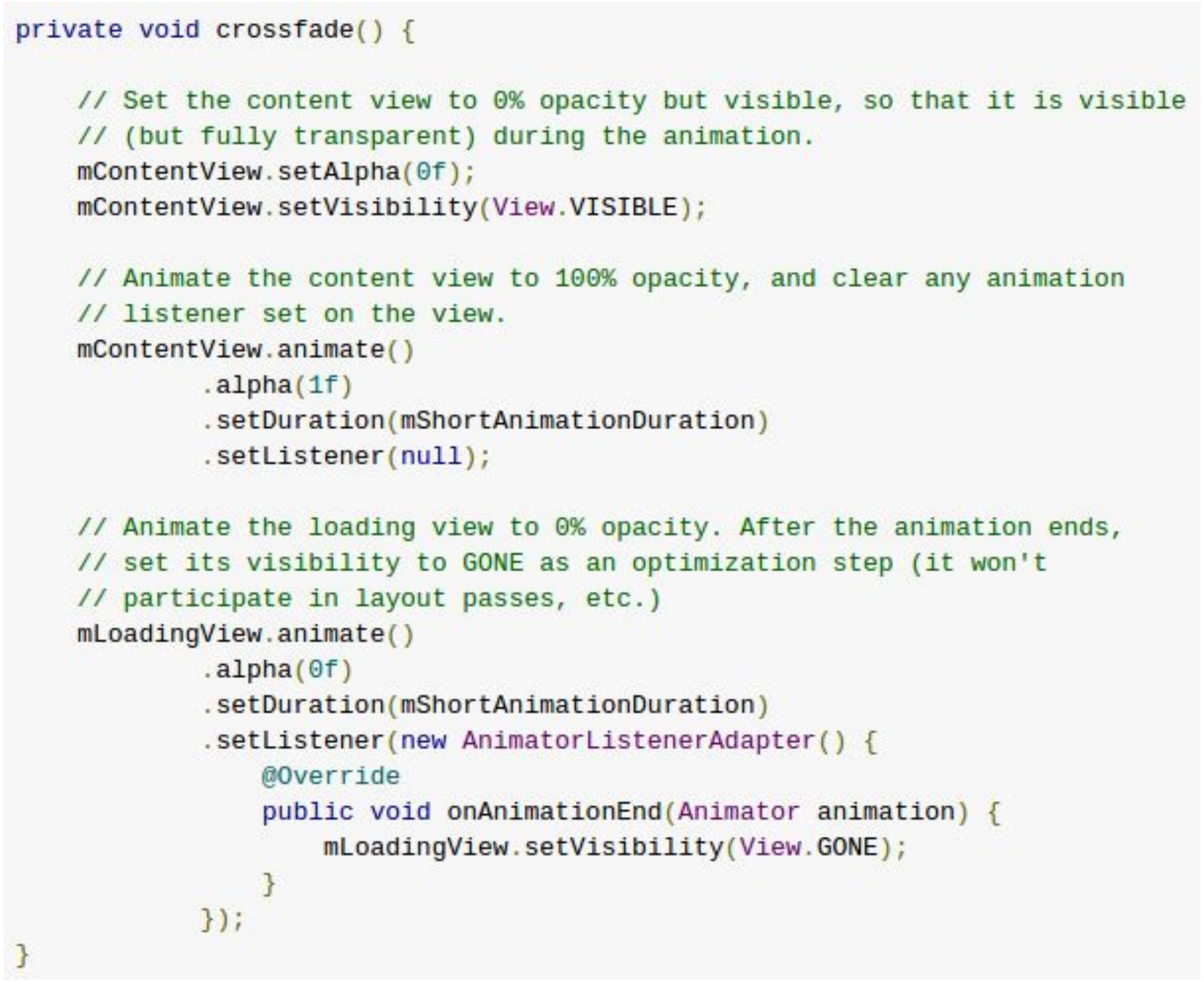
Create the Views

```xml
<FrameLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent">

    <ScrollView xmlns:android="http://schemas.android.com/apk/res/android"
        android:id="@+id/content"
        android:layout_width="match_parent"
        android:layout_height="match_parent">

        <TextView style="?android:textAppearanceMedium"
            android:lineSpacingMultiplier="1.2"
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            android:text="@string/lorem_ipsum"
            android:padding="16dp" />

    </ScrollView>

    <ProgressBar android:id="@+id/loading_spinner"
        style="?android:progressBarStyleLarge"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_gravity="center" />

</FrameLayout>
```

Set up the Animation

```java
public class CrossfadeActivity extends Activity {

    private View mContentView;
    private View mLoadingView;
    private int mShortAnimationDuration;

    ...

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_crossfade);

        mContentView = findViewById(R.id.content);
        mLoadingView = findViewById(R.id.loading_spinner);

        // Initially hide the content view.
        mContentView.setVisibility(View.GONE);

        // Retrieve and cache the system's default "short" animation time.
        mShortAnimationDuration = getResources().getInteger(
                android.R.integer.config_shortAnimTime);
    }
```

Crossfade the Views



```
private void crossfade() {

    // Set the content view to 0% opacity but visible, so that it is visible
    // (but fully transparent) during the animation.
    mContentView.setAlpha(0f);
    mContentView.setVisibility(View.VISIBLE);

    // Animate the content view to 100% opacity, and clear any animation
    // listener set on the view.
    mContentView.animate()
            .alpha(1f)
            .setDuration(mShortAnimationDuration)
            .setListener(null);

    // Animate the loading view to 0% opacity. After the animation ends,
    // set its visibility to GONE as an optimization step (it won't
    // participate in layout passes, etc.)
    mLoadingView.animate()
            .alpha(0f)
            .setDuration(mShortAnimationDuration)
            .setListener(new AnimatorListenerAdapter() {
                @Override
                public void onAnimationEnd(Animator animation) {
                    mLoadingView.setVisibility(View.GONE);
                }
            });
}
```

Using ViewPager for Screen Slides

Screen slides are transitions between one entire screen to another and are common with UIs like **setup wizards** or **slideshows**

Create the Views

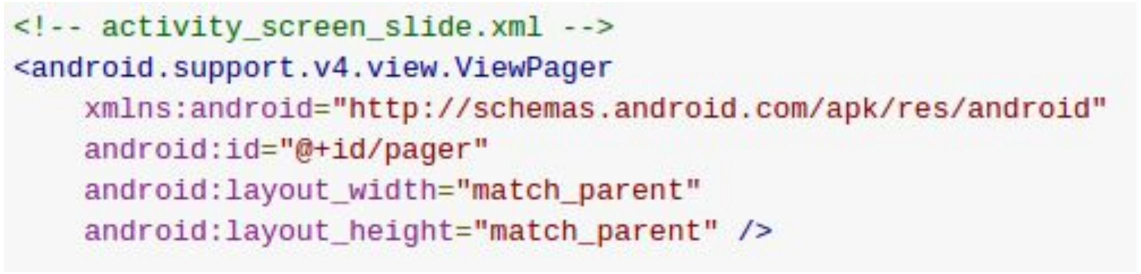
```xml
<!-- fragment_screen_slide_page.xml -->
<ScrollView xmlns:android="http://schemas.android.com/apk/res/android"
    android:id="@+id/content"
    android:layout_width="match_parent"
    android:layout_height="match_parent" >

    <TextView style="?android:textAppearanceMedium"
        android:padding="16dp"
        android:lineSpacingMultiplier="1.2"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="@string/lorem_ipsum" />
</ScrollView>
```

Create the Fragment

```
import android.support.v4.app.Fragment;
...
public class ScreenSlidePageFragment extends Fragment {

    @Override
    public View onCreateView(LayoutInflater inflater, ViewGroup container,
            Bundle savedInstanceState) {
        ViewGroup rootView = (ViewGroup) inflater.inflate(
                R.layout.fragment_screen_slide_page, container, false);

        return rootView;
    }
}
```

Add a ViewPager

ViewPagers have **built-in swipe gestures to transition through pages**, and they display screen slide animations by default, so you don't need to create any. **ViewPagers use PagerAdapters as a supply for new pages to display**, so the PagerAdapter will use the fragment class that you created earlier



```xml
<!-- activity_screen_slide.xml -->
<android.support.v4.view.ViewPager
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:id="@+id/pager"
    android:layout_width="match_parent"
    android:layout_height="match_parent" />
```

Add a ViewPager (2)

```
import android.support.v4.app.Fragment;
import android.support.v4.app.FragmentManager;
...
public class ScreenSlidePagerActivity extends FragmentActivity {
    /**
     * The number of pages (wizard steps) to show in this demo.
     */
    private static final int NUM_PAGES = 5;

    /**
     * The pager widget, which handles animation and allows swiping horizontally to
     * and next wizard steps.
     */
    private ViewPager mPager;

    /**
     * The pager adapter, which provides the pages to the view pager widget.
     */
    private PagerAdapter mPagerAdapter;
```

Add a ViewPager (3)

```
import android.support.v4.app.Fragment;
import android.support.v4.app.FragmentManager;
...
public class ScreenSlidePagerActivity extends FragmentActivity {
    /**
     * The number of pages (wizard steps) to show in this demo.
     */
    private static final int NUM_PAGES = 5;

    /**
     * The pager widget, which handles animation and allows swiping horizontally to
     * and next wizard steps.
     */
    private ViewPager mPager;

    /**
     * The pager adapter, which provides th
     */
    private PagerAdapter mPagerAdapter;
```

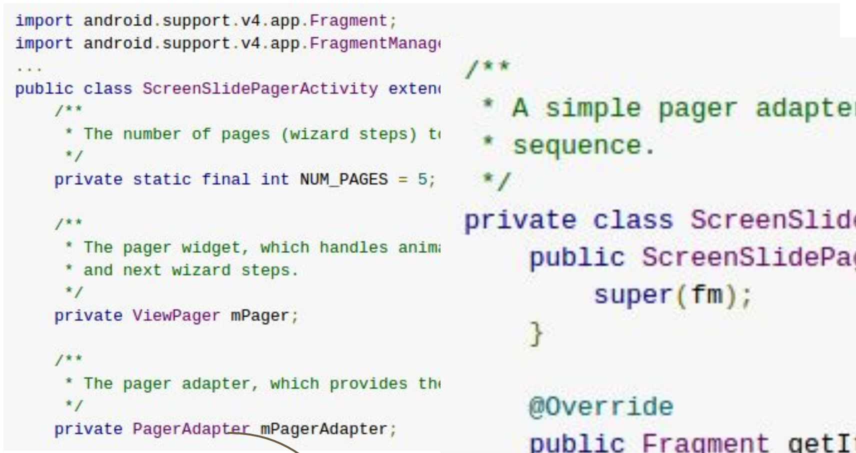
```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_screen_slide);

    // Instantiate a ViewPager and a PagerAdapter.
    mPager = (ViewPager) findViewById(R.id.pager);
    mPagerAdapter = new ScreenSlidePagerAdapter(getSupportFragmentManager());
    mPager.setAdapter(mPagerAdapter);
}
```

Add a ViewPager (4)

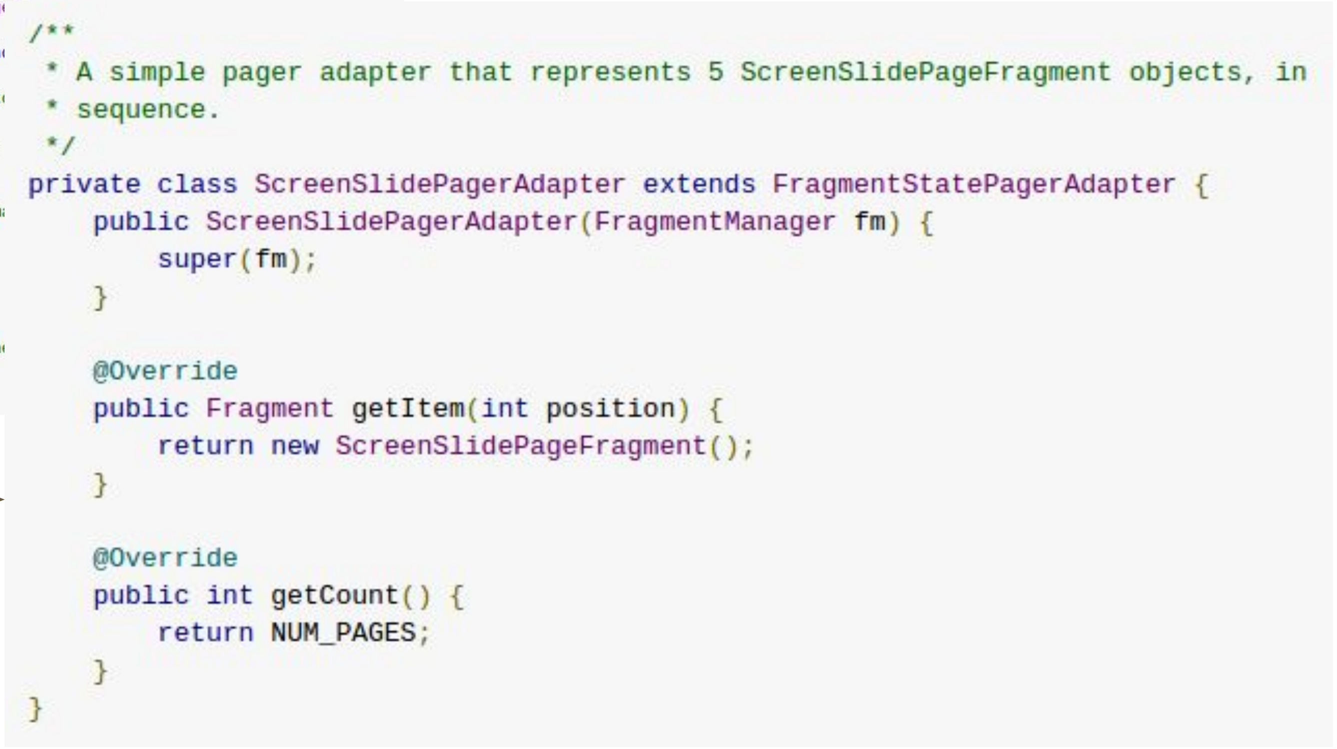
```java
import android.support.v4.app.Fragment;
import android.support.v4.app.FragmentManager;
...
public class ScreenSlidePagerActivity extends FragmentActivity {
    /**
     * The number of pages (wizard steps) to show in this demo.
     */
    private static final int NUM_PAGES = 5;

    /**
     * The pager widget, which handles animation and allows swiping horizontally to
     * and next wizard steps.
     */
    private ViewPager mPager;

    /**
     * The pager adapter, which provides th
     */
    private PagerAdapter mPagerAdapter;
```

```java
@Override
public void onBackPressed() {
    if (mPager.getCurrentItem() == 0) {
        // If the user is currently looking at the first step, allow the system to
        // Back button. This calls finish() on this activity and pops the back stac
        super.onBackPressed();
    } else {
        // Otherwise, select the previous step.
        mPager.setCurrentItem(mPager.getCurrentItem() - 1);
    }
}
```

Add a ViewPager (5)



```
import android.support.v4.app.Fragment;
import android.support.v4.app.FragmentManag
...
public class ScreenSlidePagerActivity exten
    /**
     * The number of pages (wizard steps) t
     */
    private static final int NUM_PAGES = 5;

    /**
     * The pager widget, which handles anim
     * and next wizard steps.
     */
    private ViewPager mPager;

    /**
     * The pager adapter, which provides th
     */
    private PagerAdapter mPagerAdapter;
```



```
/**
 * A simple pager adapter that represents 5 ScreenSlidePageFragment objects, in
 * sequence.
 */
private class ScreenSlidePagerAdapter extends FragmentStatePagerAdapter {
    public ScreenSlidePagerAdapter(FragmentManager fm) {
        super(fm);
    }

    @Override
    public Fragment getItem(int position) {
        return new ScreenSlidePageFragment();
    }

    @Override
    public int getCount() {
        return NUM_PAGES;
    }
}
```

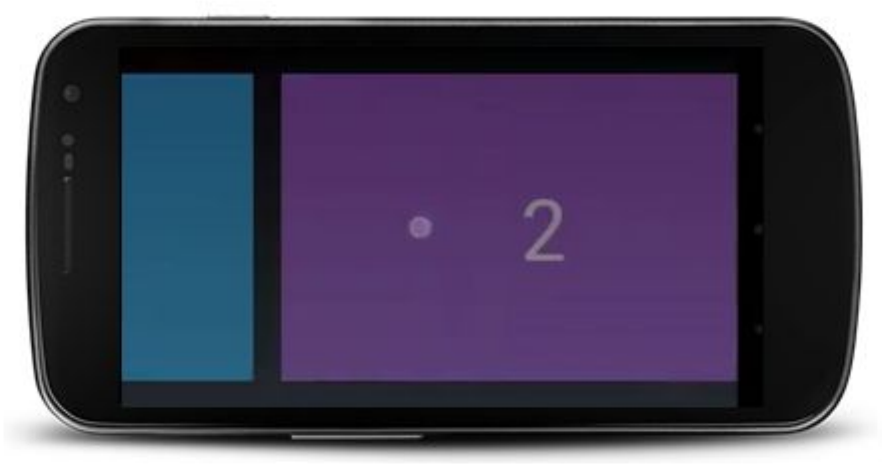
Customize the Animation with PageTransformer

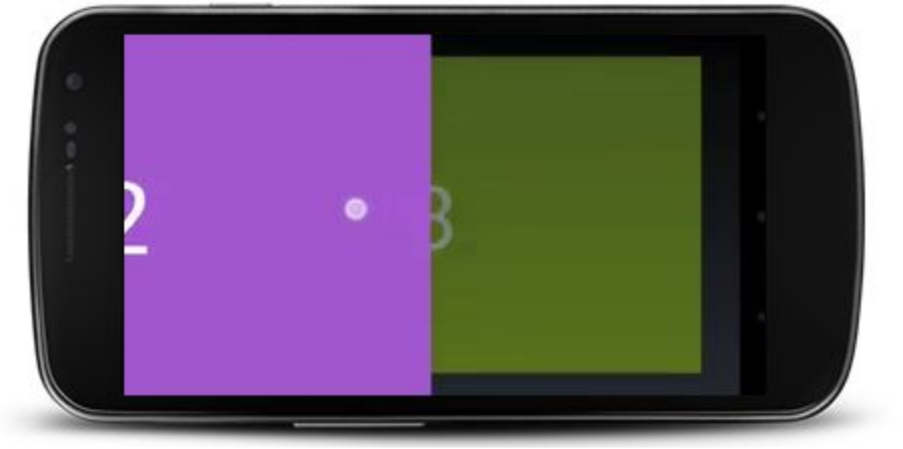
To display a different animation from the default screen slide animation, implement the ViewPager.PageTransformer interface and supply it to the view pager. The interface exposes a single method, transformPage(). At each point in the screen's transition, this method is called once for each visible page (generally there's only one visible page) and for adjacent pages just off the screen. For example, if page three is visible and the user drags towards page four, transformPage() is called for pages two, three, and four at each step of the gesture

Customize the Animation with PageTransformer

You can then create custom slide animations by determining which pages need to be transformed based on the position of the page on the screen, which is obtained from the position parameter of the transformPage() method. **The position parameter indicates where a given page is located relative to the center of the screen**. It is a dynamic property that changes as the user scrolls through the pages. **When a page fills the screen, its position value is 0**. **When a page is drawn just off the right side of the screen, its position value is 1. If the user scrolls halfway between pages one and two, page one has a position of -0.5 and page two has a position of 0.5**. Based on the position of the pages on the screen, you can create custom slide animations by setting page properties with methods such as setAlpha(), setTranslationX(), or setScaleY()

Customize the Animation with PageTransformer



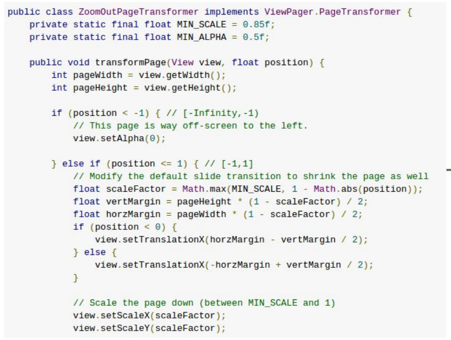


```
ViewPager mPager = (ViewPager) findViewById(R.id.pager);
...
mPager.setPageTransformer(true, new ZoomOutPageTransformer());
```

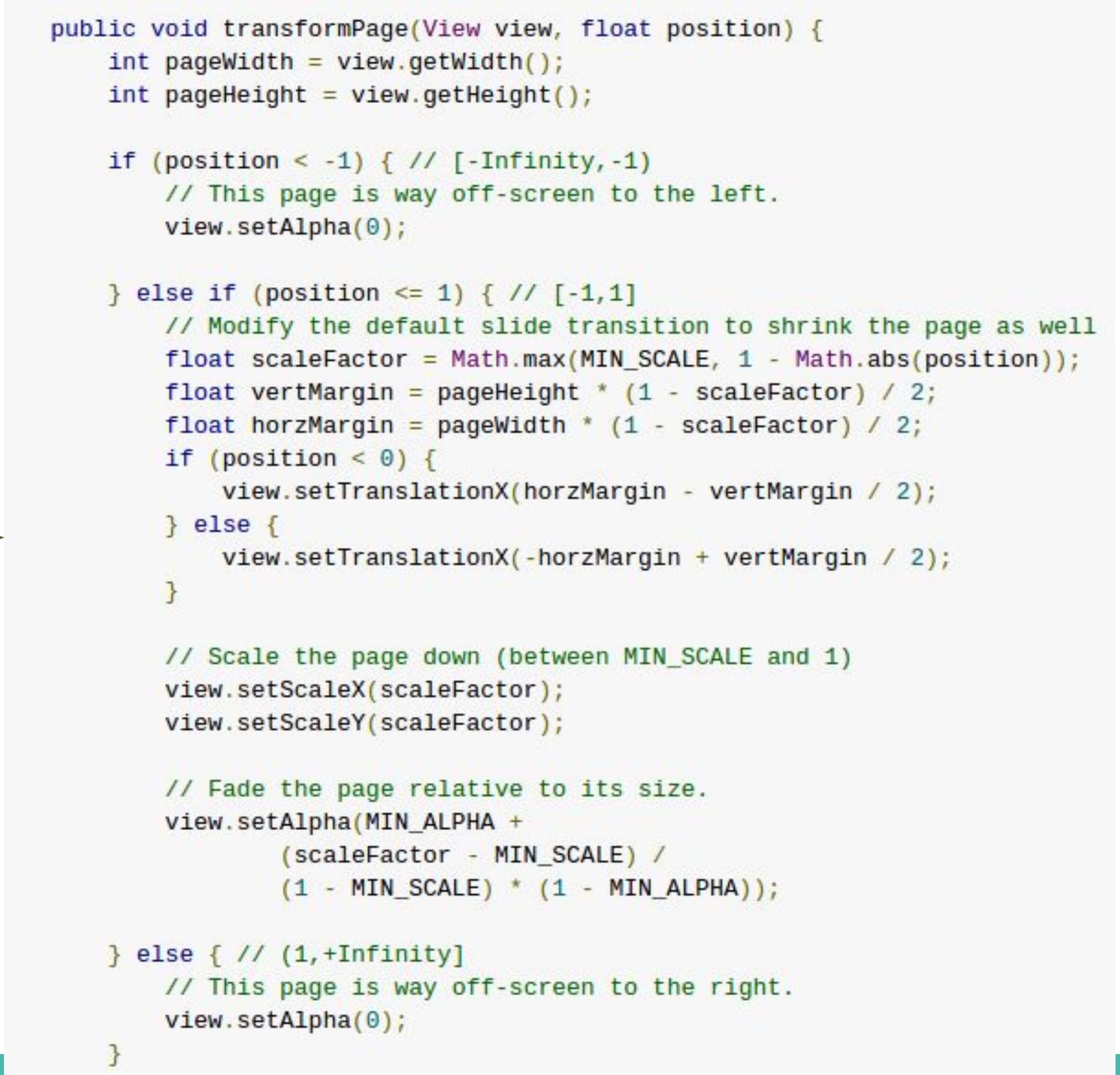
Zoom-out page transformer

```java
public class ZoomOutPageTransformer implements ViewPager.PageTransformer {
    private static final float MIN_SCALE = 0.85f;
    private static final float MIN_ALPHA = 0.5f;

    public void transformPage(View view, float position) {
        int pageWidth = view.getWidth();
        int pageHeight = view.getHeight();

        if (position < -1) { // [-Infinity,-1)
            // This page is way off-screen to the left.
            view.setAlpha(0);

        } else if (position <= 1) { // [-1,1]
            // Modify the default slide transition to shrink the page as well
            float scaleFactor = Math.max(MIN_SCALE, 1 - Math.abs(position));
            float vertMargin = pageHeight * (1 - scaleFactor) / 2;
            float horzMargin = pageWidth * (1 - scaleFactor) / 2;
            if (position < 0) {
                view.setTranslationX(horzMargin - vertMargin / 2);
            } else {
                view.setTranslationX(-horzMargin + vertMargin / 2);
            }

            // Scale the page down (between MIN_SCALE and 1)
            view.setScaleX(scaleFactor);
            view.setScaleY(scaleFactor);
```

Zoom-out page transformer

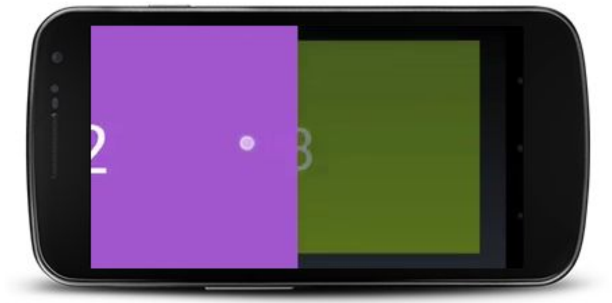


```java
public class ZoomOutPageTransformer implements ViewPager.PageTransformer {
    private static final float MIN_SCALE = 0.85f;
    private static final float MIN_ALPHA = 0.5f;

    public void transformPage(View view, float position) {
        int pageWidth = view.getWidth();
        int pageHeight = view.getHeight();

        if (position < -1) { // [-Infinity,-1)
            // This page is way off-screen to the left.
            view.setAlpha(0);

        } else if (position <= 1) { // [-1,1]
            // Modify the default slide transition to shrink the page as well
            float scaleFactor = Math.max(MIN_SCALE, 1 - Math.abs(position));
            float vertMargin = pageHeight * (1 - scaleFactor) / 2;
            float horzMargin = pageWidth * (1 - scaleFactor) / 2;
            if (position < 0) {
                view.setTranslationX(horzMargin - vertMargin / 2);
            } else {
                view.setTranslationX(-horzMargin + vertMargin / 2);
            }

            // Scale the page down (between MIN_SCALE and 1)
            view.setScaleX(scaleFactor);
            view.setScaleY(scaleFactor);
```



```java
public void transformPage(View view, float position) {
    int pageWidth = view.getWidth();
    int pageHeight = view.getHeight();

    if (position < -1) { // [-Infinity,-1)
        // This page is way off-screen to the left.
        view.setAlpha(0);

    } else if (position <= 1) { // [-1,1]
        // Modify the default slide transition to shrink the page as well
        float scaleFactor = Math.max(MIN_SCALE, 1 - Math.abs(position));
        float vertMargin = pageHeight * (1 - scaleFactor) / 2;
        float horzMargin = pageWidth * (1 - scaleFactor) / 2;
        if (position < 0) {
            view.setTranslationX(horzMargin - vertMargin / 2);
        } else {
            view.setTranslationX(-horzMargin + vertMargin / 2);
        }

        // Scale the page down (between MIN_SCALE and 1)
        view.setScaleX(scaleFactor);
        view.setScaleY(scaleFactor);

        // Fade the page relative to its size.
        view.setAlpha(MIN_ALPHA +
                (scaleFactor - MIN_SCALE) /
                (1 - MIN_SCALE) * (1 - MIN_ALPHA));

    } else { // (1,+Infinity]
        // This page is way off-screen to the right.
        view.setAlpha(0);
    }
```

Depth page transformer



```
public void transformPage(View view, float position) {
    int pageWidth = view.getWidth();

    if (position < -1) { // [-Infinity,-1)
        // This page is way off-screen to the left.
        view.setAlpha(0);

    } else if (position <= 0) { // [-1,0]
        // Use the default slide transition when moving to the left page
        view.setAlpha(1);
        view.setTranslationX(0);
        view.setScaleX(1);
        view.setScaleY(1);

    } else if (position <= 1) { // (0,1]
        // Fade the page out.
        view.setAlpha(1 - position);

        // Counteract the default slide transition
        view.setTranslationX(pageWidth * -position);

        // Scale the page down (between MIN_SCALE and 1)
        float scaleFactor = MIN_SCALE
                + (1 - MIN_SCALE) * (1 - Math.abs(position));
        view.setScaleX(scaleFactor);
        view.setScaleY(scaleFactor);

    } else { // (1,+Infinity]
        // This page is way off-screen to the right.
        view.setAlpha(0);
    }
}
```

Displaying Card Flip Animations

Create the animations for the card flips. You'll need two animators for when the front of the card animates out and to the left and in and from the left. You'll also need two animators for when the back of the card animates in and from the right and out and to the right.

card_flip_left_in.xml

```xml
<set xmlns:android="http://schemas.android.com/apk/res/android">
    <!-- Before rotating, immediately set the alpha to 0. -->
    <objectAnimator
        android:valueFrom="1.0"
        android:valueTo="0.0"
        android:propertyName="alpha"
        android:duration="0" />

    <!-- Rotate. -->
    <objectAnimator
        android:valueFrom="-180"
        android:valueTo="0"
        android:propertyName="rotationY"
        android:interpolator="@android:interpolator/accelerate_decelerate"
        android:duration="@integer/card_flip_time_full" />

    <!-- Half-way through the rotation (see startOffset), set the alpha to 1. -->
    <objectAnimator
        android:valueFrom="0.0"
        android:valueTo="1.0"
        android:propertyName="alpha"
        android:startOffset="@integer/card_flip_time_half"
        android:duration="1" />
</set>
```

card_flip_left_out.xml

```xml
<set xmlns:android="http://schemas.android.com/apk/res/android">
    <!-- Rotate. -->
    <objectAnimator
        android:valueFrom="0"
        android:valueTo="180"
        android:propertyName="rotationY"
        android:interpolator="@android:interpolator/accelerate_decelerate"
        android:duration="@integer/card_flip_time_full" />

    <!-- Half-way through the rotation (see startOffset), set the alpha to 0. -->
    <objectAnimator
        android:valueFrom="1.0"
        android:valueTo="0.0"
        android:propertyName="alpha"
        android:startOffset="@integer/card_flip_time_half"
        android:duration="1" />
</set>
```

card_flip_right_in.xml

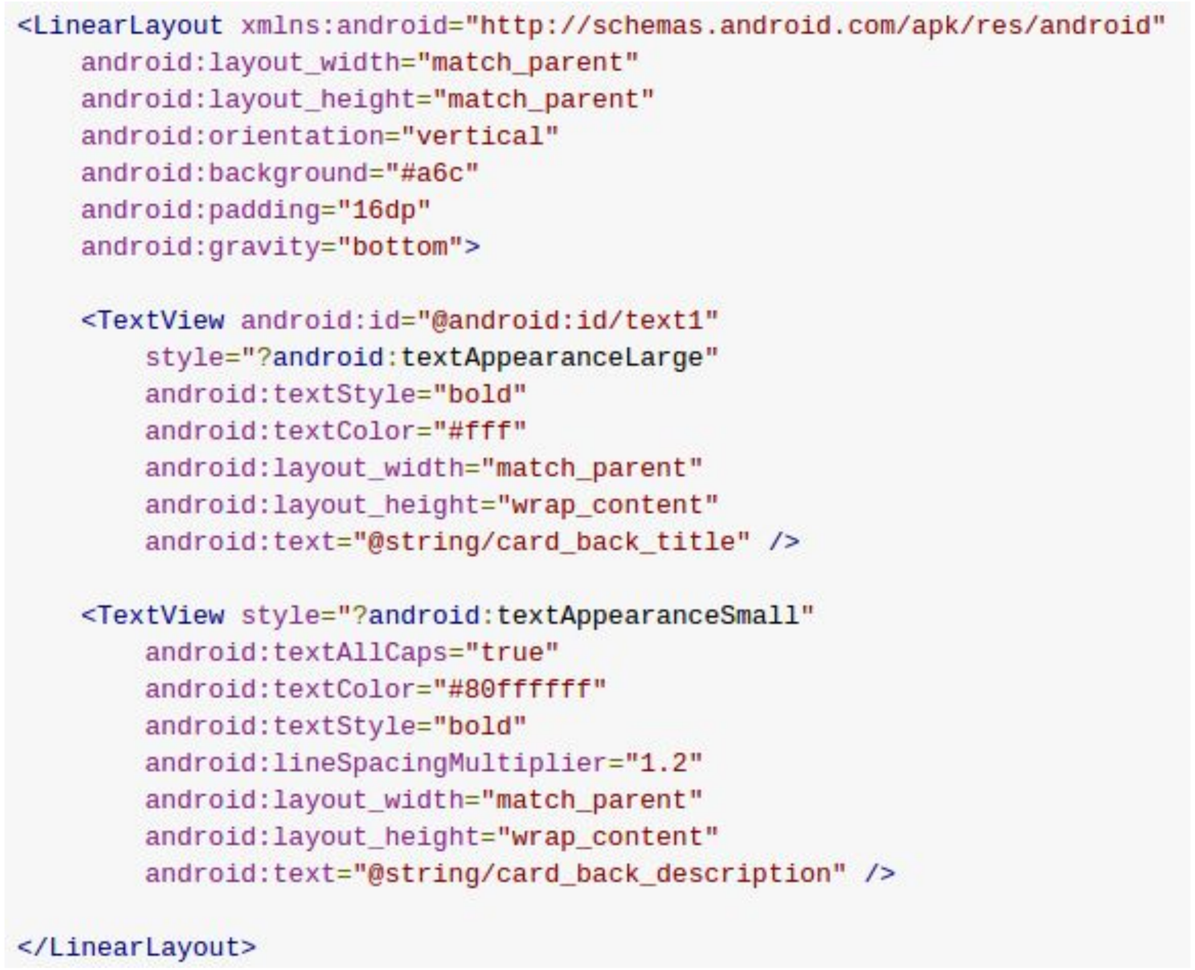
```xml
<set xmlns:android="http://schemas.android.com/apk/res/android">
    <!-- Before rotating, immediately set the alpha to 0. -->
    <objectAnimator
        android:valueFrom="1.0"
        android:valueTo="0.0"
        android:propertyName="alpha"
        android:duration="0" />

    <!-- Rotate. -->
    <objectAnimator
        android:valueFrom="180"
        android:valueTo="0"
        android:propertyName="rotationY"
        android:interpolator="@android:interpolator/accelerate_decelerate"
        android:duration="@integer/card_flip_time_full" />

    <!-- Half-way through the rotation (see startOffset), set the alpha to 1. -->
    <objectAnimator
        android:valueFrom="0.0"
        android:valueTo="1.0"
        android:propertyName="alpha"
        android:startOffset="@integer/card_flip_time_half"
        android:duration="1" />
</set>
```

card_flip_right_out.xml

```xml
<set xmlns:android="http://schemas.android.com/apk/res/android">
    <!-- Rotate. -->
    <objectAnimator
        android:valueFrom="0"
        android:valueTo="-180"
        android:propertyName="rotationY"
        android:interpolator="@android:interpolator/accelerate_decelerate"
        android:duration="@integer/card_flip_time_full" />

    <!-- Half-way through the rotation (see startOffset), set the alpha to 0. -->
    <objectAnimator
        android:valueFrom="1.0"
        android:valueTo="0.0"
        android:propertyName="alpha"
        android:startOffset="@integer/card_flip_time_half"
        android:duration="1" />
</set>
```

Create the Views

Each side of the "card" is a separate layout that can contain any content you want, such as two screens of text, two images, or any combination of views to flip between. You'll then use the two layouts in the fragments that you'll later animate



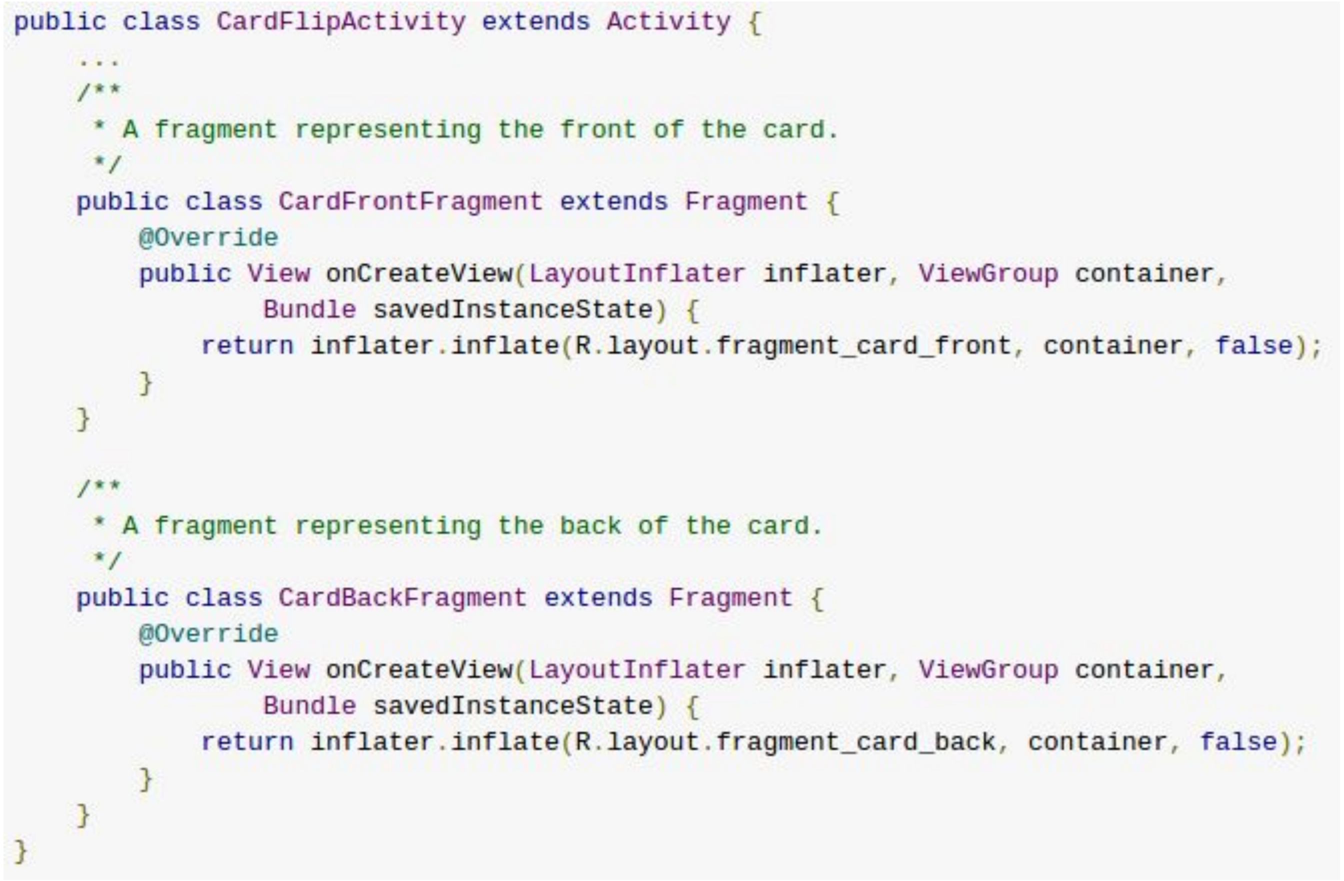
```xml
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical"
    android:background="#a6c"
    android:padding="16dp"
    android:gravity="bottom">

    <TextView android:id="@android:id/text1"
        style="?android:textAppearanceLarge"
        android:textStyle="bold"
        android:textColor="#fff"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="@string/card_back_title" />

    <TextView style="?android:textAppearanceSmall"
        android:textAllCaps="true"
        android:textColor="#80ffffff"
        android:textStyle="bold"
        android:lineSpacingMultiplier="1.2"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="@string/card_back_description" />

</LinearLayout>
```

Create the Views

the other side of the card that displays an ImageView

```
<ImageView xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:src="@drawable/image1"
    android:scaleType="centerCrop"
    android:contentDescription="@string/description_image_1" />
```

Create the Fragment



```java
public class CardFlipActivity extends Activity {
    ...
    /**
     * A fragment representing the front of the card.
     */
    public class CardFrontFragment extends Fragment {
        @Override
        public View onCreateView(LayoutInflater inflater, ViewGroup container,
                Bundle savedInstanceState) {
            return inflater.inflate(R.layout.fragment_card_front, container, false);
        }
    }

    /**
     * A fragment representing the back of the card.
     */
    public class CardBackFragment extends Fragment {
        @Override
        public View onCreateView(LayoutInflater inflater, ViewGroup container,
                Bundle savedInstanceState) {
            return inflater.inflate(R.layout.fragment_card_back, container, false);
        }
    }
}
```

Animate the Card Flip

```xml
<FrameLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:id="@+id/container"
    android:layout_width="match_parent"
    android:layout_height="match_parent" />
```

```java
public class CardFlipActivity extends Activity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_activity_card_flip);

        if (savedInstanceState == null) {
            getFragmentManager()
                    .beginTransaction()
                    .add(R.id.container, new CardFrontFragment())
                    .commit();
        }
    }
    ...
}
```

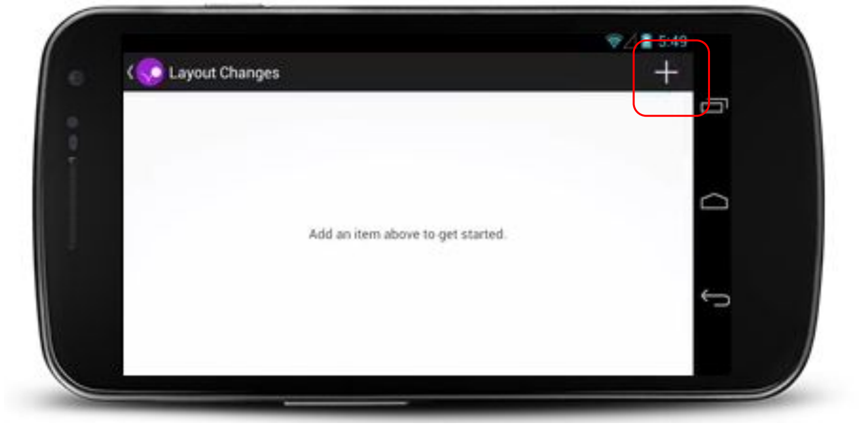
```
private void flipCard() {
    if (mShowingBack) {
        getFragmentManager().popBackStack();
        return;
    }
    // Flip to the back.
    mShowingBack = true;
    // Create and commit a new fragment transaction that adds the fragment for the back of the card, uses
custom animations, and is part of the fragment manager's back stack.

    getFragmentManager().beginTransaction()

            // Replace the default fragment animations with animator resources representing rotations when
switching to the back of the card, as well as animator resources representing rotations when flipping back to the
front (e.g. when the system Back button is pressed).
            .setCustomAnimations( R.animator.card_flip_right_in, R.animator.card_flip_right_out,
R.animator.card_flip_left_in, R.animator.card_flip_left_out)

            // Replace any fragments currently in the container view with a fragment representing the next page
(indicated by the  just-incremented currentPage variable).
            .replace(R.id.container, new CardBackFragment())
            // Add this transaction to the back stack, allowing users to press
            // Back to get to the front of the card.
            .addToBackStack(null)
            // Commit the transaction.
            .commit();
}
```

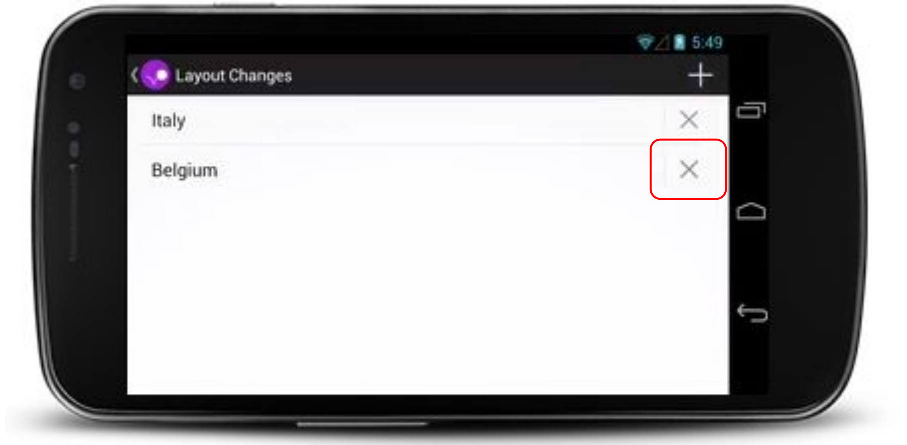
Animating Layout Changes

A layout animation is **a pre-loaded animation that the system runs each time you make a change to the layout configuration**. All you need to do is set an attribute in the layout to tell the Android system to animate these layout changes, and system-default animations are carried out for you.

Tip: If you want to supply custom layout animations, create a **LayoutTransition** object and supply it to the layout with the setLayoutTransition() method

Layout Changes





Animating Layout Changes

- Create the Layout: In your activity's layout XML file, set the android:animateLayoutChanges attribute to true for the layout that you want to enable animations for

- Add, Update, or Remove Items from the Layout

```
<LinearLayout android:id="@+id/container"
    android:animateLayoutChanges="true"
    ...
/>
```

```
private ViewGroup mContainerView;
...
private void addItem() {
    View newView;
    ...
    mContainerView.addView(newView, 0);
}
```