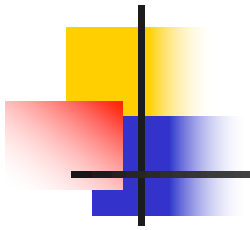
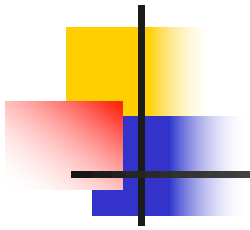
Introduction to Layouts

Includes many slides from Chapter 8



Overview

- Create user interfaces in Android by defining resource files or through programming
- Organize your user interface using layout classes of View container controls
- Use built-in layout classes such as FrameLayout, LinearLayout, TableLayout, RelativeLayout, and ConstraintLayout
- Use container control classes for lists, grids, scrolling, and switching
- An in-depth look at ConstraintLayout



Creating User Interfaces in Android

- Application user interfaces can be simple or complex, involving many different screens or only a few.
- Layouts and user interface controls can be defined as application resources or created programmatically at runtime.

Creating User Interfaces in Android

- The term layout is used for two different but related purposes in Android user interface design:
  - For resources, the res/layout directory contains XML resource definitions often called layout resource files.
    - These XML files provide a template for how to draw controls on the screen.
    - Layout resource files may contain any number of controls.
  - The term layout is also used to refer to a set of ViewGroup Java classes, such as LinearLayout, FrameLayout, TableLayout, GridLayout, and ConstraintLayout.
    - These controls are used to organize other View controls.

Creating Layouts Using XML Resources

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
    android:layout_width=”match_parent"
    android:layout_height=" match_parent"
    xmlns:tools="http://schemas.android.com/tools"
    android:orientation="vertical"
    tools:layout_editor_absoluteX="8dp"
    tools:layout_editor_absoluteY="8dp"
    xmlns:android="http://schemas.android.com/apk/res/android">

    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/hello_world" />

</LinearLayout>
```

Layout element

This layout XML file is placed in res/layout

Child element

Creating Layouts Using XML Resources

- A typical call to create a layout view, normally in the `onCreate()` callback:

```
setContentView( R.layout.activity_main );
```

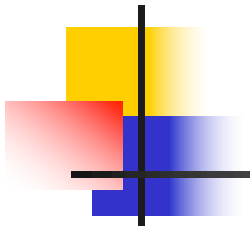
- In Android terminology, a layout is **inflated**, i.e., its rendering is created as a collection of UI Java objects.
- All XML-defined views (including layouts) have corresponding Java classes.
- A layout can be created programmatically.

Creating Layouts Programmatically

```
public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    if (getSupportActionBar() != null) {
     getSupportActionBar().setDisplayHomeAsUpEnabled(true);
    }
    TextView text1 = new TextView(this);
    text1.setText(”Some string 1”);
    TextView text2 = new TextView(this);
    text2.setText(”Some string 2”);
    text2.setTextSize(TypedValue.COMPLEX_UNIT_SP, 60);
    LinearLayout ll = new LinearLayout( this );
    ll.setOrientation(LinearLayout.VERTICAL);
    ll.addView(text1); // as the first child view
    ll.addView(text2); // as the second child view
    setContentView(ll); // set the content to this view
}
```

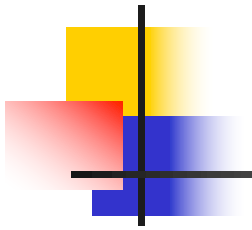
Creating Layouts in XML

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="match_parent"
    android:layout_height="match_parent">
    <TextView android:id="@+id/TextView1"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="@string/string1" />
    <TextView android:id="@+id/TextView2"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:textSize="60sp"
        android:text="@string/string2" />
</LinearLayout>
```



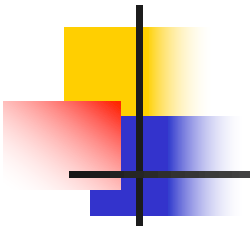
Organizing Your User Interface

- All user interface controls, such as Button, Spinner, and EditText, derive from the View class.
- A ViewGroup is a special kind of View.
- ViewGroup subclasses are broken down into two categories:
  - Layout classes
  - View container controls



Using ViewGroup Subclasses for Layout Design

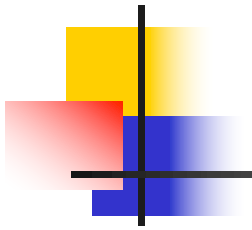
- Many of the most important subclasses of ViewGroup used for screen design end with "Layout."
- The most common layout classes are `LinearLayout`, `RelativeLayout`, `TableLayout`, `FrameLayout`, and `GridLayout`.
- `ConstraintLayout`, also a very popular layout, has been added later. Now, it is in the `androidx.constraintlayout.widget` package; it is also a subclass of `ViewGroup`.
- You can use each of these layouts to position other View controls on the screen in different ways.



Using ViewGroup Subclasses for Layout Design

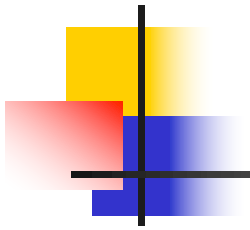
- Often, we've been using the LinearLayout to arrange various controls (e.g., TextView and EditTexts) on the screen in a single vertical column.
- Users do not generally interact with the layouts themselves.
- Instead, they interact with the View controls included in the layouts.

Using ViewGroup Subclasses as View Containers



- The second category of `ViewGroup` subclasses are the indirect "subclasses."
- These special View controls act as View containers like Layout objects do, but they also provide some kind of active functionality that enables users to interact with them like other controls.
- Unfortunately, these classes are not known by any *handy* names.
- Instead, they are named for the kind of functionality they provide.

Using ViewGroup Subclasses as View Containers

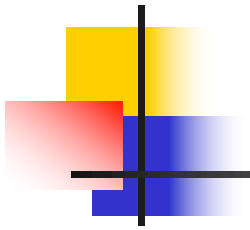


- Some of the classes that fall into this category include:
  - ListView
  - RecyclerView
  - GridView
  - ImageSwitcher
  - ScrollView
- It can be helpful to consider these objects as different kinds of View browsers, or container classes.
- A `ListView` displays each View control as a list item, and the user can browse between the individual controls using vertical scrolling capability.

Using Built-in Layout Classes

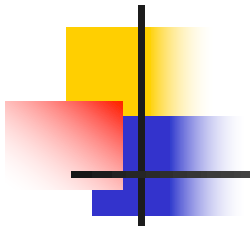
- The types of layouts built into the Android SDK framework include:
  - LinearLayout
  - RelativeLayout
  - FrameLayout
  - TableLayout
  - GridLayout
  - ConstraintLayout (newer; improves on RelativeLayout)
- These layouts are derived from:
  - android.view.ViewGroup

Using Built-in Layout Classes



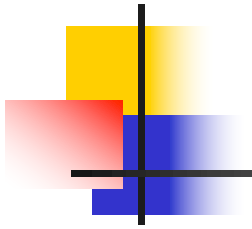
- There are several layout attributes that all ViewGroup objects share.
- These include size attributes and margin attributes.
  - Find these basic layout attributes in the `ViewGroup.LayoutParams` class.
- The margin attributes enable each child View within a layout to have extra space on each side.
  - Find these attributes in the `ViewGroup.MarginLayoutParams` classes.
- There are also a number of ViewGroup attributes for handling child View drawing bounds and animation settings.

Using Built-in Layout Classes



- A general definition of an XML layout attribute:

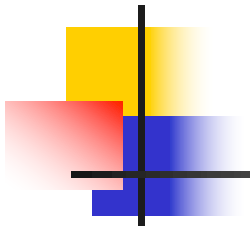
android:*layout_attribute_name*=*"value"*



Using Built-in Layout Classes

Attribute Name	Applies to	Value
`layout_height`	Parent or child `View`	Dimension value or `match_parent` or `wrap_content`
`layout_width`	Parent or child `View`	Dimension value or `match_parent` or `wrap_content`
`layout_margin`	Parent or child `View`	Dimension value. This is for all sides. You may use more specific margin attributes to control individual margin sides, if necessary.

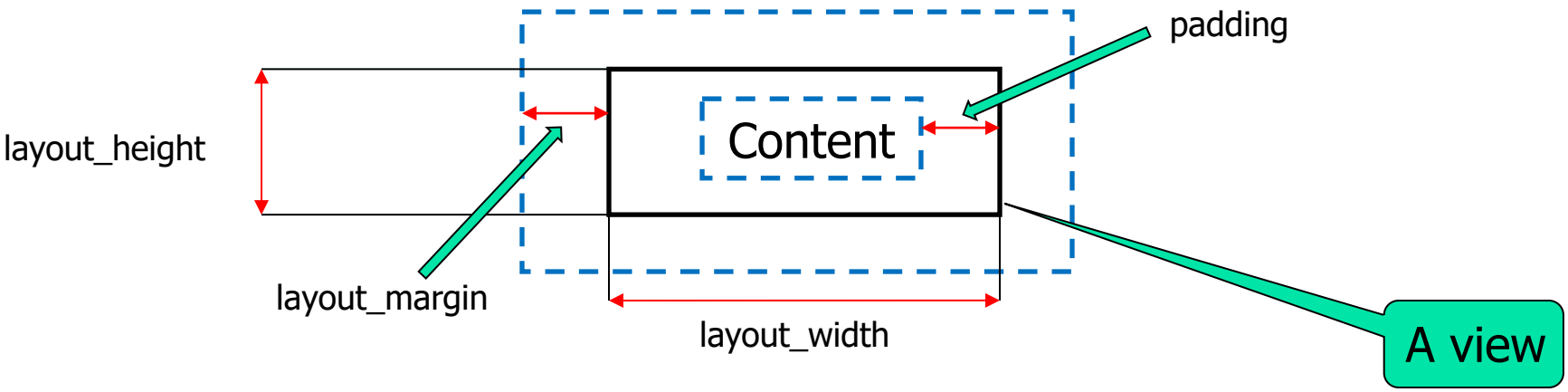
Using Built-in Layout Classes



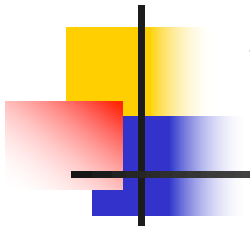
- `match_parent` - the view will be as big as its parent (minus padding)
- `wrap_content` - the view will be just big enough to enclose its content (plus padding)

Using Built-in Layout Classes

- Sizing of a view




```
android:layout_width="wrap_content"
android:layout_height="wrap_content”
android:layout_margin="20dp"
android:padding="20dp"
```



View Margins and Padding

- Margins (in ViewGroup.MarginLayoutParams):
  - android:layout_margin -- margin on all sides
  - android:layout_marginTop -- margin on top

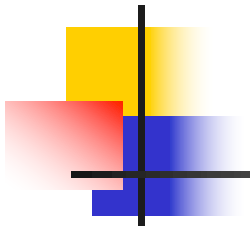
    and similarly Bottom, Left (or Start), Right (End)
  - android:layout_marginHorizontal -- left and right
  - android:layout_marginVertical -- top and bottom

View Margins and Padding

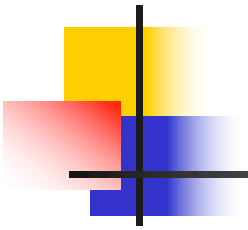
- Padding (defined in View):
  - android:padding -- padding on all sides
  - android:paddingTop -- padding on top

    and similarly Bottom, Left (or Start), Right (End)
  - android:paddingHorizontal -- left and right
  - android:paddingVertical -- top and bottom

Using Built-in Layout Classes

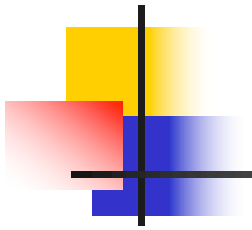


- A layout element can cover any rectangular space on the screen.
  - It doesn't need to fill the entire screen.
- Layouts can be nested within one another.
  - This provides great flexibility when developers need to organize screen elements
  - However, nesting is considered sub-optimal!



Using Built-in Layout Classes

- It is common to start with a (until recently) RelativeLayout, FrameLayout, or LinearLayout as the parent layout for the entire screen and then organize individual screen elements inside the parent layout using whichever layout type is most appropriate.
- Now, the Android team advises developers to use the newer ConstraintLayout, instead.
- We will talk about ConstraintLayout later.

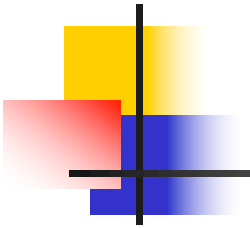


Using LinearLayout

- A LinearLayout view organizes its child View controls in a single row, or a single column, depending on whether its orientation attribute is set to horizontal or vertical.
- This is a very handy layout method for creating forms.
- You can find the layout attributes available for LinearLayout child View controls in `android.widget.LinearLayout.LayoutParams`.

Using LinearLayout

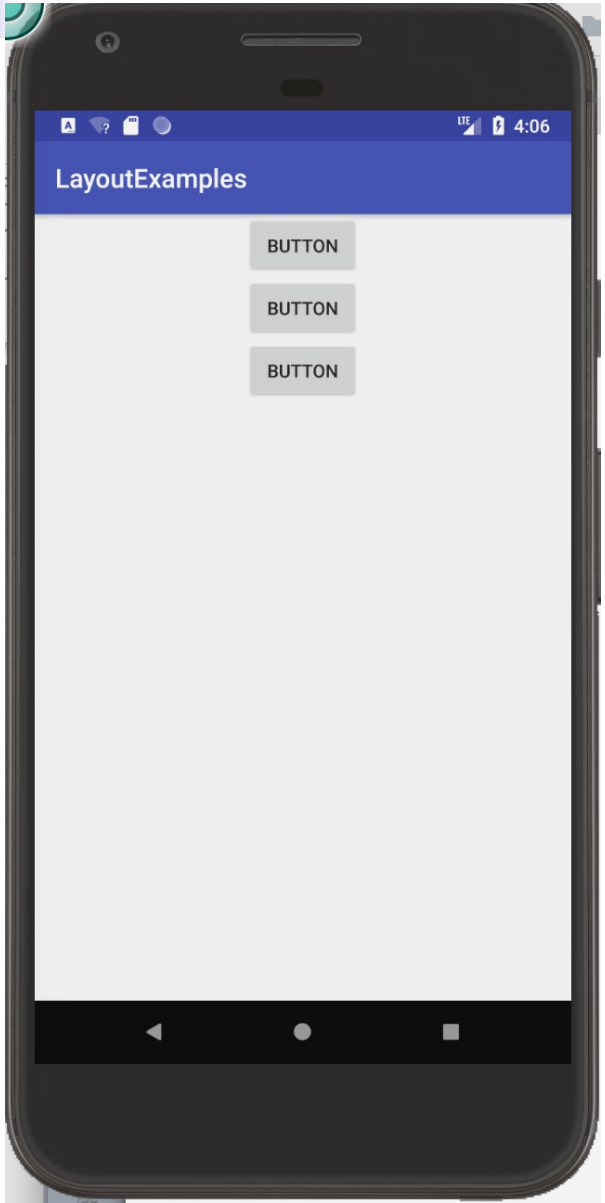
Attribute Name	Applies to	Value
`orientation`	Parent `View`	Either horizontal or vertical
`gravity`	Parent `View`	One or more constants separated by "|" such as `top`, `bottom`, `left`, `right`, and many others
`weightSum`	Parent `View`	A number that defines the sum of all child control weights; default is `1`
`layout_gravity`	Child `View`	One or more constants separated by "|" such as `top`, `bottom`, `left`, `right`, and many others
`layout_weight`	Child `View`	The sum of values across all child views in a parent `View` must equal the `weightSum` attribute of the parent `LinearLayout` control, e.g., `.3` or `.7`



Using LinearLayout

- All views must have specified *height* in a vertical layout (or *width* in a horizontal layout).
- Relative sizing can be specified with the use of the `layout_height` for vertical (`layout_width` for horizontal) and `layout_weight`.
- Relative positioning can be specified with the use of `gravity` or `layout_gravity` (with values of left, right, top, bottom, center, etc.)

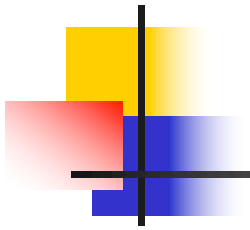
Using LinearLayout




```
<LinearLayout  …
  android:orientation="vertical" >
  <Button
      android:id="@+id/button"
      android:layout_width="wrap_content”
      android:layout_height="wrap_content"
      android:layout_gravity="center"
      android:text="Button" />
   <Button
      android:id="@+id/button2"
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:layout_gravity="center"
      android:text="Button" />
   <Button
      android:id="@+id/button3"
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:layout_gravity="center"
      android:text="Button" />
```

Using LinearLayout


```
<LinearLayout  …
  android:orientation="vertical" >
  <Button
      android:id="@+id/button"
      android:layout_width="wrap_content”
      android:layout_height="wrap_content"
      android:layout_gravity="center"
      android:text="Button" />
  <Button
      android:id="@+id/button2"
      android:layout_width="match_parent"
      android:layout_height="wrap_content"
      android:layout_gravity="center"
      android:text="Button" />
  <Button
      android:id="@+id/button3"
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:layout_gravity="center"
      android:text="Button" />
```

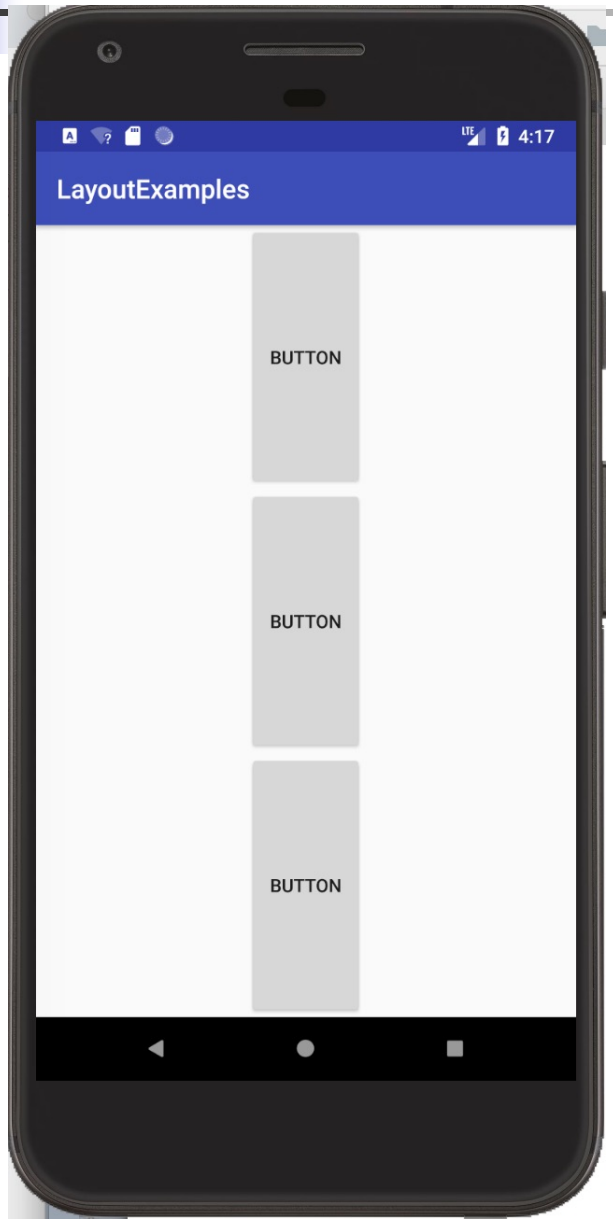


Using LinearLayout

For an equal distribution of views within a layout, i.e., in which each child uses the same amount of space on the screen:

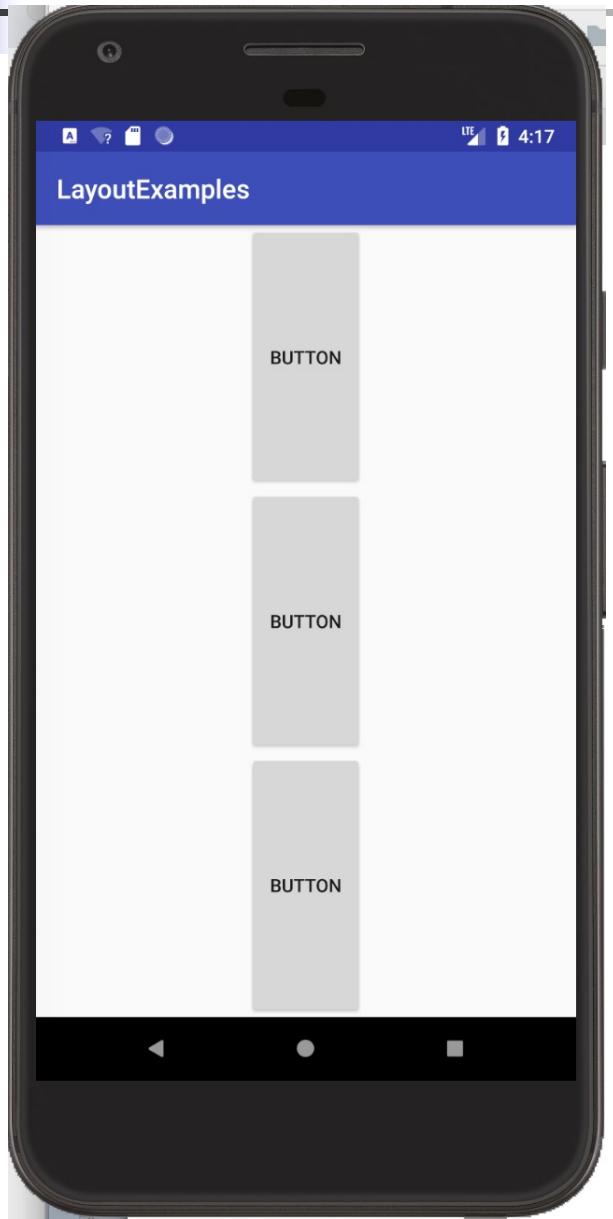
1. set the `layout_height` of each view to "`0dp`" (for a vertical layout) or the `layout_width` of each view to "`0dp`" (horizontal layout). With some views, you may use `wrap_content`.
2. set the `layout_weight` of each view to "`1`".

Using LinearLayout

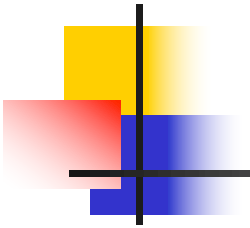



```
<Button
    android:id="@+id/button"
    android:layout_width="wrap_content”
    android:layout_height="wrap_content"
    android:layout_gravity="center”
    android:layout_weight="1"
    android:text="Button" />
<Button
    android:id="@+id/button2"
    android:layout_width=”wrap_content"
    android:layout_height="wrap_content"
    android:layout_gravity="center”
    android:layout_weight="1"
    android:text="Button" />
<Button
    android:id="@+id/button3"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_gravity="center”
    android:layout_weight="1”
    android:text="Button" />
```

Using LinearLayout




```
<Button
   android:id="@+id/button"
   android:layout_width="wrap_content”
   android:layout_height="0dp"
   android:layout_gravity="center”
   android:layout_weight="1"
   android:text="Button" />
<Button
   android:id="@+id/button2"
   android:layout_width="wrap_content"
   android:layout_height=”0dp"
   android:layout_gravity="center”
   android:layout_weight="1"
   android:text="Button" />
<Button
   android:id="@+id/button3"
   android:layout_width="wrap_content"
   android:layout_height=”0dp"
   android:layout_gravity="center”
   android:layout_weight="1”
   android:text="Button" />
```

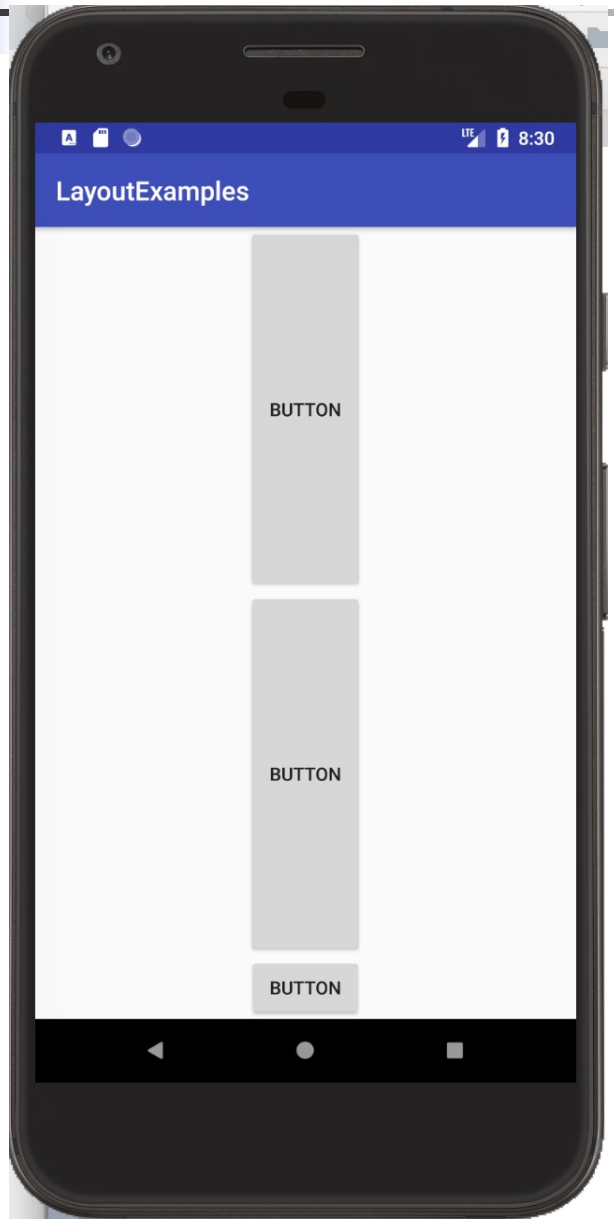


Using LinearLayout

For an unequal distribution of views within a layout, i.e., in which some children are larger than others:

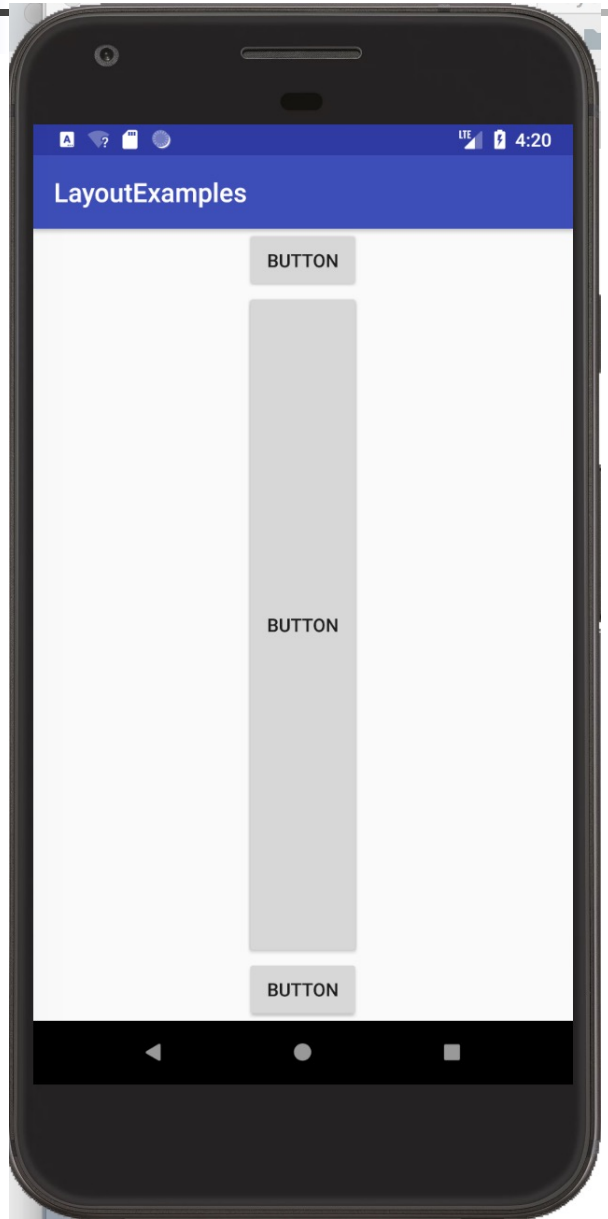
1. Assuming a few views declare a weight of 1, while one other view has no weight, the view without the weight doesn't grow and occupies only the area required by its content. The other views expand equally to fill the space remaining after the other views have been placed.

Using LinearLayout

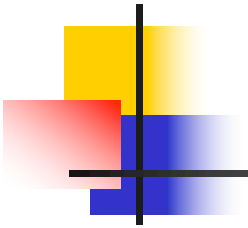



```
<Button
    android:id="@+id/button"
    android:layout_width="wrap_content”
    android:layout_height="wrap_content"
    android:layout_gravity="center”
    android:layout_weight="1”
    android:text="Button" />
  <Button
    android:id="@+id/button2"
    android:layout_height="wrap_content”
    android:layout_height="wrap_content"
    android:layout_gravity="center”
    android:layout_weight="1"
    android:text="Button" />
  <Button
    android:id="@+id/button3"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_gravity="center”
    android:text="Button" />
```

Using LinearLayout



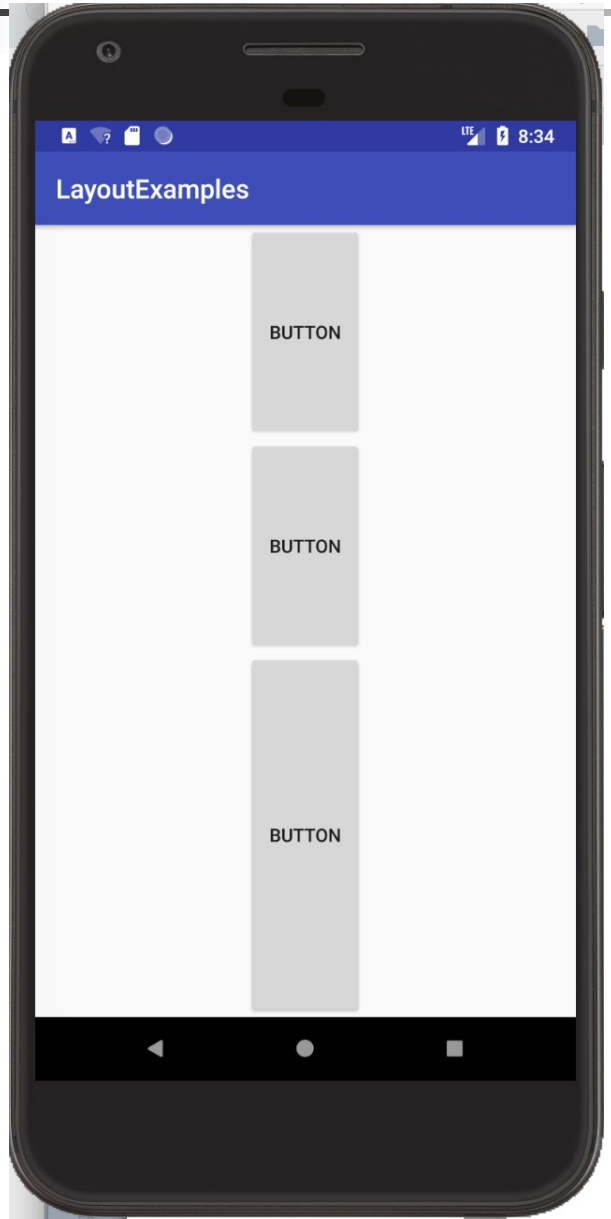

```
<Button
    android:id="@+id/button"
    android:layout_width="wrap_content”
    android:layout_height="wrap_content"
    android:layout_gravity="center”
    android:text="Button" />
  <Button
    android:id="@+id/button2"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:layout_gravity="center”
    android:layout_weight="1"
    android:text="Button" />
  <Button
    android:id="@+id/button3"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_gravity="center”
    android:text="Button" />
```



Using LinearLayout

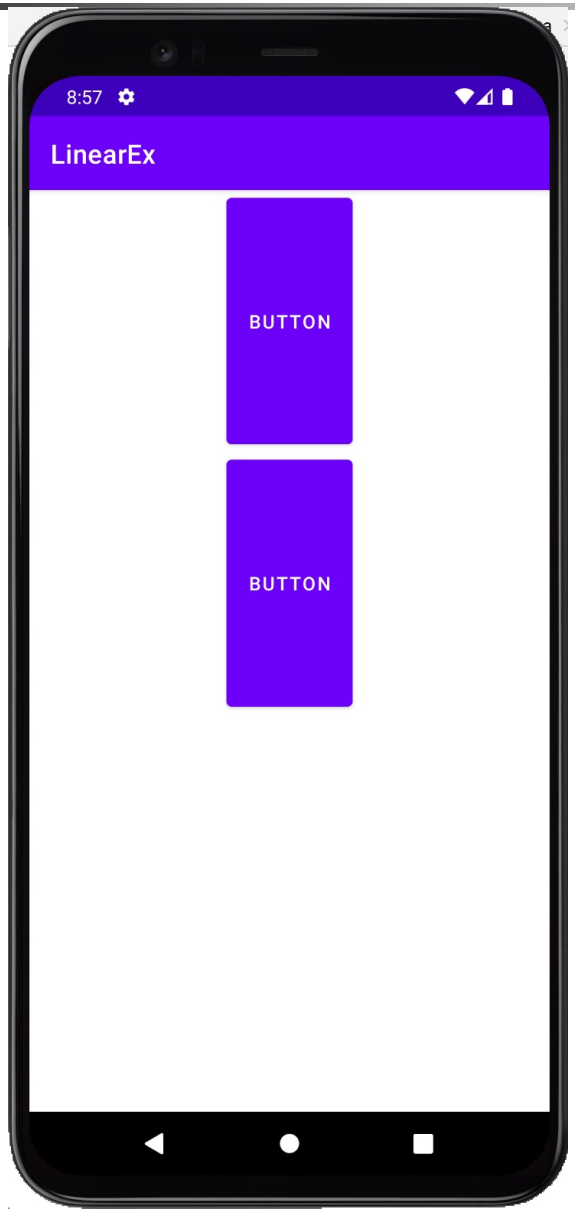
2. Assume there are three views and two of them have a weight of 1, while the third view is then given a weight of 2 (instead of 0), then it's now declared more important than both the others, so it gets half the total remaining space, while the first two share the rest equally.

Using LinearLayout



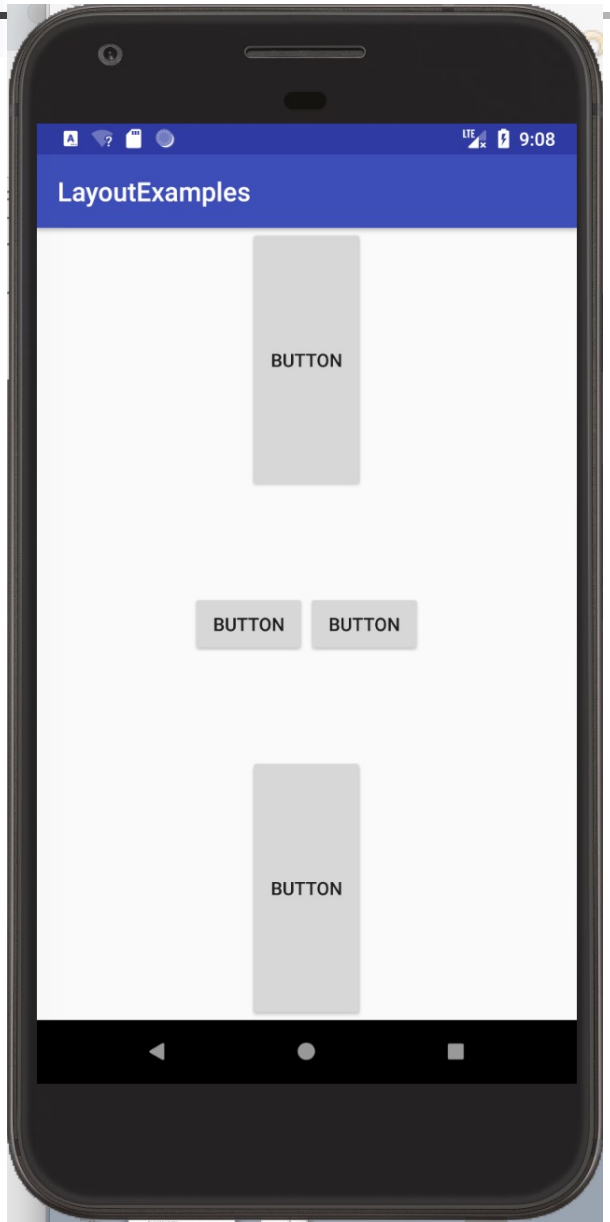

```
<Button
    android:id="@+id/button"
    android:layout_width="wrap_content”
    android:layout_height="wrap_content"
    android:layout_gravity="center”
    android:layout_weight="1”
    android:text="Button" />
  <Button
    android:id="@+id/button2"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:layout_gravity="center”
    android:layout_weight="1"
    android:text="Button" />
  <Button
    android:id="@+id/button3"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_gravity="center”
    android:layout_weight=“2”
    android:text="Button" />
```

Using LinearLayout: weightSum

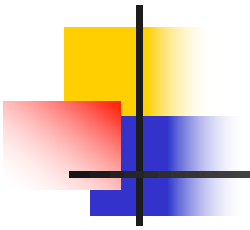



```
<LinearLayout
    android:weightSum="1.0"
    ...

    <Button
        android:id="@+id/button"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_gravity="center"
        android:layout_weight="0.25"
        android:text="Button" />

    <Button
        android:id="@+id/button2"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_gravity="center"
        android:layout_weight="0.25"
        android:text="Button" />
</LinearLayout>
```

Nesting LinearLayouts




```
<Button
   android:id="@+id/button2"
   android:layout_width="wrap_content"
   android:layout_height="wrap_content"
   android:layout_gravity="center"
   android:layout_weight="1"
   android:text="Button" />
<LinearLayout
   android:layout_width="wrap_content"
   android:layout_height="wrap_content"
   android:layout_gravity="center"
   android:layout_weight="1"
   android:orientation="horizontal”>
   <Button
      android:id="@+id/button4"
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:layout_gravity="center"
      android:layout_weight="1"
      android:text="Button" />
   <Button
      android:id="@+id/button"
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:layout_gravity="center"
      android:text="Button" />
</LinearLayout>
<Button
   android:id="@+id/button3"
   android:layout_width="wrap_content"
   android:layout_height="wrap_content"
   android:layout_gravity="center"
   android:layout_weight="1"
   android:text="Button" />
```



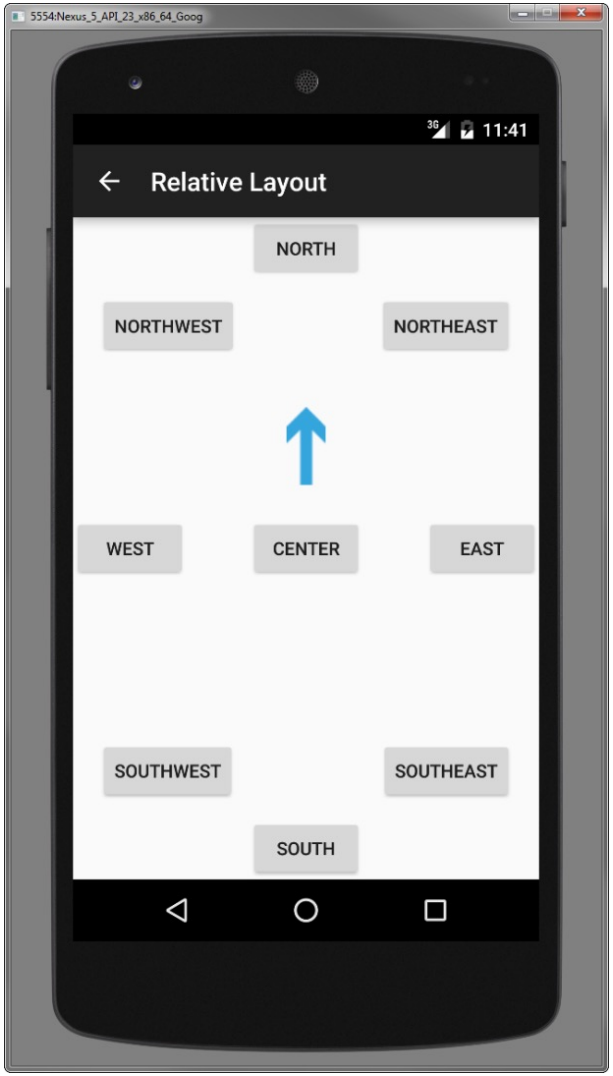
Using RelativeLayout

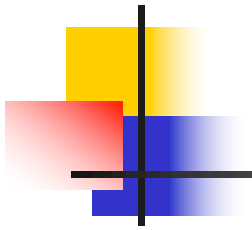
- The RelativeLayout* view enables you to specify where the child View controls are in relation to each other.
  - For instance, you can set a child View to be positioned “above” or “below” or “to the left of” or “to the right of” another View.
  - You can also align child View controls relative to one another or the parent layout edges.
- Combining RelativeLayout attributes can simplify the creation of interesting user interfaces without resorting to multiple layout groups to achieve a desired effect.

*) It is considered a “legacy” layout now.

Using RelativeLayout

This example from your textbook is a bit misleading as RelativeLayout does not have any "geographic" regions





Using RelativeLayout

- You can find the layout attributes available for RelativeLayout child View controls in `android.widget.RelativeLayout.LayoutParams`

Using RelativeLayout

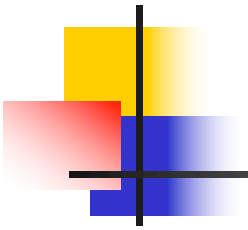
Attribute Name	Applies to	Value
`gravity`	Parent `View`	One or more constants separated by "|" such as `top`, `bottom`, `left`, `right`, and more
`layout_centerInParent`	Child `View`	`True` or `false`
`layout_centerHorizontal`	Child `View`	`True` or `false`
`layout_centerVertical`	Child `View`	`True` or `false`
`layout_alignParentTop`	Child `View`	`True` or `false`
`layout_alignParentBottom`	Child `View`	`True` or `false`
`layout_alignParentLeft`	Child `View`	`True` or `false`
`layout_alignParentRight`	Child `View`	`True` or `false`
`layout_alignParentStart`	Child `View`	`True` or `false`
`Layout_alignParentEnd`	Child `View`	`True` or `false`

Using RelativeLayout

Attribute Name	Applies To	Value
layout_alignRight	Child View	A View ID; e.g., @id/Button1
layout_alignLeft	Child View	A View ID; e.g., @id/Button1
layout_alignStart	Child View	A View ID; e.g., @id/Button1
layout_alignEnd	Child View	A View ID; e.g., @id/Button1
layout_alignTop	Child View	A View ID; e.g., @id/Button1
layout_alignBottom	Child View	A View ID; e.g., @id/Button1
layout_above	Child View	A View ID; e.g., @id/Button1
layout_below	Child View	A View ID; e.g., @id/Button1
layout_toLeftOf	Child View	A View ID; e.g., @id/Button1
layout_toRightOf	Child View	A View ID; e.g., @id/Button1

Using a RelativeLayout

```
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:id="@+id/RelativeLayout01"
    android:layout_height="match_parent"
    android:layout_width="match_parent">
    <Button android:id="@+id/ButtonCenter"
        android:text="Center" android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_centerInParent="true" />
    <ImageView android:id="@+id/ImageView01"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_above="@id/ButtonCenter"
        android:layout_centerHorizontal="true"
        android:src="@drawable/arrow" />
</RelativeLayout>
```



Using a FrameLayout

- A `FrameLayout` view is designed to display a stack of child View items.
- Child views are drawn as if on a stack, with the most recently added child (last) on top. Basically, the size of the `FrameLayout` is the size of its largest child.
- You can add multiple views to this layout, but each View is drawn from the top-left corner of the layout.
- You can find the layout attributes available for `FrameLayout` child View controls in `android.widget.FrameLayout.LayoutParams`.

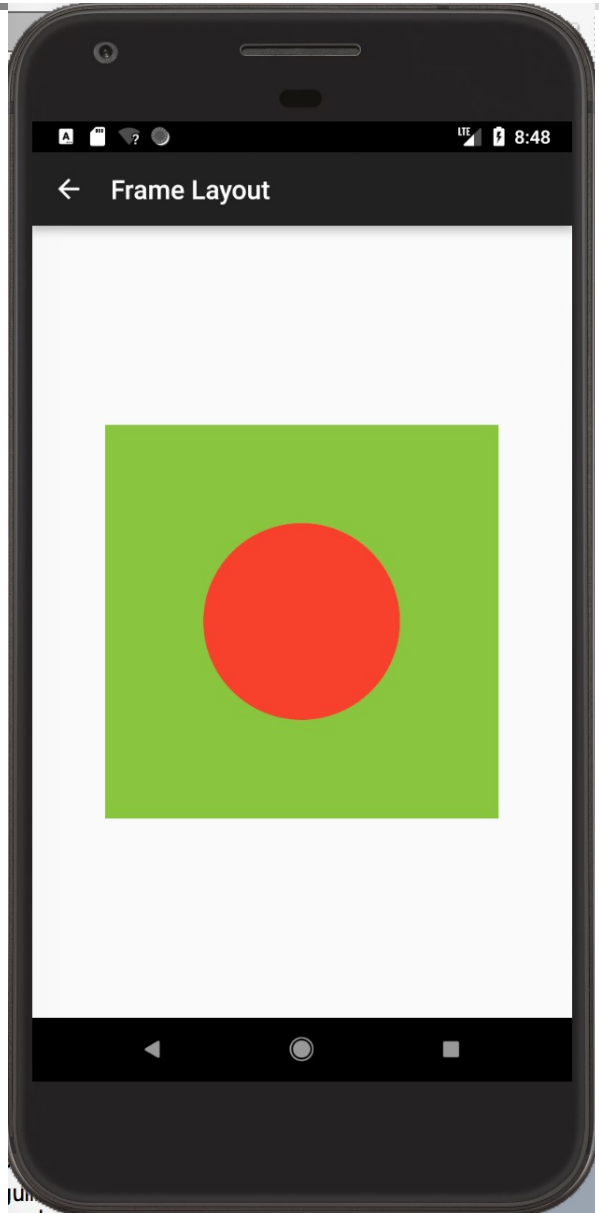
Using FrameLayout

Attribute Name	Applies to	Value
`foreground`	Parent `View`	Drawable resource
`foregroundGravity`	Parent `View`	One or more constants separated by "|" such as `top`, `bottom`, `left`, `right`, and many others
`measureAllChildren`	Parent `View`	`True` or `false`
`layout_gravity`	Child `View`	One or more constants separated by "|" such as `top`, `bottom`, `left`, `right`, and many others

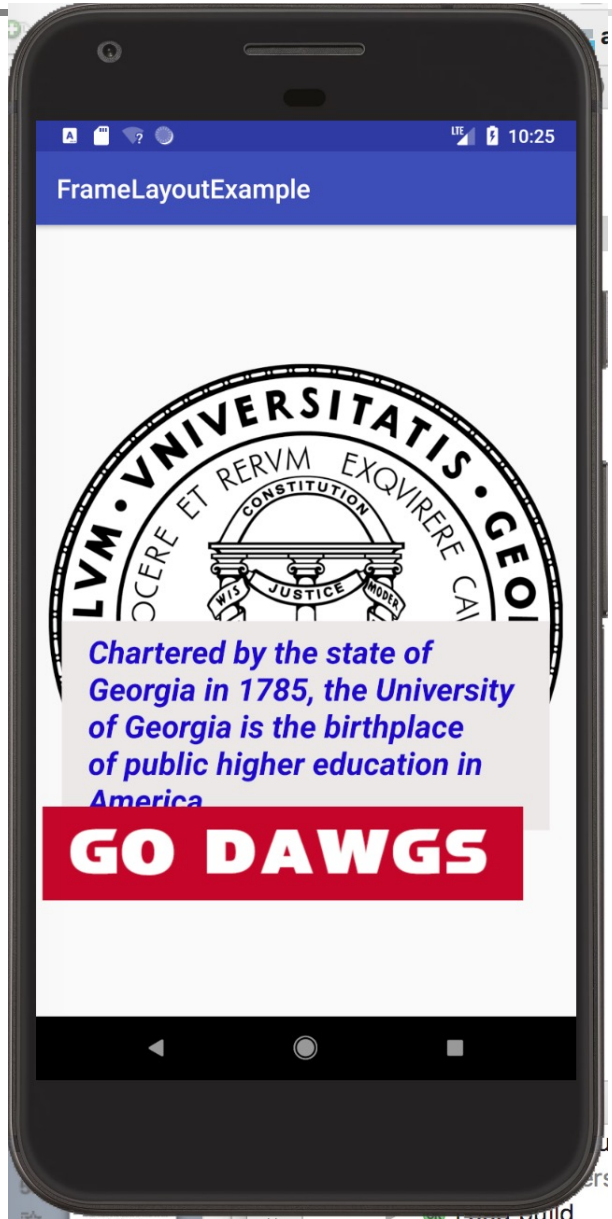
Using FrameLayout

```
<FrameLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:id="@+id/FrameLayout01"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_gravity="center">
    <ImageView android:id="@+id/ImageView01"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:src="@drawable/green_rect"
        android:contentDescription="@string/green_rect"
        android:minHeight="200dp" android:minWidth="200dp" />
    <ImageView android:id="@+id/ImageView02"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:src="@drawable/red_oval"
        android:contentDescription="@string/red_oval"
        android:minHeight="100dp" android:minWidth="100dp"
        android:layout_gravity="center" />
</FrameLayout>
```

Using FrameLayout



Using FrameLayout



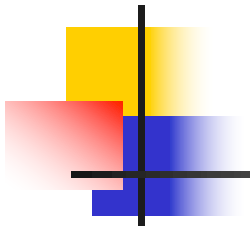
You can create a collage-like arrangement

Using FrameLayout

The GO DAWGS image
Has been rotated by
30 degrees here, by adding

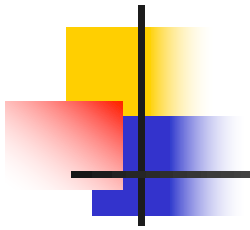
android:rotation="30"

Using TableLayout



- A TableLayout view organizes children into rows.
- You add individual View controls within each row of the table using a TableRow layout View for each row of the table.
- Each column of the TableRow can contain one View (or layout with child View controls).
- You place View items added to a TableRow in columns in the order they are added.
- You can specify the column number (zero based) to skip columns as necessary; otherwise, the View control is put in the next column to the right.

Using TableLayout



- Columns scale to the size of the largest View of that column.
- You can also include normal View controls instead of TableRow elements, if you want the View to take up an entire row.
- You can find the layout attributes available for TableLayout child View controls in `android.widget.TableLayout.LayoutParams`.
- You can find the layout attributes available for TableRow child View controls in `android.widget.TableRow.LayoutParams`.

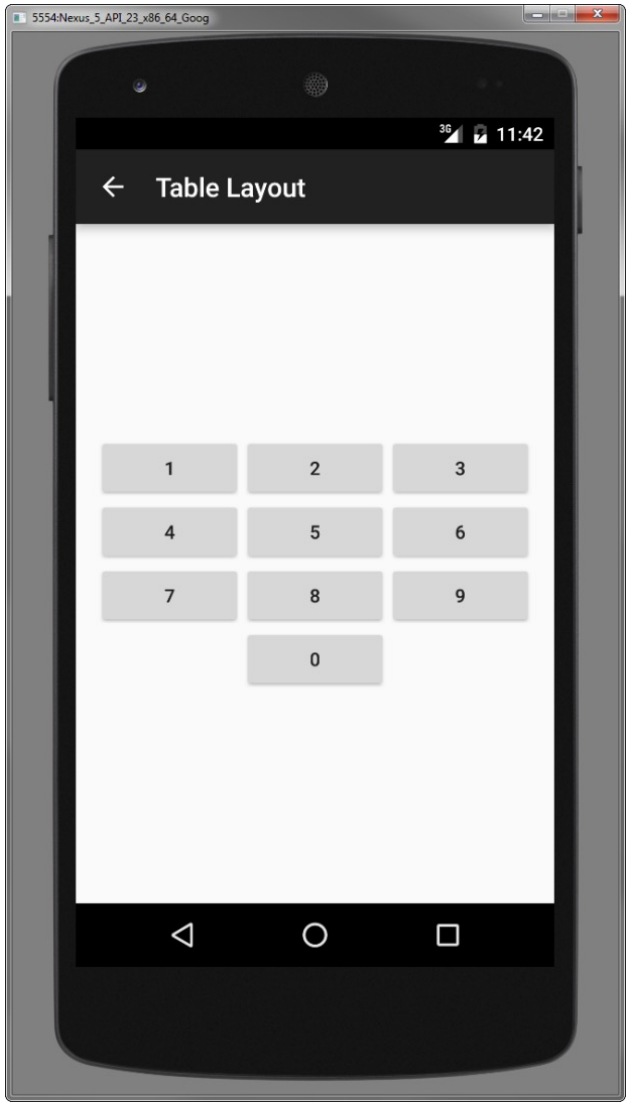
Using TableLayout

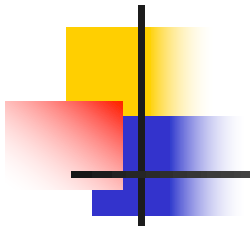
Attribute Name	Applies to	Value
`collapseColumns`	`TableLayout`	String or string resource; for example, `"0,1,3,5"`
`shrinkColumns`	`TableLayout`	String or string resource; use “*” for all columns; for example, `"0,1,3,5"`
`stretchColumns`	`TableLayout`	String or string resource; use “*” for all columns; for example, `"0,1,3,5"`
`layout_column`	`TableRow` child `View`	Integer or integer resource; for example, `1`
`layout_span`	`TableRow` child `View`	Integer or integer resource greater than or equal to 1; for example, `3`

Using TableLayout

```
<TableLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:id="@+id/TableLayout01"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:stretchColumns="*">
    <TableRow android:id="@+id/TableRow01">
        <Button android:id="@+id/ButtonLeft" android:text="Left Door"/>
        <Button android:id="@+id/ButtonMiddle" android:text="Middle Door"/>
        <Button android:id="@+id/ButtonRight" android:text="Right Door"/>
    </TableRow>
    <TableRow android:id="@+id/TableRow02">
        <Button
            android:id="@+id/ButtonBack"
            android:text="Go Back"
            android:layout_column="1"/>
    </TableRow>
</TableLayout>
```

Using TableLayout

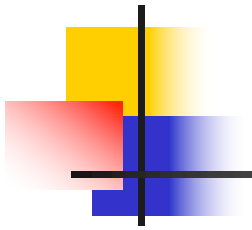




Using GridLayout

- GridLayout* was introduced in Android 4.0 (API Level 14), but it is already considered a “legacy” layout!
- The GridLayout organizes its children inside a grid.
- But don’t confuse it with GridView.
  - This layout grid is dynamically created.
- Unlike a TableLayout, child View controls in a GridLayout can span rows and columns, and are flatter and more efficient in terms of layout rendering.

*) It is considered a “legacy” layout now.



Using GridLayout

- In fact, it is the child View controls of a GridLayout that tell the layout where they want to be placed.
- You can find the layout attributes available for GridLayout child View controls in `android.widget.GridLayout.LayoutParams`.

Using GridLayout

Attribute Name	Applies to	Value
`columnCount`	`GridLayout`	A whole number; e.g., `4`
`rowCount`	`GridLayout`	A whole number; e.g., `3`
`orientation`	`GridLayout`	Can be vertical (down a row) or horizontal (over a column)
`layout_column`	Child `View` of `GridLayout`	Integer or integer resource; e.g., `1`
`layout_columnSpan`	Child `View` of `GridLayout`	Integer or integer resource greater than or equal to 1; e.g., `3`
`layout_row`	Child `View` of `GridLayout`	Integer or integer resource; e.g., `1`
`layout_rowSpan`	Child `View` of `GridLayout`	Integer or integer resource greater than or equal to 1; e.g., `3`
`layout_gravity`	Child `View` of `GridLayout`	One or more constants separated by "|"; `baseline`, `top`, `bottom`, `left`, `right`, and others

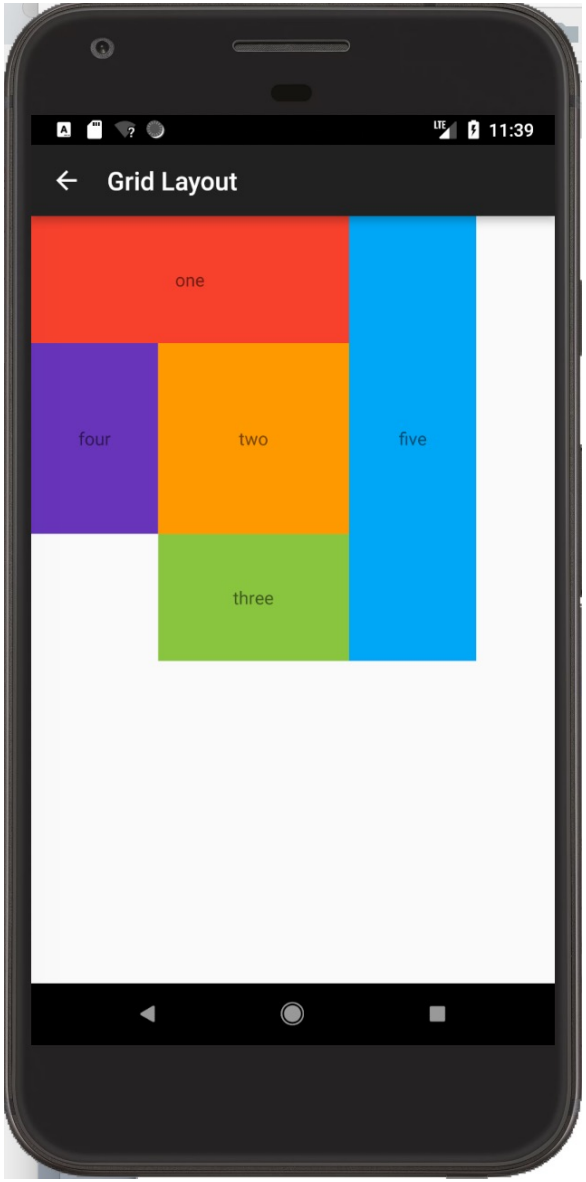
Using GridLayout

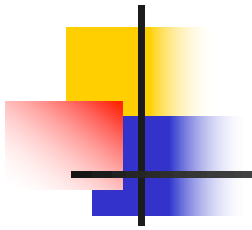
```
<GridLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:id="@+id/gridLayout1" android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:columnCount="4"
    android:rowCount="4" >
    <TextView android:layout_width="150dp"
                  android:layout_height="50dp"
       android:layout_column="0" android:layout_columnSpan="3"
       android:layout_row="0" android:background="#ff0000"
       android:gravity="center" android:text="one" />
    …
```

Using GridLayout

```
...
    <TextView android:layout_width="50dp"
        android:layout_height="50dp"
        android:layout_column="2" android:layout_row="3"
        android:background="#00ff00"  android:gravity="center"
        android:text="three" />
    <TextView android:layout_width="50dp"
        android:layout_height="50dp"
        android:layout_column="0" android:layout_row="1"
        android:background="#0000ff" android:gravity="center"
        android:text="four" />
    <TextView android:layout_width="50dp"
        android:layout_height="200dp"
        android:layout_column="3" android:layout_row="0"
        android:layout_rowSpan="4"
        android:background="#0077ff"
        android:gravity="center" android:text="five" />
</GridLayout>
```

Using GridLayout



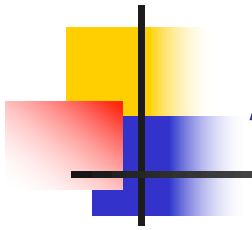


Using Multiple Layouts on a Screen

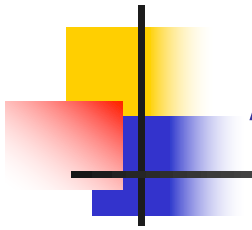
- Combining different layout methods on a single screen can create complex layouts.
- Remember that because a layout contains View controls and is, itself, a View control, it can contain other layouts.

Using Multiple Layouts on a Screen

Adding Scrolling Support



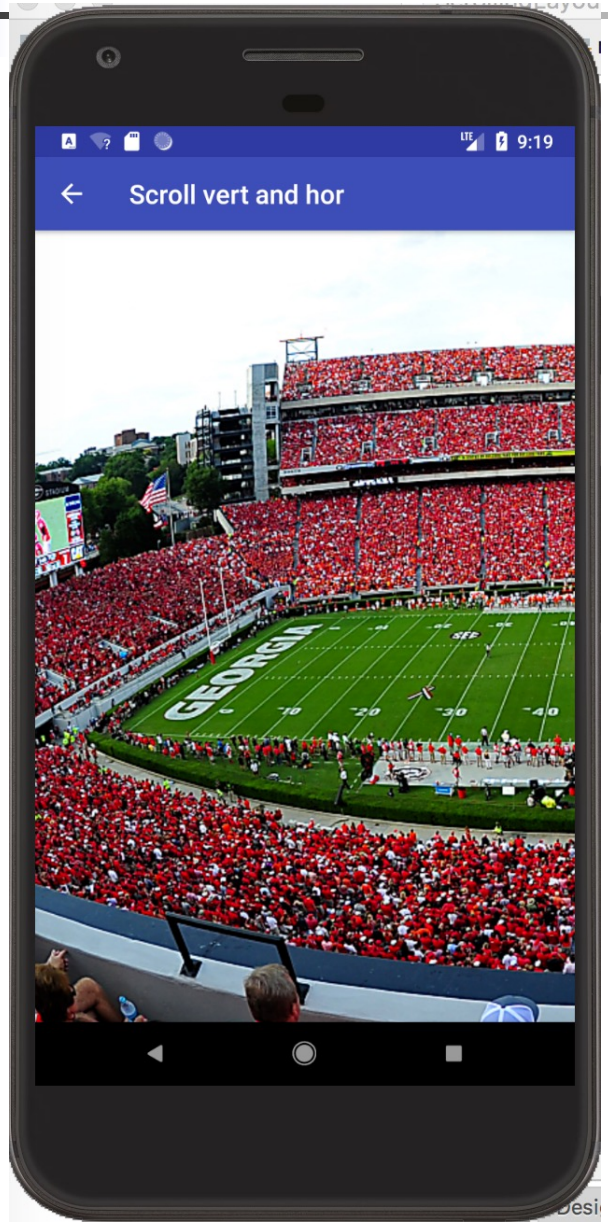
- One of the easiest ways to provide vertical scrolling for a screen is by using these controls:
  - `ScrollView` (vertical scrolling)
  - `HorizontalScrollView` (horizontal scrolling)
- Either control can be used as a wrapper container, causing all children View controls to have one continuous scroll bar.



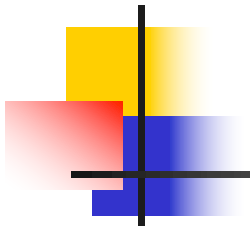
Adding Scrolling Support

- However, the `ScrollView` and `HorizontalScrollView` controls can have only one child, so it's customary to have that child be a layout, such as a `LinearLayout`, which then contains all the "real" child controls to be scrolled through.

Adding Scrolling Support




```
<ScrollView
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:id="@+id/ScrollView01"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:scrollbars="horizontal|vertical”>
    <HorizontalScrollView
        android:id="@+id/HorizontalScrollView01"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content”>
        <LinearLayout
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:orientation="vertical”>
            <ImageView
                android:id="@+id/ImageView01"
                android:layout_width="wrap_content"
                android:layout_height="wrap_content"
                android:contentDescription="UGA game"
                android:scaleType="matrix"
                android:src="@drawable/georgia_game" />
        </LinearLayout>
    </HorizontalScrollView>
</ScrollView>
```

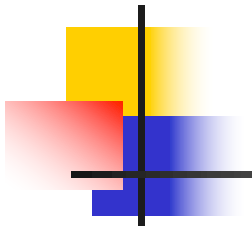


Using Container Control Classes

- Layouts are not the only controls that can contain other View controls.
- Although layouts are useful for positioning other View controls on the screen, they aren't interactive.
- Now let's talk about the other kind of ViewGroup, the containers.
- These View controls encapsulate other, simpler View types and give the user the ability to interactively browse the child View controls in a standard fashion.
- Much like layouts, each of these controls has a special, well-defined purpose (functionality).

Using Container Control Classes

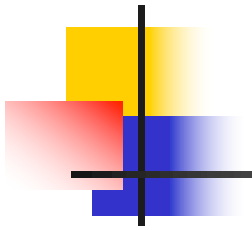
- The types of ViewGroup containers built into the Android SDK framework include:
  - Lists and grids
  - ScrollView and HorizontalScrollView for scrolling
  - ViewFlipper, ViewSwitcher, ImageSwitcher and TextSwitcher for switching views, images and text views using animations



Using Data-Driven Containers

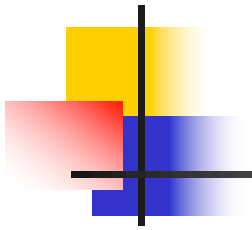
- Some of the View container controls are designed for displaying repetitive View controls in a particular way.
- Examples are (both considered legacy, now):
  - ListView
  - GridView
- These containers are all types of `AdapterView` controls.
- An `AdapterView` control contains a set of child View controls to display data from some data source.
- An Adapter generates these child View controls from a data source.

Using Data-Driven Containers

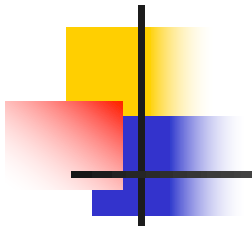


- In the Android SDK, an Adapter reads data from some data source and generates the data for a View control based on some rules, depending on the type of Adapter used.
- This View is used to populate the child View controls of a particular `AdapterView`.
- The most common Adapter classes are the `CursorAdapter` and the `ArrayAdapter`.
- The `CursorAdapter` gathers data from a `Cursor`, whereas the `ArrayAdapter` gathers data from an array.

Using Data-Driven Containers



- A `CursorAdapter` is a good choice when using data from a database (we will talk about it later).
- An `ArrayAdapter` is a good choice when there is only a single column of data or when the data comes from a resource array.



Using Data-Driven Containers

- When creating an Adapter, you provide a layout identifier.
- This layout is the template for filling in each row of data.
- The template you create contains identifiers for controls to which the Adapter assigns data.
- A simple layout can contain as little as a single TextView control.
- When making an Adapter, refer to both the layout resource and the identifier of the TextView control.
- The Android SDK provides common layout resources.

Using ArrayAdapter

- An ArrayAdapter binds each element of the array to a single View control within the layout resource.
- Here is an example of creating an ArrayAdapter:

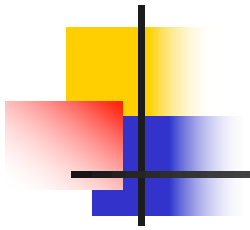
```
private String[] items = {
    "Item 1", "Item 2", "Item 3" };
ArrayAdapter adapt =
    new ArrayAdapter<String>
        (this, R.layout.textview, items);
```

Using ArrayAdapter

Layout for a list item is defined in

`textview.xml`:

```
<TextView
    xmlns:android= "http://schemas.android.com/apk/res/android"
      android:layout_width="match_parent"
      android:layout_height="wrap_content"
      android:textSize="20sp" />
```



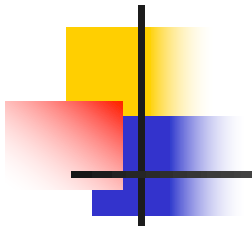
Using CursorAdapter

- A CursorAdapter binds one or more columns of data to one or more View controls within the layout resource provided.

Using CursorAdapter (Cont'd)

```
CursorLoader loader = new CursorLoader( this,
    ContactsContract.CommonDataKinds.Phone.CONTENT_URI,
    null, null, null, null);
Cursor contacts = loader.loadInBackground();
ListAdapter adapter = new SimpleCursorAdapter(
    this, R.layout.scratch_layout,
    contacts, new String[] {
        ContactsContract.CommonDataKinds.Phone.DISPLAY_NAME,
        ContactsContract.CommonDataKinds.Phone.NUMBER },
        new int[] {
           R.id.scratch_text1,
           R.id.scratch_text2
        },
     0);
```

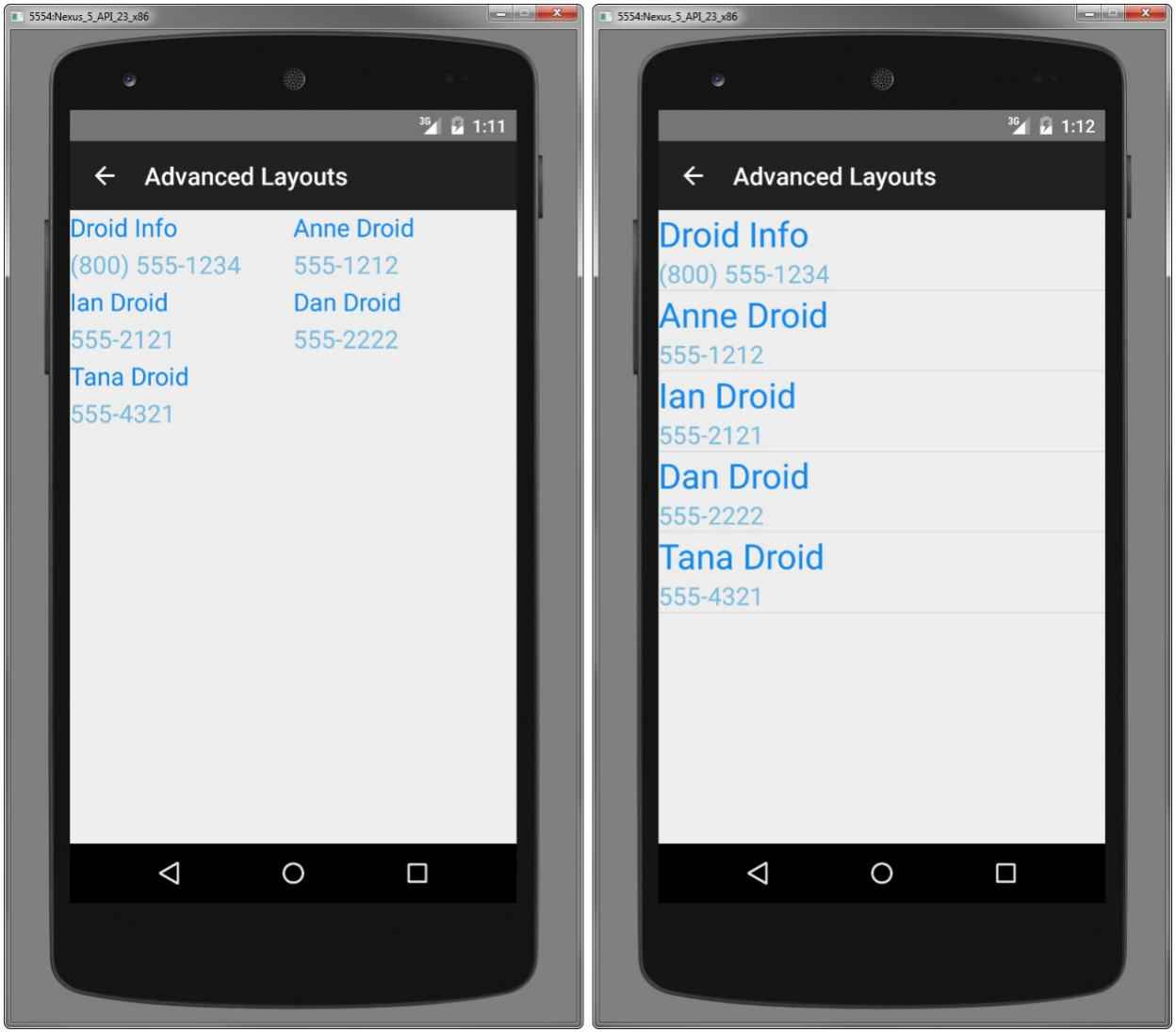
Binding Data to the AdapterView

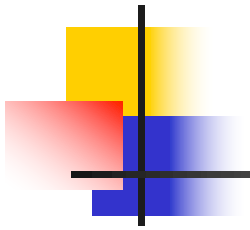


- Now that you have an Adapter object, you can apply this to one of the AdapterView controls.
- Either of them will work.
- Here is an example of this with a ListView, continuing on from the previous sample code:

```
((ListView) findViewById( R.id.scratch_adapter_view ))
                                        .setAdapter( adapter );
```

Binding Data to the AdapterView





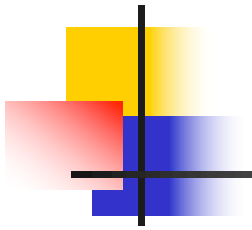
Handling Selection Events

- You often use AdapterView controls to present data from which the user should select.
- Both of the discussed controls — ListView and GridView — enable your application to monitor for click events in the same way.
- You need to call setOnItemClickListener() on your AdapterView and pass in an implementation of the AdapterView.OnItemClickListener class.

Handling Selection Events

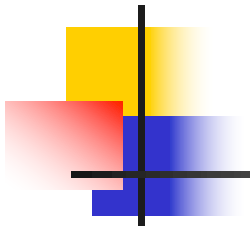
```
av.setOnItemClickListener(
    new AdapterView.OnItemClickListener() {
    @Override
    public void onItemClick(
        AdapterView<?> parent, View view,
        int position, long id) {
        Toast.makeText(Scratch.this, "Clicked _id="+id,
          Toast.LENGTH_SHORT).show();
    }
});
```

Using ListView with ListFragment



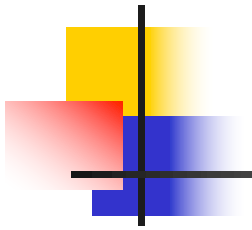
- The ListView control is commonly used for full-screen menus or lists of items from which a user selects (it is a legacy view now).
- You might consider using ListFragment as the base class for such screens and adding the ListFragment to your View.
- Using the ListFragment can simplify these types of screens.
- We will discuss fragments later (also discussed in Chapter 9).

Using ListView with ListFragment



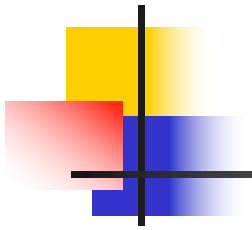
- First, to handle item events, you now need to provide an implementation in your ListFragment.
- For instance, the equivalent of `OnItemClickListener` is to implement the `onItemClick()` method within your ListFragment that implements the `AdapterView.OnItemClickListener` interface.

Using ListView with ListFragment

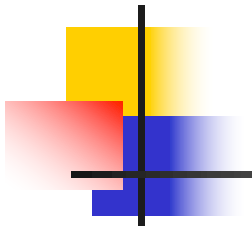


- Second, to assign an Adapter, you need a call to the `setListAdapter()` method.
- You do this after the call to the `setContentView()` method of your Activity, in the ListFragment method named `onActivityCreated()`.
- However, this hints at some of the limitations of using ListFragment.

Using ListView with ListFragment



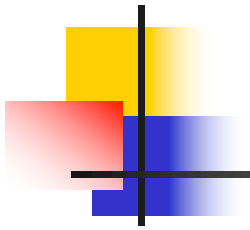
- To use ListFragment, the layout that is inflated inside your View Fragment with the `onCreateView()` method of the ListFragment must contain a ListView with the identifier set to `@android:id/list`; this cannot be changed.
- Second, you can also have a View with an identifier set to `@android:id/empty` to have a View display when no data is returned from the Adapter.



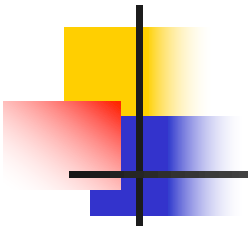
Using ListView with ListFragment

- Finally, this works only with ListView controls, so it has limited use.
- However, when it does work for your application, it can save some coding.

Exploring Other View Containers



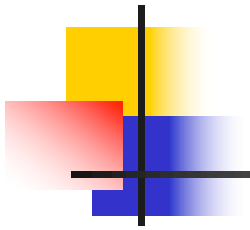
- Many other user interface controls are available within the Android SDK.
- Some of these controls are listed here (we will take a closer look at some of them, later):
  - Toolbar
  - SwipeRefreshLayout
  - RecyclerView
  - CardView
  - ViewPager and ViewPager2 (newer)
  - DrawerLayout



Using ConstraintLayout

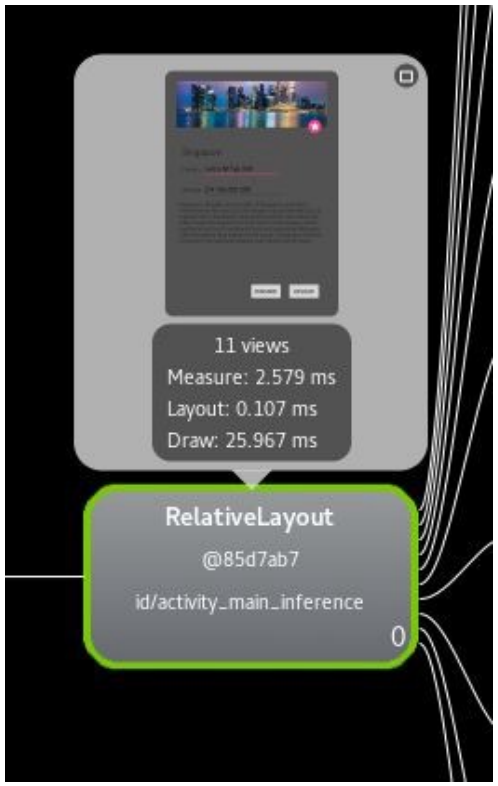
- `ConstraintLayout` was introduced to:
  - separate this layout from the Android framework for faster upgrade route,
  - reduce layout nesting – reduce the time overhead (tests do not prove this point!)
  - offer comprehensive support within the Android Studio – provide a great point-and-click support for layout design.

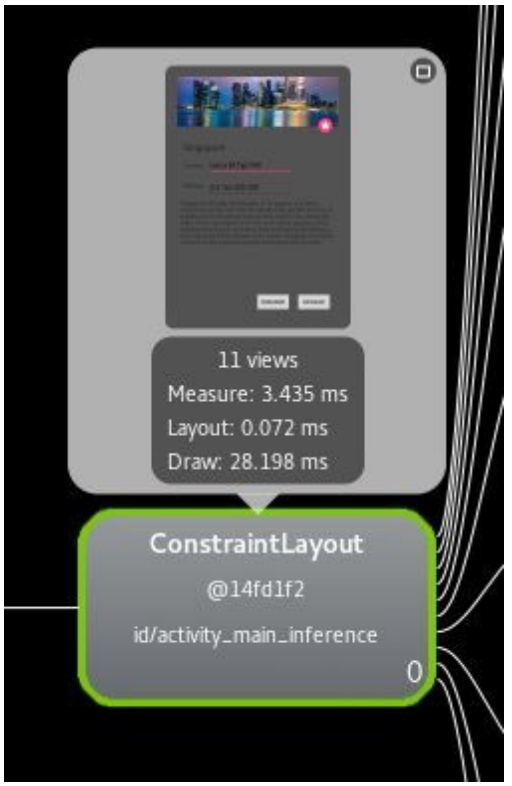
Reasons For ConstraintLayout



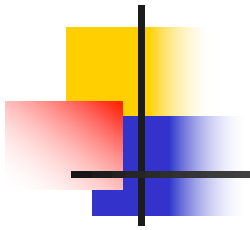
- Faster development and distribution
  - up to developers, not phone manufacturers
  - distributed just like other support libraries
- ConstraintLayout code is placed in:
  - constraint-layout-2.1.4.aar
  - constraintlayout-solver-2.0.4.jar
- compatible with API 9 and up
  - 99.9% of Android devices

Caution: May Be Slower



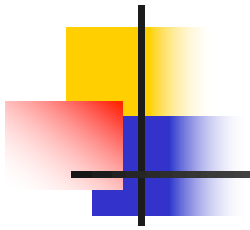


2.579	Measure	3.435
0.107	Layout	0.072
25.967	Draw	28.198



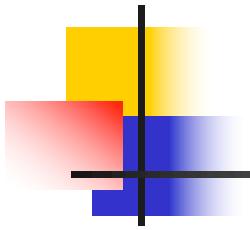
Reasons For ConstraintLayout

- Available excellent support within Android Studio.
- New blueprint mode since AS 2.2 Preview
  - for both old layouts and new ConstraintLayout
- First time when UI editor is actually usable; before, developers used to code directly in XML or to create the UI programmatically.



Using ConstraintLayout

- Layouts are flat (or they can be).
- Somewhat similar, but more flexible than RelativeLayout.
- Includes some capabilities of LinearLayout.



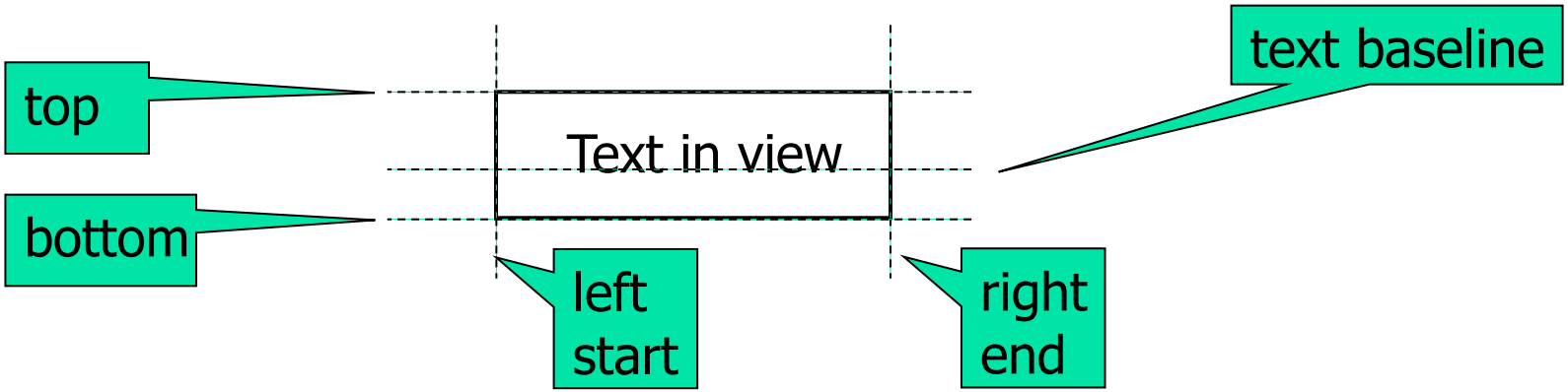
Using ConstraintLayout

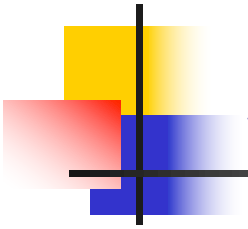
- A constraint defines a requirement on how two Views or a View and the parent are positioned, relative to each other.
- At least one vertical and one horizontal constraints are needed to define a View's position.
- An optional margin can be used to create a gap between Views or a View and the parent.

Using ConstraintLayout

More specifically, positioning of a View can be constrained relative to (with respect to) another View or parent:

- along its horizontal axis:
  - left, right, start and end sides
- along its vertical axis:
  - top, bottom sides and text baseline



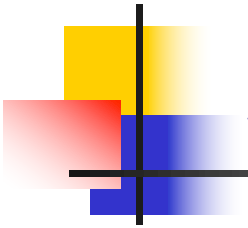


Available constraint types

All of the constraints below are relative to another View (given by id) or to the parent.

These are order constraints:

```
layout_constraintLeft_toRightOf
layout_constraintRight_toLeftOf
layout_constraintTop_toBottomOf
layout_constraintBottom_toTopOf
layout_constraintStart_toEndOf
layout_constraintEnd_toStartOf
```



Available constraint types

All of the constraints below are relative to another View (given by id) or to the parent.

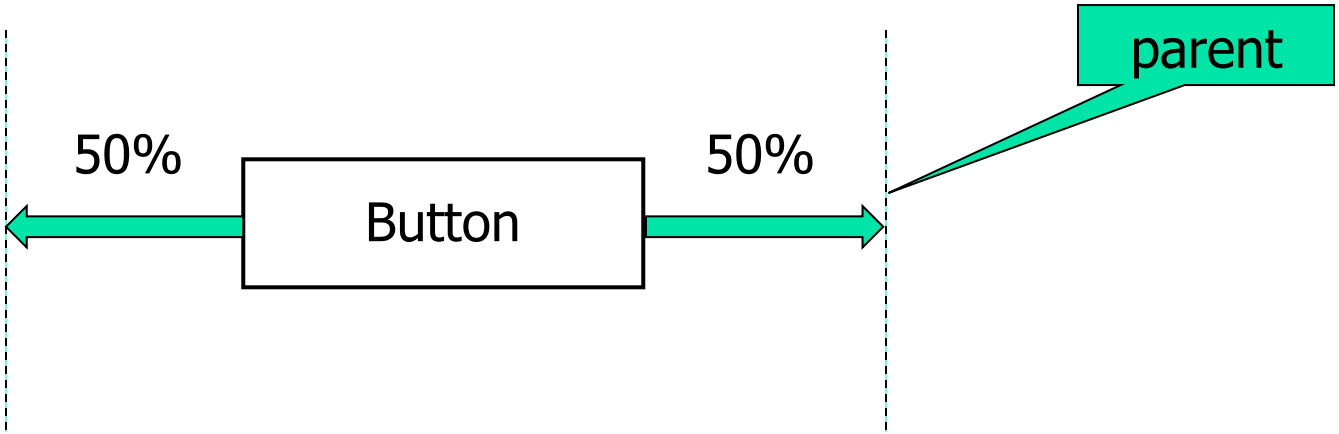
These are alignment constraints:

```
layout_constraintLeft_toLeftOf
layout_constraintRight_toRightOf
layout_constraintTop_toTopOf
layout_constraintBottom_toBottomOf
layout_constraintBaseline_toBaselineOf
layout_constraintStart_toStartOf
layout_constraintEnd_toEndOf
```

Bias and Centering

If a View is narrower than the parent, the following constraints will, by default, center the View (similarly for height):

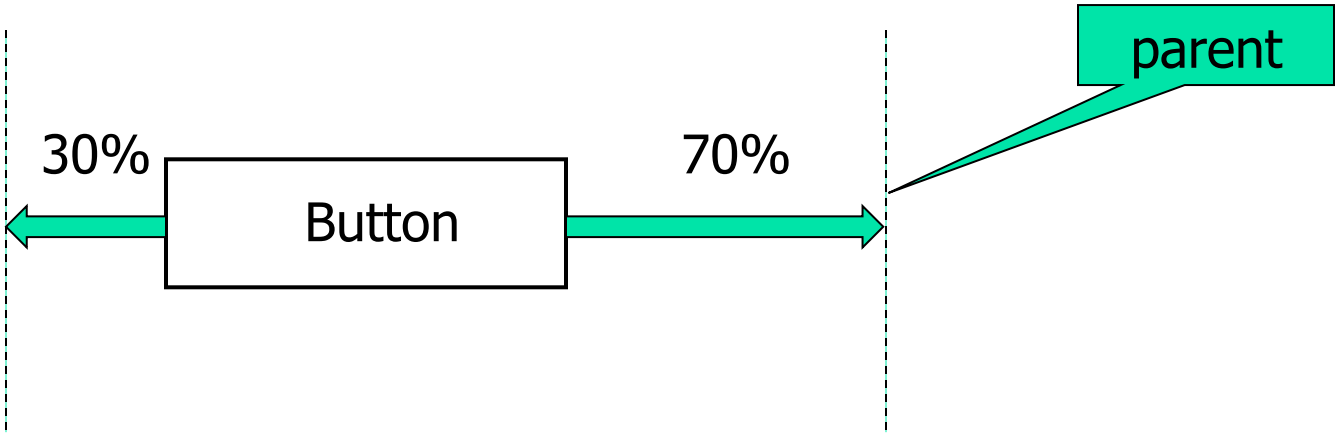
```
app:layout_constraintLeft_toLeftOf="parent”
app:layout_constraintRight_toRightOf="parent”
```



Bias and Centering

It is possible to move the View off-center by specifying a bias attribute:

```
app:layout_constraintHorizontal_bias="0.3"
app:layout_constraintLeft_toLeftOf="parent”
app:layout_constraintRight_toRightOf="parent”
```

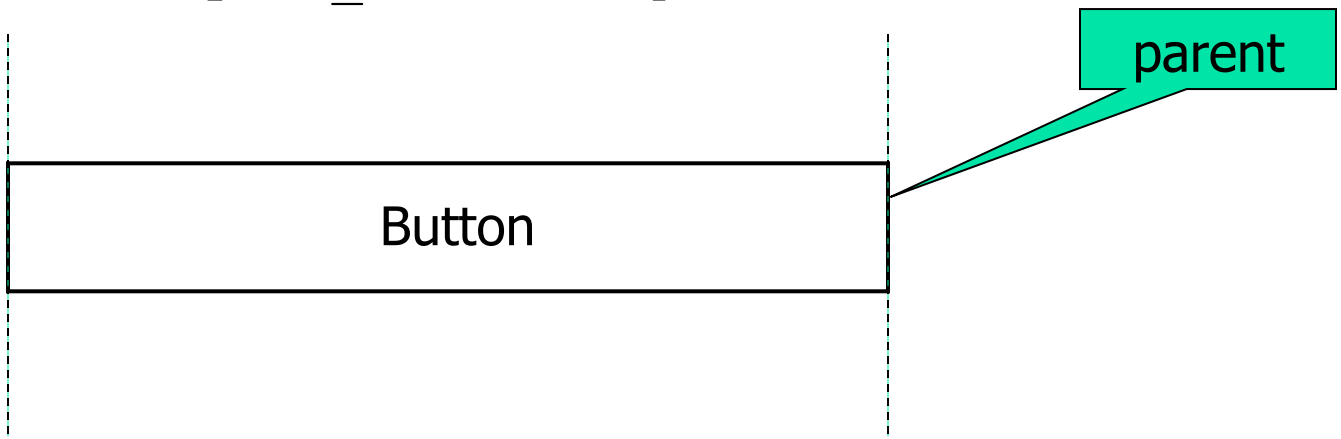


View sizing: 0dp (match_constraint)

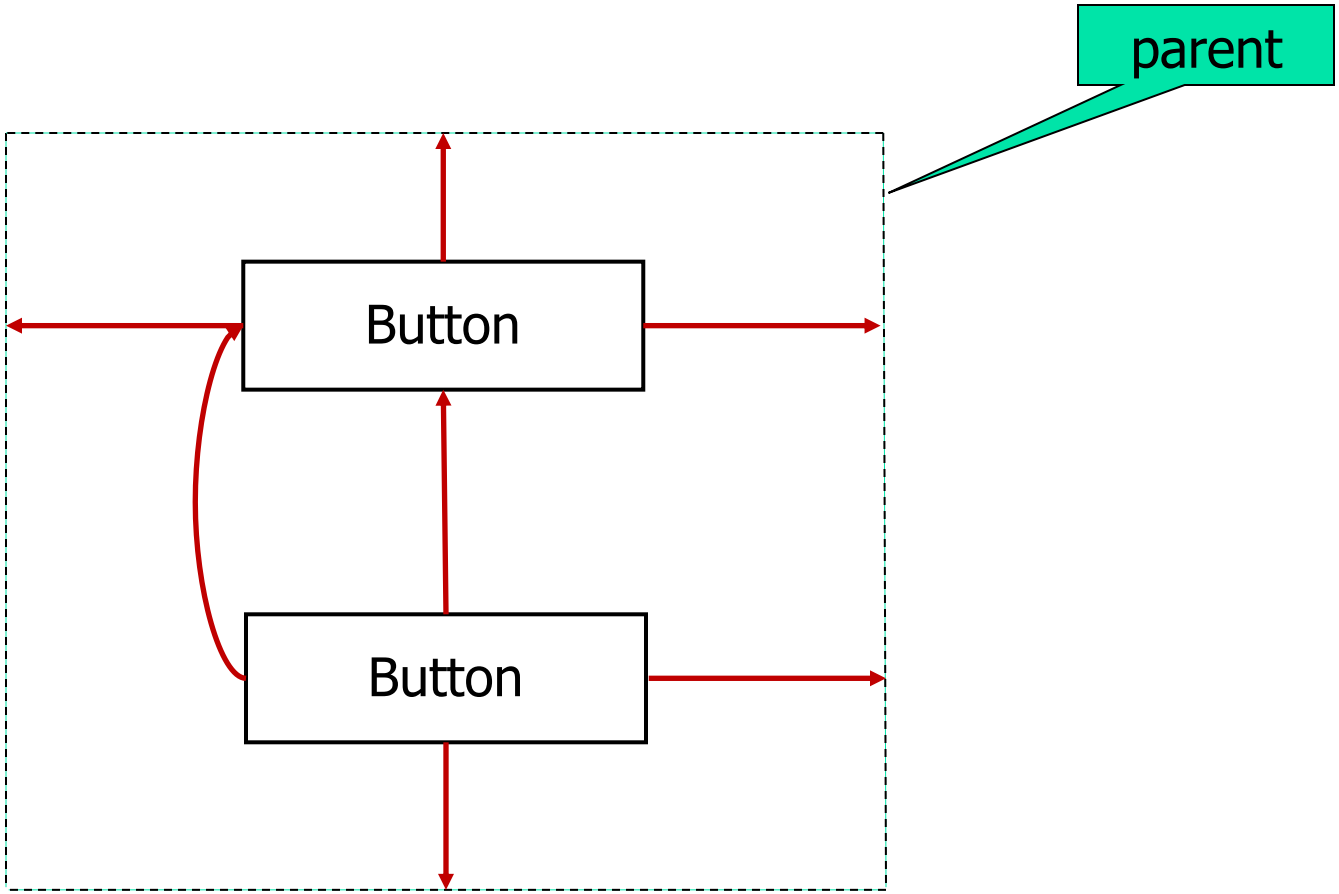
A view may be sized as:

- wrap_content
- match_constraint (or 0dp)

```
app:layout_constraintLeft_toLeftOf="parent”
app:layout_constraintRight_toRightOf="parent”
android:layout_width="0dp"
```

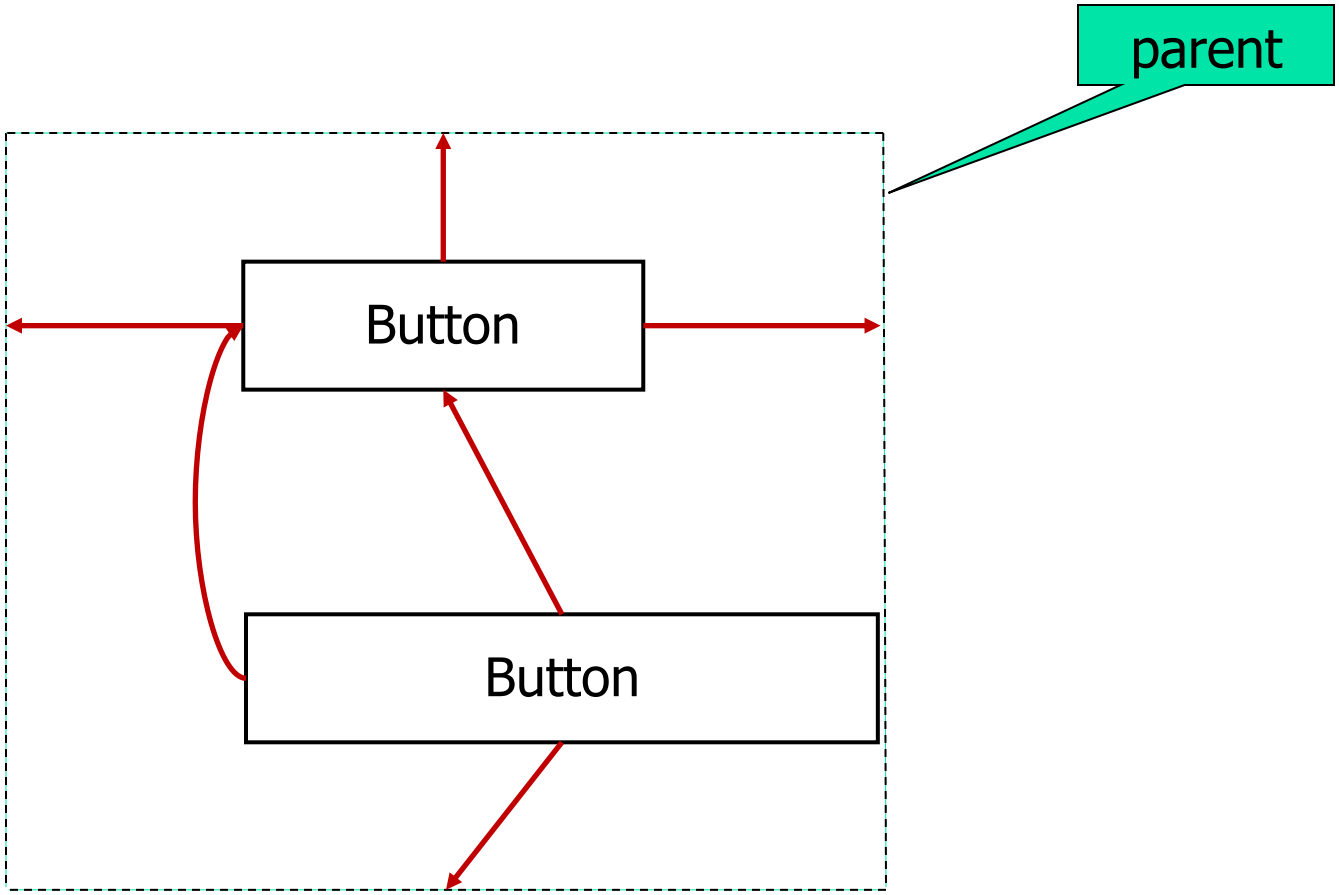


View sizing: 0dp (match_constraint)



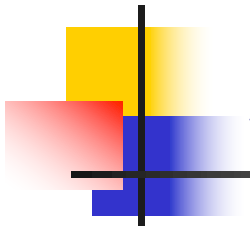

```
app:layout_constraintStart_toStartOf="@+id/button”
layout_width=”wrap_content"
```

View sizing: 0dp (match_constraint)

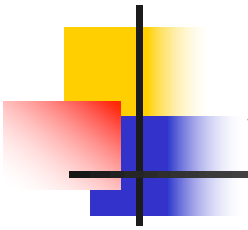



```
layout_width="0dp”
app:layout_constraintStart_toStartOf="@+id/button”
app:layout_constraintRight_toRightOf="parent”
```

Additional constraint types



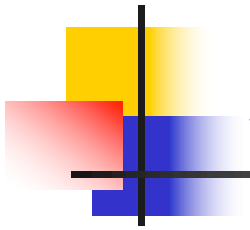
- View's constraints may also be relative to:
  - a vertical (or horizontal) guideline, or
  - a barrier (start, left, right, top, bottom)
- A guideline is an invisible line, which is placed at a specified position.
- Other Views can then be constrained (as usual) to an existing guideline.
- It is a convenient anchor for positioning Views.



Additional constraint types

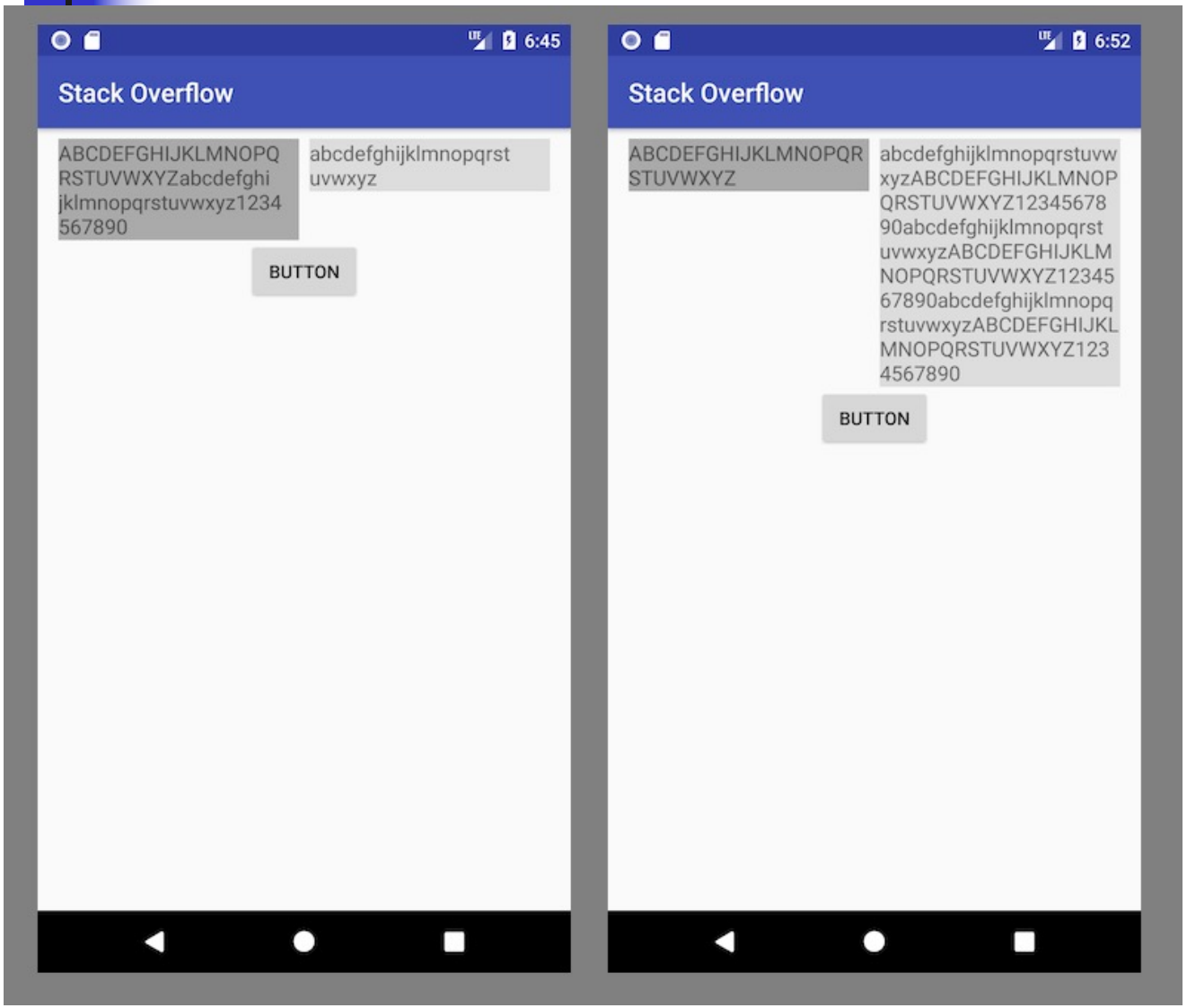
- A guideline can be positioned by providing:
  - a fixed distance from the left or the top of a layout (layout_constraintGuide_begin), usually in dp units,
  - a fixed distance from the right or the bottom of a layout (layout_constraintGuide_end), usually in dp units, or
  - a percentage of the width or the height of a layout (layout_constraintGuide_percent), in decimal fraction units, e.g., 0.4.

Additional constraint types



- A barrier is an invisible guideline which is not at a fixed in position. It moves, as the most extreme (dominant) View in it changes size or is repositioned.
- Other Views can then be constrained to an existing barrier.

Additional constraint types

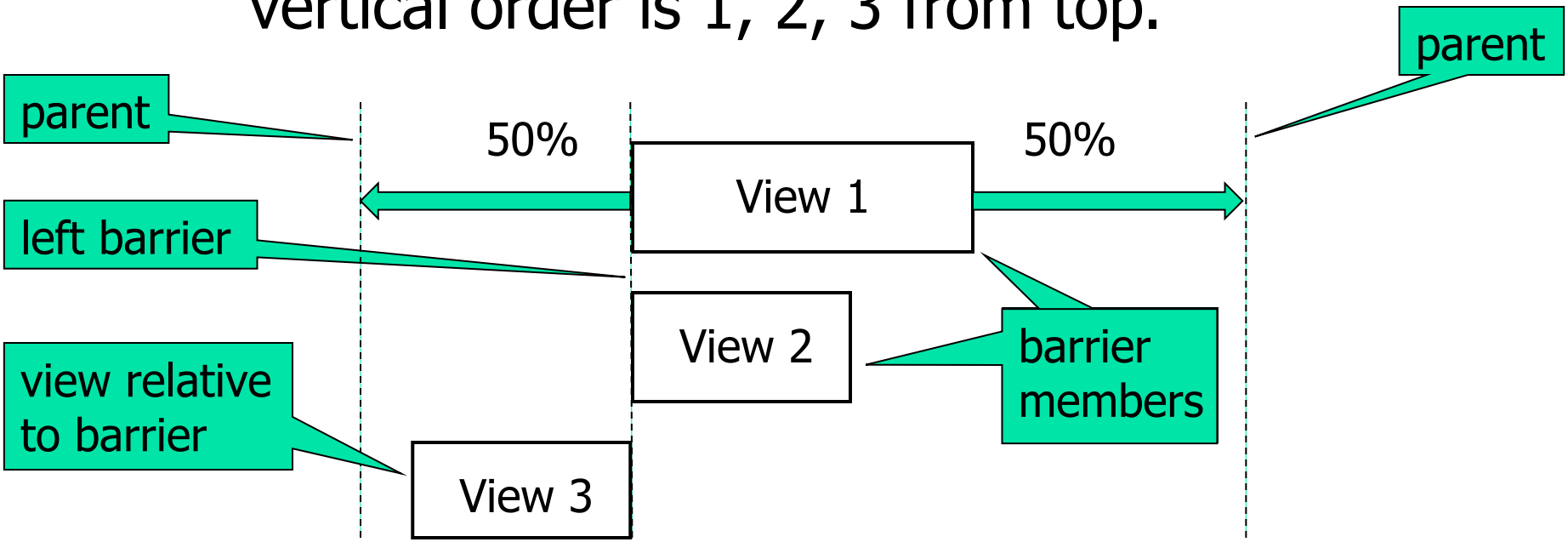


Example from stackexchange.com

Example:
Use a **barrier** to make sure that the button is always below both TextViews, irrespective of their vertical sizes

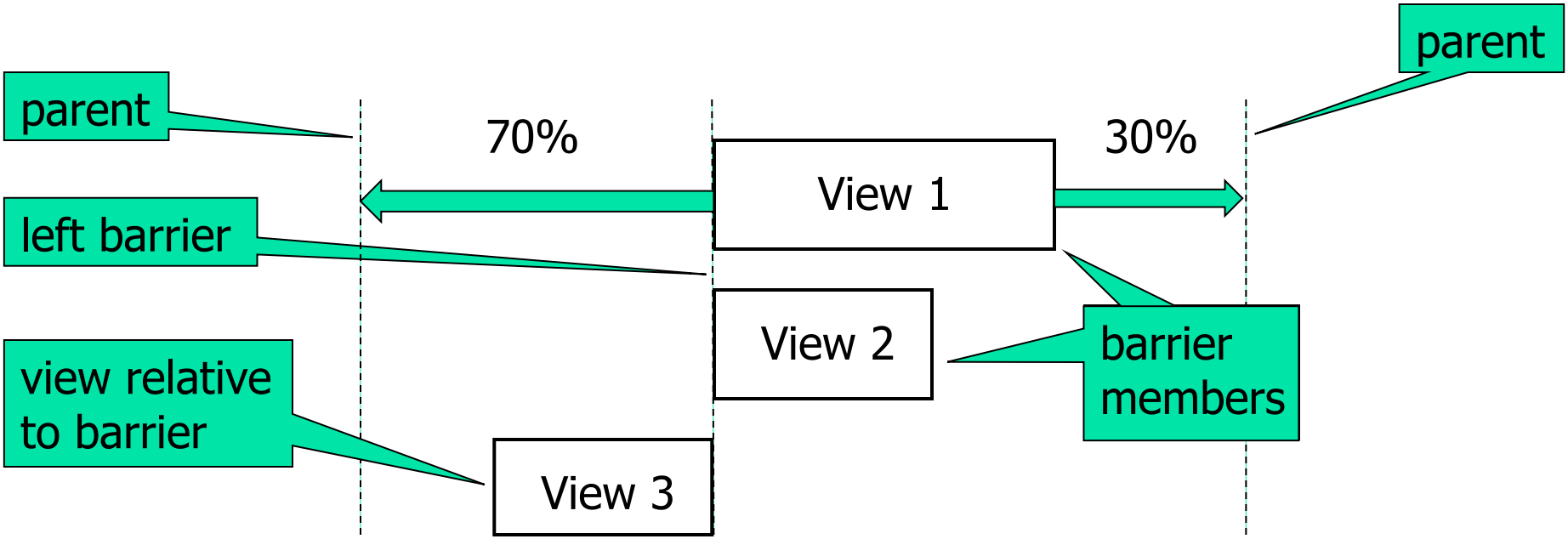
Additional constraint types

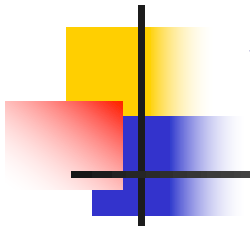
- For example, 2 Views (1 and 2) belong to a left barrier (the barrier aligns on the left of the most dominant View, i.e., View 1). View 3 is positioned relative to the barrier. The vertical order is 1, 2, 3 from top.



Additional constraint types

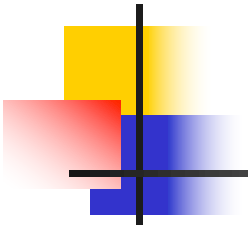
- Now, the dominant View 1 was repositioned (its bias has changed) and so the barrier moved, too. As a result, the dependent View 3 repositioned, as well.





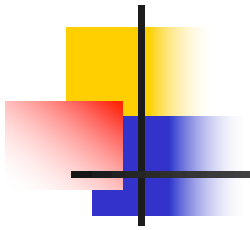
View Sizing

- A View’s size can be adjusted as:
  - fixed: The view has a specific dimension, usually expressed in dp units.
  - wrap content: The view expands only as much as needed to fit its contents.
  - match constraint: The view expands as much as possible to meet the constraints on each side (after accounting for the view's margins).



Chains

- A linear group of Views can form a chain, where the views are linked to each other with bi-directional ordering constraints.
- A chain controls the relative positioning of views included in the chain.
- A chain can be either vertical or horizontal.



Chains

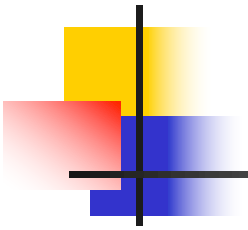
- To create a chain, select the views to be included in the chain, right-click, and then select:

  Chains -> Create Vertical Chain

  or

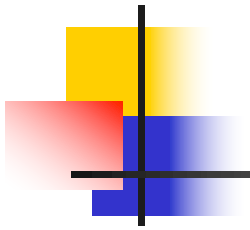
  Chains -> Create Horizontal Chain

- The head View (topmost, in a vertical chain, or leftmost, in a horizontal chain) controls the chain style.



Chains

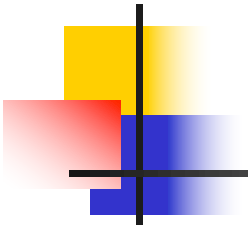
- A chain can be:
  - spread: the views are evenly distributed
  - packed: the views are packed (closely) together
  - spread inside: the first and last view are affixed to the constraints on each end of the chain and the rest are evenly distributed.
  - weighted: either spread or spread inside; fill the remaining space by setting one or more views to "match constraints" (0dp).
- A view can be included in a horizontal chain and in a vertical chain, at the same time.



More on the ConstraintLayout

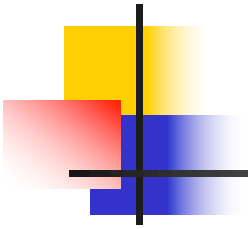
Build a Responsive UI with ConstraintLayout

https://developer.android.com/training/constraint-layout



Summary

- We have learned how to create layouts using XML resources.
- We have learned how to create layouts programmatically.
- We have explored many useful ViewGroups and ViewGroup subclass attributes.
- We are now able to define the LinearLayout, FrameLayout, RelativeLayout, TableLayout, and GridLayout.



Summary

- We have learned how to place multiple layouts on a screen.
- We have learned about the AdapterView controls and how to populate them with data.
- We have learned how to implement scrolling using the ScrollView control.
- We have learned how to use ConstraintLayout.