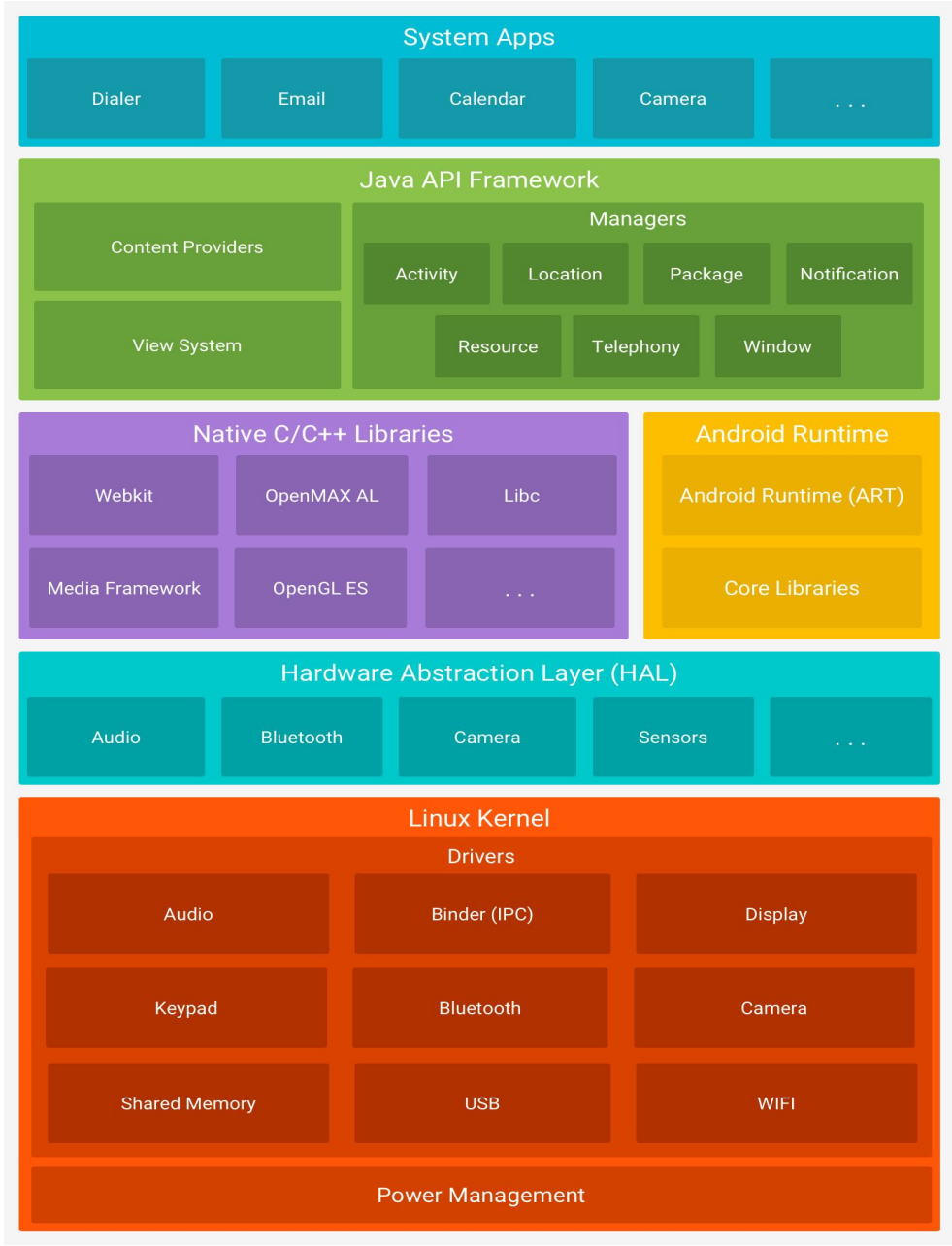
Android Versions

Code name	Version numbers	API level	Release date
No codename	1.0	1	September 23, 2008
No codename	1.1	2	February 9, 2009
Cupcake	1.5	3	April 27, 2009
Donut	1.6	4	September 15, 2009
Eclair	2.0 - 2.1	5 - 7	October 26, 2009
Froyo	2.2 - 2.2.3	8	May 20, 2010
Gingerbread	2.3 - 2.3.7	9 - 10	December 6, 2010
Honeycomb	3.0 - 3.2.6	11 - 13	February 22, 2011
Ice Cream Sandwich	4.0 - 4.0.4	14 - 15	October 18, 2011
Jelly Bean	4.1 - 4.3.1	16 - 18	July 9, 2012
KitKat	4.4 - 4.4.4	19 - 20	October 31, 2013
Lollipop	5.0 - 5.1.1	21- 22	November 12, 2014
Marshmallow	6.0 - 6.0.1	23	October 5, 2015
Nougat	7.0	24	August 22, 2016
Nougat	7.1.0 - 7.1.2	25	October 4, 2016
Oreo	8.0	26	August 21, 2017
Oreo	8.1	27	December 5, 2017
Pie	9.0	28	August 6, 2018
Android 10	10.0	29	September 3, 2019
Android 11	11	30	September 8, 2020

https://en.wikipedia.org/wiki/Android_version_history

Android Architecture



https://developer.android.com/guide/platform

...

The Android framework includes the following key services:

- **Activity Manager** – Controls all aspects of the application lifecycle and activity stack.
- **Content Providers** – Allows applications to publish and share data with other applications.
- **Resource Manager** – Provides access to non-code embedded resources such as strings, color settings and user interface layouts.
- **Notifications Manager** – Allows applications to display alerts and notifications to the user.
- **View System** – An extensible set of views used to create application user interfaces.
- **Package Manager** – The system by which applications are able to find out information about other applications currently installed on the device.
- **Telephony Manager** – Provides information to the application about the telephony services available on the device such as status and subscriber information.
- **Location Manager** – Provides access to the location services allowing an application to receive updates about location changes.

...

...

App components

App components are the essential building blocks of an Android app. Each component is an entry point through which the system or a user can enter your app. Some components depend on others.

There are four different types of app components:

- Activities
- Services
- Broadcast receivers
- Content providers

Each type serves a distinct purpose and has a distinct lifecycle that defines how the component is created and destroyed. The following sections describe the four types of app components.

https://developer.android.com/guide/components/fundamentals

...

Activities

An *activity* is the entry point for interacting with the user. It represents a single screen with a user interface. For example, an email app might have one activity that shows a list of new emails, another activity to compose an email, and another activity for reading emails. Although the activities work together to form a cohesive user experience in the email app, each one is independent of the others. As such, a different app can start any one of these activities if the email app allows it. For example, a camera app can start the activity in the email app that composes new mail to allow the user to share a picture. An activity facilitates the following key interactions between system and app:

- Keeping track of what the user currently cares about (what is on screen) to ensure that the system keeps running the process that is hosting the activity.
- Knowing that previously used processes contain things the user may return to (stopped activities), and thus more highly prioritize keeping those processes around.
- Helping the app handle having its process killed so the user can return to activities with their previous state restored.
- Providing a way for apps to implement user flows between each other, and for the system to coordinate these flows. (The most classic example here being share.)

You implement an activity as a subclass of the `Activity` class. For more information about the `Activity` class, see the Activities developer guide.

...

...

Services

A *service* is a general-purpose entry point for keeping an app running in the background for all kinds of reasons. It is a component that runs in the background to perform long-running operations or to perform work for remote processes. A service does not provide a user interface. For example, a service might play music in the background while the user is in a different app, or it might fetch data over the network without blocking user interaction with an activity. Another component, such as an activity, can start the service and let it run or bind to it in order to interact with it. There are actually two very distinct semantics services tell the system about how to manage an app: Started services tell the system to keep them running until their work is completed. This could be to sync some data in the background or play music even after the user leaves the app. Syncing data in the background or playing music also represent two different types of started services that modify how the system handles them:

- Music playback is something the user is directly aware of, so the app tells the system this by saying it wants to be foreground with a notification to tell the user about it; in this case the system knows that it should try really hard to keep that service's process running, because the user will be unhappy if it goes away.
- A regular background service is not something the user is directly aware as running, so the system has more freedom in managing its process. It may allow it to be killed (and then restarting the service sometime later) if it needs RAM for things that are of more immediate concern to the user.

...

...

Broadcast receivers

A *broadcast receiver* is a component that enables the system to deliver events to the app outside of a regular user flow, allowing the app to respond to system-wide broadcast announcements. Because broadcast receivers are another well-defined entry into the app, the system can deliver broadcasts even to apps that aren't currently running. So, for example, an app can schedule an alarm to post a notification to tell the user about an upcoming event... and by delivering that alarm to a BroadcastReceiver of the app, there is no need for the app to remain running until the alarm goes off. Many broadcasts originate from the system—for example, a broadcast announcing that the screen has turned off, the battery is low, or a picture was captured. Apps can also initiate broadcasts—for example, to let other apps know that some data has been downloaded to the device and is available for them to use. Although broadcast receivers don't display a user interface, they may create a status bar notification to alert the user when a broadcast event occurs. More commonly, though, a broadcast receiver is just a *gateway* to other components and is intended to do a very minimal amount of work. For instance, it might schedule a `JobService` to perform some work based on the event with `JobScheduler`

A broadcast receiver is implemented as a subclass of `BroadcastReceiver` and each broadcast is delivered as an `Intent` object. For more information, see the `BroadcastReceiver` class.

...

...

Content providers

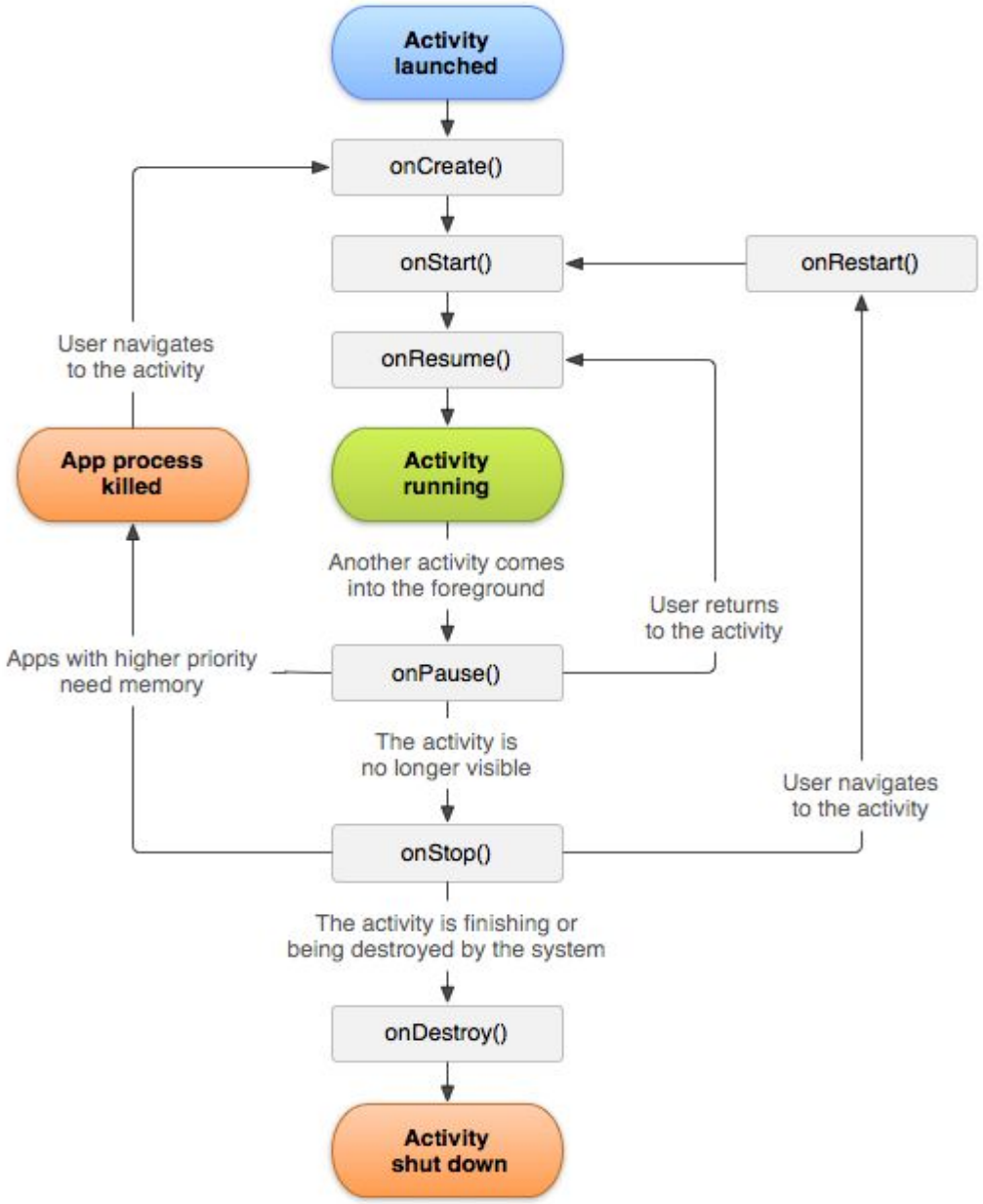
A *content provider* manages a shared set of app data that you can store in the file system, in a SQLite database, on the web, or on any other persistent storage location that your app can access. Through the content provider, other apps can query or modify the data if the content provider allows it. For example, the Android system provides a content provider that manages the user's contact information. As such, any app with the proper permissions can query the content provider, such as `ContactsContract.Data`, to read and write information about a particular person. It is tempting to think of a content provider as an abstraction on a database, because there is a lot of API and support built in to them for that common case. However, they have a different core purpose from a system-design perspective. To the system, a content provider is an entry point into an app for publishing named data items, identified by a URI scheme. Thus an app can decide how it wants to map the data it contains to a URI namespace, handing out those URIs to other entities which can in turn use them to access the data. There are a few particular things this allows the system to do in managing an app:

- Assigning a URI doesn't require that the app remain running, so URIs can persist after their owning apps have exited. The system only needs to make sure that an owning app is still running when it has to retrieve the app's data from the corresponding URI.
- These URIs also provide an important fine-grained security model. For example, an app can place the URI for an image it has on the clipboard, but leave its content provider locked up so that other apps cannot freely access it. When a second app attempts to access that URI on the clipboard, the system can allow that app to access the data via a temporary *URI permission grant* so that it is allowed to access the data only behind that URI, but nothing else in the second app.

Content providers are also useful for reading and writing data that is private to your app and not shared.

...

Activity Lifecycle



Method	Description
onCreate	called when activity is first created.
onStart	called when activity is becoming visible to the user.
onResume	called when activity will start interacting with the user.
onPause	called when activity is not visible to the user.
onStop	called when activity is no longer visible to the user.
onRestart	called after your activity is stopped, prior to start.
onDestroy	called before the activity is destroyed.

https://developer.android.com/reference/android/app/Activity

Log Methods

`Log` provides methods that correspond to different level of priority (importance) of the messages being recorded. From low to high priority:

- `Log.v()` : VERBOSE output. This is the most detailed, for everyday messages. This is often the go-to, default level for logging.

  Ideally, `Log.v()` calls should only be compiled into an application during development, and removed for production versions.

- `Log.d()` : DEBUG output. This is intended for lower-level, less detailed messages (but still code-level, that is referring to specific programming messages).

  These messages can be compiled into the code but are removed at runtime in production builds through Gradle.

- `Log.i()` : INFO output. This is intended for "high-level" information, such at the user level (rather than specifics about code)

- `Log.w()` : WARN output. For warnings

- `Log.e()` : ERROR output. For errors

- Also if you look at the API... `Log.wtf()` !

...

Resources Types

- `res/drawable/` : contains graphics (PNG, JPEG, etc)
- `res/layout/` : contains UI XML layout files
- `res/mipmap/` : conatins launcher icon files in different resolutions
- `res/values/` : contains XML definitions for general constants
  - `/strings` : short string constants (e.g., labels)
  - `/colors` : color constants
  - `/styles` : constants for style and theme details
  - `/dimen` : dimensional constants (like default margins); not created by default in Android Studio 2.3+.

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <string name="app_name">Mon Application</string>
</resources>
```

...

Resources Tags

Path's	Tag's
res/values/strings.xml	<plurals>
res/values/strings.xml	<string>
res/values/strings.xml	<string-array>
res/values/arrays.xml	<string-array>
res/values/bools.xml	<bool>
res/values/colors.xml	<color>
res/values/styles.xml	<style>
res/values/themes.xml	<style>
res/values/dimens.xml	<dimen>
res/values/ids.xml	<item>
res/values/integers.xml	<integer>
res/values/integers.xml	<integer-array>
res/color/	<selector>
res/menu/	<menu>
res/xml/	
res/drawable/	
res/drawable/	<animation-list>
res/animator/	<set>, <objectAnimator>, <valueAnimator>
res/anim/	<set>, <alpha>, <rotate>, <scale>, <translate>
res/raw/	
res/layout/	

Layouts

As mentioned above, a Layout is a grouping of Views (specifically, a `ViewGroup`). A Layout acts as a container for other Views, to help organize things. Layouts are all subclasses of `ViewGroup`, so you can use its inheritance documentation to see a (mostly) complete list of options, though many of the listed classes are deprecated in favor of later, more generic/powerful options.

An Android layout is a class that handles arranging the way its children appear on the screen. Anything that is a View (or inherits from **View**) can be a child of a layout. All of the layouts inherit from **ViewGroup** (which inherits from **View**) so you can nest layouts. You could also create your own custom layout by making a class that inherits from **ViewGroup**.

...

View Properties

Before we get into how to group Views, let's focus on the individual, basic `View` classes. As an example, consider the `activity_main` layout in the lecture code. This layout contains two individual `View` elements (inside a `Layout`): a `TextView` and a `Button`.

All View have **properties** which define the state of the View. Properties are usually defined within the resource XML as element *attributes*. Some examples of these property attributes are described below.

- `android:id` specifies a unique identifier for the View. This identifier needs to be unique within the layout, though ideally is unique within the entire app (for clarity).

  Identifiers must be legal Java variable names (because they are turned into a variable name in the `R` class), and by convention are named in `lower_case` format.

  - *Style tip*: it is useful to prefix each View's id with its type (e.g., `btn`, `txt`, `edt`). This helps with making the code self-documenting.

  You should give each interactive `View` a unique id, which will allow its state to automatically be saved as a `Bundle` when the Activity is destroyed. See here for details.

...

layout_width and layout_height

- `android:layout_width` and `android:layout_height` are used to specify the View's size on the screen (see ViewGroup.LayoutParams for documentation). These values can be a specific value (e.g., `12dp`), but more commonly is one of two special values:
  - `wrap_content`, meaning the dimension should be as large as the content requires, plus padding.
  - `match_parent`, meaning the dimension should be as large as the *parent* (container) element, minus padding. This value was renamed from `fill_parent` (which has now been deprecated).

...

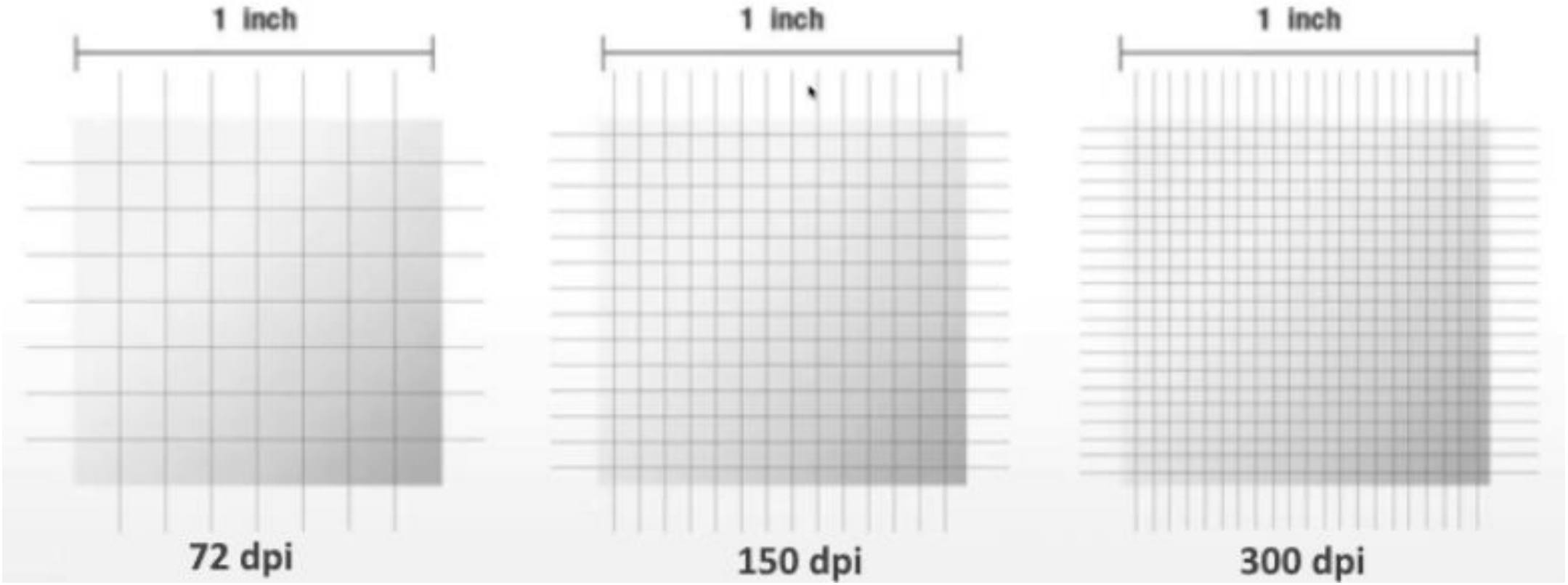
Dimension Units

Android utilizes the following dimensions or units:

- `dp` is a "density-independent pixel". On a 160-dpi (dots-per-inch) screen, `1dp` equals `1px` (pixel). But as dpi increases, the number of pixels per `dp` increases. These values should be used instead of `px`, as it allows dimensions to work independent of the hardware's dpi (which is *highly* variable).
- `px` is an actual screen pixel. *DO NOT USE THIS* (use `dp` instead!)
- `sp` is a "scale-independent pixel". This value is like `dp`, but is scale by the system's font preference (e.g., if the user has selected that the device should display in a larger font, `1sp` will cover more `dp`). *You should **always** use `sp` for text dimensions, in order to support user preferences and accessibility.*
- `pt` is 1/72 of an inch of the physical screen. Similar units `mm` and `in` are available. *Not recommended for use.*

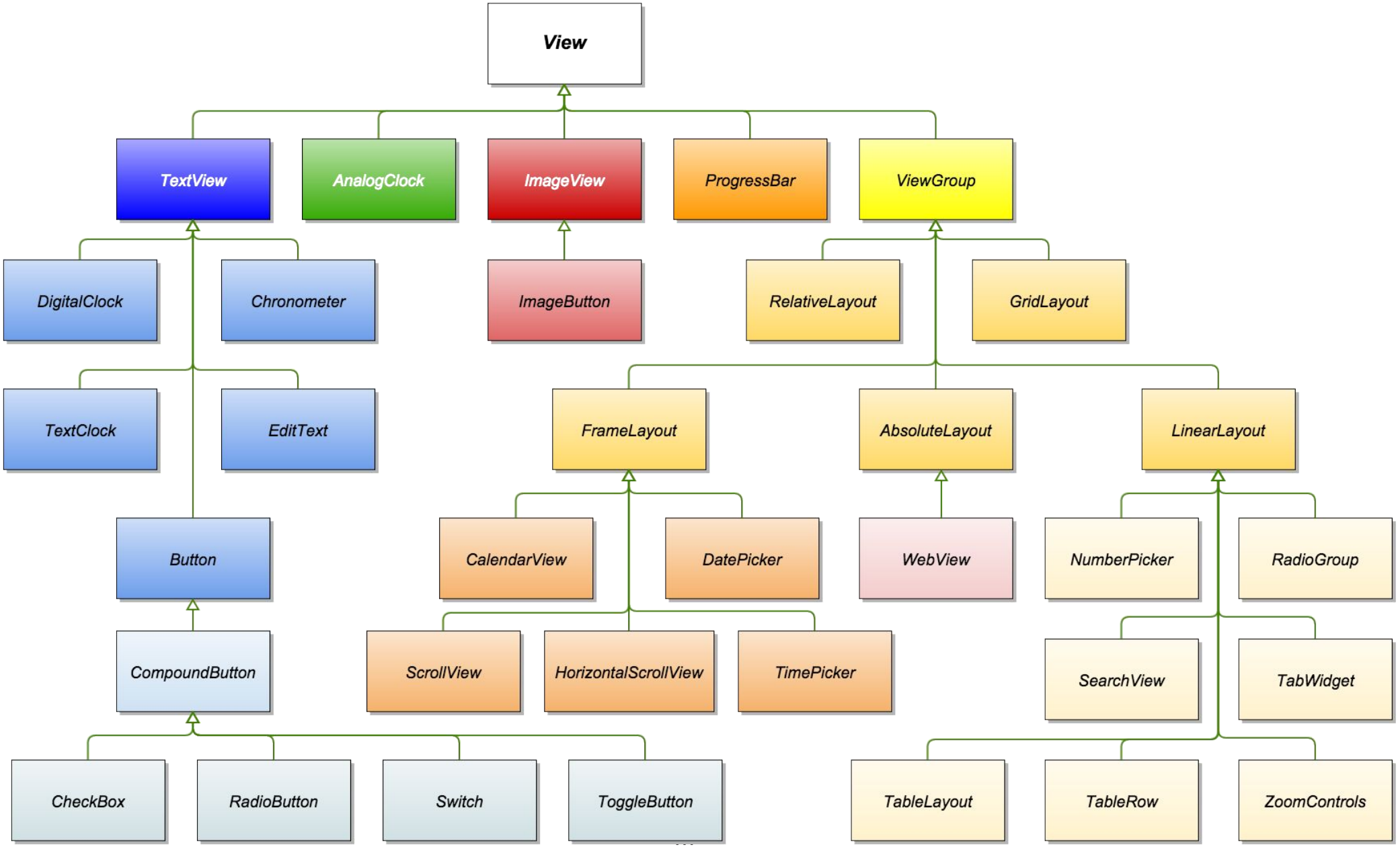
...

Dimension Units

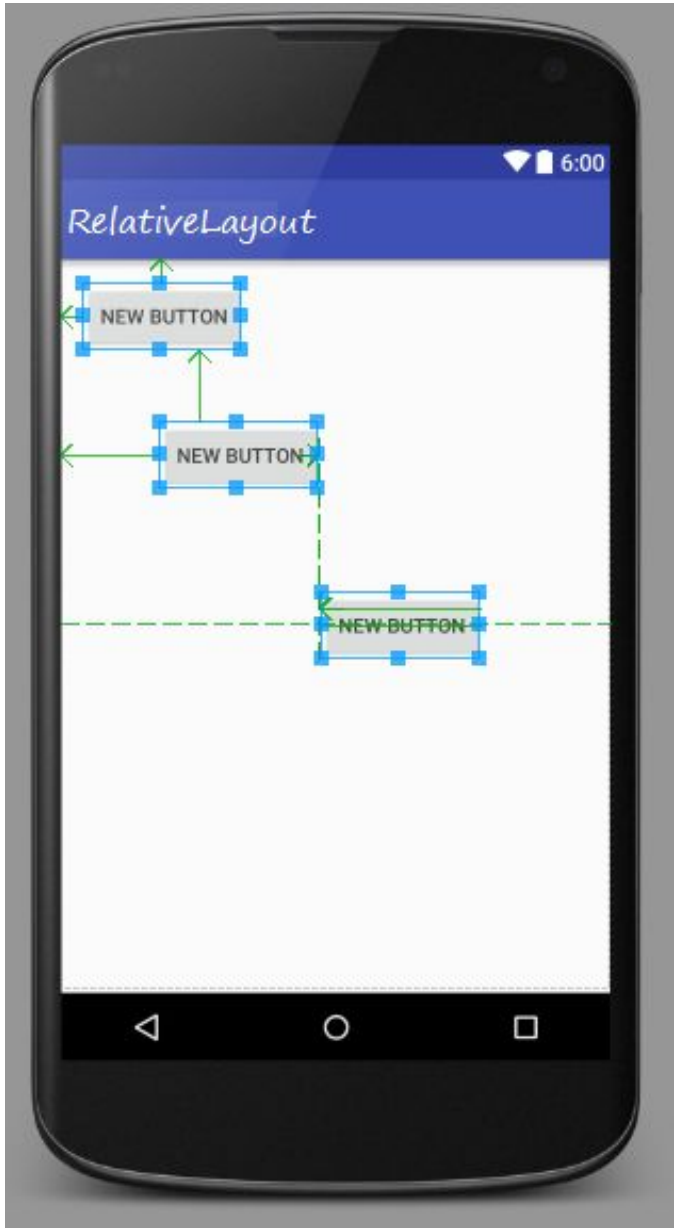


...

View Classes



RelativeLayout



RelativeLayout is a view group that displays child views in relative positions. The position of each view can be specified as relative to sibling elements (such as to the left-of or below another view) or in positions relative to the parent **RelativeLayout** area (such as aligned to the bottom, left or center).

...

RelativeLayout Properties

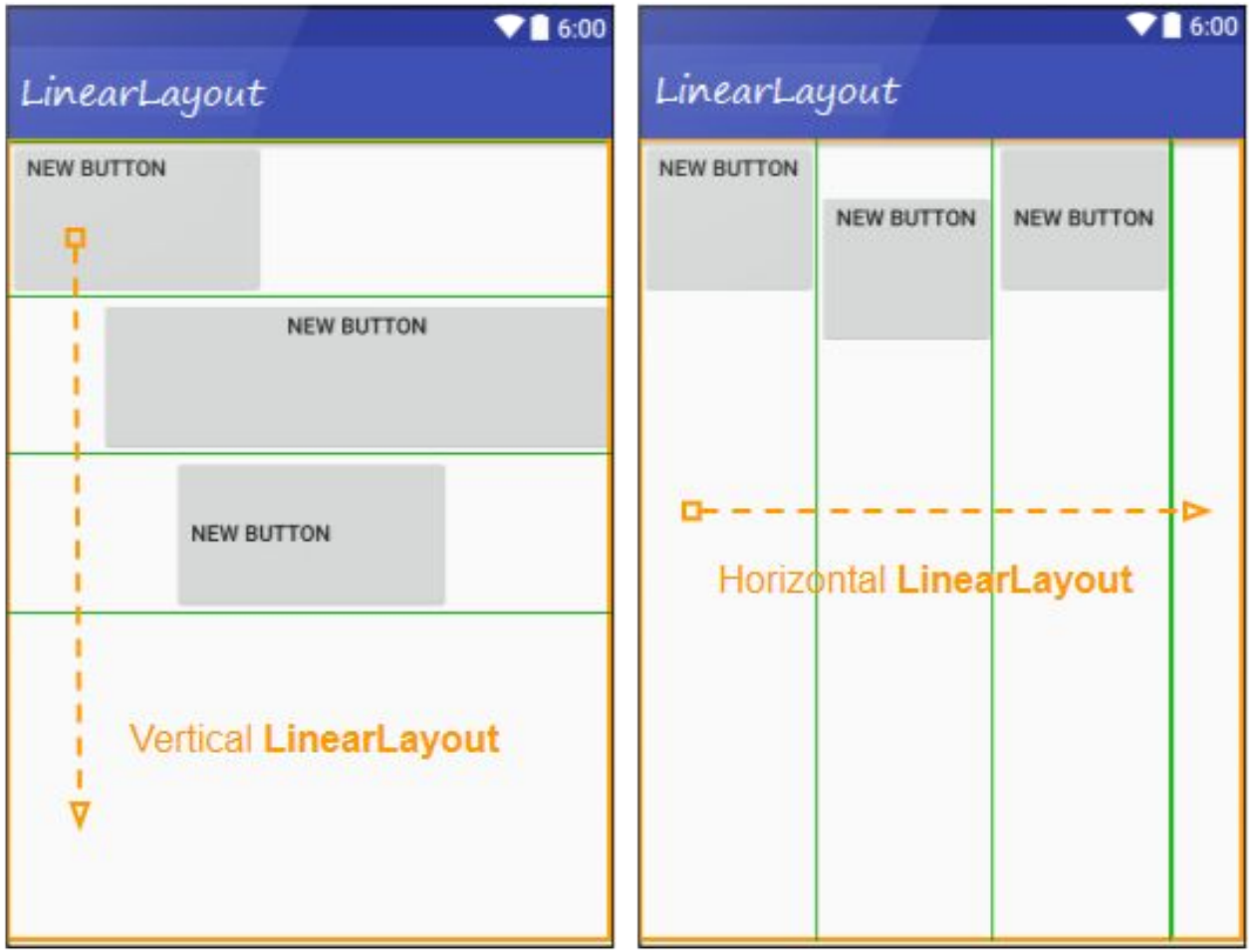
android:layout_above
android:layout_below
android:layout_toLeftOf
android:layout_toRightOf
android:layout_toStartOf
android:layout_toEndOf

android:layout_alignBottom
android:layout_alignLeft
android:layout_alignRight
android:layout_alignStart
android:layout_alignEnd
android:layout_alignTop
android:layout_alignBaseline

android:layout_alignParentBottom
android:layout_alignParentRight
android:layout_alignParentLeft
android:layout_alignParentStart
android:layout_alignParentEnd
android:layout_alignParentTop
android:layout_centerInParent
android:layout_centerHorizontal
android:layout_centerVertical

...

LinearLayout



LinearLayout is a ViewGroup that arranges the child View(s) in a single direction, either vertically or horizontally.

```
1  <!-- Horizontal LinearLayout (Default) -->
2  <LinearLayout
3     ...
4     android:orientation="horizontal">
5
6     ...
7  </LinearLayout>
8
9
10 <!-- Vertical LinearLayout -->
11 <LinearLayout
12    ...
13    android:orientation="vertical">
14
15    ...
16 </LinearLayout>
```

...

LinearLayout Properties

android:orientation = "vertical"|"horizontal"

android:weightSum = "10“

android:layout_weight = "1"

android:layout_gravity = "top"|"bottom"|"left"|"right"|

"center_vertical"|"center_horizontal"|"center"|

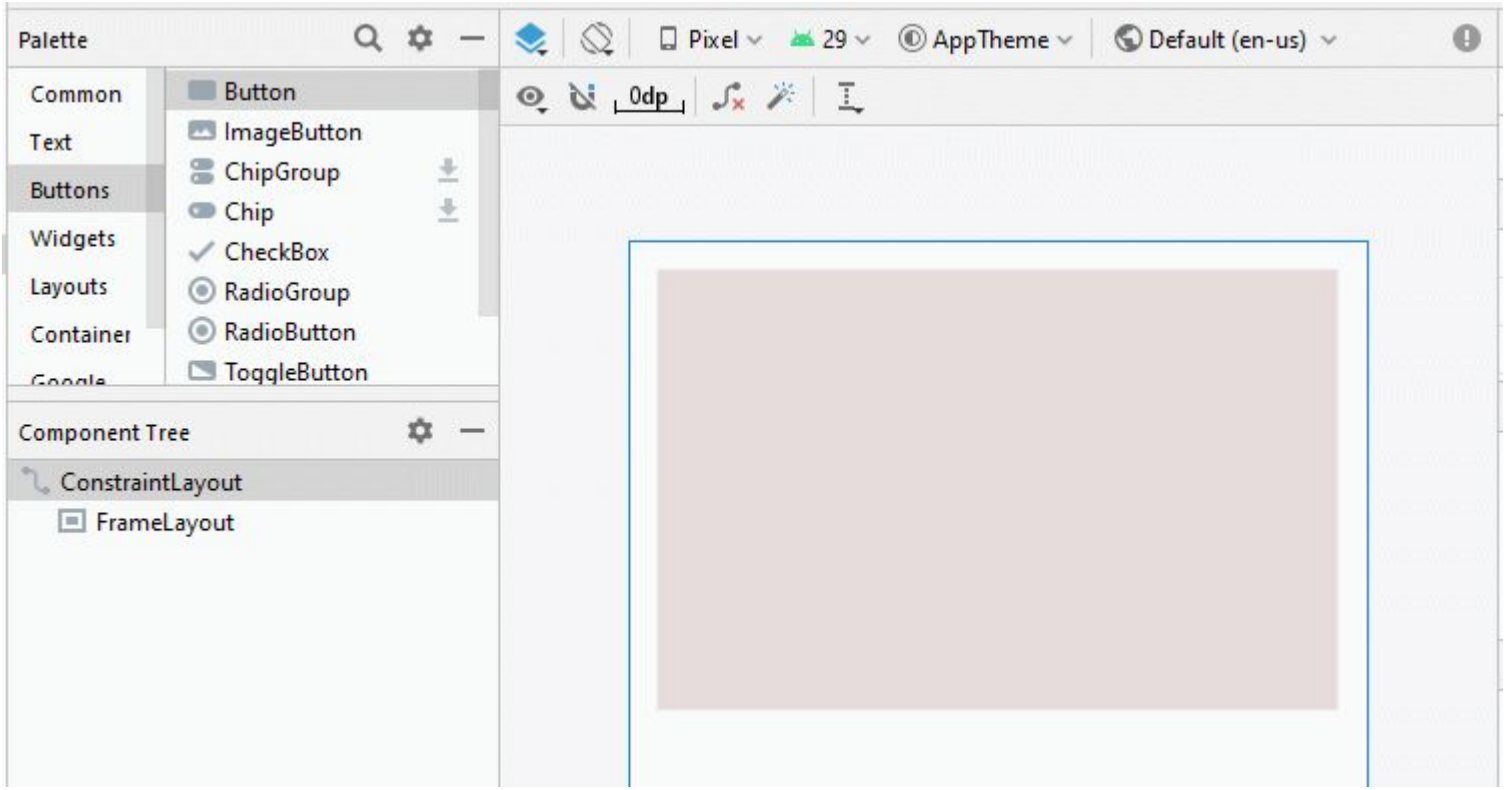
"fill_vertical"|"fill_horizontal"|"fill"|

"clip_vertical"|"clip_horizontal"|

"start"|"end"

...

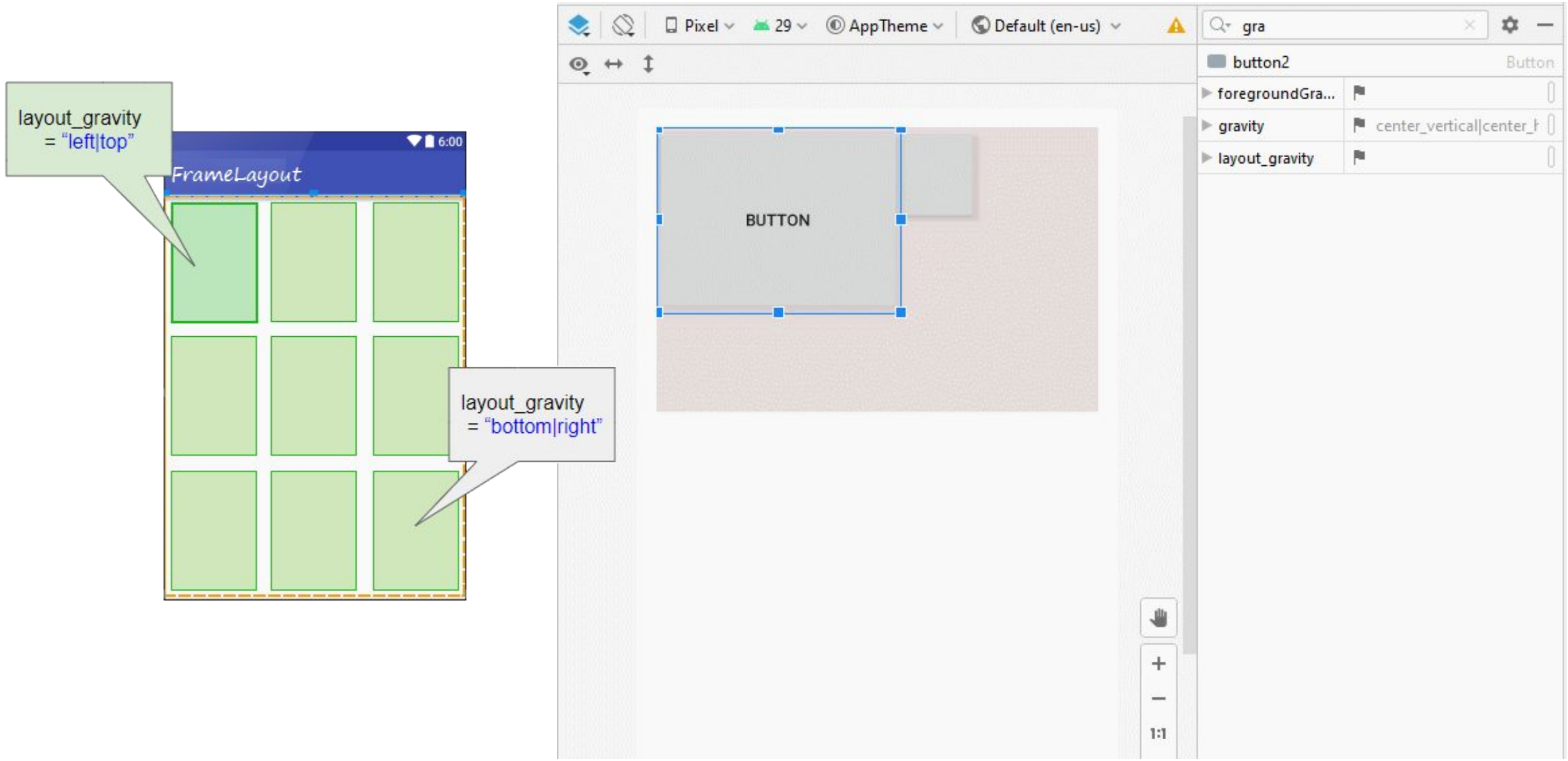
FrameLayout



FrameLayout is a simple layout. It can contain one or more child **View**(s), and they can overlap each other. Therefore, the **android:layout_gravity** attribute is used to locate the child **View**(s).

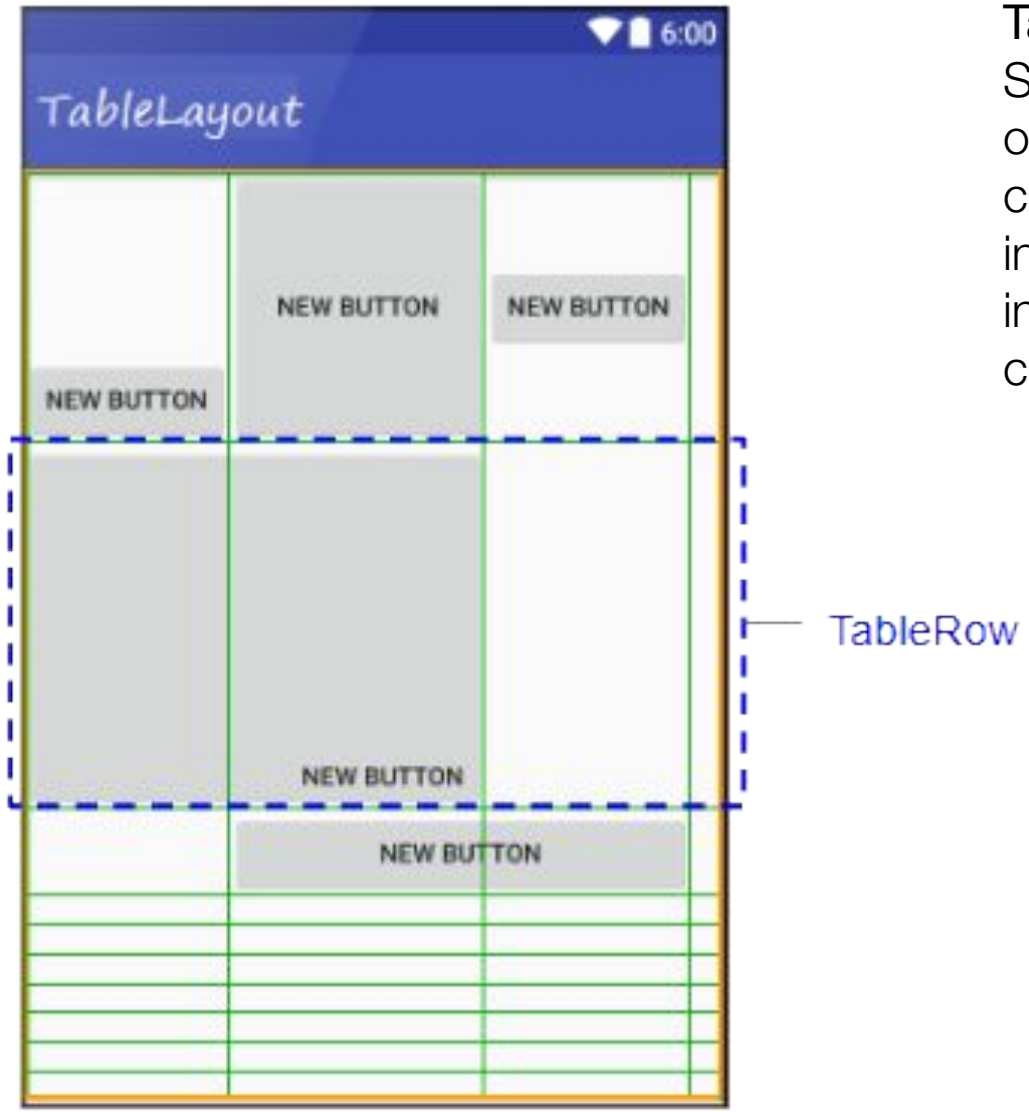
...

android:layout_gravity



...

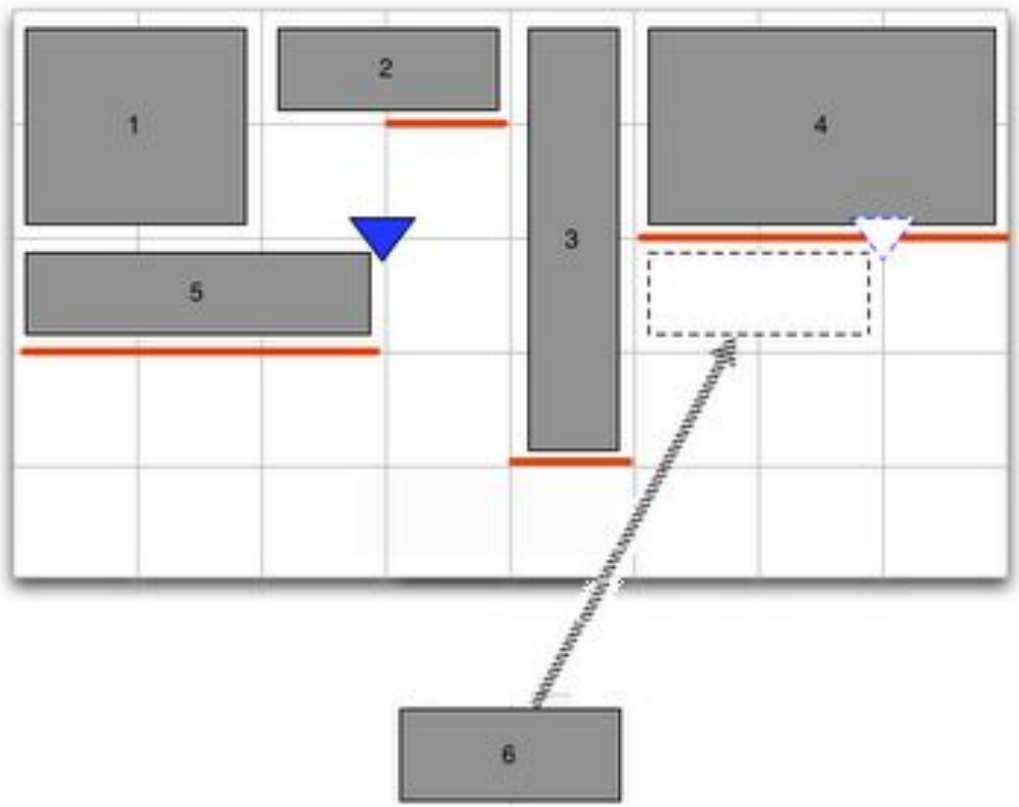
TableLayout



TableLayout arranges the View(s) in table format. Specifically, TableLayout is a ViewGroup containing one or more TableRow(s). Each TableRow is a row in the table containing cells. Child View(s) can be placed in one cell or in a merged cell from adjacent cells of a row. Unlike tables in HTML, you cannot merge consecutive cells in the one column.

...

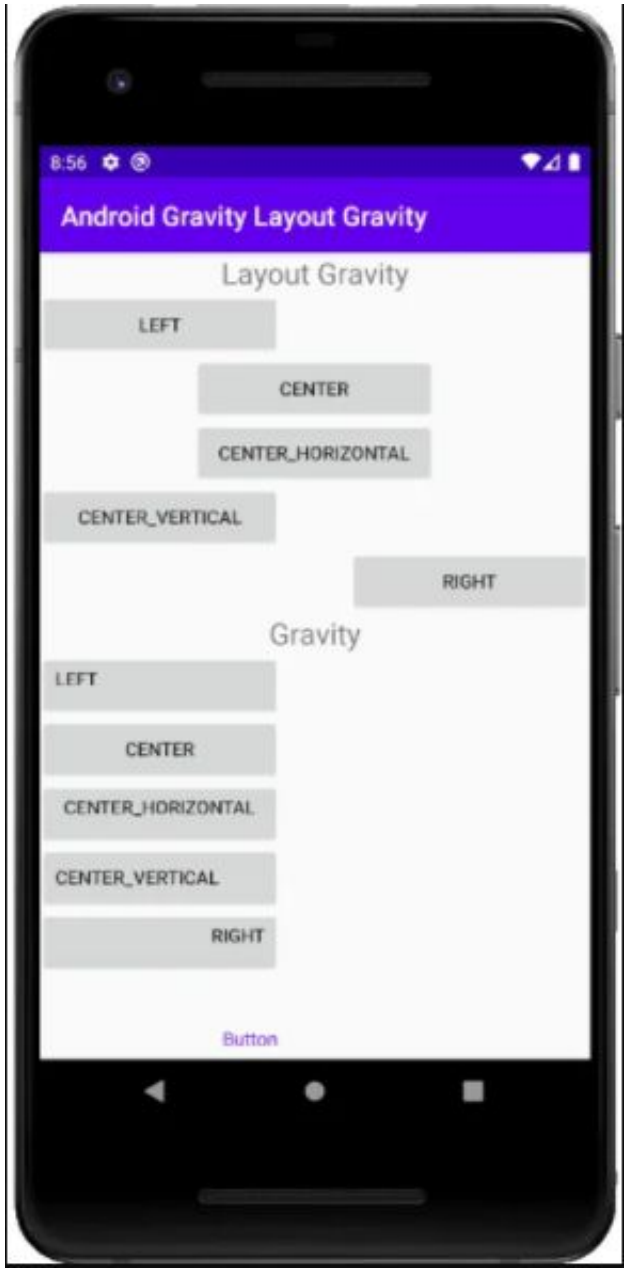
GridLayout

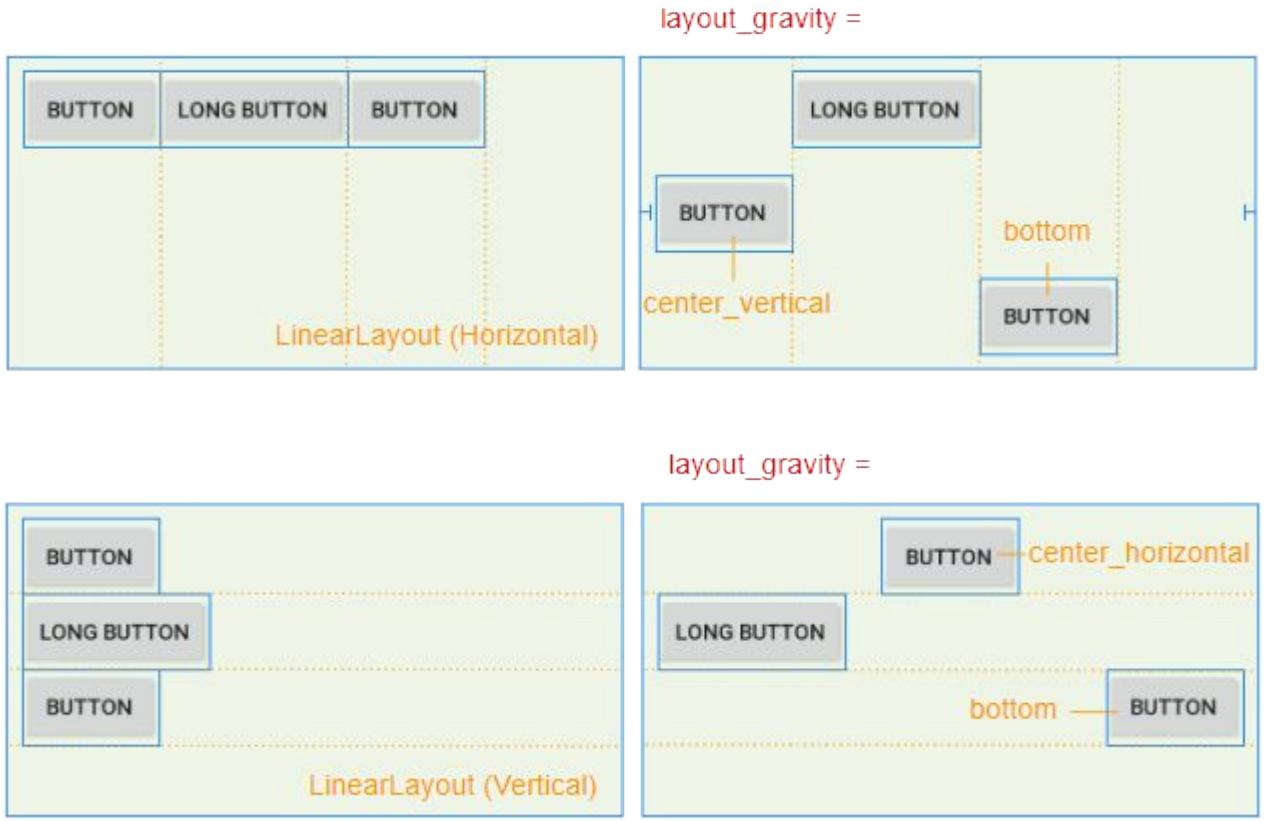


GridLayout uses a grid of infinitely-thin lines to separate its drawing area into: rows, columns, and cells. It supports both row and column spanning, this means it is possible to merge adjacent cells into a large cell (a rectangle) to contain a **View**.

...

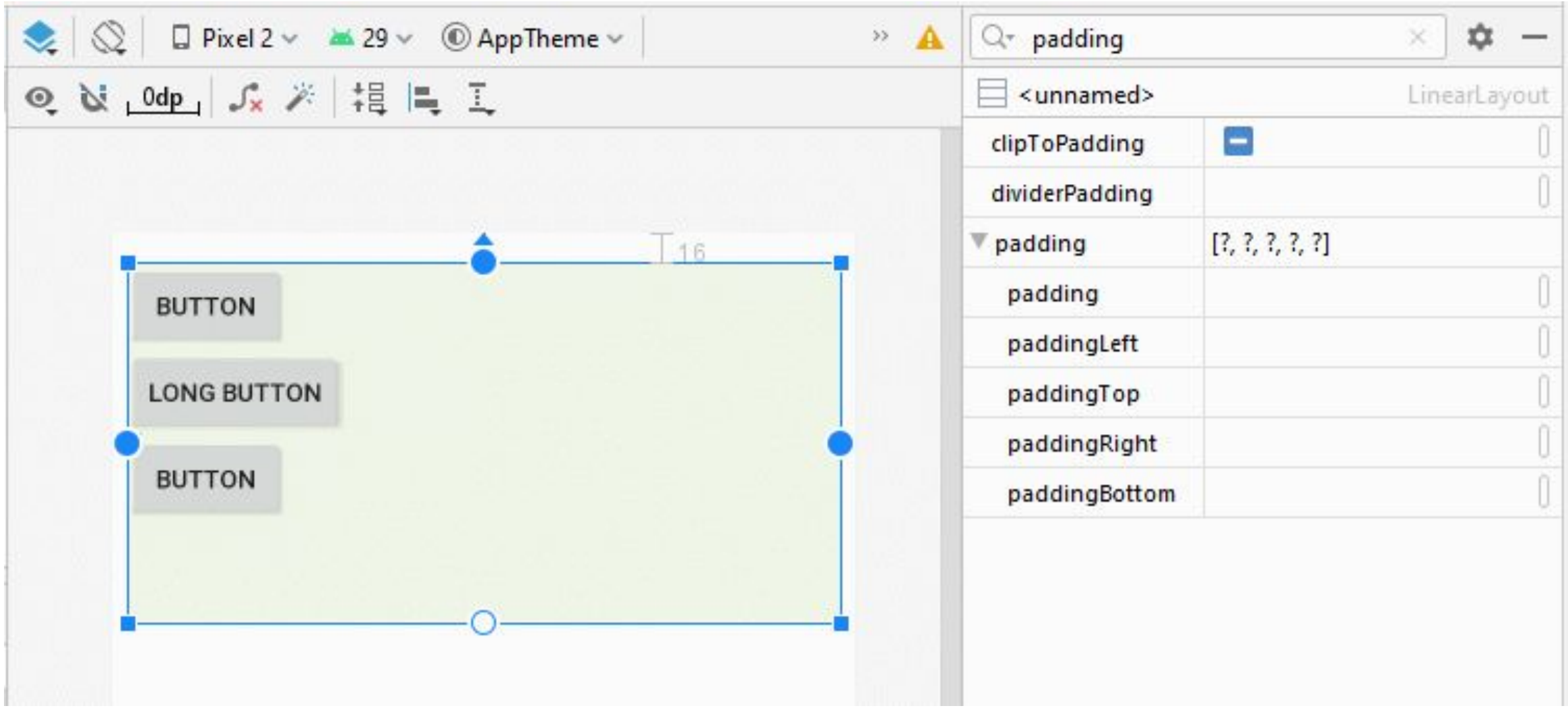
Gravity and Layout_Gravity





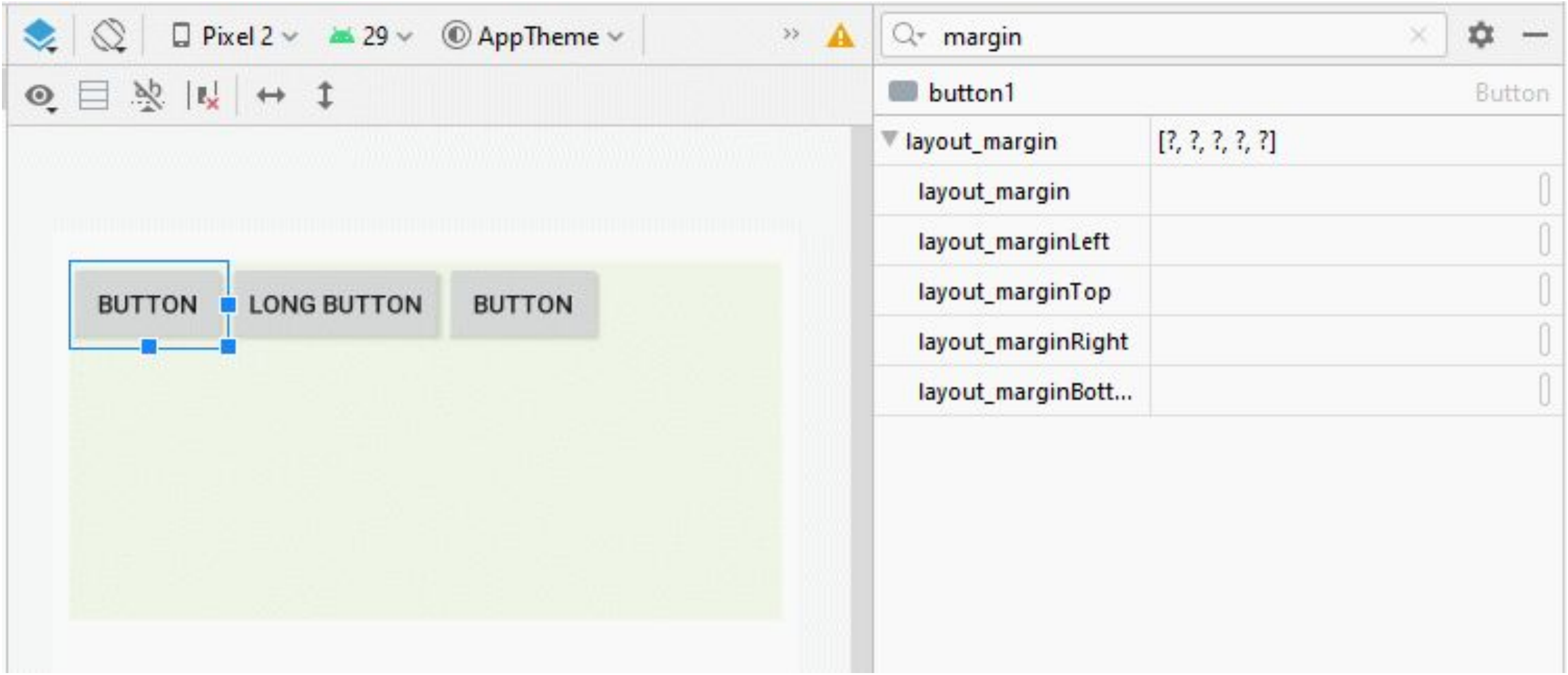
https://learntodroid.com/what-is-the-difference-between-gravity-and-layout_gravity-in-android/

Padding


Pixel 2
29
AppTheme
0dp
16
BUTTON
LONG BUTTON
BUTTON
padding
<unnamed>
LinearLayout
clipToPadding
dividerPadding
padding
[?, ?, ?, ?, ?]
padding
paddingLeft
paddingTop
paddingRight
paddingBottom

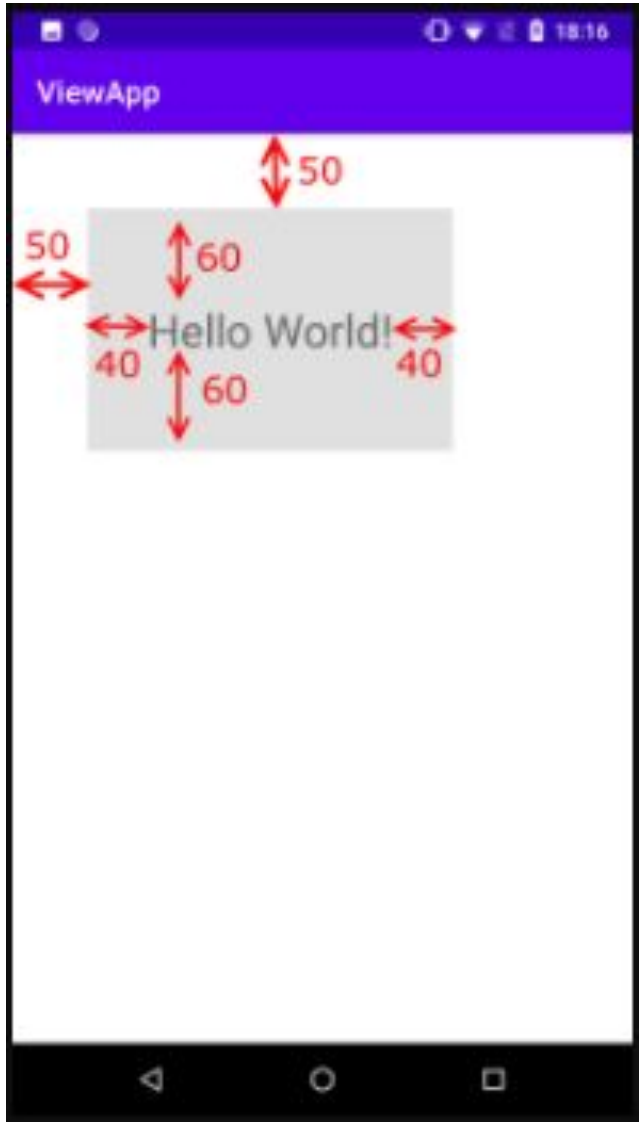
...

Margins


Pixel 2
29
AppTheme
margin
button1
Button
layout_margin
[?, ?, ?, ?, ?]
layout_margin
layout_marginLeft
layout_marginTop
layout_marginRight
layout_marginBott...
BUTTON
LONG BUTTON
BUTTON

...

Example




```
5     xmlns:tools="http://schemas.android.com/tools"
6     android:layout_width="match_parent"
7     android:layout_height="match_parent"
8     android:padding="50dp"
9     tools:context=".MainActivity">
10
11    <TextView
12        android:layout_height="wrap_content"
13        android:layout_width="wrap_content"
14        android:paddingTop="60dp"
15        android:paddingLeft="40dp"
16        android:paddingRight="40dp"
17        android:paddingBottom="60dp"
18        android:text="Hello World!"
19        android:textSize="30sp"
20        android:background="#e0e0e0"
21        app:layout_constraintLeft_toLeftOf="parent"
22        app:layout_constraintTop_toTopOf="parent"
23        />
24
```

...

ConstraintLayout

ConstraintLayout – Introduced in Android 7, use of this layout manager is recommended for most layout requirements. ConstraintLayout allows the positioning and behavior of the views in a layout to be defined by simple constraint settings assigned to each child view. The flexibility of this layout allows complex layouts to be quickly and easily created without the necessity to nest other layout types inside each other, resulting in improved layout performance. ConstraintLayout is also tightly integrated into the Android Studio Layout Editor tool. Unless otherwise stated, this is the layout of choice for the majority of examples in this book.

ConstraintLayout

```
<androidx.constraintlayout.widget.ConstraintLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".MainActivity">

    <TextView
        android:layout_width="0dp"
        android:layout_height="0dp"
        android:text="Hello World!"
        android:textSize="30sp"
        android:background="#e0e0e0"
        app:layout_constraintLeft_toLeftOf="parent"
        app:layout_constraintTop_toTopOf="parent"
        app:layout_constraintRight_toRightOf="parent"
        app:layout_constraintBottom_toBottomOf="parent"
        />

</androidx.constraintlayout.widget.ConstraintLayout>
```

...

ConstraintLayout Properties

app:layout_constraintDimensionRatio="1:0.5"
app:layout_constraintHorizontal_bias="0.5"
app:layout_constraintVertical_bias="0.5"

app:layout_constraintWidth_default="percent"
app:layout_constraintWidth_percent="0.5"
app:layout_constraintHeight_default="percent"
app:layout_constraintHeight_percent="0.5"

app:layout_constraintHorizontal_chainStyle = "spread"|"spread_inside"|"packed"
app:layout_constraintHorizontal_weight = "1"
app:layout_constraintVertical_chainStyle = "spread"|"spread_inside"|"packed"
app:layout_constraintVertical_weight = "1"

...

ConstraintLayout Properties2

layout_constraintLeft_toLeftOf

layout_constraintLeft_toRightOf

layout_constraintRight_toLeftOf

layout_constraintRight_toRightOf

layout_constraintTop_toTopOf

layout_constraintTop_toBottomOf

layout_constraintBottom_toBottomOf

layout_constraintBottom_toTopOf

layout_constraintBaseline_toBaselineOf

layout_constraintStart_toEndOf

layout_constraintStart_toStartOf

layout_constraintEnd_toStartOf

layout_constraintEnd_toEndOf

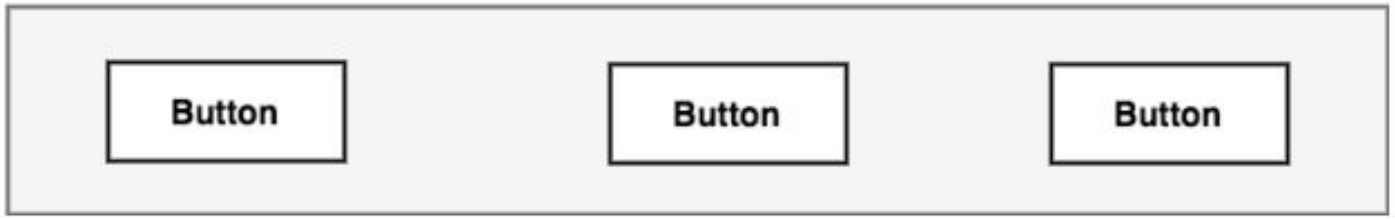
...

include

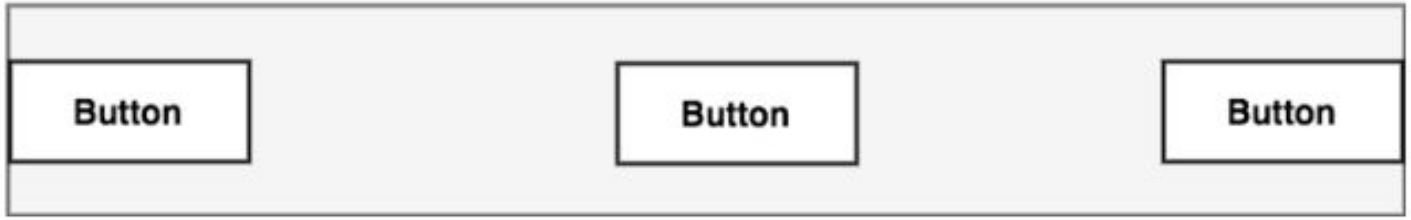
```xml
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.ConstraintLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:padding="16dp"
    tools:context=".MainActivity">

    <include
        android:id="@+id/textView"
        layout="@layout/text_panel"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        app:layout_constraintLeft_toLeftOf="parent"
        app:layout_constraintTop_toTopOf="parent"
        app:layout_constraintBottom_toTopOf="@+id/button"
        />
    <include
        android:id="@+id/button"
        layout="@layout/button_panel"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        app:layout_constraintLeft_toLeftOf="parent"
        app:layout_constraintTop_toBottomOf="@+id/textView"
        />

</androidx.constraintlayout.widget.ConstraintLayout>
```

ConstraintLayout ChainStyle

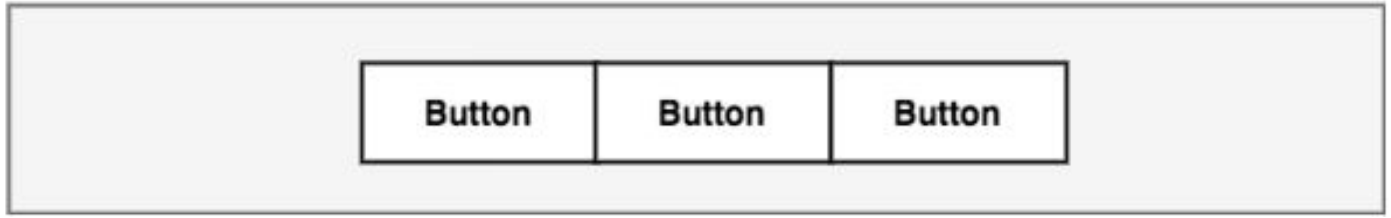
- **Spread Chain** – The widgets contained within the chain are distributed evenly across the available space. This is the default behavior for chains.



- **Spread Inside Chain** – The widgets contained within the chain are spread evenly between the chain head and the last widget in the chain. The head and last widgets are not included in the distribution of spacing.



- **Packed Chain** – The widgets that make up the chain are packed together without any spacing. A bias may be applied to control the horizontal or vertical positioning of the chain in relation to the parent container.



...

Android …

Android …