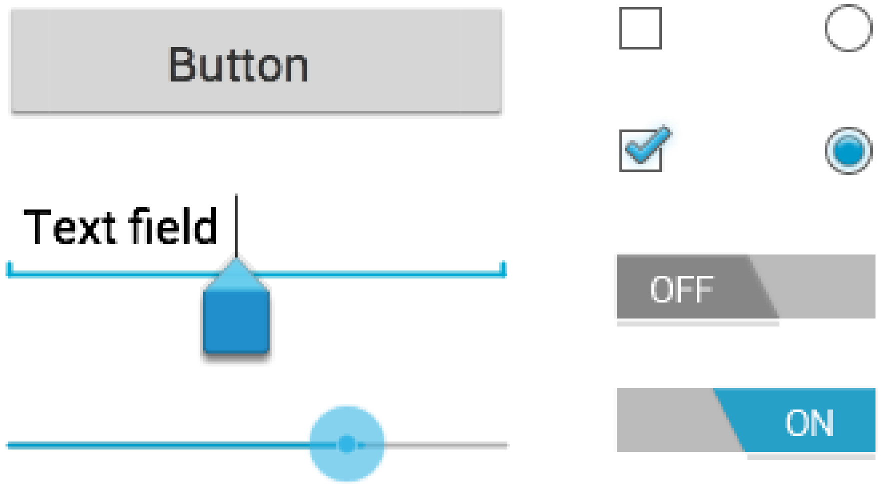
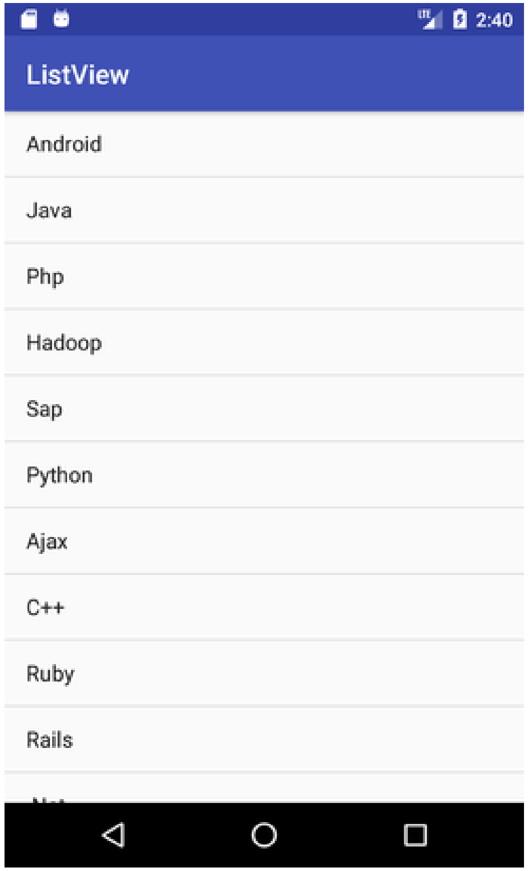
Android Components

Your app's user interface is everything that the user can see and interact with. Android provides a variety of pre-built UI components such as structured layout objects and UI controls that allow you to build the graphical user interface for your app. Android also provides other UI modules for special interfaces such as dialogs, notifications, and menus.

Example



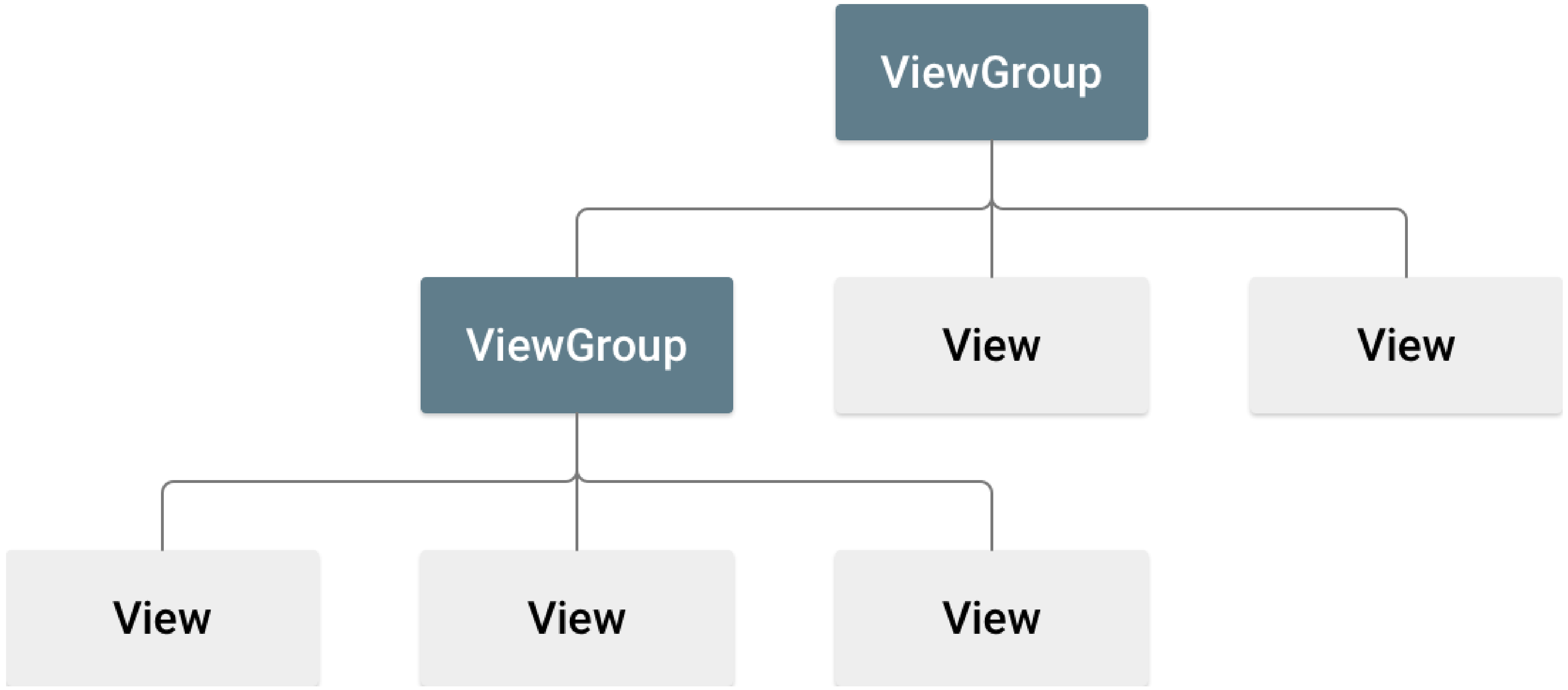


Android
Java
Php
Hadoop
Sap
Python
Ajax
C++
Ruby
Rails

Layouts

A layout defines the structure for a user interface in your app, such as in an activity. All elements in the layout are built using a hierarchy of View and ViewGroup objects. A View usually draws something the user can see and interact with. Whereas a ViewGroup is an invisible container that defines the layout structure for View and other ViewGroup objects

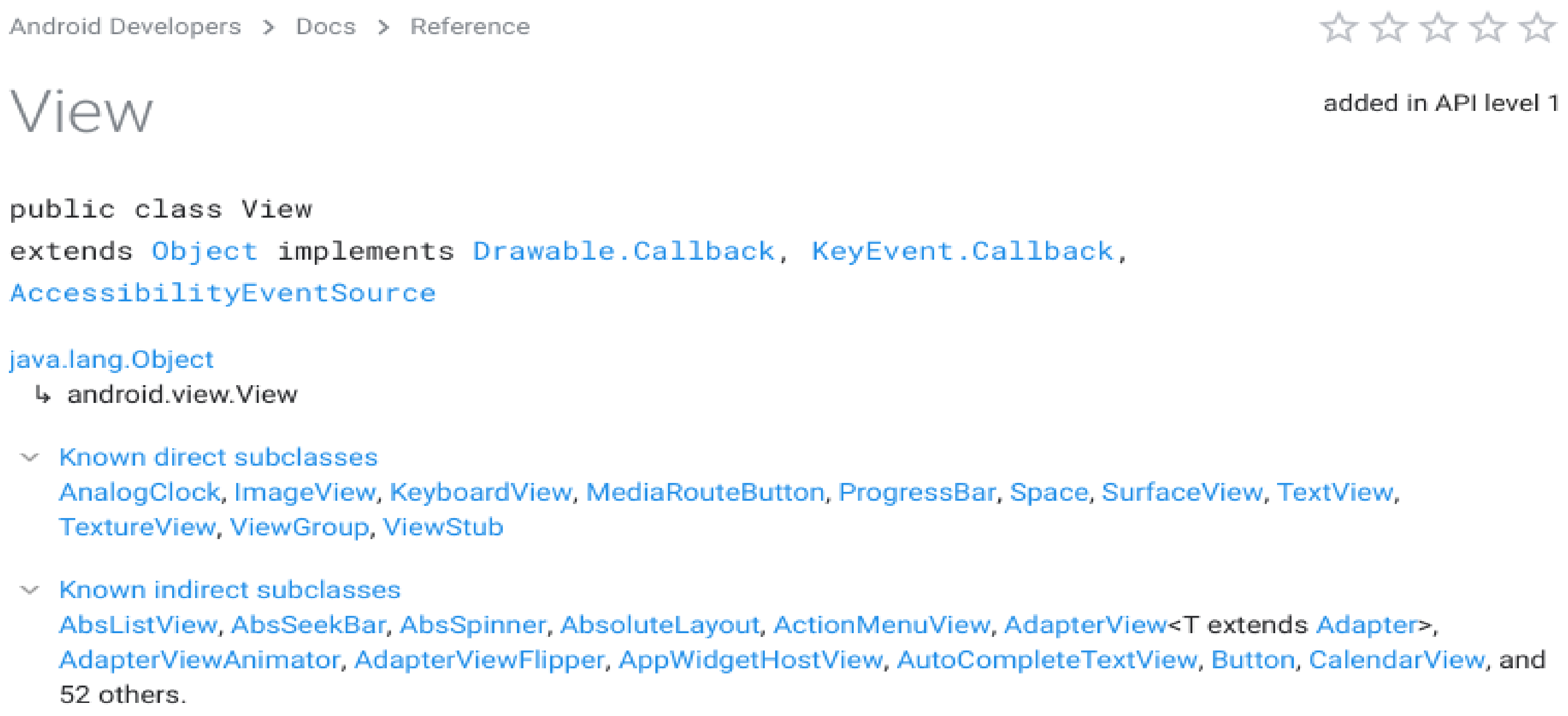
Example



View and ViewGroup

- View :-   The View objects are usually called "widgets" and can be one of many subclasses, such as Button or TextView.
- ViewGroup :-   The ViewGroup objects are usually called "layouts" can be one of many types that provide a different layout structure, such as LinearLayout or ConstraintLayout .

View hierarchy



Android Developers > Docs > Reference

☆ ☆ ☆ ☆ ☆

View

added in API level 1

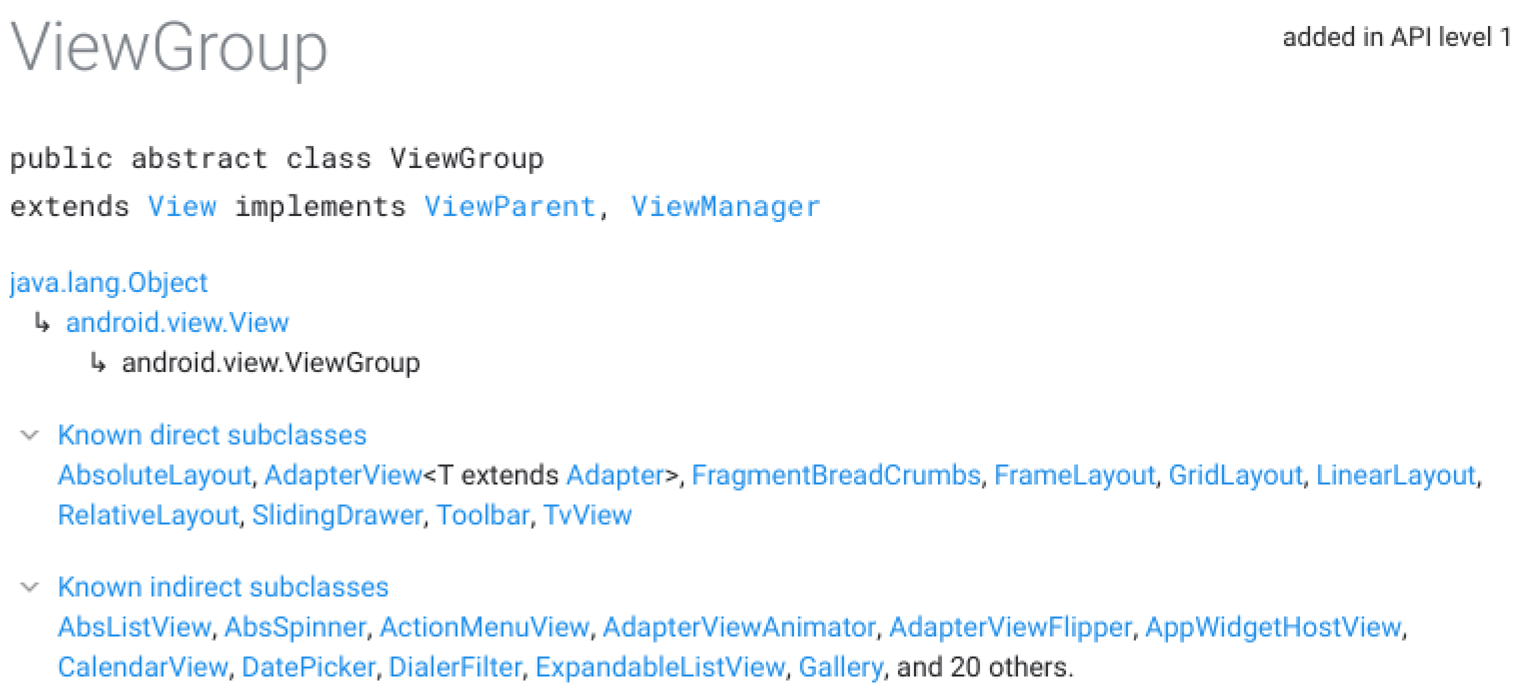
```
public class View
extends Object implements Drawable.Callback, KeyEvent.Callback,
AccessibilityEventSource
```

java.lang.Object
↳ android.view.View

Known direct subclasses
AnalogClock, ImageView, KeyboardView, MediaRouteButton, ProgressBar, Space, SurfaceView, TextView, TextureView, ViewGroup, ViewStub

Known indirect subclasses
AbsListView, AbsSeekBar, AbsSpinner, AbsoluteLayout, ActionMenuView, AdapterView<T extends Adapter>, AdapterViewAnimator, AdapterViewFlipper, AppWidgetHostView, AutoCompleteTextView, Button, CalendarView, and 52 others.

ViewGroup hierarchy



ViewGroup

added in API level 1

```
public abstract class ViewGroup
extends View implements ViewParent, ViewManager
```

java.lang.Object
  ↳ android.view.View
    ↳ android.view.ViewGroup

Known direct subclasses
AbsoluteLayout, AdapterView<T extends Adapter>, FragmentBreadCrumbs, FrameLayout, GridLayout, LinearLayout, RelativeLayout, SlidingDrawer, Toolbar, TvView

Known indirect subclasses
AbsListView, AbsSpinner, ActionMenuView, AdapterViewAnimator, AdapterViewFlipper, AppWidgetHostView, CalendarView, DatePicker, DialerFilter, ExpandableListView, Gallery, and 20 others.

Declaring layouts

You can declare a layout in two ways:

- **Declare UI elements in XML**. Android provides a straightforward XML vocabulary that corresponds to the View classes and subclasses, such as those for widgets and layouts.You can also use Android Studio's Layout Editor to build your XML layout using a drag-and-drop interface.

- **Instantiate layout elements at runtime.** Your app can create View and ViewGroup objects (and manipulate their properties) programmatically.

Write the XML

Using Android's XML vocabulary, you can quickly design UI layouts and the screen elements they contain, in the same way you create web pages in HTML — with a series of nested elements.

Each layout file must contain exactly one root element, which must be a View or ViewGroup object. Once you've defined the root element, you can add additional layout objects or widgets as child elements to gradually build a View hierarchy that defines your layout.

Attributes

Every View and ViewGroup object supports their own variety of XML attributes. Some attributes are specific to a View object (for example, TextView supports the textSize attribute), but these attributes are also inherited by any View objects that may extend this class. Some are common to all View objects, because they are inherited from the root View class (like the id attribute). And, other attributes are considered "layout parameters," which are attributes that describe certain layout orientations of the View object, as defined by that object's parent ViewGroup object.

ID Attribute

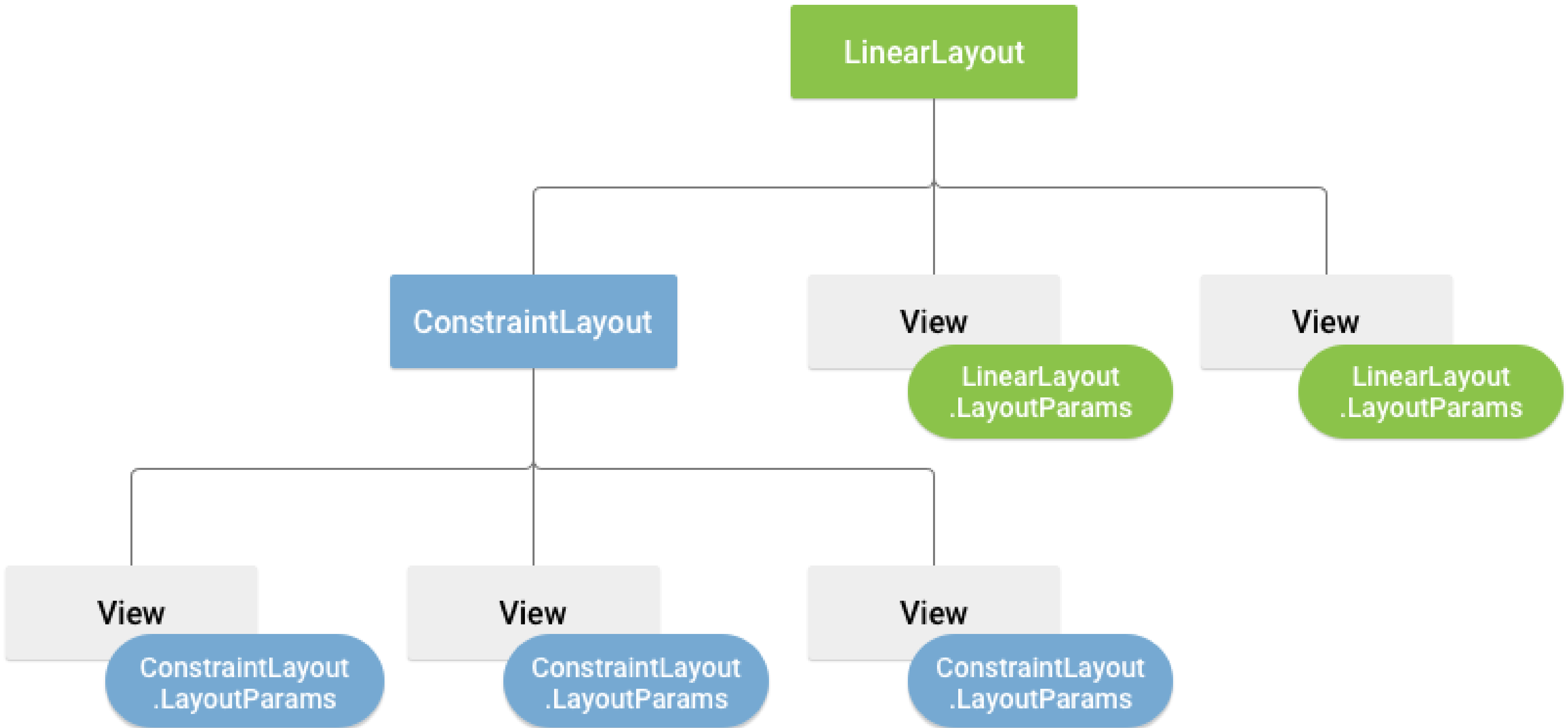
Any View object may have an integer ID associated with it, to uniquely identify the View within the tree. When the app is compiled, this ID is referenced as an integer, but the ID is typically assigned in the layout XML file as a string, in the id attribute. This is an XML attribute common to all View objects (defined by the View class) and you will use it very often.

```
android:id="@+id/textView2"
```

Layout Parameters

- XML layout attributes named layout_something define layout parameters for the View that are appropriate for the ViewGroup in which it resides.
- Every ViewGroup class implements a nested class that extends ViewGroup.LayoutParams. This subclass contains property types that define the size and position for each child view, as appropriate for the view group.

Layout params Hierarchy


LinearLayout
ConstraintLayout
View
LinearLayout .LayoutParams
View
LinearLayout .LayoutParams
View
ConstraintLayout .LayoutParams
View
ConstraintLayout .LayoutParams
View
ConstraintLayout .LayoutParams

Layout Position

The geometry of a view is that of a rectangle. A view has a location, expressed as a pair of left and top coordinates, and two dimensions, expressed as a width and a height. The unit for location and dimensions is the pixel.

It is possible to retrieve the location of a view by invoking the methods **getLeft()** and **getTop()**. The former returns the left, or X, coordinate of the rectangle representing the view. The latter returns the top, or Y, coordinate of the rectangle representing the view. These methods both return the location of the view relative to its parent. For instance, when **getLeft()** returns **20**, that means the view is located 20 pixels to the right of the left edge of its direct parent.

Size, Padding and Margins

The size of a view is expressed with a width and a height. A view actually possesses two pairs of width and height values.

The first pair is known as measured width and measured height. These dimensions define how big a view wants to be within its parent. The measured dimensions can be obtained by calling getMeasuredWidth() and getMeasuredHeight().

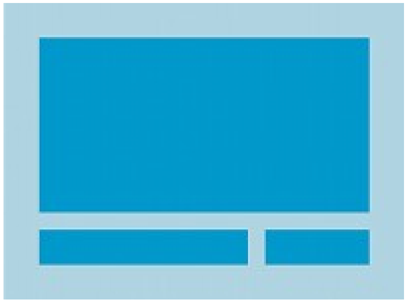
Size, Padding and Margins

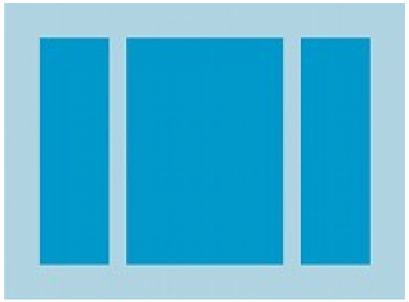
To measure its dimensions, a view takes into account its padding. The padding is expressed in pixels for the left, top, right and bottom parts of the view. Padding can be used to offset the content of the view by a specific number of pixels. For instance, a left padding of 2 will push the view's content by 2 pixels to the right of the left edge. Padding can be set using the setPadding(int, int, int, int) method and queried by calling getPaddingLeft(), getPaddingTop(), getPaddingRight() and getPaddingBottom().

Common Layouts

Each subclass of the ViewGroup class provides a unique way to display the views you nest within it. Below are some of the more common layout types that are built into the Android platform.

Common Layouts





- Relative

  Enables you to specify the location of child objects relative to each other (child A to the left of child B) or to the parent (aligned to the top of the parent).

LineraLayut

A layout that organizes its children into a single horizontal or vertical row. It creates a scrollbar if the length of the window exceeds the length of the screen.

WebLayout

Displays web pages.

CoordinatorLayout

A CoordinatorLayout (a ViewGroup) brings the different elements (child Views) of a ~~(a complex activity or an organization)~~ layout into a harmonious or efficient relationship:

With the help of a CoordinatorLayout, child views work together Efficient to implement awesome behaviours such as
drags, swipes, flings, or any other gestures.

UI with ConstraintLayout

allows you to create large and complex layouts with a flat view hierarchy (no nested view groups). It's similar to RelativeLayout in that all views are laid out according to relationships between sibling views and the parent layout, but it's more flexible than RelativeLayout and easier to use with Android Studio's Layout Editor.

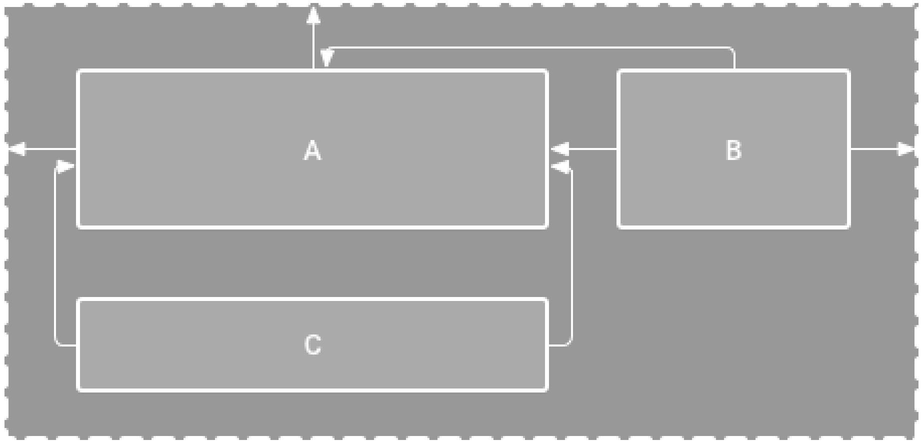
All the power of ConstraintLayout is available directly from the Layout Editor's visual tools, because the layout API and the Layout Editor were specially built for each other. So you can build your layout with ConstraintLayout entirely by drag-and-dropping instead of editing the XML

Constraints overview

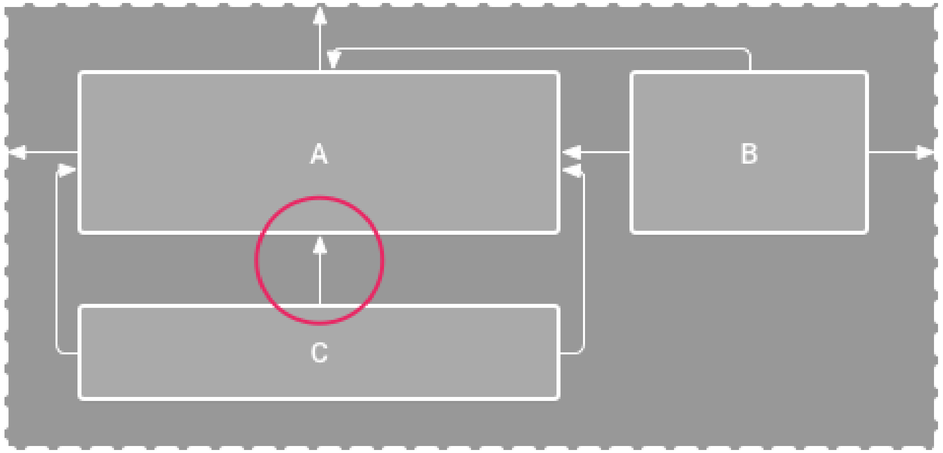
To define a view's position in ConstraintLayout, you must add at least one horizontal and one vertical constraint for the view. Each constraint represents a connection or alignment to another view, the parent layout, or an invisible guideline. Each constraint defines the view's position along either the vertical or horizontal axis; so each view must have a minimum of one constraint for each axis, but often more are necessary.

Constraints overview

The editor shows view C below A, but it has no vertical constraint



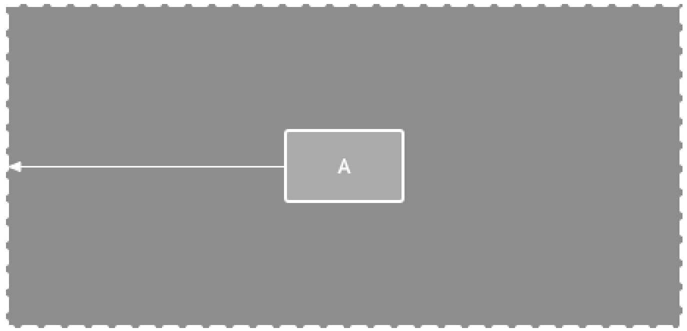
View C is now vertically constrained below view A



Parent position

Constrain the side of a view to the corresponding edge of the layout.

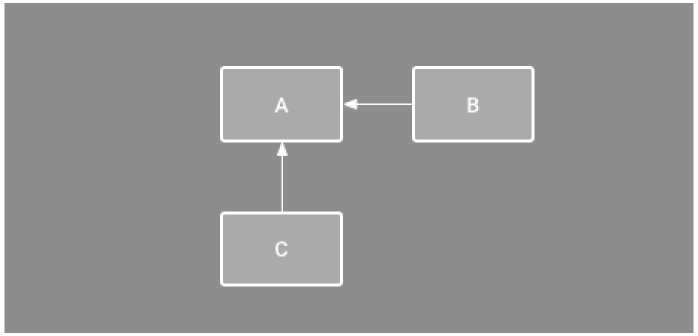
The left side of the view is connected to the left edge of the parent layout. You can define the distance from the edge with margin.



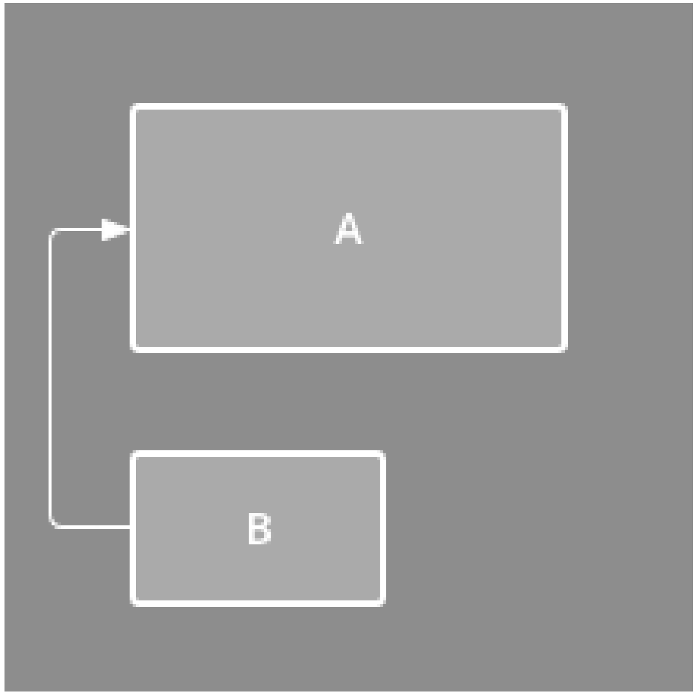
Order position

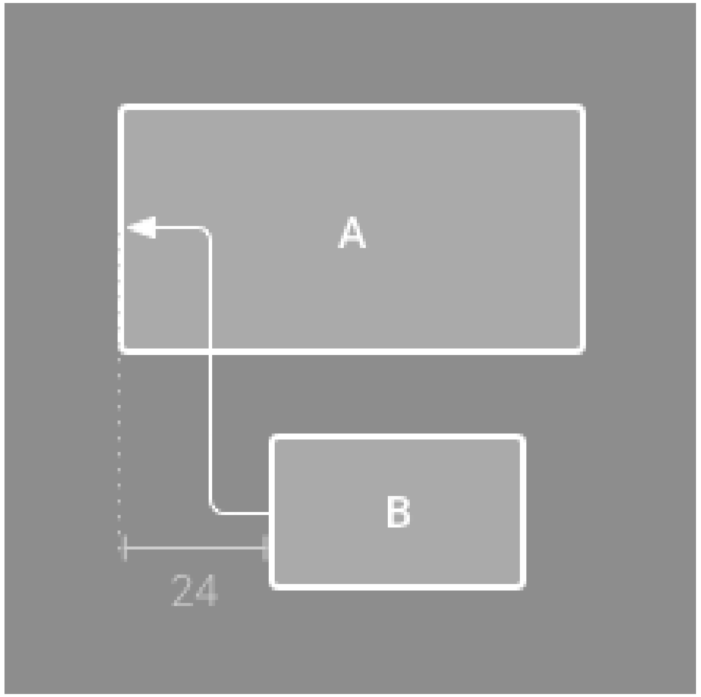
Define the order of appearance for two views, either vertically or horizontally.

B is constrained to always be to the right of A, and C is constrained below A. However, these constraints do not imply alignment, so B can still move up and down.



Alignment



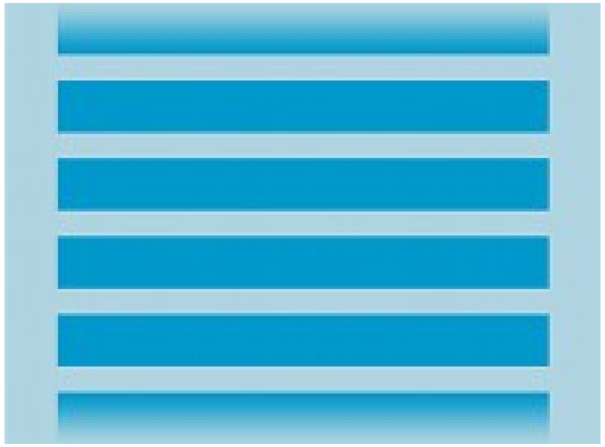


Building Layouts with an Adapter

When the content for your layout is dynamic or not pre-determined, you can use a layout that subclasses AdapterView to populate the layout with views at runtime. A subclass of the AdapterView class uses an Adapter to bind data to its layout. The Adapter behaves as a middleman between the data source and the AdapterView layout—the Adapter retrieves the data (from a source such as an array or a database query) and converts each entry into a view that can be added into the AdapterView layout.

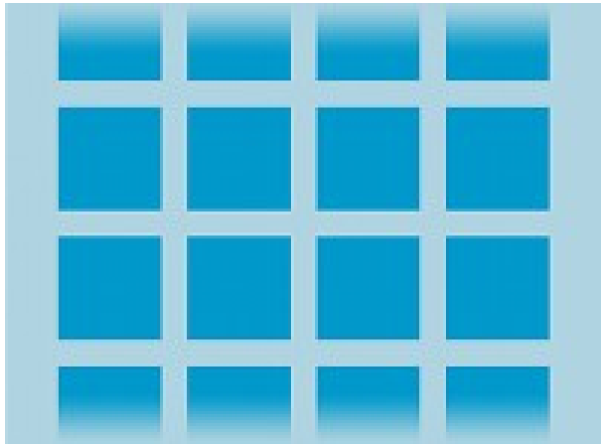
Building Layouts with an Adapter

ListView



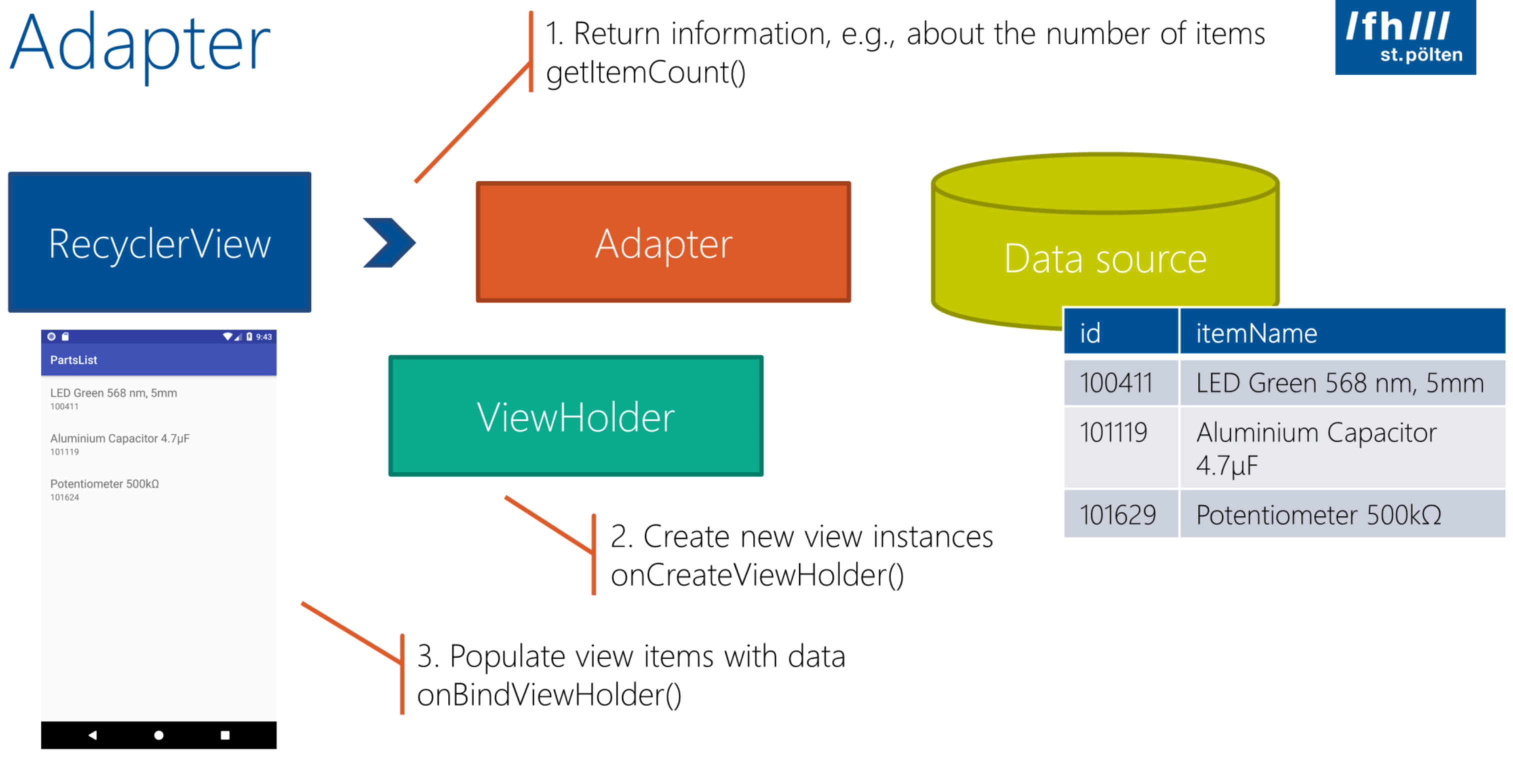
Displays a scrolling single column list.

GridView



Displays a scrolling grid of columns and rows.

Adapter



Adapter

1. Return information, e.g., about the number of items
getItemCount()

2. Create new view instances
onCreateViewHolder()

3. Populate view items with data
onBindViewHolder()

LED Green 568 nm, 5mm
100411
Aluminium Capacitor 4.7µF
101119
Potentiometer 500kΩ
101624

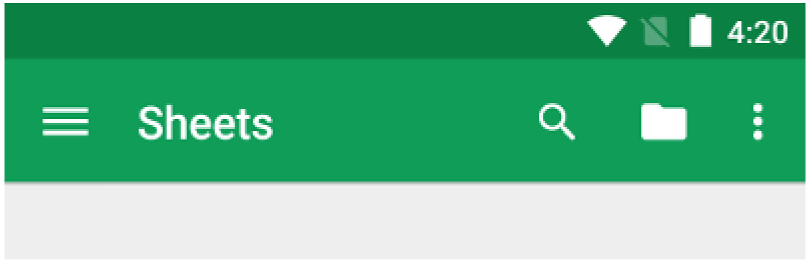
id	itemName
100411	LED Green 568 nm, 5mm
101119	Aluminium Capacitor 4.7µF
101629	Potentiometer 500kΩ

Action Bar (Toolbar/Appbar)

The app bar, also known as the action bar, is one of the most important design elements in your app's activities, because it provides a visual structure and interactive elements that are familiar to users. Using the app bar makes your app consistent with other Android apps, allowing users to quickly understand how to operate your app and have a great experience

- A dedicated space for giving your app an identity and indicating the user's location in the app.
- Access to important actions in a predictable way, such as search.
- Support for navigation and view switching (with tabs or drop-down lists).

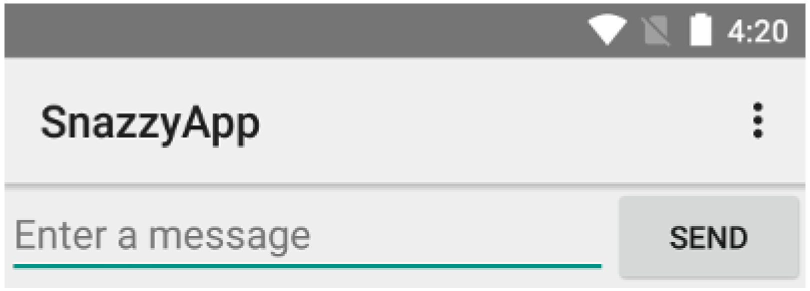
Action Bar (Toolbar/Appbar)



This class describes how to use the v7 appcompat support library's Toolbar widget as an app bar. There are other ways to implement an app bar—for example, some themes set up an ActionBar as an app bar by default—but using the appcompat Toolbar makes it easy to set up an app bar that works on the widest range of devices, and also gives you room to customize your app bar later on as your app develops.

Set up the app bar

In its most basic form, the action bar displays the title for the activity on one side and an overflow menu on the other. Even in this simple form, the app bar provides useful information to the users, and helps to give Android apps a consistent look and feel.



Add a Toolbar to an Activity

These steps describe how to set up a Toolbar as your activity's app bar:

- Add the v7 appcompat support library to your project, as described in Support Library Setup.
- Make sure the activity extends AppCompatActivity:

```
public class MainActivity extends AppCompatActivity {
```

- In the app manifest, set the <application> element to use one of appcompat's NoActionBar themes. Using one of these themes prevents the app from using the native ActionBar class to provide the app bar. For example:
- Add a Toolbar to the activity's layout. For example, the follow

Add a Toolbar to an Activity

```
<android.support.v7.widget.Toolbar
    android:id="@+id/toolbar"
    android:layout_width="match_parent"
    android:layout_height="?attr/actionBarSize"
    android:background="?attr/colorPrimary"
    app:popupTheme="@style/AppTheme.PopupOverlay" />
```

Add a Toolbar to an Activity

-
  ```
  Override
  public boolean onCreateOptionsMenu(Menu menu) {
      // Inflate the menu; this adds items to the action bar if it is present.
      getMenuInflater().inflate(R.menu.menu_main, menu);
      return true;
  }

  @Override
  public boolean onOptionsItemSelected(MenuItem item) {
      // Handle action bar item clicks here. The action bar will
      // automatically handle clicks on the Home/Up button, so long
      // as you specify a parent activity in AndroidManifest.xml.
      int id = item.getItemId();

      //noinspection SimplifiableIfStatement
      if (id == R.id.action_settings) {
          return true;
      }

      return super.onOptionsItemSelected(item);
  }
  ```