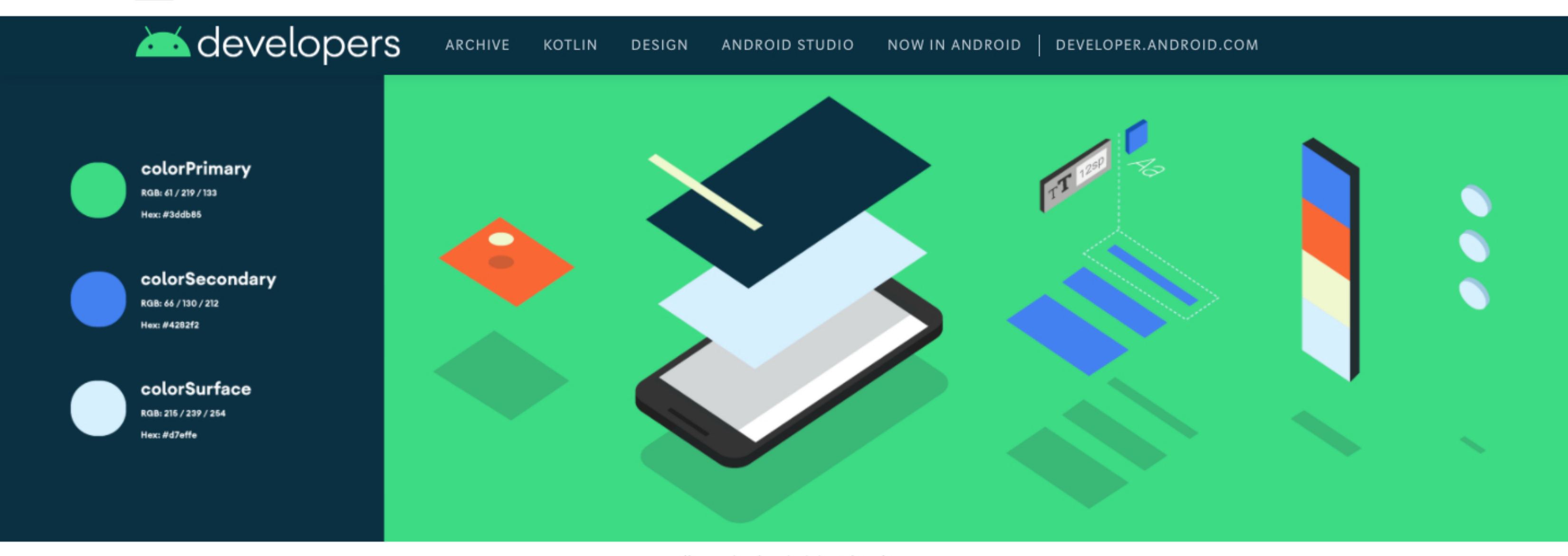


Illustration by Virginia Poltrack

# Android Styling: Themes vs Styles

Nick Butcher Following
Feb 5 · 6 min read

The Android styling system offers a powerful way to specify your app's visual design, but it can be easy to misuse. Proper use of it can make themes and styles easier to maintain, make branding updates less scary and make it straightforward to support dark modes. This is the first in a series of articles where Chris Banes and I will set out to demystify Android styling so that you can make stylish apps without pulling your hair out.

In this first article, I'll take a look at the building blocks of the styling system: themes and styles.

## Theme != Style

Both themes and styles use the same `<style>` syntax but serve very different purposes. You can think of both as key-value stores where the keys are attributes and the values are resources. Let's take a look at each.

## What's in a style?

A style is a collection of view attribute values. You can think of a style as a `Map<view attribute, resource>`. That is the keys are all view attributes i.e. attributes that a widget declares and you might set in a layout file. Styles are specific to a single type of widget because different widgets support different sets of attributes:

> *Styles are a collection of view attributes; specific to a single type of widget*

```xml
<!-- Copyright 2019 Google LLC.
   SPDX-License-Identifier: Apache-2.0 -->
<style name="Widget.Plaid.Button.InlineAction" parent="…">
  <item name="android:gravity">center_horizontal</item>
  <item name="android:textAppearance">@style/TextAppearance.CommentAuthor</item>
  <item name="android:drawablePadding">@dimen/spacing_micro</item>
</style>
```

themes_vs_styles_style.xml hosted with ❤ by GitHub view raw

As you can see, each of the keys in the style are things you *could* set in a layout:

```xml
<!-- Copyright 2019 Google LLC.
   SPDX-License-Identifier: Apache-2.0 -->
<Button …
  android:gravity="center_horizontal"
  android:textAppearance="@style/TextAppearance.CommentAuthor"
  android:drawablePadding="@dimen/spacing_micro"/>
```

themes_vs_styles_view_attrs.xml hosted with ❤ by GitHub view raw

Extracting them to a style makes it easy to reuse across multiple views and maintain.

### Usage

Styles are used by individual views from a layout:

```xml
<!-- Copyright 2019 Google LLC.
   SPDX-License-Identifier: Apache-2.0 -->
<Button …
  style="@style/Widget.Plaid.Button.InlineAction"/>
```

themes_vs_styles_style_usage.xml hosted with ❤ by GitHub view raw

Views can only apply a single style — contrast this to other styling systems such as css on the web where components can set multiple css classes.

### Scope

A style applied to a view **only** applies to *that* view, not to any of its children. For example, if you have a `ViewGroup` with three buttons, setting the `InlineAction` style on the `ViewGroup` will not apply that style to the buttons. The values provided by the style are combined with those set directly in the layout (resolved using the styling precedence order).

## What's a theme?

A theme is a collection of named resources which can be referenced later by styles, layouts etc. They provide semantic names to Android resources so you can refer to them later e.g. `colorPrimary` is a semantic name for a given color:

```xml
<!-- Copyright 2019 Google LLC.
   SPDX-License-Identifier: Apache-2.0 -->
<style name="Theme.Plaid" parent="…">
  <item name="colorPrimary">@color/teal_500</item>
  <item name="colorSecondary">@color/pink_200</item>
  <item name="android:windowBackground">@color/white</item>
</style>
```

themes_vs_styles_theme.xml hosted with ❤ by GitHub view raw

These named resources are known as theme attributes, so a theme is `Map<theme attribute, resource>`. Theme attributes are different from view attributes because they're not properties specific to an individual view type but *semantically named* pointers to values which are applicable more broadly in an app. A theme provides concrete values for these named resources. In the example above the `colorPrimary` attribute specifies that the primary color for this theme is teal. By abstracting the resource with a theme, we can provide different concrete values (such as `colorPrimary` =orange) in different themes.

> *Themes are a collection of named resources, useful broadly across an app*

Top highlight

A theme is similar to an interface. Programming to an interface allows you to decouple the public contract from the implementation allowing you to provide *different* implementations. Themes play a similar role; by writing our layouts and styles against theme attributes, we can use them under different themes, providing different concrete resources.

Roughly equivalent pseudo-code:

```kotlin
/* Copyright 2019 Google LLC.
   SPDX-License-Identifier: Apache-2.0 */
interface ColorPalette {
  @ColorInt val colorPrimary
  @ColorInt val colorSecondary
}

class MyView(colors: ColorPalette) {
  fab.backgroundTint = colors.colorPrimary
}
```

themes_vs_styles_theme_psuedo_interface.kt hosted with ❤ by GitHub view raw

Which allows you to vary the way that `MyView` is rendered, without having to create variants of it:

```kotlin
/* Copyright 2019 Google LLC.
   SPDX-License-Identifier: Apache-2.0 */
val lightPalette = object : ColorPalette { … }
val darkPalette = object : ColorPalette { … }
val view = MyView(if (isDarkTheme) darkPalette else lightPalette)
```

themes_vs_styles_theme_psuedo_interface_usage.kt hosted with ❤ by GitHub view raw

### Usage

You can specify a theme on components which have (or are) a `Context` e.g. `Activity` or `Views` / `ViewGroup` s:

```xml
<!-- Copyright 2019 Google LLC.
   SPDX-License-Identifier: Apache-2.0 -->

<!-- AndroidManifest.xml -->
<application …
  android:theme="@style/Theme.Plaid">
<activity …
  android:theme="@style/Theme.Plaid.About"/>

<!-- layout/foo.xml -->
<ConstraintLayout …
  android:theme="@style/Theme.Plaid.Foo">

```

themes_vs_styles_theme_usage.xml hosted with ❤ by GitHub view raw

You can also set a theme in code by wrapping an existing `Context` with a `ContextThemeWrapper` which you could then use to inflate a layout etc.

The power of themes really comes from how you use them; you can build more flexible widgets by referencing theme attributes. Different themes provide concrete values at a later time. For example, you might wish to set a background color on a section of your view hierarchy:

```xml
<!-- Copyright 2019 Google LLC.
   SPDX-License-Identifier: Apache-2.0 -->
<ViewGroup …
  android:background="?attr/colorSurface">
```

themes_vs_styles_theme_attr_usage.xml hosted with ❤ by GitHub view raw

Rather than setting a static color ( `#ffffff` or a `@color` resource) we can delegate to the theme by using the `?attr/themeAttributeName` syntax. This syntax means: query the theme for the value of this semantic attribute. This level of indirection allows us to provide different behavior (e.g. providing a different background color in light and dark themes) without having to create multiple layouts or styles which are mostly identical but for a few color variations. It isolates the elements that are changing within the theme.

> *Use the `?attr/themeAttributeName` syntax to query the theme for the value of this semantic attribute*

### Scope

A `Theme` is accessed as a property of a `Context` and can be obtained from any object which is or has a `Context` e.g. `Activity`, `View` or `ViewGroup`. These objects exist in a tree, where an `Activity` contains `ViewGroup` s which contain `View` s etc. Specifying a theme at any level of this tree cascades to descendent nodes e.g. setting a theme on a `ViewGroup` applies to all the `View` s within it (in contrast to styles which only apply to a single view).

```xml
<!-- Copyright 2019 Google LLC.
   SPDX-License-Identifier: Apache-2.0 -->
<ViewGroup …
  android:theme="@style/Theme.App.SomeTheme">
  <! - SomeTheme also applies to all child views. -->
</ViewGroup>
```

themes_vs_styles_theme_cascade.xml hosted with ❤ by GitHub view raw

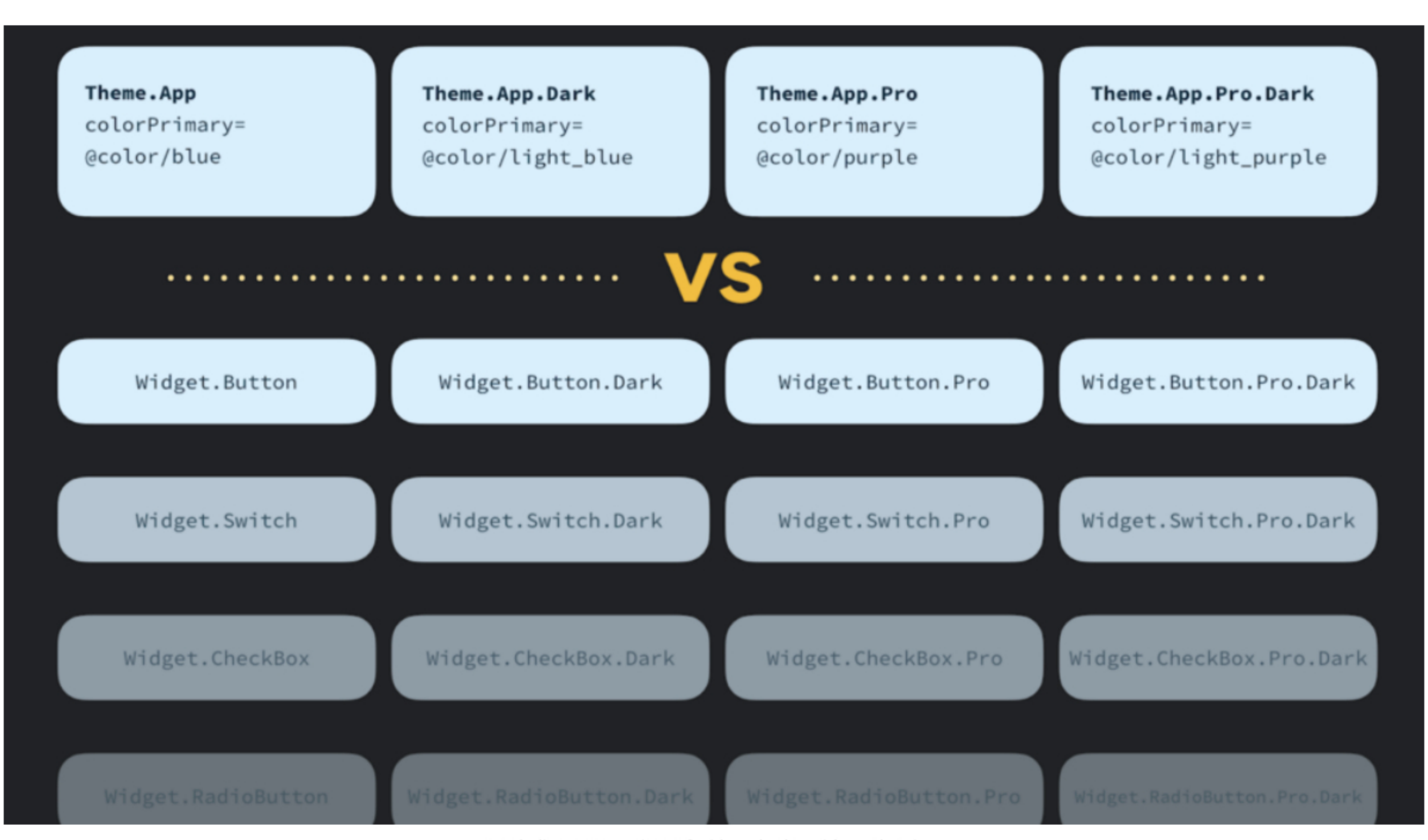
This can be extremely useful, say if you want a dark themed section of an otherwise light screen. Read more about this behavior here.

Note that this behavior only applies at layout inflation time. While `Context` offers a `setTheme` method, or `Theme` offers an `applyStyle` method, these need to be called *before* inflation. Setting a new theme or applying a style after inflation will not update existing views.

## Separate Concerns

Understanding the different responsibilities and the interaction of styles and themes, helps to keep your styling resources more manageable.

For example, say you have a blue theme for your app, but some Pro screens get a fancy purple look **and** you want to provide dark themes with tweaked colors. If you tried to achieve this using only styles, you would have to create 4 styles for the permutations of Pro/non-Pro and light/dark. As styles are specific to a type of view ( `Button`, `Switch` etc) you'd need to create these permutations for each view type in your app.



| Theme.App<br>colorPrimary=<br>@color/blue | Theme.App.Dark<br>colorPrimary=<br>@color/light_blue | Theme.App.Pro<br>colorPrimary=<br>@color/purple | Theme.App.Pro.Dark<br>colorPrimary=<br>@color/light_purple |
|---|---|---|---|

VS

| | | | |
|---|---|---|---|
| Widget.Button | Widget.Button.Dark | Widget.Button.Pro | Widget.Button.Pro.Dark |
| Widget.Switch | Widget.Switch.Dark | Widget.Switch.Pro | Widget.Switch.Pro.Dark |
| Widget.CheckBox | Widget.CheckBox.Dark | Widget.CheckBox.Pro | Widget.CheckBox.Pro.Dark |
| Widget.RadioButton | Widget.RadioButton.Dark | Widget.RadioButton.Pro | Widget.RadioButton.Pro.Dark |

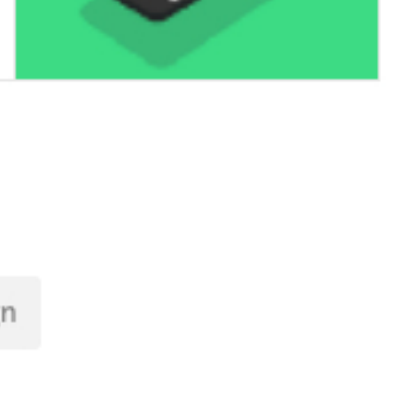
Exploding permutations of widgets/styles without theming

If instead we use styles **and** themes we can isolate the parts which alter by theme as theme attributes so we only need to define a single style per view type. For the above example we might define 4 themes which each provide different values for the `colorPrimary` theme attribute, which these styles then refer to and automatically reflect the correct value from the theme.

This approach might seem more complicated as you need to consider the interaction of styles and themes, but it has the benefit of isolating the parts that change per theme. So if your app rebrands from blue to orange, you only need to change this in a single place, not scattered throughout your styling. It also helps fight a proliferation of styles. Ideally you only have a small number of styles per view type. If you don't take advantage of theming, it's easy for your `styles.xml` file to get out of hand and explode with different variations of similar styles, which becomes a maintenance headache.

Join us in the next article where we explore some common theme attributes and how to create your own:

**Android Styling: Common Theme Attributes**
In the previous article in this series on Android styling, we looked at the difference between themes and styles and…
medium.com



Thanks to Florina Muntenescu and Chris Banes.

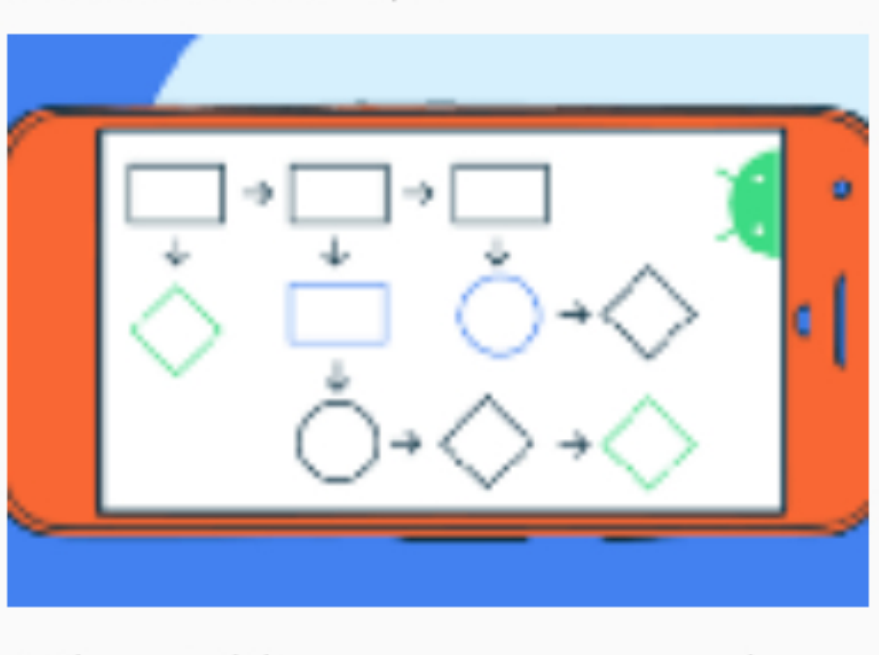
Android · Android App Development · AndroidDev · Android Styles · Design

2,900 claps
Applause
from you and
495 others

WRITTEN BY
**Nick Butcher** Following
Android designer + developer @ Google

**Android Developers** Following
The official Android Developers publication on Medium

See responses (6)



Discover Medium
Welcome to a place where words matter. On Medium, smart voices and original ideas take center stage - with no ads in sight. Watch

Make Medium yours
Follow all the topics you care about, and we'll deliver the best stories for you to your homepage and inbox. Explore

Become a member
Get unlimited access to the best stories on Medium — and support writers while you're at it. Just $5/month. Upgrade

Medium About Help Legal