Mobile Application Development

By,

Prof. Himanshu H Patel,

Prof. Hiten M Sadani

U. V. Patel College of Engineering, Ganpat University

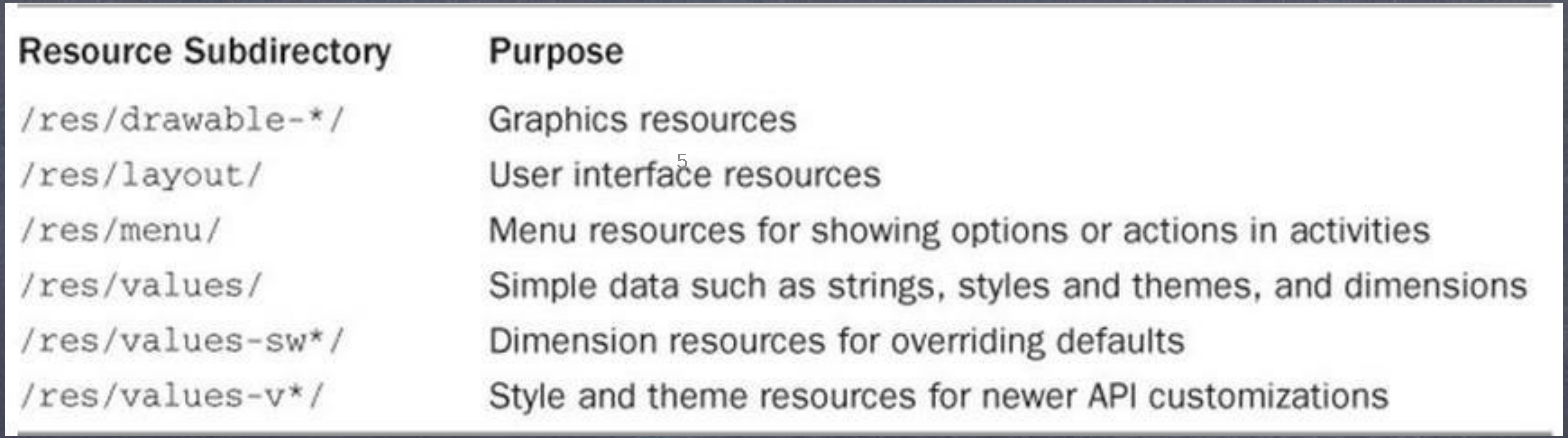
Android Resources

What Are Resources?

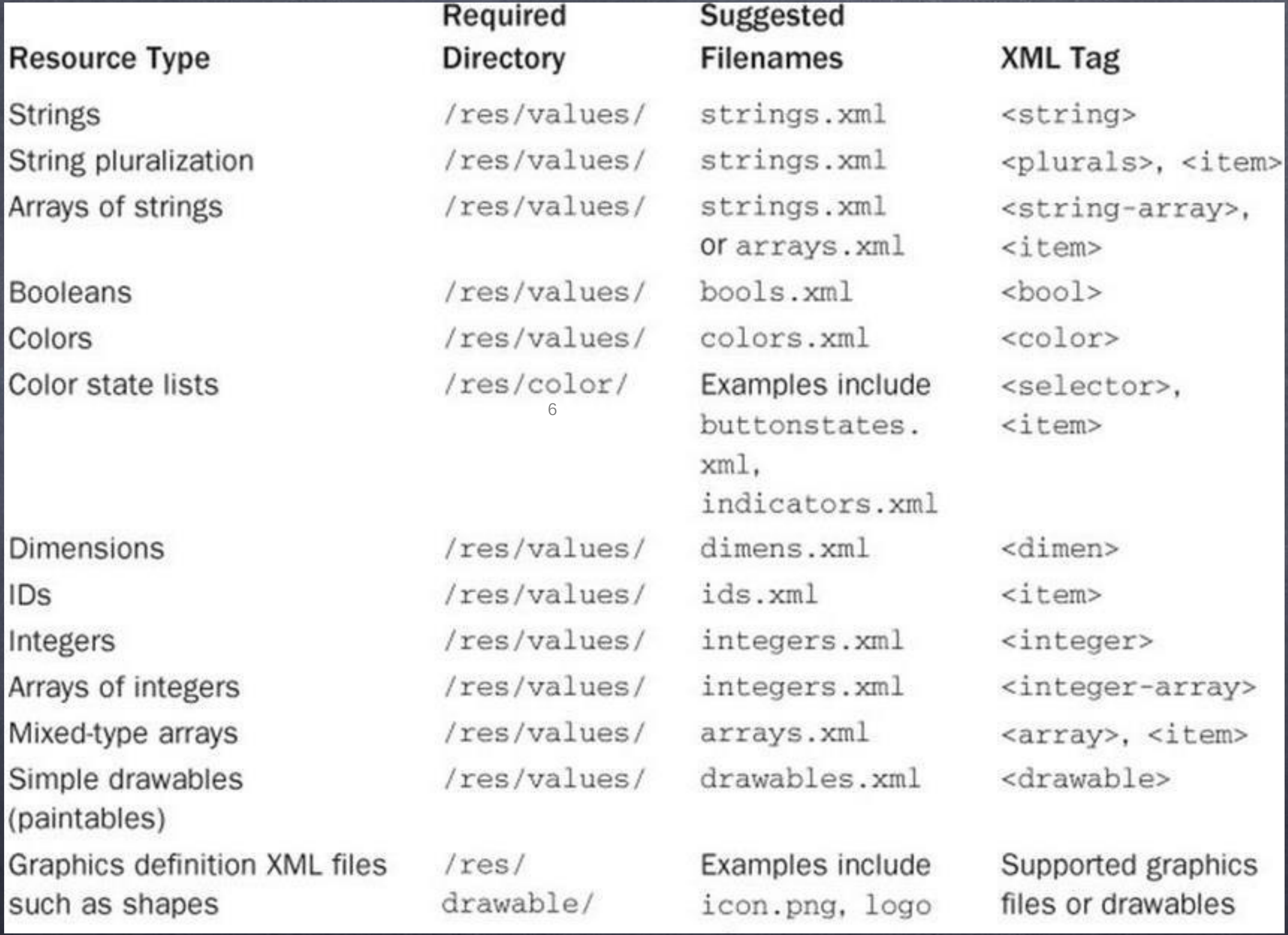
- All Android applications are composed of two things: functionality (code instructions) and data (resources). The functionality is the code that determines how your application behaves. This includes any algorithms that make the application run. Resources include text strings, styles and themes, dimensions, images and icons, audio files, videos, and other data used by the application.

Storing Application Resources

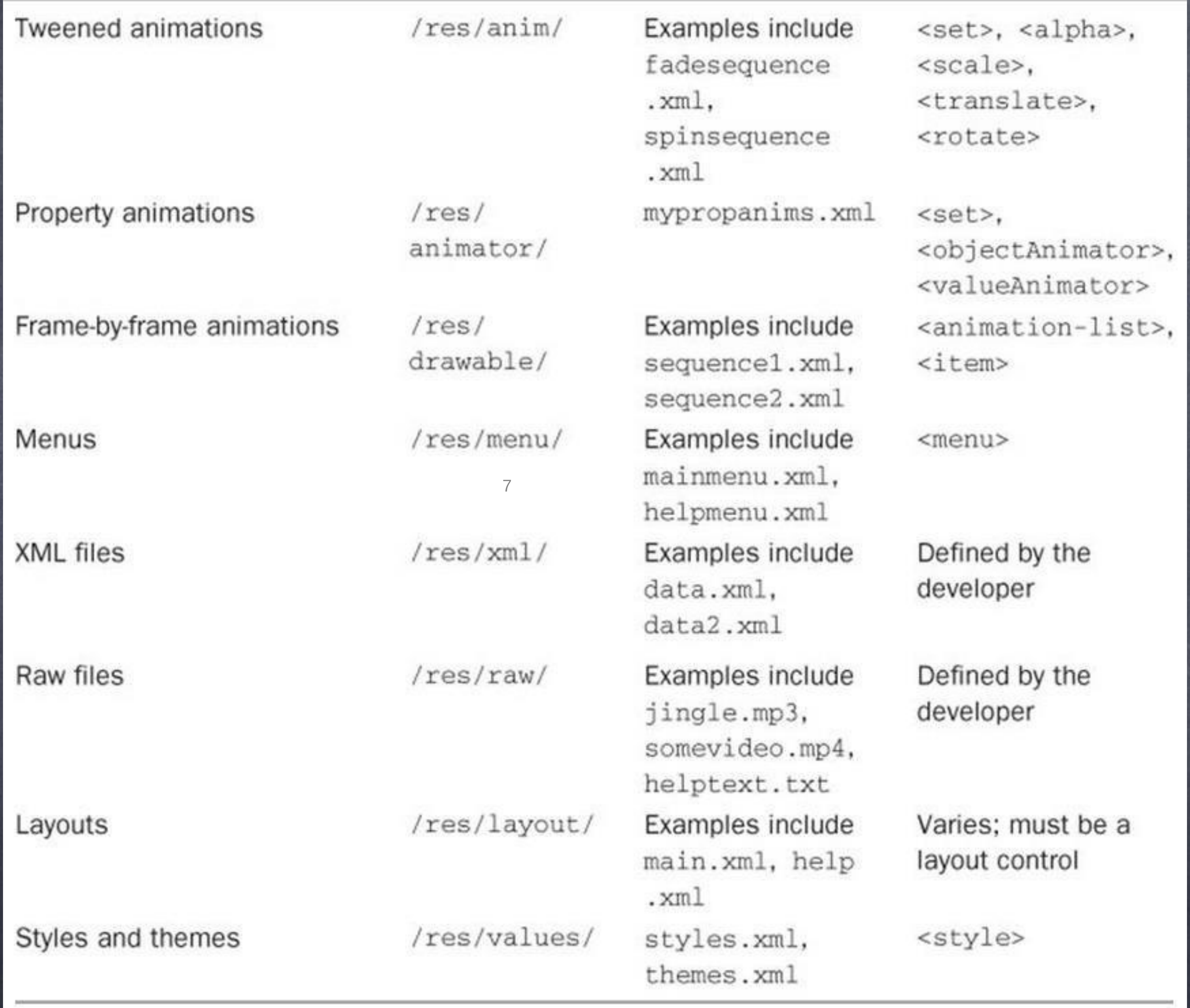
- Android resource files are stored separately from the .java/Kotlin class files in the Android project. Most common resource types are stored in XML. You can also store raw data files and graphics as resources. Resources are organized in a strict directory hierarchy. All resources must be stored under the /res project directory in specially named subdirectories whose names must be lowercase.
- Different resource types are stored in different directories. The resource subdirectories generated when you create an Android project



Resource Subdirectory	Purpose
/res/drawable-*/	Graphics resources
/res/layout/	User interface resources
/res/menu/	Menu resources for showing options or actions in activities
/res/values/	Simple data such as strings, styles and themes, and dimensions
/res/values-sw*/	Dimension resources for overriding defaults
/res/values-v*/	Style and theme resources for newer API customizations

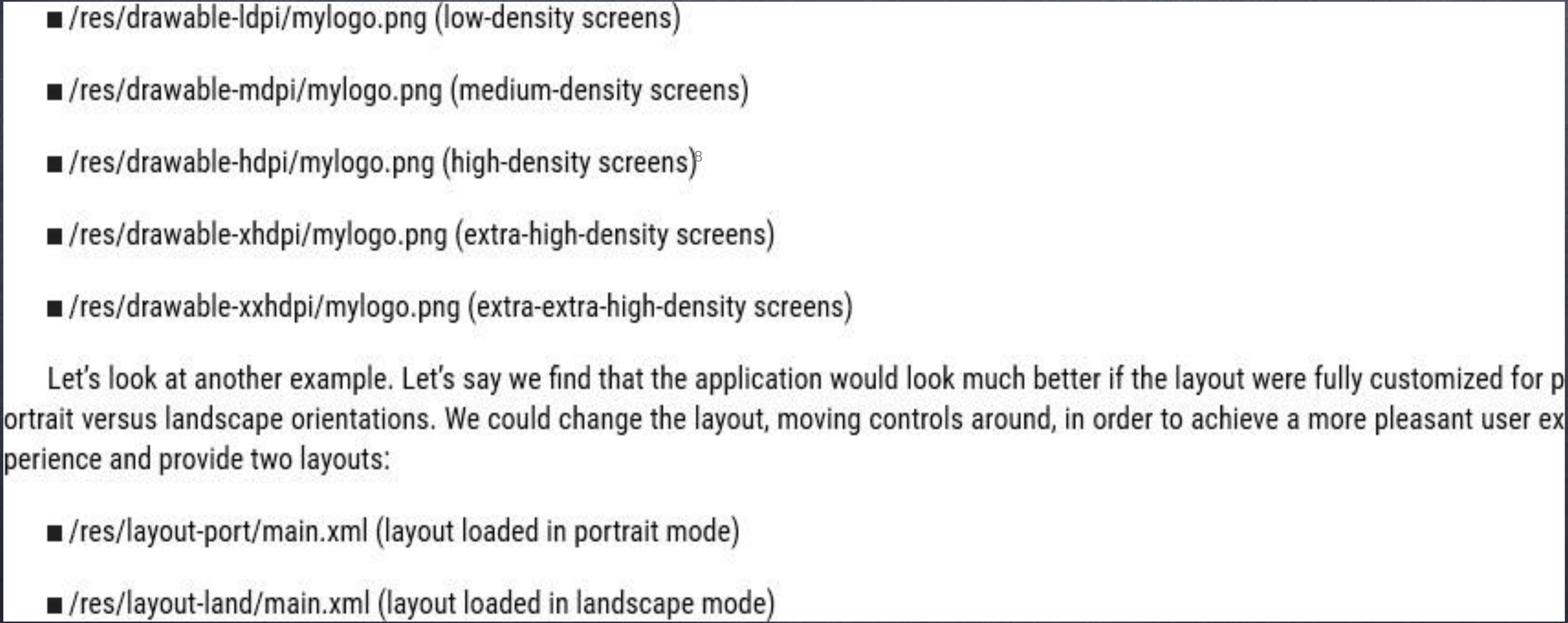


Resource Type	Required Directory	Suggested Filenames	XML Tag
Strings	`/res/values/`	`strings.xml`	`<string>`
String pluralization	`/res/values/`	`strings.xml`	`<plurals>`, `<item>`
Arrays of strings	`/res/values/`	`strings.xml` or `arrays.xml`	`<string-array>`, `<item>`
Booleans	`/res/values/`	`bools.xml`	`<bool>`
Colors	`/res/values/`	`colors.xml`	`<color>`
Color state lists	`/res/color/`	Examples include `buttonstates.xml`, `indicators.xml`	`<selector>`, `<item>`
Dimensions	`/res/values/`	`dimens.xml`	`<dimen>`
IDs	`/res/values/`	`ids.xml`	`<item>`
Integers	`/res/values/`	`integers.xml`	`<integer>`
Arrays of integers	`/res/values/`	`integers.xml`	`<integer-array>`
Mixed-type arrays	`/res/values/`	`arrays.xml`	`<array>`, `<item>`
Simple drawables (paintables)	`/res/values/`	`drawables.xml`	`<drawable>`
Graphics definition XML files such as shapes	`/res/drawable/`	Examples include `icon.png`, `logo`	Supported graphics files or drawables



Tweened animations	`/res/anim/`	Examples include `fadesequence.xml`, `spinsequence.xml`	`<set>`, `<alpha>`, `<scale>`, `<translate>`, `<rotate>`
Property animations	`/res/animator/`	`mypropanims.xml`	`<set>`, `<objectAnimator>`, `<valueAnimator>`
Frame-by-frame animations	`/res/drawable/`	Examples include `sequence1.xml`, `sequence2.xml`	`<animation-list>`, `<item>`
Menus	`/res/menu/`	Examples include `mainmenu.xml`, `helpmenu.xml`	`<menu>`
XML files	`/res/xml/`	Examples include `data.xml`, `data2.xml`	Defined by the developer
Raw files	`/res/raw/`	Examples include `jingle.mp3`, `somevideo.mp4`, `helptext.txt`	Defined by the developer
Layouts	`/res/layout/`	Examples include `main.xml`, `help.xml`	Varies; must be a layout control
Styles and themes	`/res/values/`	`styles.xml`, `themes.xml`	`<style>`

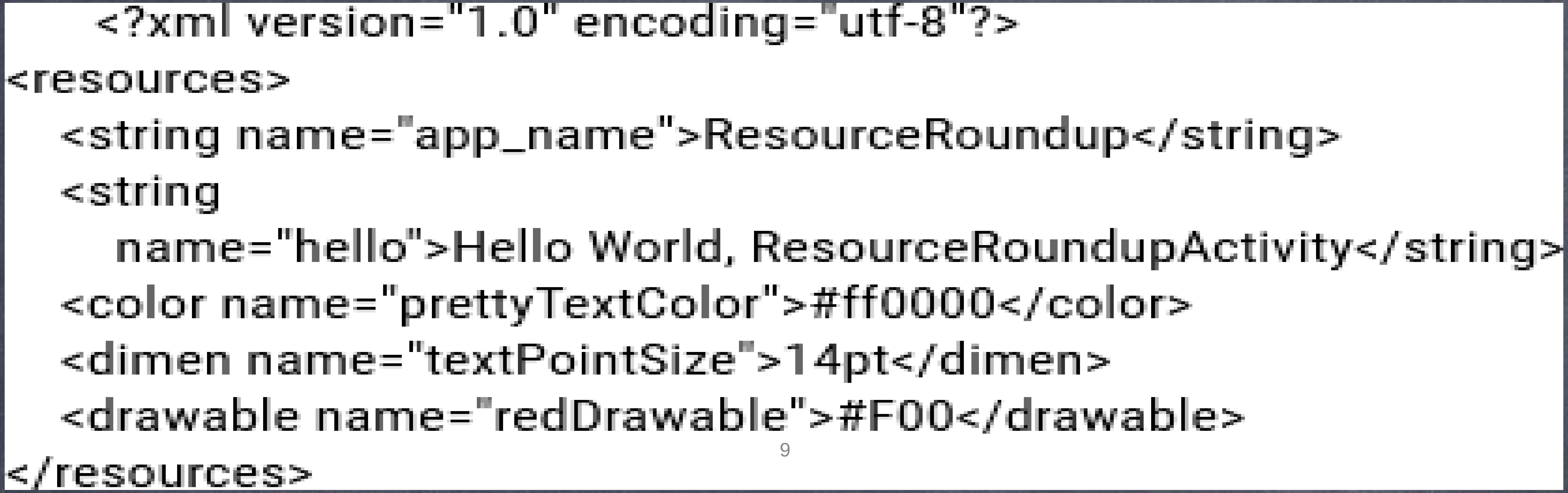
Table 6.2 How Common Resource Types Are Stored in the Project File Hierarchy



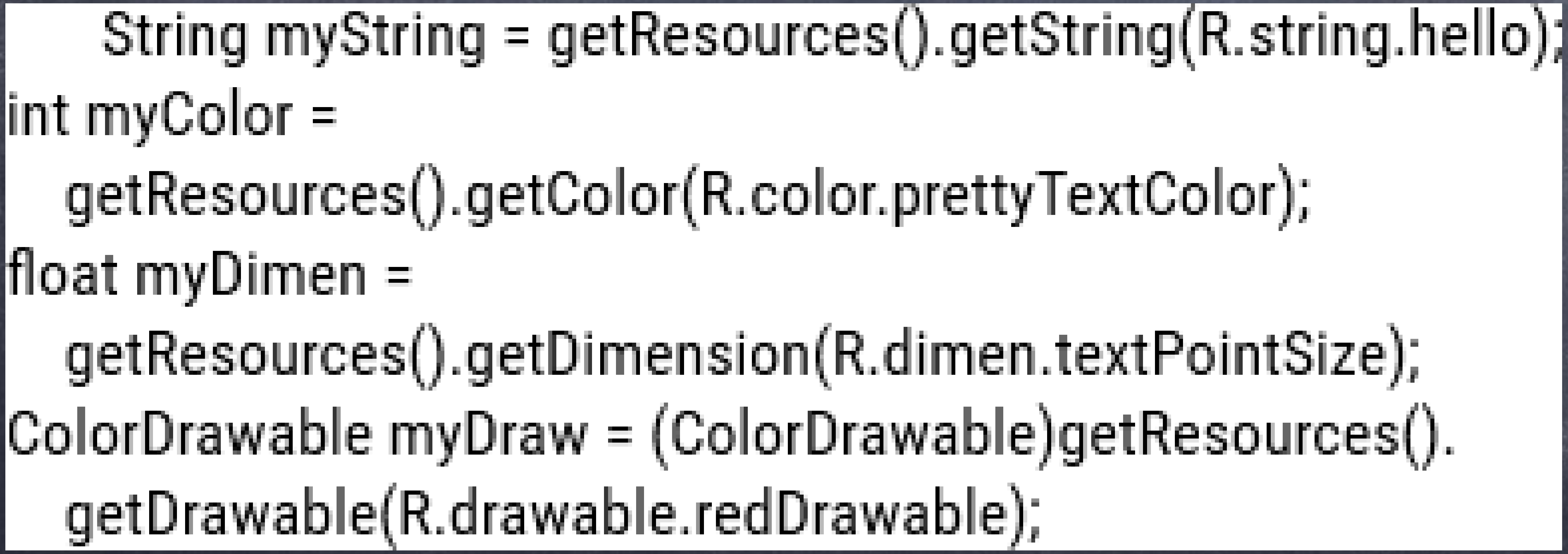
- /res/drawable-ldpi/mylogo.png (low-density screens)
- /res/drawable-mdpi/mylogo.png (medium-density screens)
- /res/drawable-hdpi/mylogo.png (high-density screens)
- /res/drawable-xhdpi/mylogo.png (extra-high-density screens)
- /res/drawable-xxhdpi/mylogo.png (extra-extra-high-density screens)

Let's look at another example. Let's say we find that the application would look much better if the layout were fully customized for p
ortrait versus landscape orientations. We could change the layout, moving controls around, in order to achieve a more pleasant user ex
perience and provide two layouts:

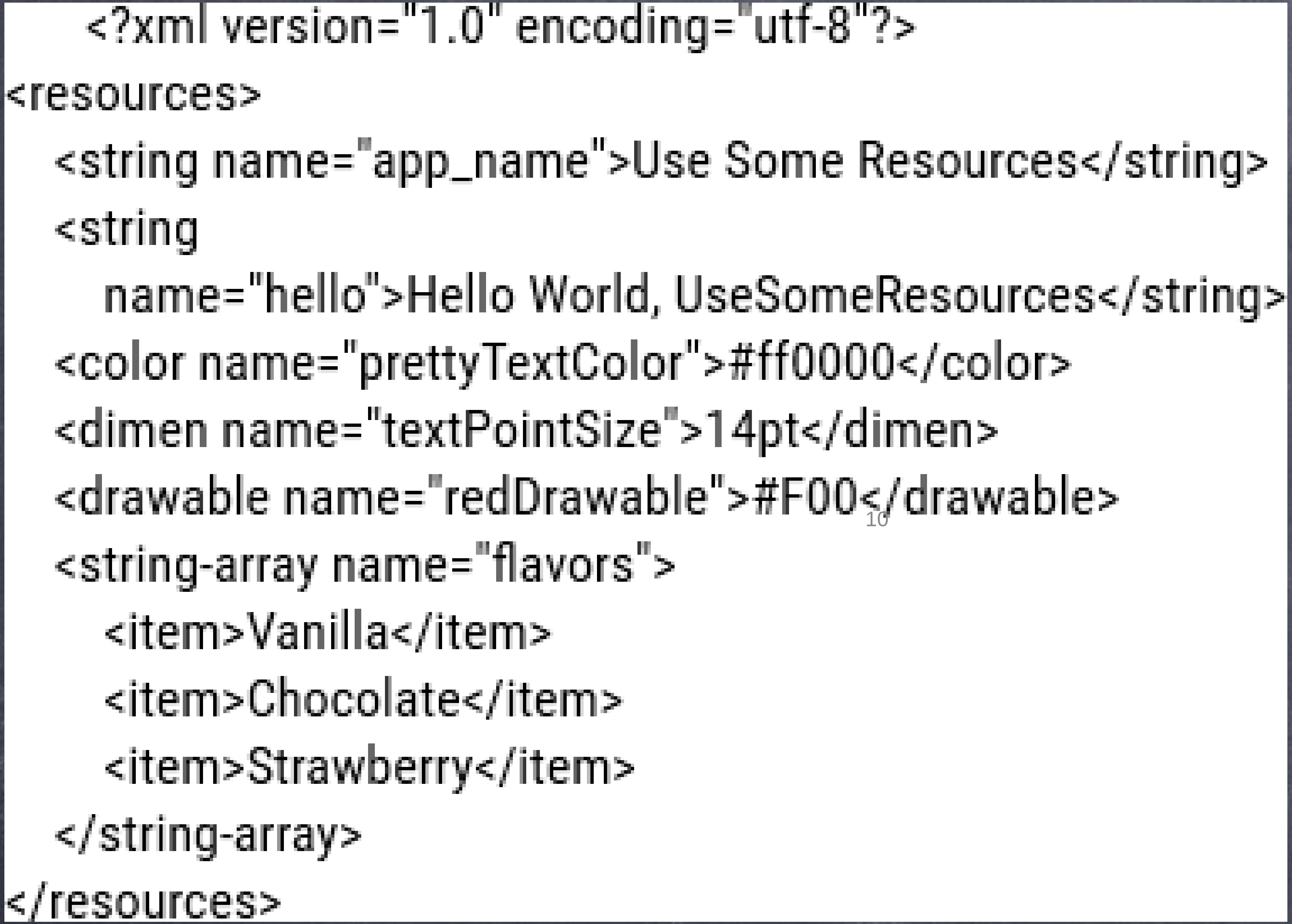
- /res/layout-port/main.xml (layout loaded in portrait mode)
- /res/layout-land/main.xml (layout loaded in landscape mode)



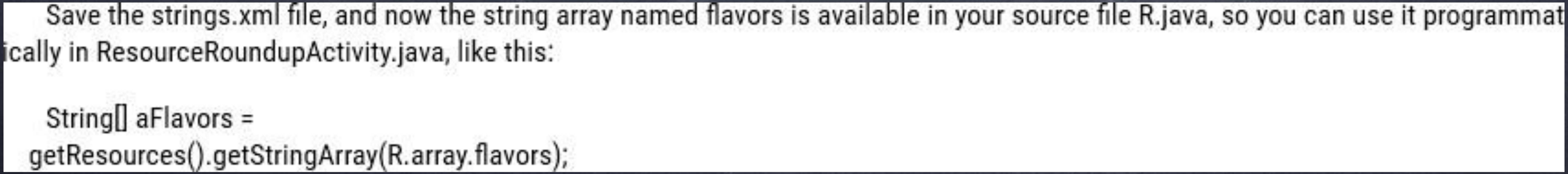
```xml
      <?xml version="1.0" encoding="utf-8"?>
<resources>
    <string name="app_name">ResourceRoundup</string>
    <string
        name="hello">Hello World, ResourceRoundupActivity</string>
    <color name="prettyTextColor">#ff0000</color>
    <dimen name="textPointSize">14pt</dimen>
    <drawable name="redDrawable">#F00</drawable>
</resources>
```



```java
     String myString = getResources().getString(R.string.hello);
int myColor =
    getResources().getColor(R.color.prettyTextColor);
float myDimen =
    getResources().getDimension(R.dimen.textPointSize);
ColorDrawable myDraw = (ColorDrawable)getResources().
    getDrawable(R.drawable.redDrawable);
```

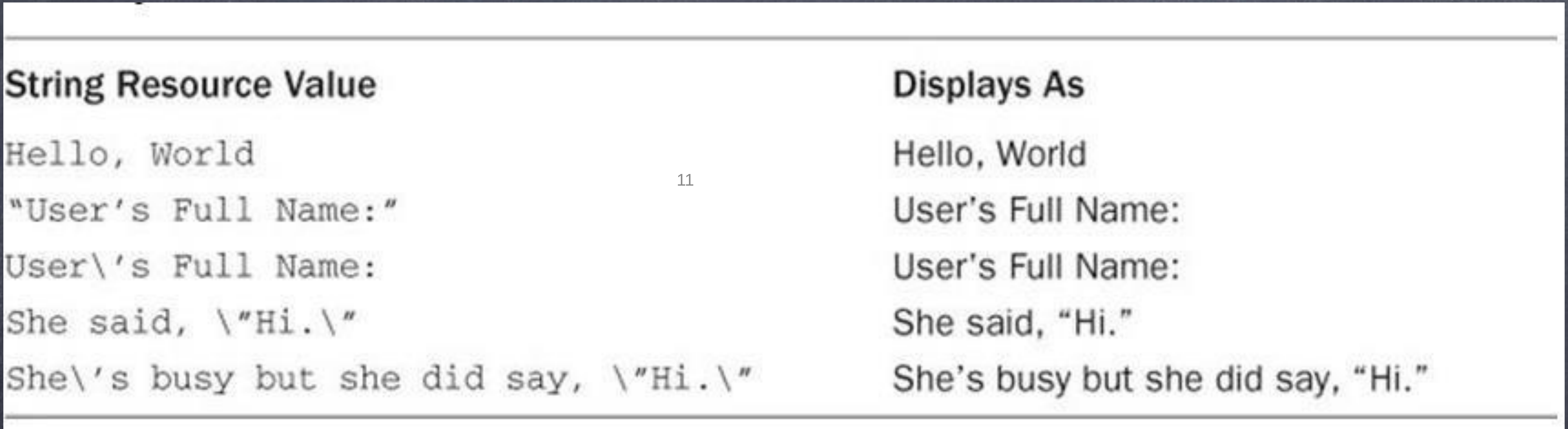


```
    <?xml version="1.0" encoding="utf-8"?>
<resources>
  <string name="app_name">Use Some Resources</string>
  <string
    name="hello">Hello World, UseSomeResources</string>
  <color name="prettyTextColor">#ff0000</color>
  <dimen name="textPointSize">14pt</dimen>
  <drawable name="redDrawable">#F00</drawable>
  <string-array name="flavors">
    <item>Vanilla</item>
    <item>Chocolate</item>
    <item>Strawberry</item>
  </string-array>
</resources>
```



Save the strings.xml file, and now the string array named flavors is available in your source file R.java, so you can use it programmat ically in ResourceRoundupActivity.java, like this:

```
  String[] aFlavors =
getResources().getStringArray(R.array.flavors);
```



String Resource Value	Displays As
`Hello, World`	Hello, World
`“User’s Full Name:”`	User’s Full Name:
`User\’s Full Name:`	User’s Full Name:
`She said, \”Hi.\”`	She said, “Hi.”
`She\’s busy but she did say, \”Hi.\”`	She’s busy but she did say, “Hi.”

Table 6.3 String Resource Formatting Examples

Defining Boolean Resources in XML

Boolean values are appropriately tagged with the <bool> tag and represent a name/value pair. The name attribute is how you refer t
o the specific Boolean value programmatically, so name these resources wisely.

Here is an example of the Boolean resource file /res/values/bools.xml:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
  <bool name="onePlusOneEqualsTwo">true</bool>
  <bool name="isAdvancedFeaturesEnabled">false</bool>
</resources>
```

Using Boolean Resources Programmatically

To use a Boolean resource in code, you can load it using the getBoolean() method of the Resources class. The following code acce
sses your application's Boolean resource named bAdvancedFeaturesEnabled:

```
boolean isAdvancedMode =
  getResources().getBoolean(R.bool.isAdvancedFeaturesEnabled);
```

Working with Integer Resources

In addition to strings and Boolean values, you can also store integers as resources. Integer resources are defined in XML under the /res/values project directory and compiled into the application package at build time.

Defining Integer Resources in XML

Integer values are appropriately tagged with the <integer> tag and represent a name/value pair. The name attribute is how you refer to the specific integer programmatically, so name these resources wisely.

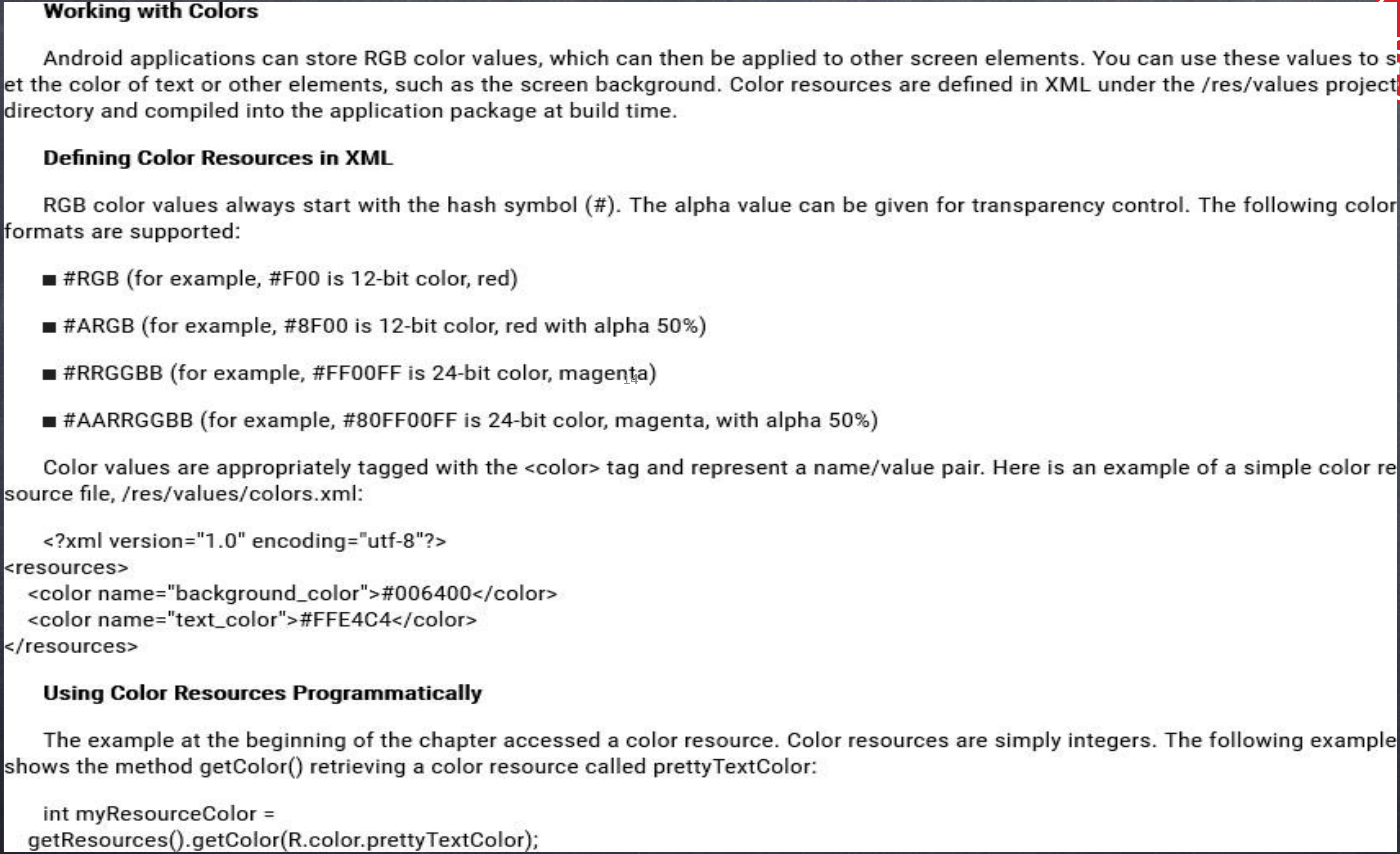
Here is an example of the integer resource file /res/values/nums.xml:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
   <integer name="numTimesToRepeat">25</integer>
   <integer name="startingAgeOfCharacter">3</integer>
</resources>
```

Using Integer Resources Programmatically

To use the integer resource, you must load it using the Resources class. The following code accesses your application's integer res ource named numTimesToRepeat:

```
int repTimes = getResources().getInteger(R.integer.numTimesToRepeat);
```



Working with Colors

Android applications can store RGB color values, which can then be applied to other screen elements. You can use these values to s
et the color of text or other elements, such as the screen background. Color resources are defined in XML under the /res/values project
directory and compiled into the application package at build time.

Defining Color Resources in XML

RGB color values always start with the hash symbol (#). The alpha value can be given for transparency control. The following color
formats are supported:

- #RGB (for example, #F00 is 12-bit color, red)
- #ARGB (for example, #8F00 is 12-bit color, red with alpha 50%)
- #RRGGBB (for example, #FF00FF is 24-bit color, magenta)
- #AARRGGBB (for example, #80FF00FF is 24-bit color, magenta, with alpha 50%)

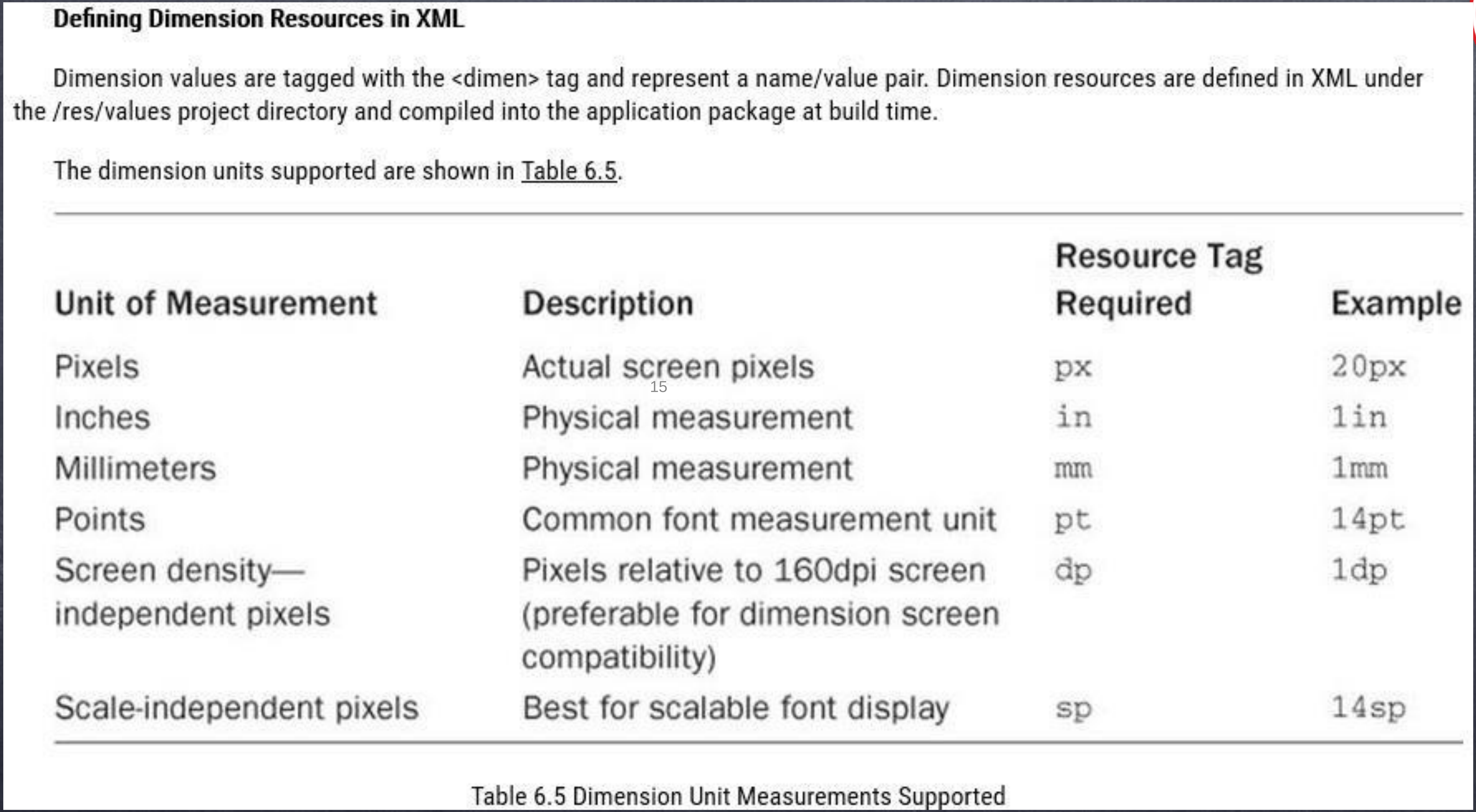
Color values are appropriately tagged with the <color> tag and represent a name/value pair. Here is an example of a simple color re
source file, /res/values/colors.xml:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
  <color name="background_color">#006400</color>
  <color name="text_color">#FFE4C4</color>
</resources>
```

Using Color Resources Programmatically

The example at the beginning of the chapter accessed a color resource. Color resources are simply integers. The following example
shows the method getColor() retrieving a color resource called prettyTextColor:

```
int myResourceColor =
  getResources().getColor(R.color.prettyTextColor);
```



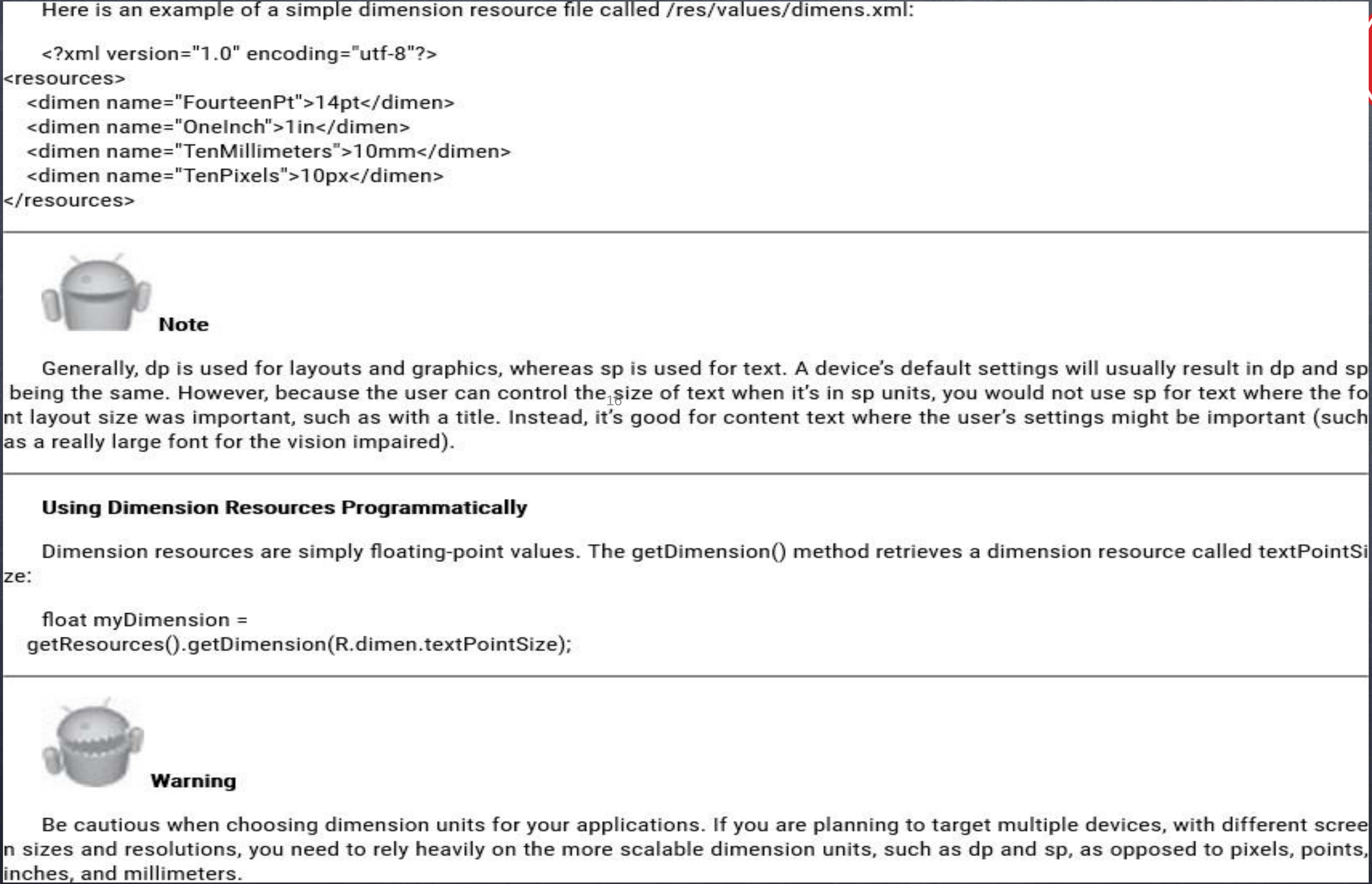
Defining Dimension Resources in XML

Dimension values are tagged with the <dimen> tag and represent a name/value pair. Dimension resources are defined in XML under the /res/values project directory and compiled into the application package at build time.

The dimension units supported are shown in Table 6.5.

Unit of Measurement	Description	Resource Tag Required	Example
Pixels	Actual screen pixels	px	20px
Inches	Physical measurement	in	1in
Millimeters	Physical measurement	mm	1mm
Points	Common font measurement unit	pt	14pt
Screen density—independent pixels	Pixels relative to 160dpi screen (preferable for dimension screen compatibility)	dp	1dp
Scale-independent pixels	Best for scalable font display	sp	14sp

Table 6.5 Dimension Unit Measurements Supported



Here is an example of a simple dimension resource file called /res/values/dimens.xml:

```
    <?xml version="1.0" encoding="utf-8"?>
<resources>
   <dimen name="FourteenPt">14pt</dimen>
   <dimen name="OneInch">1in</dimen>
   <dimen name="TenMillimeters">10mm</dimen>
   <dimen name="TenPixels">10px</dimen>
</resources>
```

Note

Generally, dp is used for layouts and graphics, whereas sp is used for text. A device's default settings will usually result in dp and sp being the same. However, because the user can control the size of text when it's in sp units, you would not use sp for text where the fo
nt layout size was important, such as with a title. Instead, it's good for content text where the user's settings might be important (such as a really large font for the vision impaired).

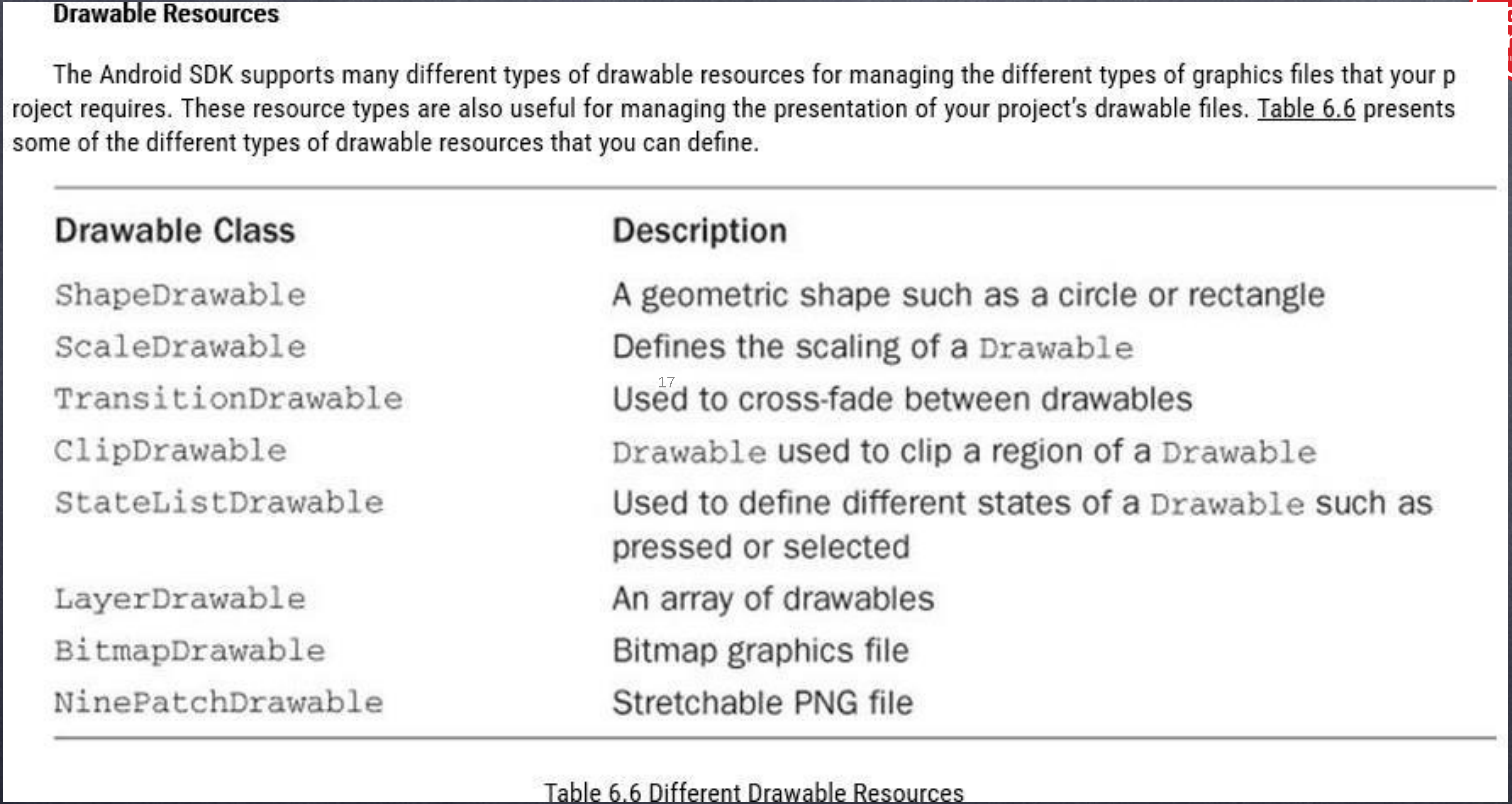
Using Dimension Resources Programmatically

Dimension resources are simply floating-point values. The getDimension() method retrieves a dimension resource called textPointSi
ze:

```
    float myDimension =
  getResources().getDimension(R.dimen.textPointSize);
```

Warning

Be cautious when choosing dimension units for your applications. If you are planning to target multiple devices, with different scree
n sizes and resolutions, you need to rely heavily on the more scalable dimension units, such as dp and sp, as opposed to pixels, points, inches, and millimeters.

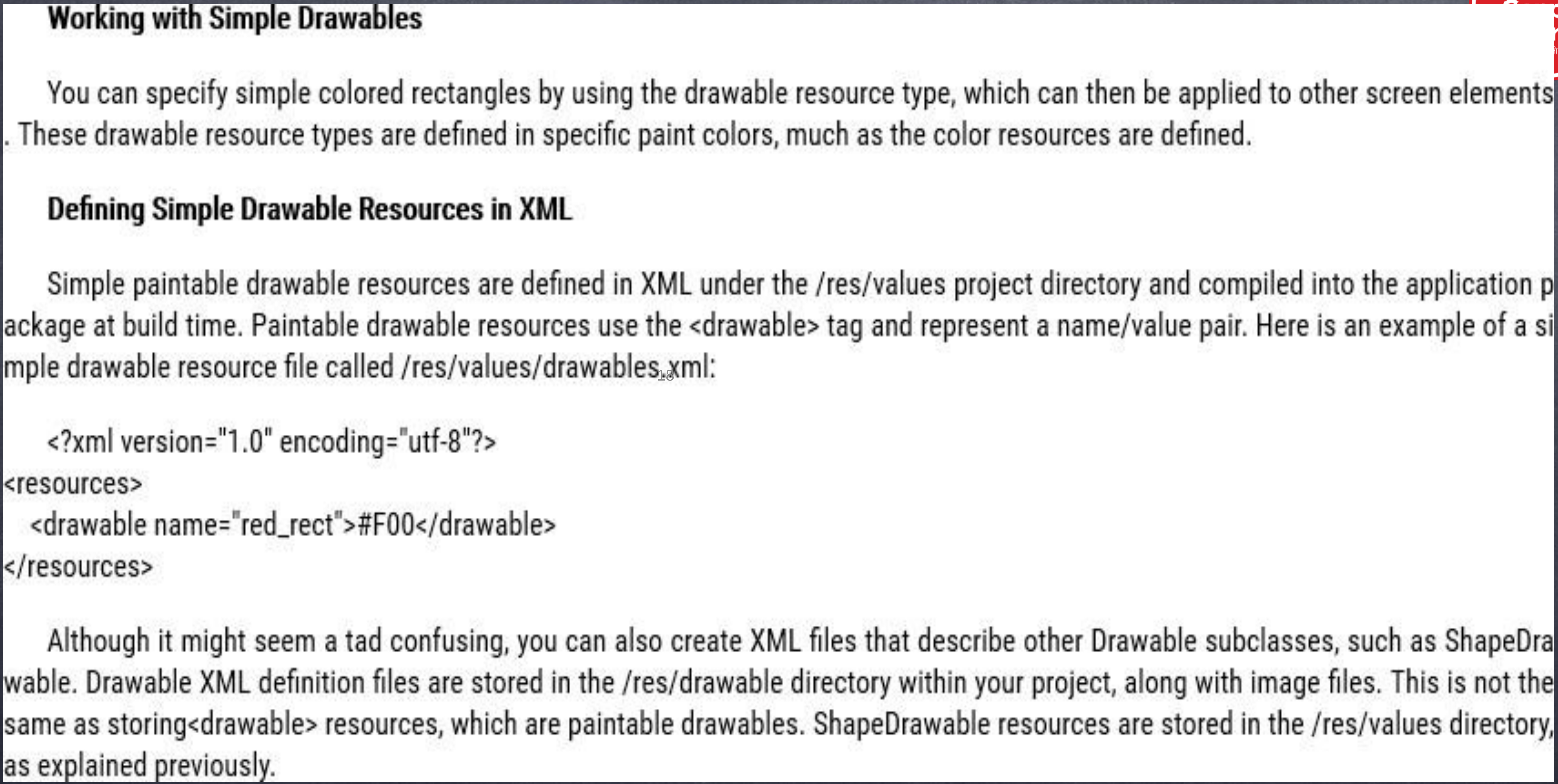


Drawable Resources

The Android SDK supports many different types of drawable resources for managing the different types of graphics files that your p roject requires. These resource types are also useful for managing the presentation of your project's drawable files. Table 6.6 presents some of the different types of drawable resources that you can define.

Drawable Class	Description
`ShapeDrawable`	A geometric shape such as a circle or rectangle
`ScaleDrawable`	Defines the scaling of a `Drawable`
`TransitionDrawable`	Used to cross-fade between drawables
`ClipDrawable`	`Drawable` used to clip a region of a `Drawable`
`StateListDrawable`	Used to define different states of a `Drawable` such as pressed or selected
`LayerDrawable`	An array of drawables
`BitmapDrawable`	Bitmap graphics file
`NinePatchDrawable`	Stretchable PNG file

Table 6.6 Different Drawable Resources



Working with Simple Drawables

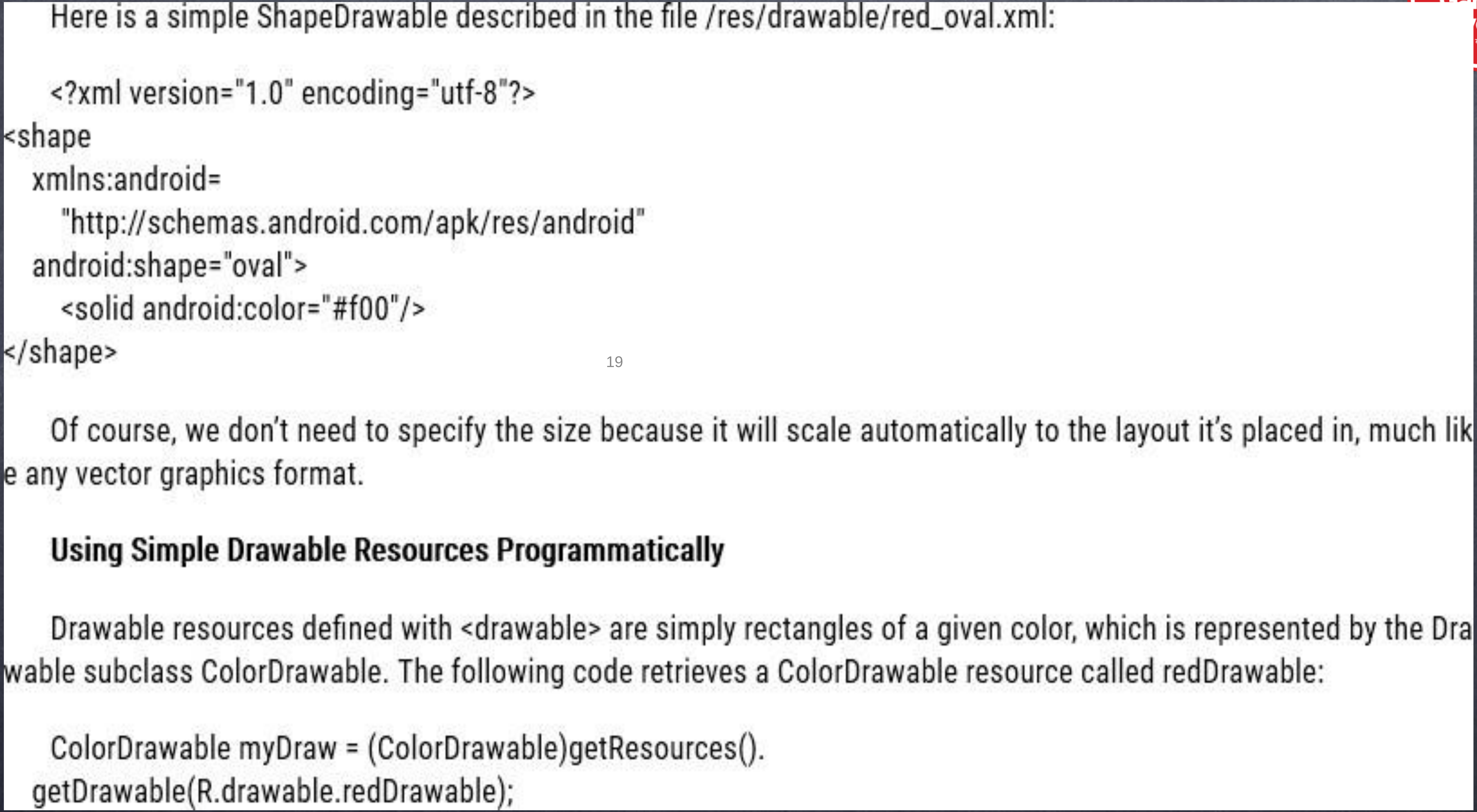
You can specify simple colored rectangles by using the drawable resource type, which can then be applied to other screen elements
. These drawable resource types are defined in specific paint colors, much as the color resources are defined.

Defining Simple Drawable Resources in XML

Simple paintable drawable resources are defined in XML under the /res/values project directory and compiled into the application p
ackage at build time. Paintable drawable resources use the <drawable> tag and represent a name/value pair. Here is an example of a si
mple drawable resource file called /res/values/drawables.xml:

```
    <?xml version="1.0" encoding="utf-8"?>
<resources>
  <drawable name="red_rect">#F00</drawable>
</resources>
```

Although it might seem a tad confusing, you can also create XML files that describe other Drawable subclasses, such as ShapeDra
wable. Drawable XML definition files are stored in the /res/drawable directory within your project, along with image files. This is not the
same as storing<drawable> resources, which are paintable drawables. ShapeDrawable resources are stored in the /res/values directory,
as explained previously.



Here is a simple ShapeDrawable described in the file /res/drawable/red_oval.xml:

```
<?xml version="1.0" encoding="utf-8"?>
<shape
  xmlns:android=
    "http://schemas.android.com/apk/res/android"
  android:shape="oval">
    <solid android:color="#f00"/>
</shape>
```

Of course, we don’t need to specify the size because it will scale automatically to the layout it’s placed in, much lik
e any vector graphics format.

Using Simple Drawable Resources Programmatically

Drawable resources defined with <drawable> are simply rectangles of a given color, which is represented by the Dra
wable subclass ColorDrawable. The following code retrieves a ColorDrawable resource called redDrawable:

```
ColorDrawable myDraw = (ColorDrawable)getResources().
  getDrawable(R.drawable.redDrawable);
```

Working with Images

Applications often include visual elements such as icons and graphics. Android supports several image formats that can be directly included as resources for your application. These image formats are shown in Table 6.7.

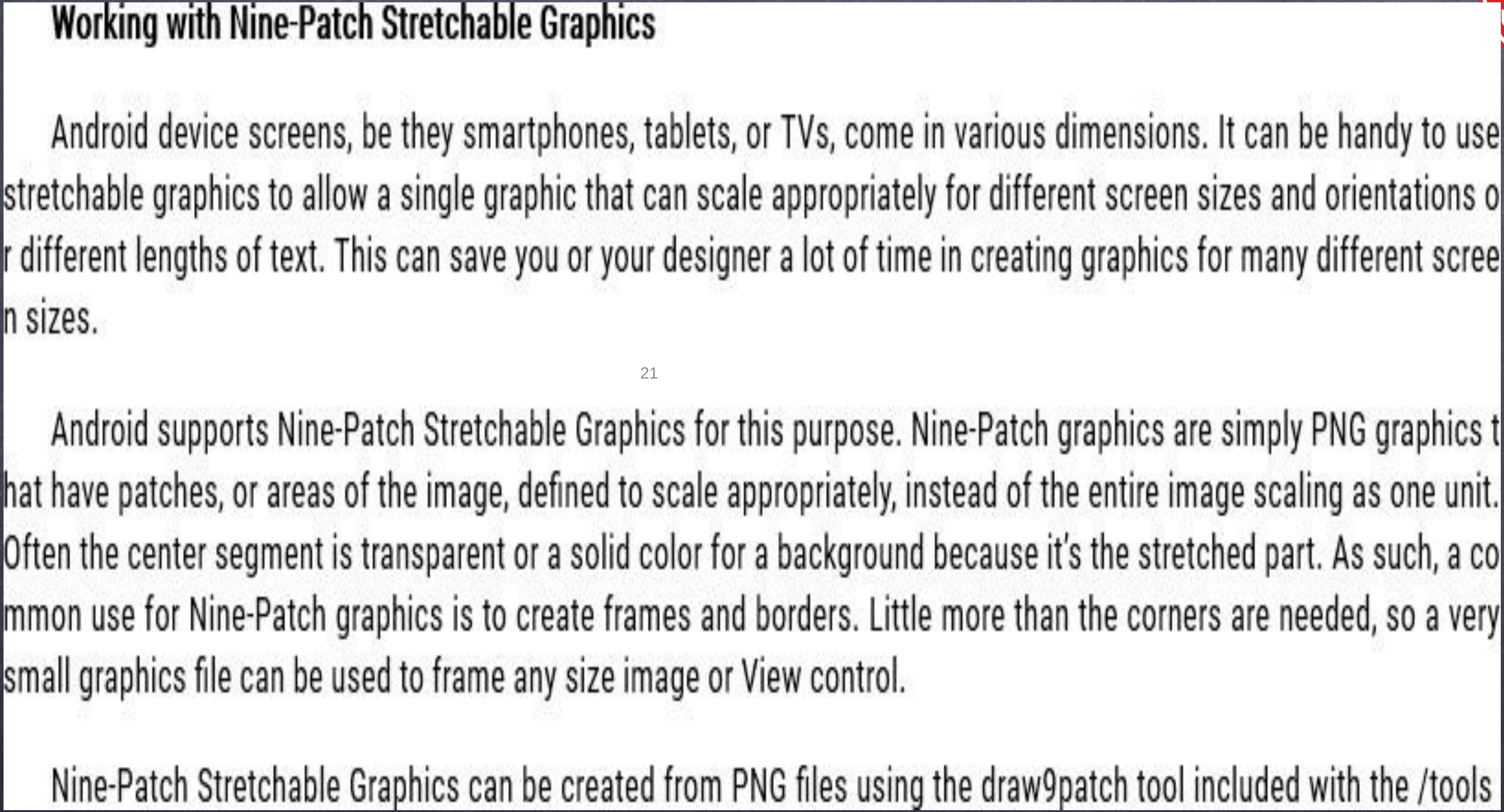
Supported Image Format	Description	Required Extension
Portable Network Graphics (PNG)	Preferred format (lossless)	`.png`
Nine-Patch Stretchable Graphics	Preferred format (lossless)	`.9.png`
Joint Photographic Experts Group (JPEG)	Acceptable format (lossy)	`.jpg, .jpeg`
Graphics Interchange Format (GIF)	Discouraged format	`.gif`
WebP (WEBP)	Android 4.0+	`.webp`

Table 6.7 Image Formats Supported in Android

These image formats are all well supported by popular graphics editors such as Adobe Photoshop, GIMP, and Microsoft Paint. Adding image resources to your project is easy. Simply drag the image asset into the /res/drawable resource directory hierarchy and it will automatically be included in the application package.

Warning

All resource filenames must be lowercase and simple (letters, numbers, and underscores only). This rule applies to all files, including graphics.

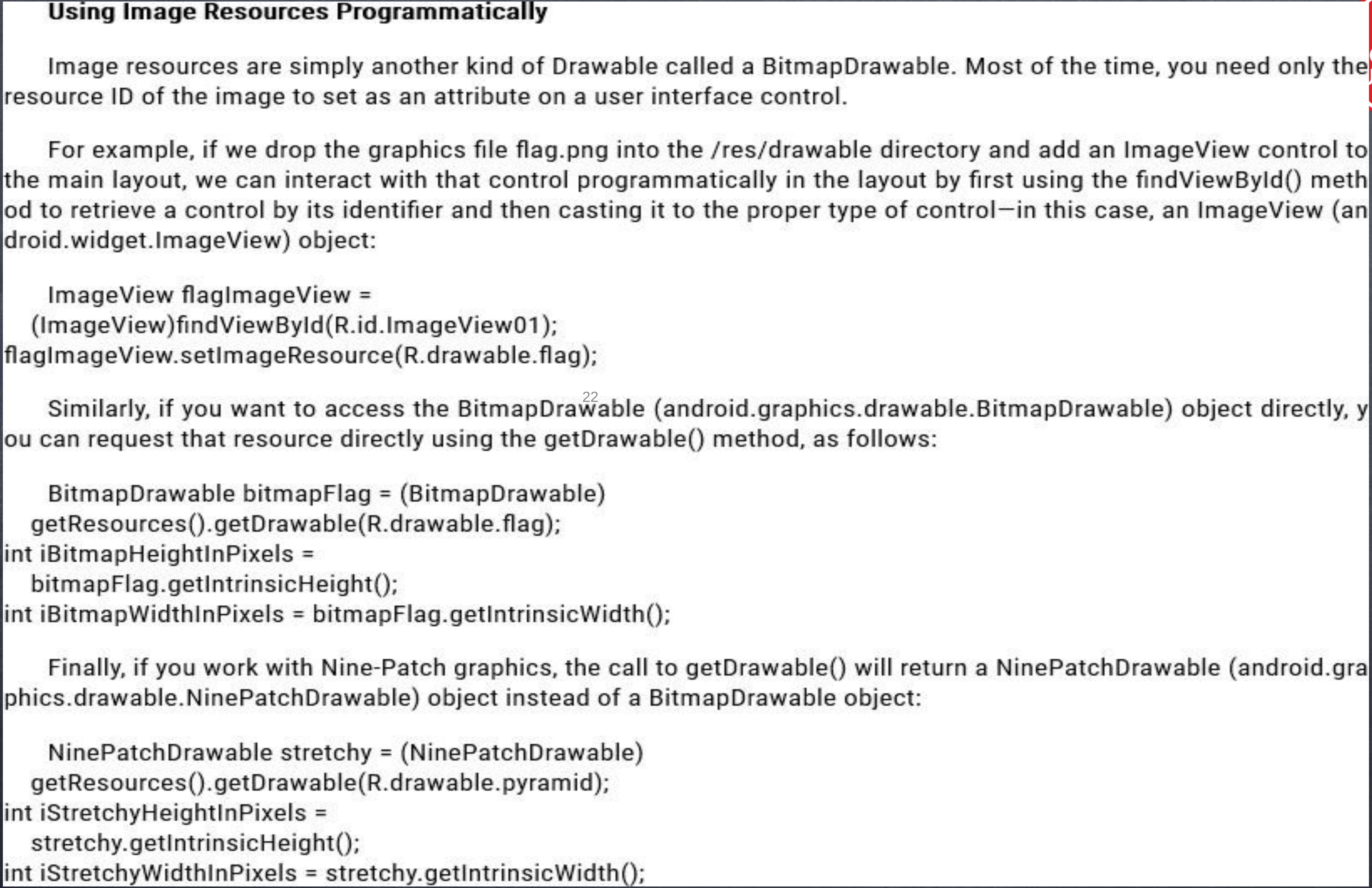


Working with Nine-Patch Stretchable Graphics

Android device screens, be they smartphones, tablets, or TVs, come in various dimensions. It can be handy to use stretchable graphics to allow a single graphic that can scale appropriately for different screen sizes and orientations or different lengths of text. This can save you or your designer a lot of time in creating graphics for many different screen sizes.

Android supports Nine-Patch Stretchable Graphics for this purpose. Nine-Patch graphics are simply PNG graphics that have patches, or areas of the image, defined to scale appropriately, instead of the entire image scaling as one unit. Often the center segment is transparent or a solid color for a background because it's the stretched part. As such, a common use for Nine-Patch graphics is to create frames and borders. Little more than the corners are needed, so a very small graphics file can be used to frame any size image or View control.

Nine-Patch Stretchable Graphics can be created from PNG files using the draw9patch tool included with the /tools



Using Image Resources Programmatically

Image resources are simply another kind of Drawable called a BitmapDrawable. Most of the time, you need only the resource ID of the image to set as an attribute on a user interface control.

For example, if we drop the graphics file flag.png into the /res/drawable directory and add an ImageView control to the main layout, we can interact with that control programmatically in the layout by first using the findViewById() meth
od to retrieve a control by its identifier and then casting it to the proper type of control—in this case, an ImageView (an
droid.widget.ImageView) object:

```
ImageView flagImageView =
   (ImageView)findViewById(R.id.ImageView01);
flagImageView.setImageResource(R.drawable.flag);
```

Similarly, if you want to access the BitmapDrawable (android.graphics.drawable.BitmapDrawable) object directly, y
ou can request that resource directly using the getDrawable() method, as follows:

```
BitmapDrawable bitmapFlag = (BitmapDrawable)
   getResources().getDrawable(R.drawable.flag);
int iBitmapHeightInPixels =
   bitmapFlag.getIntrinsicHeight();
int iBitmapWidthInPixels = bitmapFlag.getIntrinsicWidth();
```

Finally, if you work with Nine-Patch graphics, the call to getDrawable() will return a NinePatchDrawable (android.gra
phics.drawable.NinePatchDrawable) object instead of a BitmapDrawable object:

```
NinePatchDrawable stretchy = (NinePatchDrawable)
   getResources().getDrawable(R.drawable.pyramid);
int iStretchyHeightInPixels =
   stretchy.getIntrinsicHeight();
int iStretchyWidthInPixels = stretchy.getIntrinsicWidth();
```

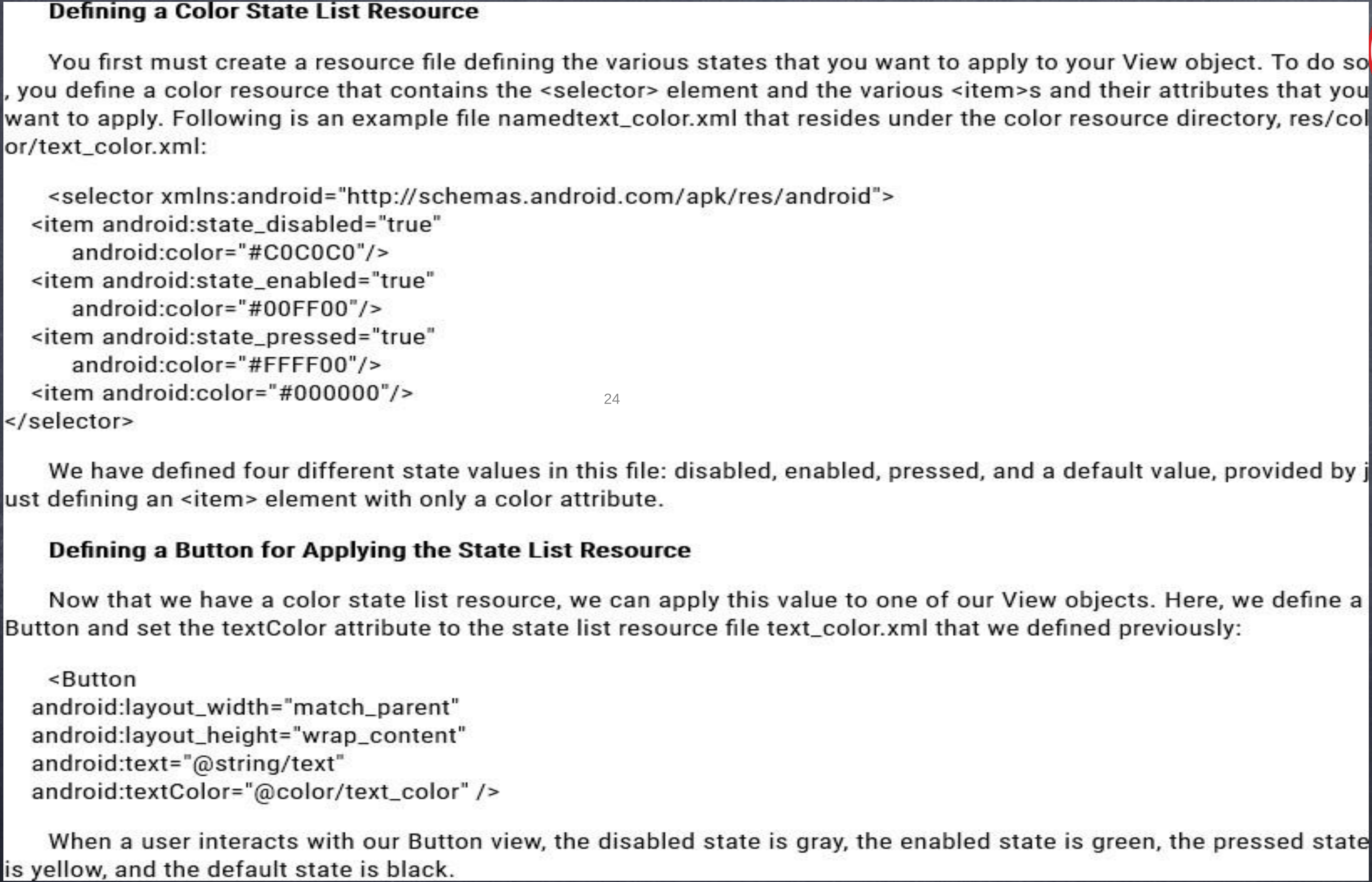
Working with Color State Lists

A special resource type called <selector> can be used to define different colors or drawables to be used depending
on a control's state. For example, you could define a color state list for a Button control: gray when the button is disabl
ed, green when it is enabled, and yellow when it is being pressed. Similarly, you could provide different drawables base
d on the state of an ImageButton control.

The <selector> element can have one or more child <item> elements that define different colors for different states
. There are quite a few attributes that you are able to define for the <item> element, and you can define one or more for
supporting many different states for your View objects.Table 6.8 shows many of the attributes that you are able to defi
ne for the <item> element.

Attribute	Values
`color`	Required attribute for specifying a hexadecimal color in one of the following formats: `#RGB`, `#ARGB`, `#RRGGBB`, or `#AARRGGBB`, where `A` is alpha, `R` is red, `G` is green, and `B` is blue
`state_enabled`	`Boolean` value denoting whether this object is capable of receiving touch or click events, `true` or `false`
`state_checked`	`Boolean` value denoting whether this object is checked or unchecked, `true` or `false`
`state_checkable`	`Boolean` value denoting whether this object is checkable or not checkable, `true` or `false`
`state_selected`	`Boolean` value denoting whether this object is selected or not selected, `true` or `false`
`state_focused`	`Boolean` value denoting whether this object is focused or not focused, `true` or `false`
`state_pressed`	`Boolean` value denoting whether this object is pressed or not pressed, `true` or `false`

Table 6.8 Color State List <item> Attributes



Defining a Color State List Resource

You first must create a resource file defining the various states that you want to apply to your View object. To do so , you define a color resource that contains the <selector> element and the various <item>s and their attributes that you want to apply. Following is an example file namedtext_color.xml that resides under the color resource directory, res/col or/text_color.xml:

```
<selector xmlns:android="http://schemas.android.com/apk/res/android">
  <item android:state_disabled="true"
      android:color="#C0C0C0"/>
  <item android:state_enabled="true"
      android:color="#00FF00"/>
  <item android:state_pressed="true"
      android:color="#FFFF00"/>
  <item android:color="#000000"/>
</selector>
```

We have defined four different state values in this file: disabled, enabled, pressed, and a default value, provided by j ust defining an <item> element with only a color attribute.

Defining a Button for Applying the State List Resource

Now that we have a color state list resource, we can apply this value to one of our View objects. Here, we define a Button and set the textColor attribute to the state list resource file text_color.xml that we defined previously:

```
<Button
  android:layout_width="match_parent"
  android:layout_height="wrap_content"
  android:text="@string/text"
  android:textColor="@color/text_color" />
```

When a user interacts with our Button view, the disabled state is gray, the enabled state is green, the pressed state is yellow, and the default state is black.

Working with Animation

Android provides two categories of animations. The first category, property animation, allows you to animate an ob
ject's properties. The second category of animation is view animation. There are two different types of view animation
s: *frame-by-frame* animation and *tween* animations.

Frame-by-frame animation involves the display of a sequence of images in rapid succession. Tweened animation in
volves applying standard graphical transformations such as rotations and fades to a single image.

The Android SDK provides some helper utilities for loading and using animation resources. These utilities are foun
d in the android.view.animation.AnimationUtils class. Let's look at how you define the different view animations in ter
ms of resources.

Defining and Using Frame-by-Frame Animation Resources

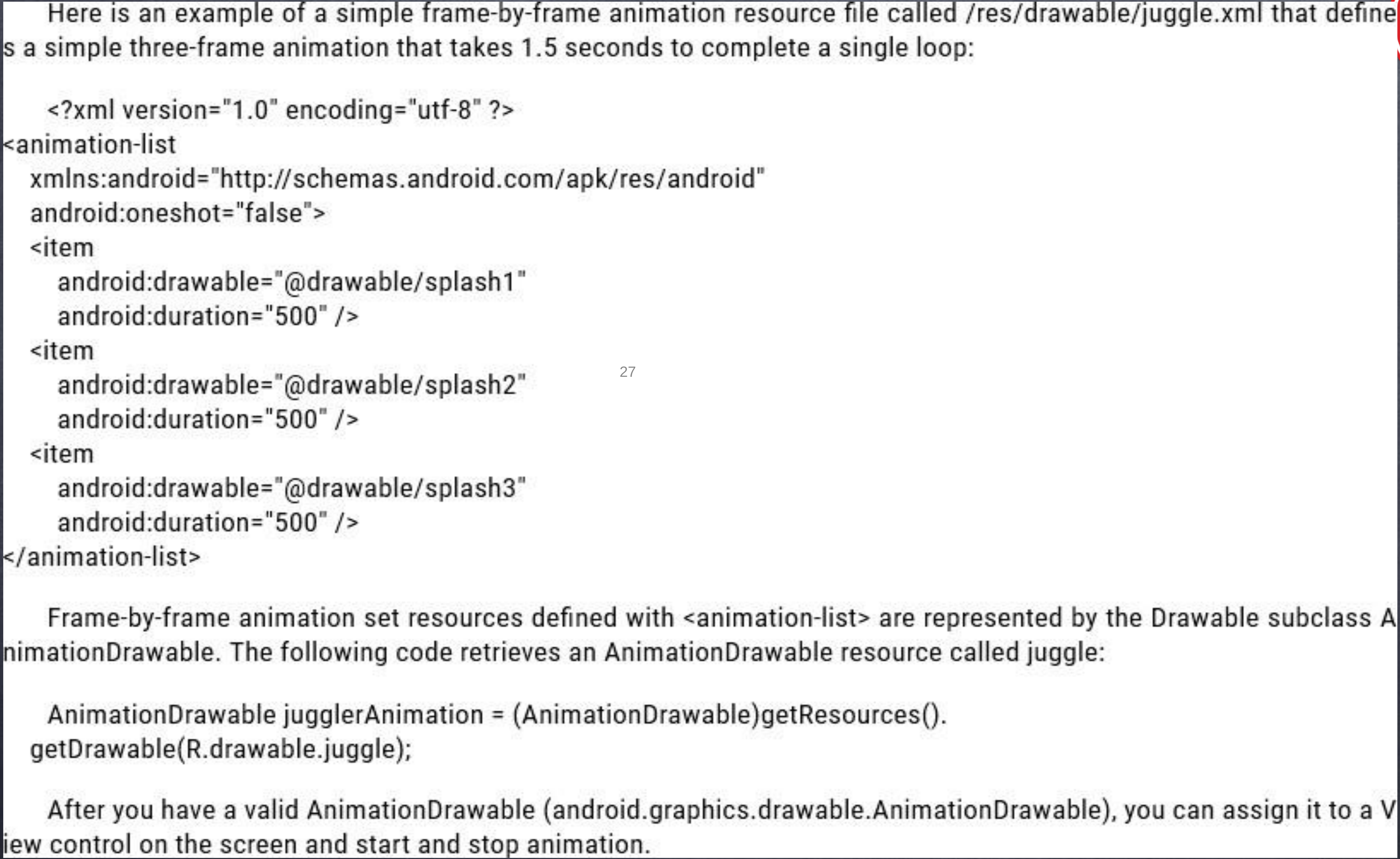
Frame-by-frame animation is often used when the content changes from frame to frame. This type of animation ca
n be used for complex frame transitions—much like a kid's flip book.

To define frame-by-frame resources, take the following steps:

1. Save each frame graphic as an individual drawable resource. It may help to name your graphics sequentially, in t
he order in which they are displayed—for example, frame1.png, frame2.png, and so on.

2. Define the animation set resource in an XML file within the /res/drawable/ resource directory hierarchy.

3. Load, start, and stop the animation programmatically.



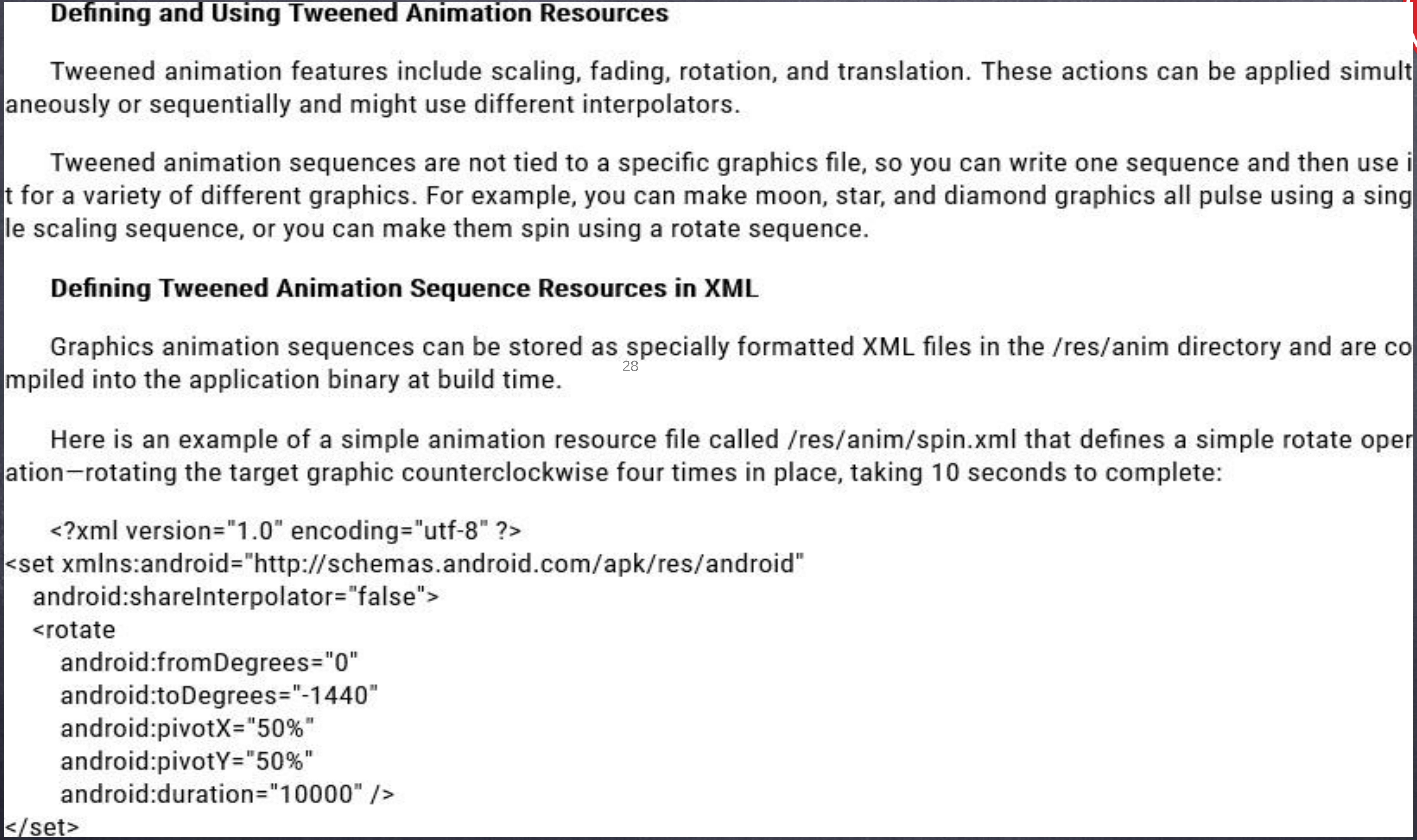
Here is an example of a simple frame-by-frame animation resource file called /res/drawable/juggle.xml that define
s a simple three-frame animation that takes 1.5 seconds to complete a single loop:

```
   <?xml version="1.0" encoding="utf-8" ?>
<animation-list
  xmlns:android="http://schemas.android.com/apk/res/android"
  android:oneshot="false">
  <item
    android:drawable="@drawable/splash1"
    android:duration="500" />
  <item
    android:drawable="@drawable/splash2"
    android:duration="500" />
  <item
    android:drawable="@drawable/splash3"
    android:duration="500" />
</animation-list>
```

Frame-by-frame animation set resources defined with <animation-list> are represented by the Drawable subclass A
nimationDrawable. The following code retrieves an AnimationDrawable resource called juggle:

```
   AnimationDrawable jugglerAnimation = (AnimationDrawable)getResources().
  getDrawable(R.drawable.juggle);
```

After you have a valid AnimationDrawable (android.graphics.drawable.AnimationDrawable), you can assign it to a V
iew control on the screen and start and stop animation.



Defining and Using Tweened Animation Resources

Tweened animation features include scaling, fading, rotation, and translation. These actions can be applied simultaneously or sequentially and might use different interpolators.

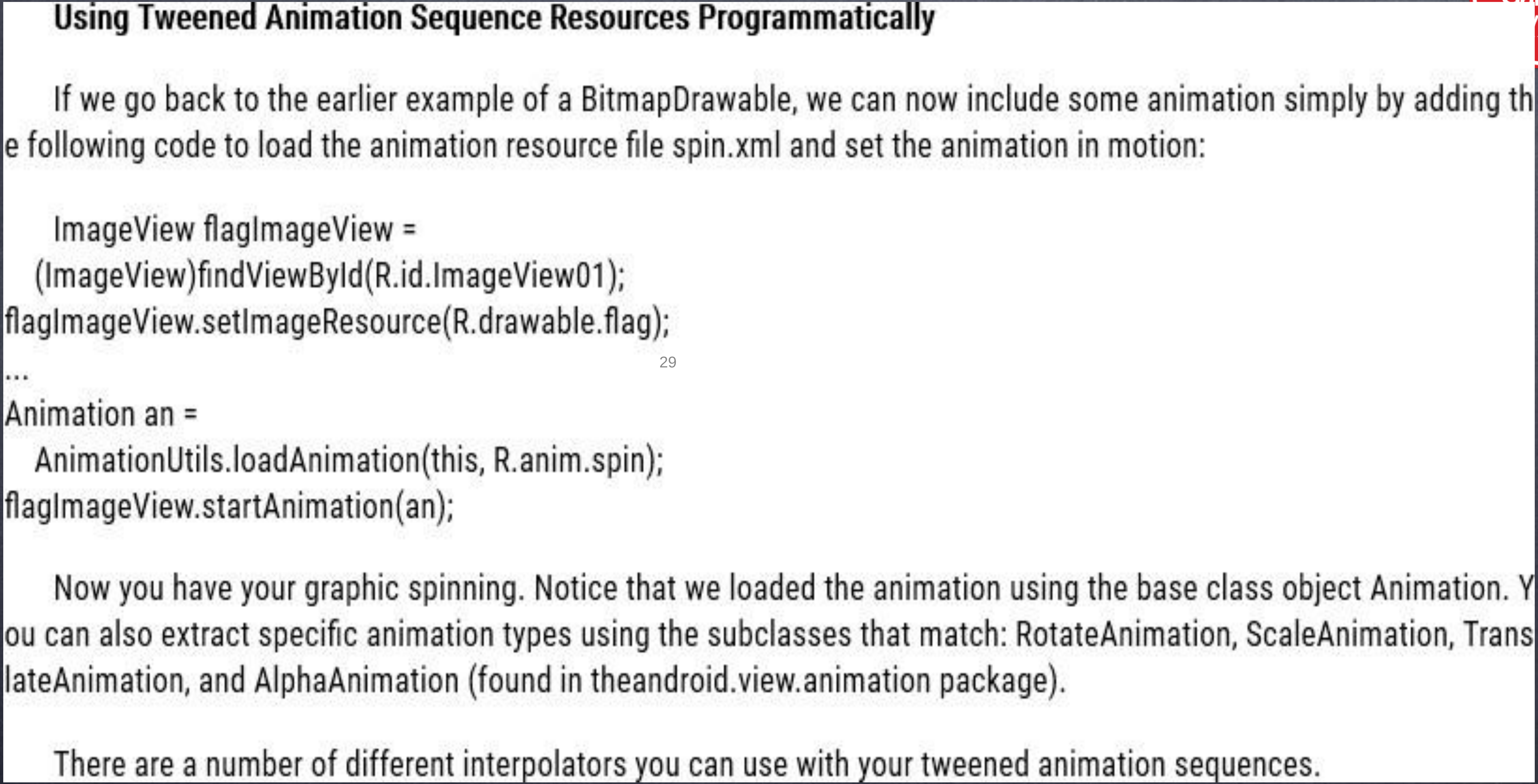
Tweened animation sequences are not tied to a specific graphics file, so you can write one sequence and then use it for a variety of different graphics. For example, you can make moon, star, and diamond graphics all pulse using a single scaling sequence, or you can make them spin using a rotate sequence.

Defining Tweened Animation Sequence Resources in XML

Graphics animation sequences can be stored as specially formatted XML files in the /res/anim directory and are compiled into the application binary at build time.

Here is an example of a simple animation resource file called /res/anim/spin.xml that defines a simple rotate operation—rotating the target graphic counterclockwise four times in place, taking 10 seconds to complete:

```
<?xml version="1.0" encoding="utf-8" ?>
<set xmlns:android="http://schemas.android.com/apk/res/android"
  android:shareInterpolator="false">
  <rotate
    android:fromDegrees="0"
    android:toDegrees="-1440"
    android:pivotX="50%"
    android:pivotY="50%"
    android:duration="10000" />
</set>
```



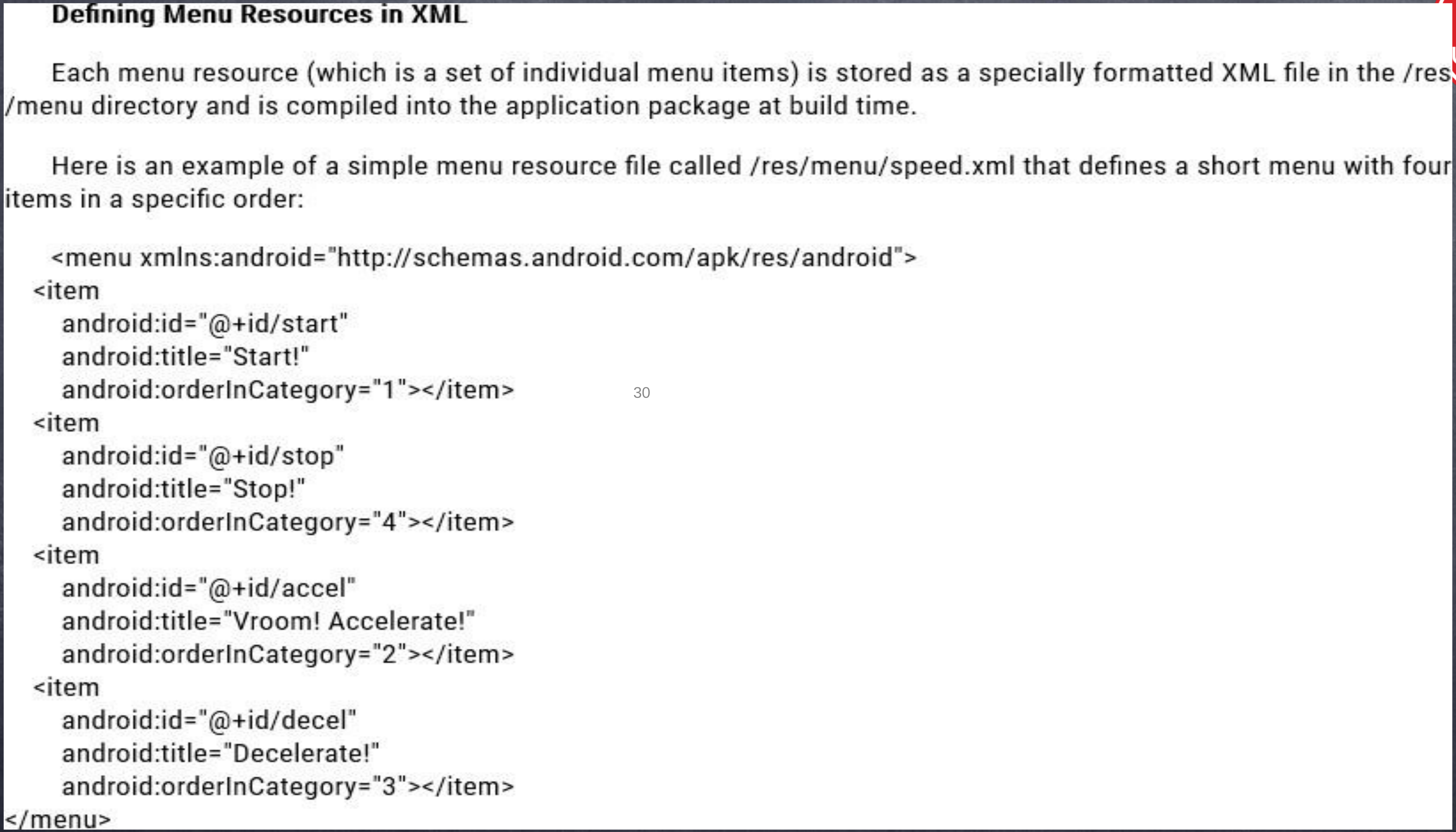
Using Tweened Animation Sequence Resources Programmatically

If we go back to the earlier example of a BitmapDrawable, we can now include some animation simply by adding th
e following code to load the animation resource file spin.xml and set the animation in motion:

```
  ImageView flagImageView =
 (ImageView)findViewById(R.id.ImageView01);
flagImageView.setImageResource(R.drawable.flag);
...
Animation an =
 AnimationUtils.loadAnimation(this, R.anim.spin);
flagImageView.startAnimation(an);
```

Now you have your graphic spinning. Notice that we loaded the animation using the base class object Animation. Y
ou can also extract specific animation types using the subclasses that match: RotateAnimation, ScaleAnimation, Trans
lateAnimation, and AlphaAnimation (found in theandroid.view.animation package).

There are a number of different interpolators you can use with your tweened animation sequences.

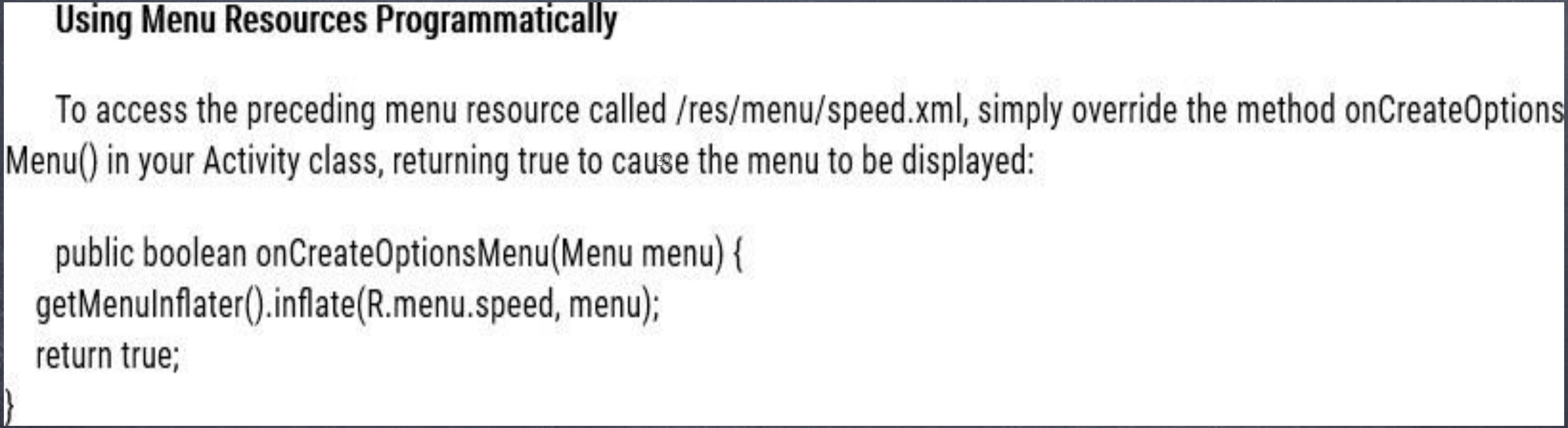


Defining Menu Resources in XML

Each menu resource (which is a set of individual menu items) is stored as a specially formatted XML file in the /res /menu directory and is compiled into the application package at build time.

Here is an example of a simple menu resource file called /res/menu/speed.xml that defines a short menu with four items in a specific order:

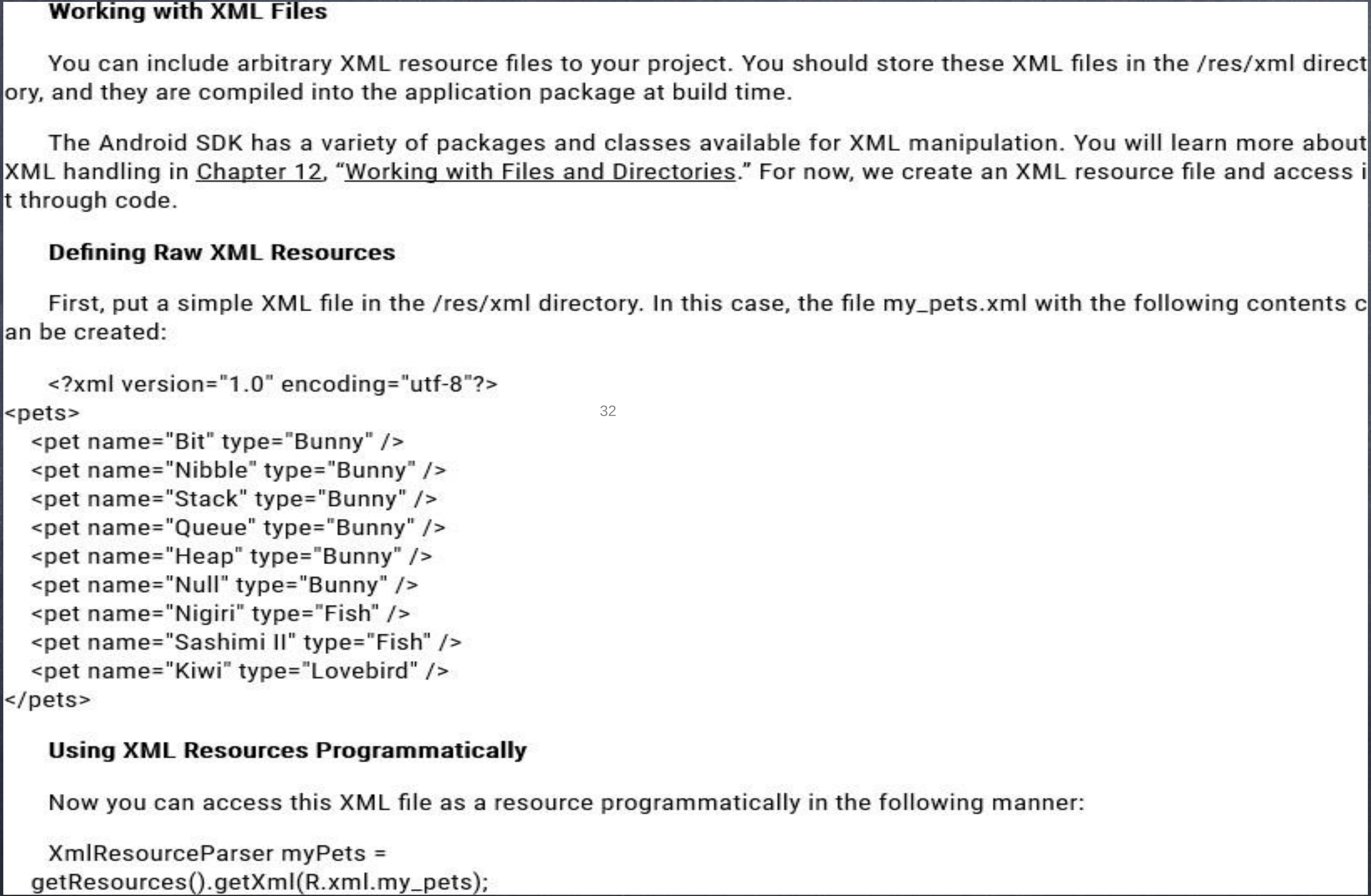
```
  <menu xmlns:android="http://schemas.android.com/apk/res/android">
 <item
   android:id="@+id/start"
   android:title="Start!"
   android:orderInCategory="1"></item>
 <item
   android:id="@+id/stop"
   android:title="Stop!"
   android:orderInCategory="4"></item>
 <item
   android:id="@+id/accel"
   android:title="Vroom! Accelerate!"
   android:orderInCategory="2"></item>
 <item
   android:id="@+id/decel"
   android:title="Decelerate!"
   android:orderInCategory="3"></item>
</menu>
```



Using Menu Resources Programmatically

To access the preceding menu resource called /res/menu/speed.xml, simply override the method onCreateOptions Menu() in your Activity class, returning true to cause the menu to be displayed:

```
public boolean onCreateOptionsMenu(Menu menu) {
getMenuInflater().inflate(R.menu.speed, menu);
return true;
}
```



Working with XML Files

You can include arbitrary XML resource files to your project. You should store these XML files in the /res/xml direct
ory, and they are compiled into the application package at build time.

The Android SDK has a variety of packages and classes available for XML manipulation. You will learn more about XML handling in Chapter 12, "Working with Files and Directories." For now, we create an XML resource file and access i
t through code.

Defining Raw XML Resources

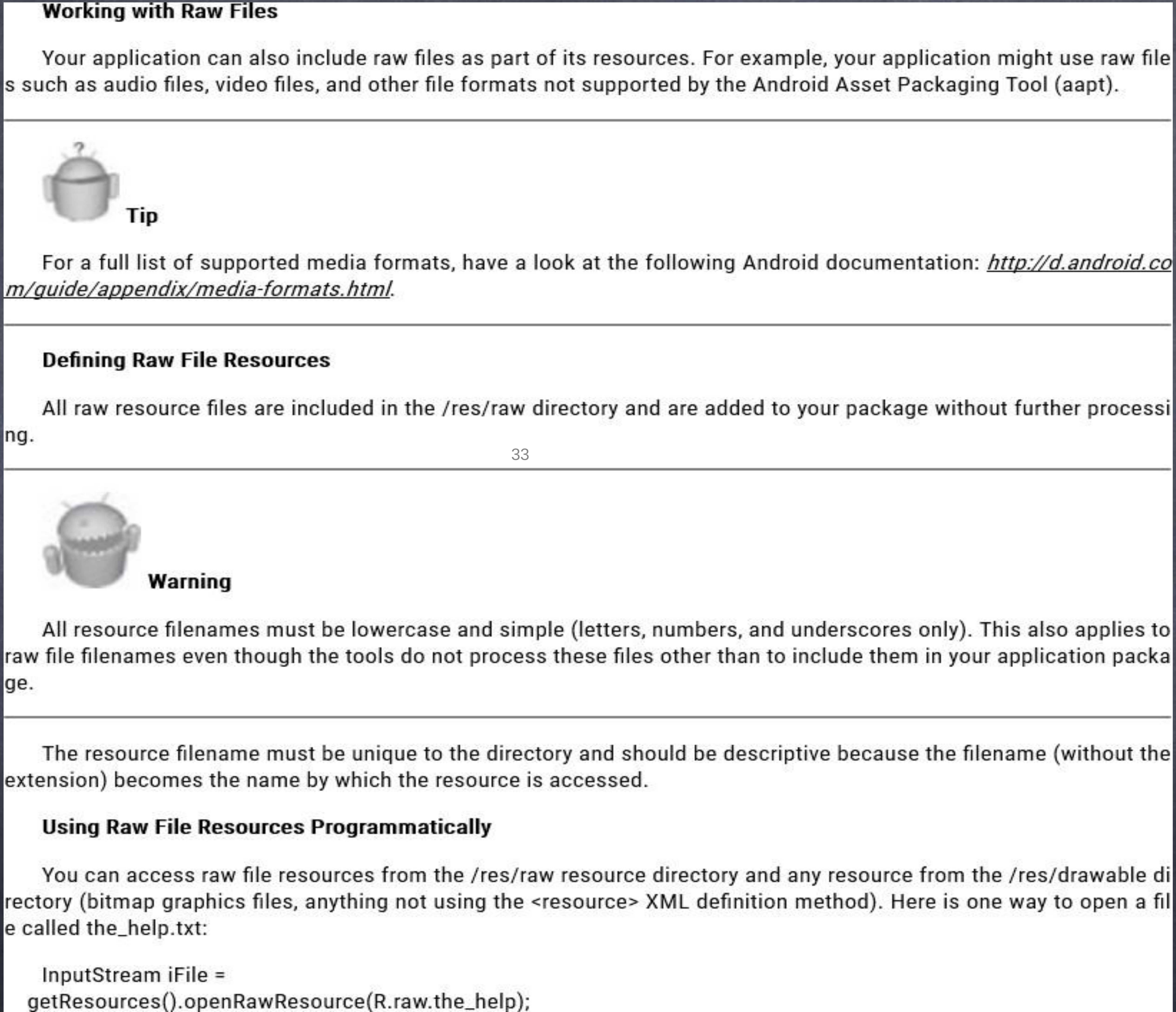
First, put a simple XML file in the /res/xml directory. In this case, the file my_pets.xml with the following contents c
an be created:

```
    <?xml version="1.0" encoding="utf-8"?>
<pets>
   <pet name="Bit" type="Bunny" />
   <pet name="Nibble" type="Bunny" />
   <pet name="Stack" type="Bunny" />
   <pet name="Queue" type="Bunny" />
   <pet name="Heap" type="Bunny" />
   <pet name="Null" type="Bunny" />
   <pet name="Nigiri" type="Fish" />
   <pet name="Sashimi II" type="Fish" />
   <pet name="Kiwi" type="Lovebird" />
</pets>
```

Using XML Resources Programmatically

Now you can access this XML file as a resource programmatically in the following manner:

```
    XmlResourceParser myPets =
   getResources().getXml(R.xml.my_pets);
```



Working with Raw Files

Your application can also include raw files as part of its resources. For example, your application might use raw file
s such as audio files, video files, and other file formats not supported by the Android Asset Packaging Tool (aapt).

Tip

For a full list of supported media formats, have a look at the following Android documentation: *http://d.android.co
m/guide/appendix/media-formats.html*.

Defining Raw File Resources

All raw resource files are included in the /res/raw directory and are added to your package without further processi
ng.

Warning

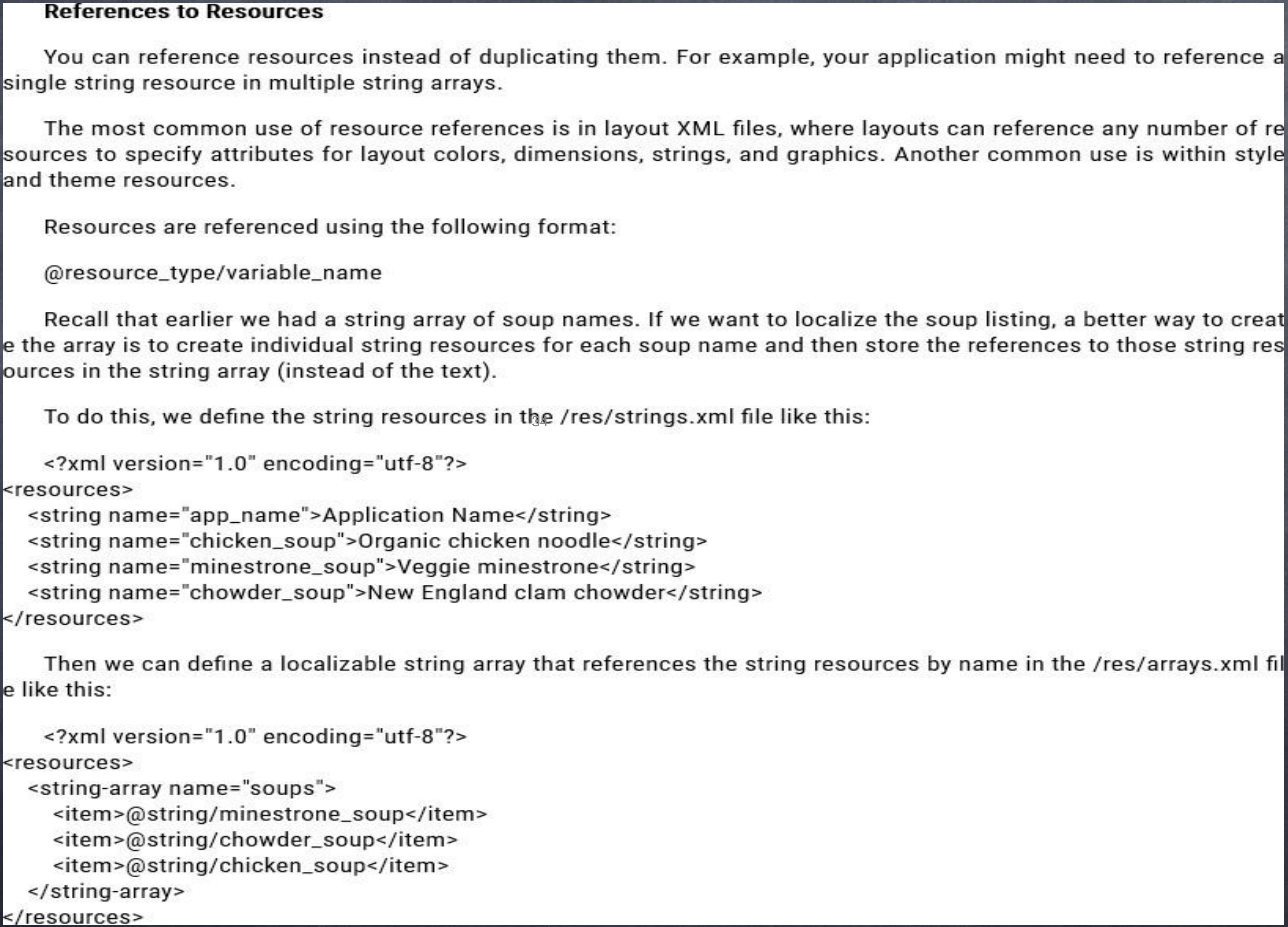
All resource filenames must be lowercase and simple (letters, numbers, and underscores only). This also applies to
raw file filenames even though the tools do not process these files other than to include them in your application packa
ge.

The resource filename must be unique to the directory and should be descriptive because the filename (without the
extension) becomes the name by which the resource is accessed.

Using Raw File Resources Programmatically

You can access raw file resources from the /res/raw resource directory and any resource from the /res/drawable di
rectory (bitmap graphics files, anything not using the <resource> XML definition method). Here is one way to open a fil
e called the_help.txt:

```
InputStream iFile =
getResources().openRawResource(R.raw.the_help);
```



References to Resources

You can reference resources instead of duplicating them. For example, your application might need to reference a single string resource in multiple string arrays.

The most common use of resource references is in layout XML files, where layouts can reference any number of resources to specify attributes for layout colors, dimensions, strings, and graphics. Another common use is within style and theme resources.

Resources are referenced using the following format:

@resource_type/variable_name

Recall that earlier we had a string array of soup names. If we want to localize the soup listing, a better way to create the array is to create individual string resources for each soup name and then store the references to those string resources in the string array (instead of the text).

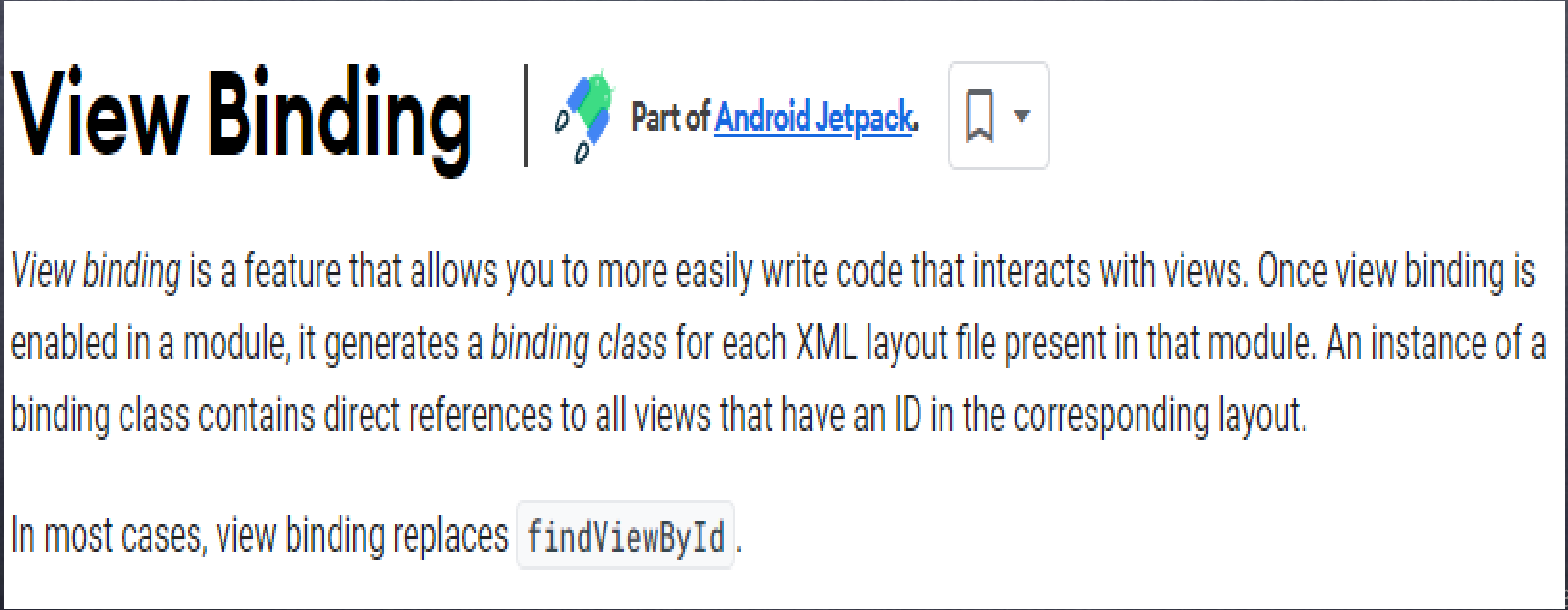
To do this, we define the string resources in the /res/strings.xml file like this:

```
    <?xml version="1.0" encoding="utf-8"?>
<resources>
  <string name="app_name">Application Name</string>
  <string name="chicken_soup">Organic chicken noodle</string>
  <string name="minestrone_soup">Veggie minestrone</string>
  <string name="chowder_soup">New England clam chowder</string>
</resources>
```

Then we can define a localizable string array that references the string resources by name in the /res/arrays.xml file like this:

```
    <?xml version="1.0" encoding="utf-8"?>
<resources>
  <string-array name="soups">
    <item>@string/minestrone_soup</item>
    <item>@string/chowder_soup</item>
    <item>@string/chicken_soup</item>
  </string-array>
</resources>
```

View Binding



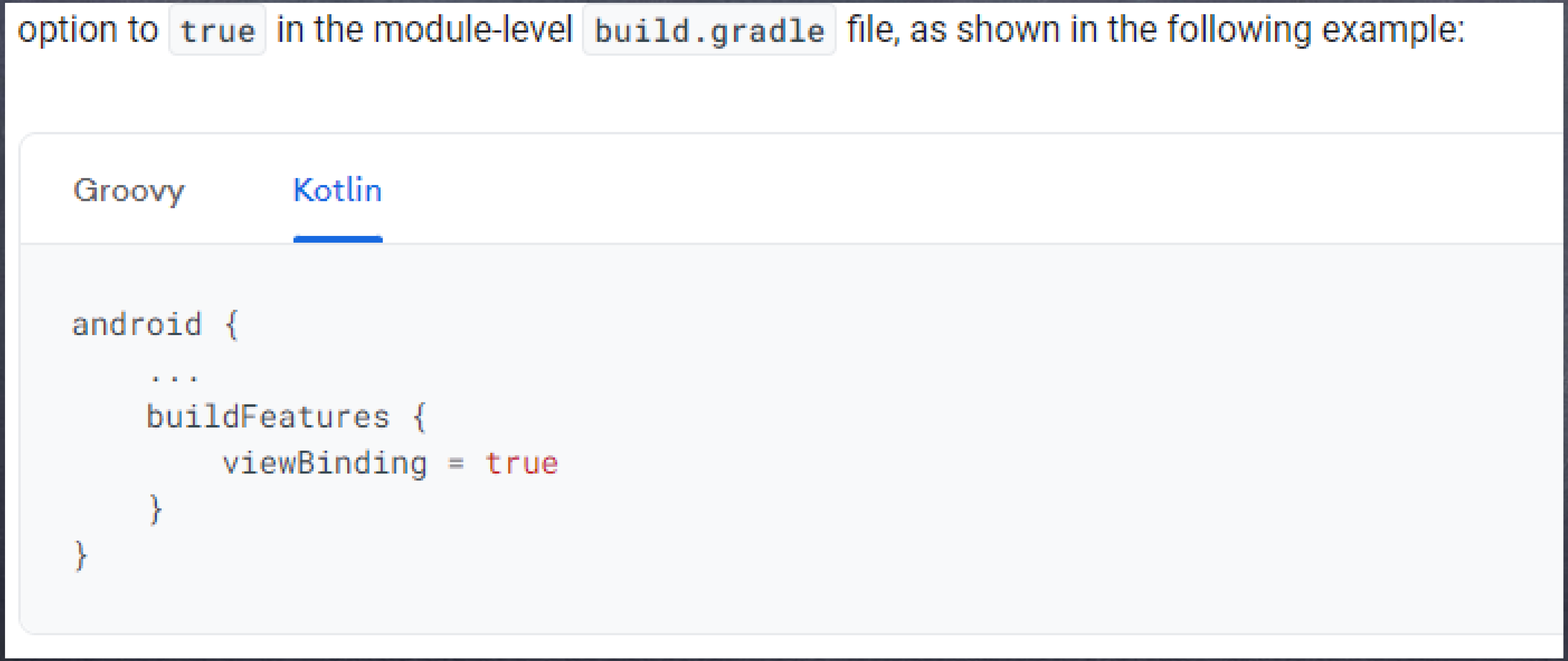
View Binding

Part of Android Jetpack.

View binding is a feature that allows you to more easily write code that interacts with views. Once view binding is enabled in a module, it generates a *binding class* for each XML layout file present in that module. An instance of a binding class contains direct references to all views that have an ID in the corresponding layout.

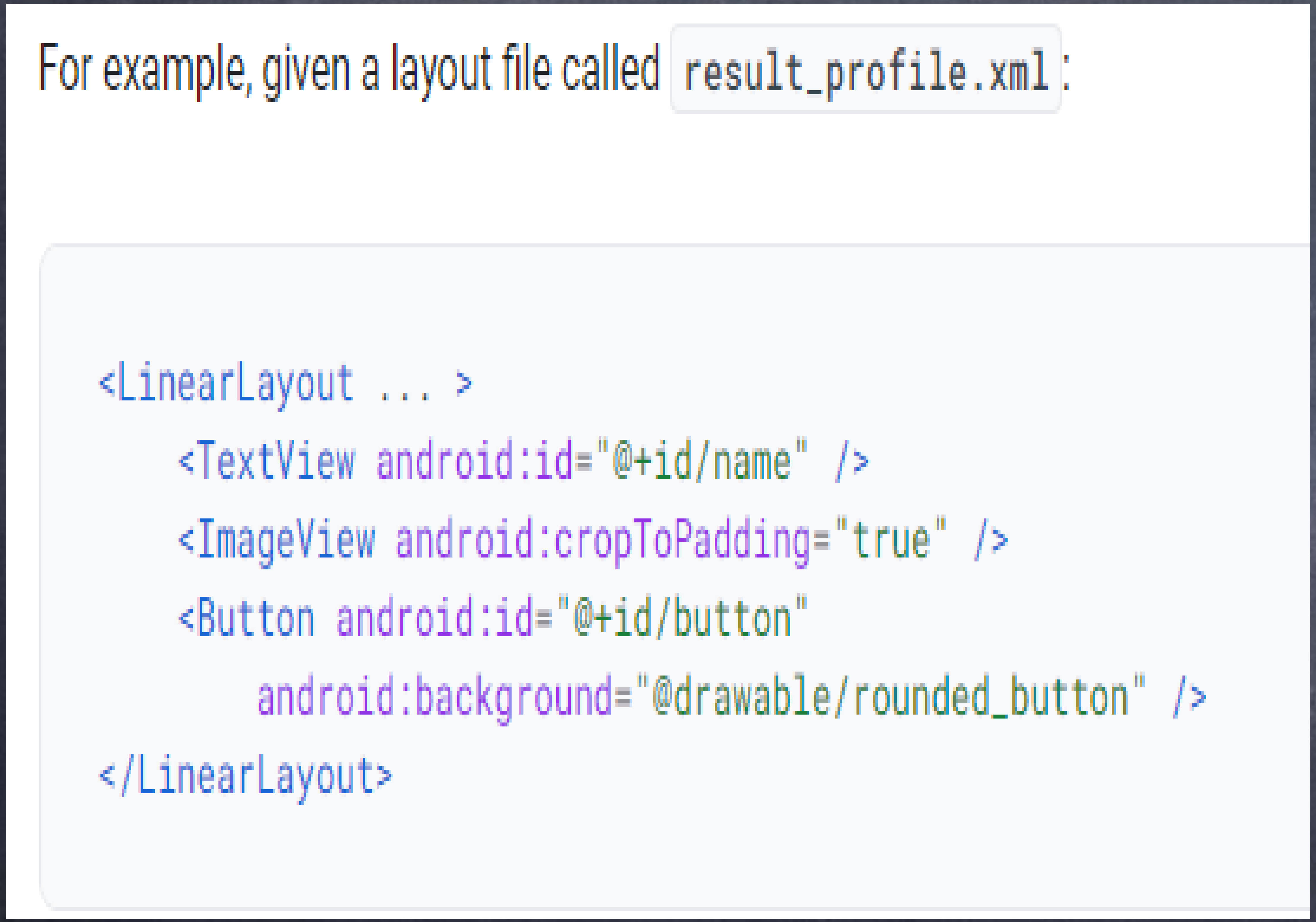
In most cases, view binding replaces `findViewById`.

View Binding


option to `true` in the module-level `build.gradle` file, as shown in the following example:

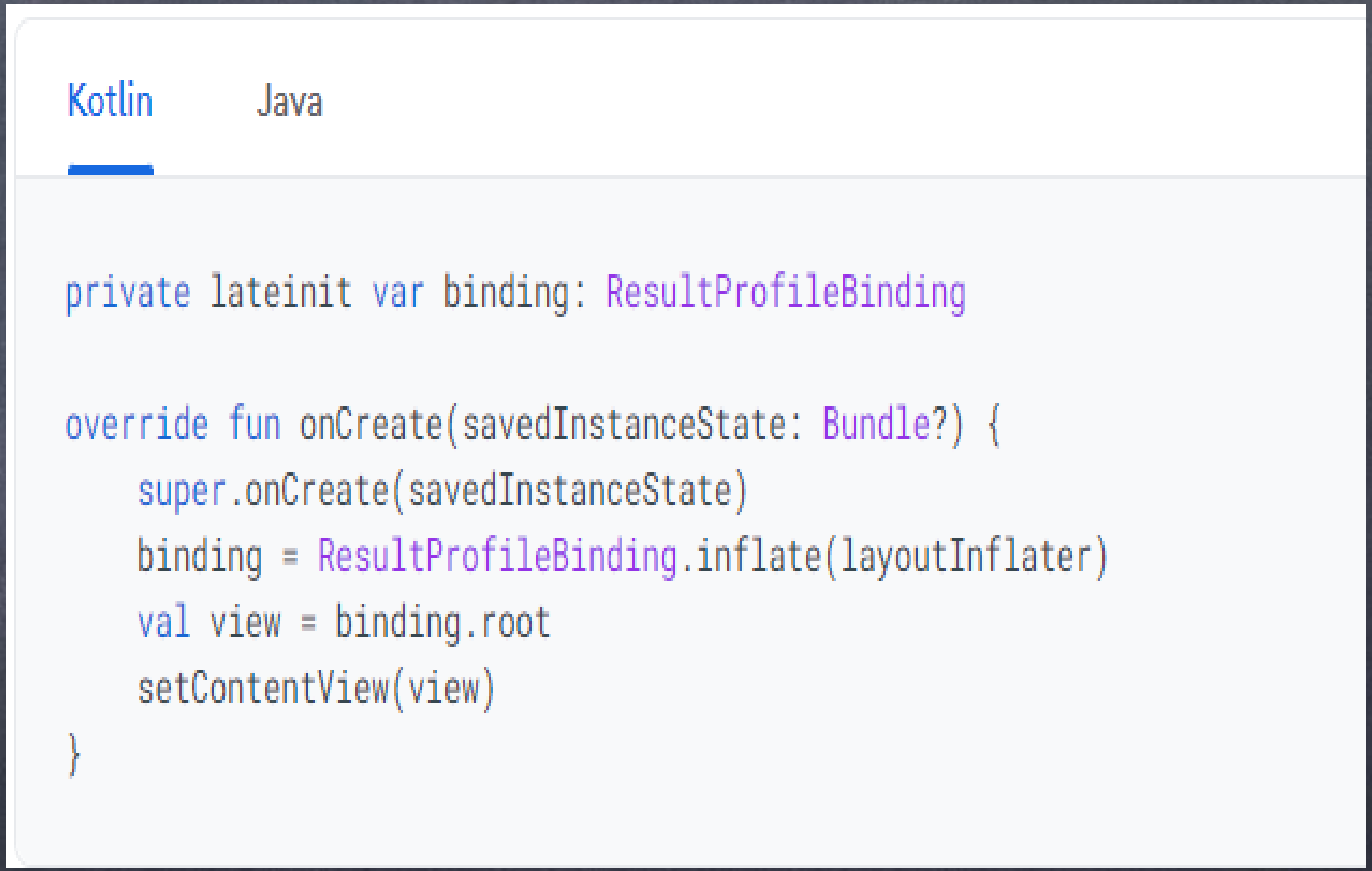
```
android {
    ...
    buildFeatures {
        viewBinding = true
    }
}
```

View Binding



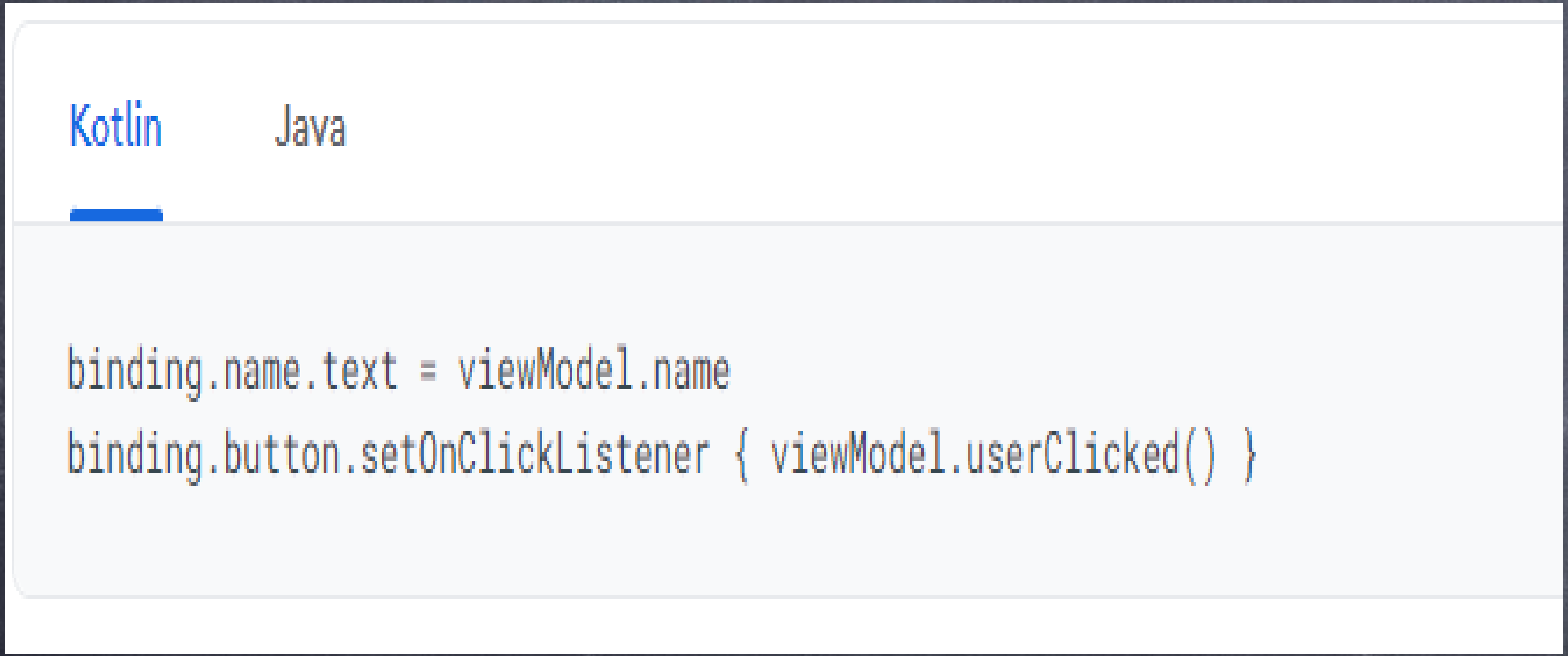
For example, given a layout file called `result_profile.xml`:

```xml
<LinearLayout ... >
    <TextView android:id="@+id/name" />
    <ImageView android:cropToPadding="true" />
    <Button android:id="@+id/button"
        android:background="@drawable/rounded_button" />
</LinearLayout>
```



```kotlin
private lateinit var binding: ResultProfileBinding

override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    binding = ResultProfileBinding.inflate(layoutInflater)
    val view = binding.root
    setContentView(view)
}
```

View Binding



```
binding.name.text = viewModel.name
binding.button.setOnClickListener { viewModel.userClicked() }
```

Android Resources:

What Are Resources?

- All Android applications are composed of two things: functionality (code instructions) and data (resources). The functionality is the code that determines how your application behaves. This includes any algorithms that make the application run. Resources include text strings, styles and themes, dimensions, images and icons, audio files, videos, and other data used by the application.

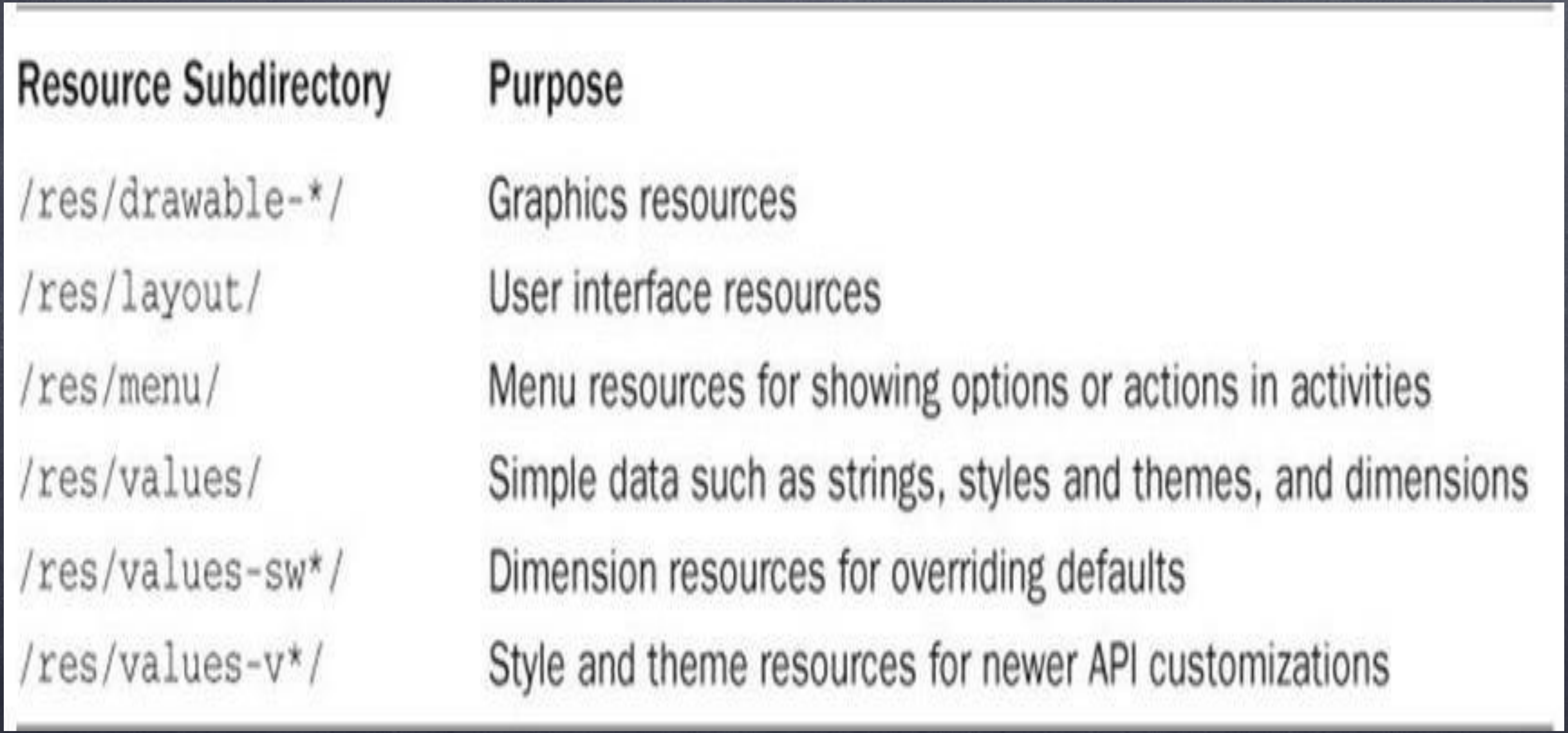
Storing Application Resources

- Android resource files are stored separately from the .java/.kotlin class files in the Android project. Most common resource types are stored in XML.
- You can also store raw data files and graphics as resources. Resources are organized in a strict directory hierarchy.
- All resources must be stored under the /res project directory in specially named subdirectories whose names must be lowercase.
- Different resource types are stored in different directories. The resource subdirectories generated when you create an Android project

Storing Application Resources

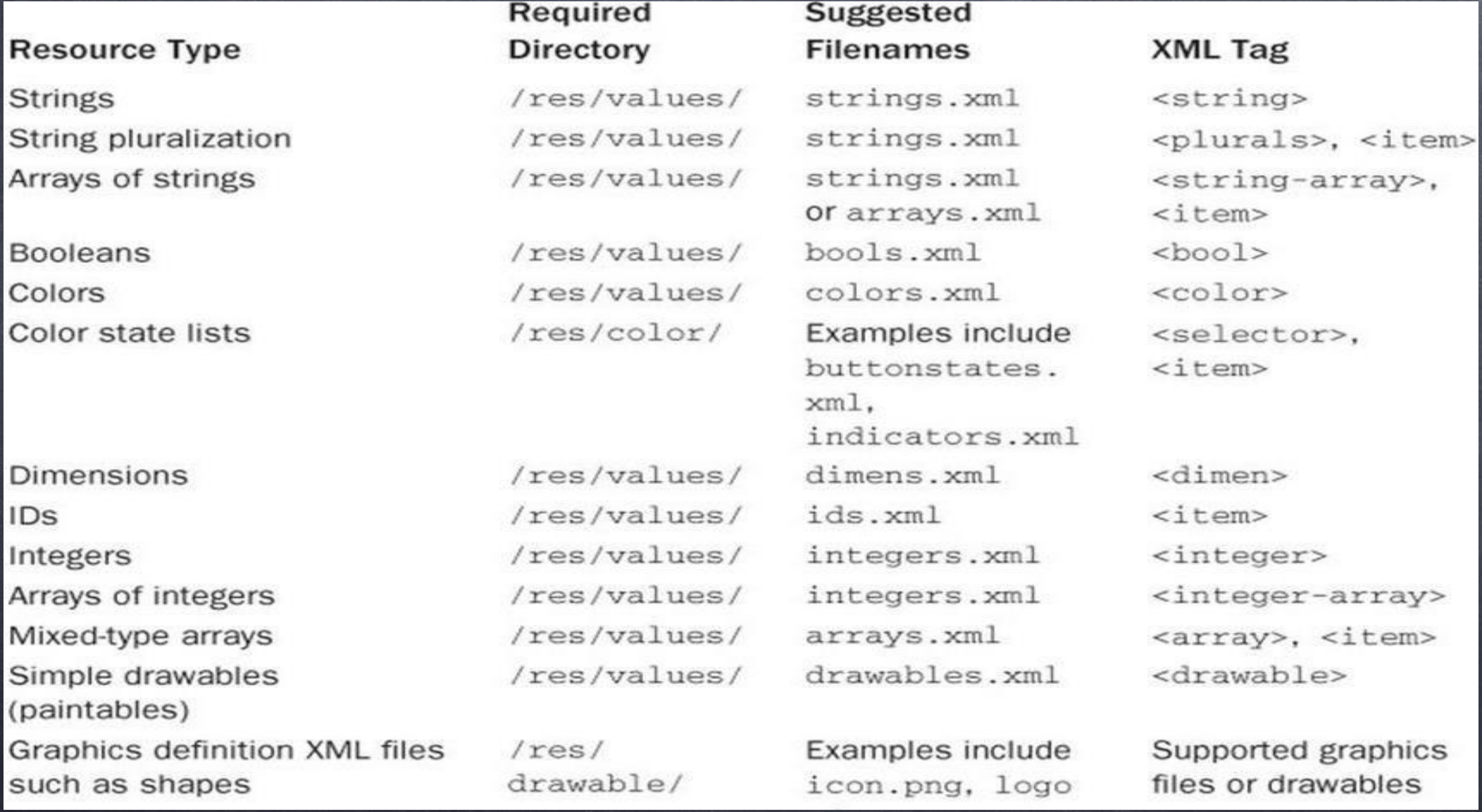
```
MyProject/
    src/
        MyActivity.java
    res/
        drawable/
            graphic.png
        layout/
            main.xml
            info.xml
        mipmap/
            icon.png
        values/
            strings.xml
```

Storing Application Resources



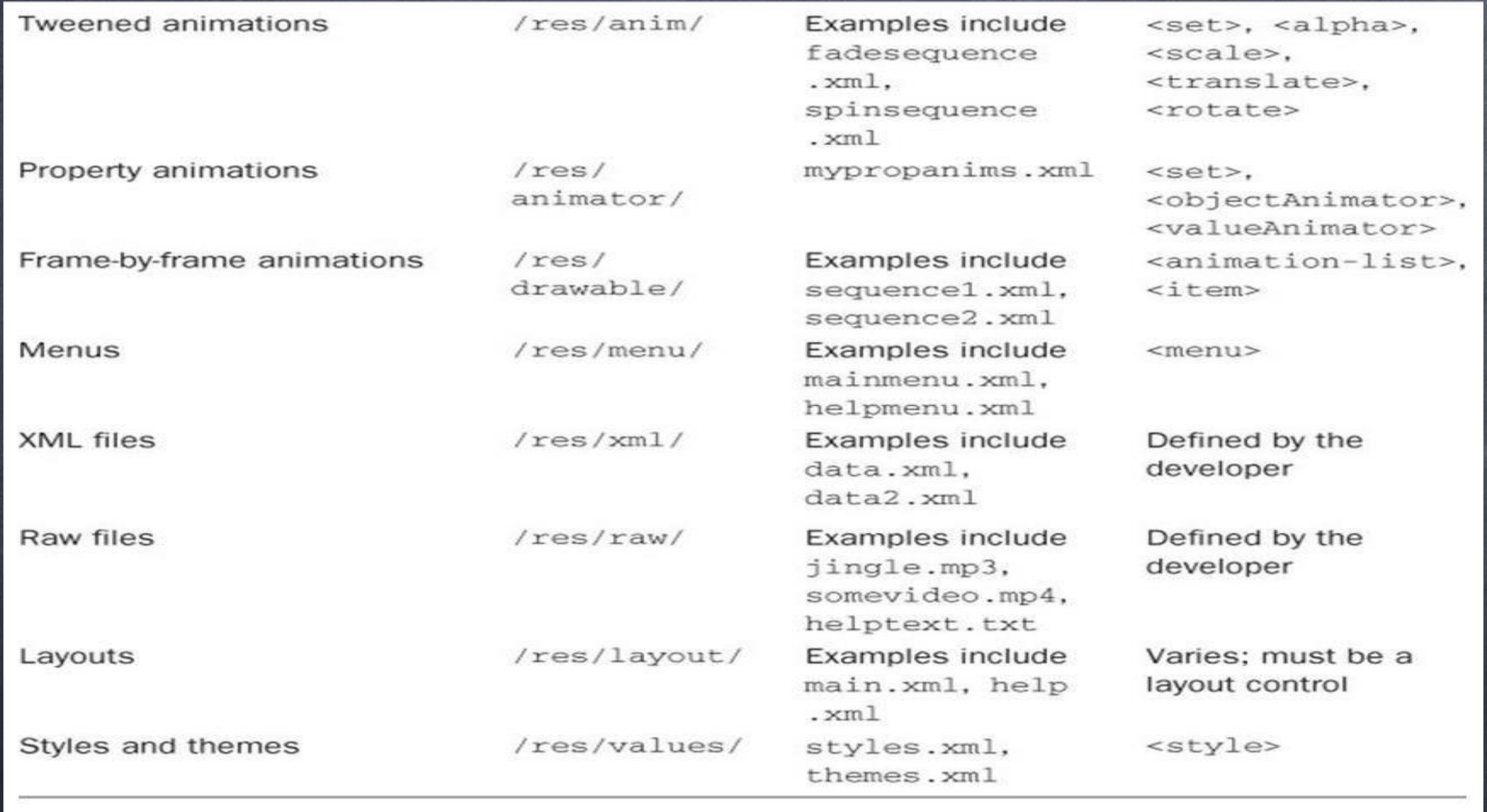
Resource Subdirectory	Purpose
/res/drawable-*/	Graphics resources
/res/layout/	User interface resources
/res/menu/	Menu resources for showing options or actions in activities
/res/values/	Simple data such as strings, styles and themes, and dimensions
/res/values-sw*/	Dimension resources for overriding defaults
/res/values-v*/	Style and theme resources for newer API customizations

Storing Application Resources



Resource Type	Required Directory	Suggested Filenames	XML Tag
Strings	/res/values/	strings.xml	<string>
String pluralization	/res/values/	strings.xml	<plurals>, <item>
Arrays of strings	/res/values/	strings.xml or arrays.xml	<string-array>, <item>
Booleans	/res/values/	bools.xml	<bool>
Colors	/res/values/	colors.xml	<color>
Color state lists	/res/color/	Examples include buttonstates.xml, indicators.xml	<selector>, <item>
Dimensions	/res/values/	dimens.xml	<dimen>
IDs	/res/values/	ids.xml	<item>
Integers	/res/values/	integers.xml	<integer>
Arrays of integers	/res/values/	integers.xml	<integer-array>
Mixed-type arrays	/res/values/	arrays.xml	<array>, <item>
Simple drawables (paintables)	/res/values/	drawables.xml	<drawable>
Graphics definition XML files such as shapes	/res/drawable/	Examples include icon.png, logo	Supported graphics files or drawables

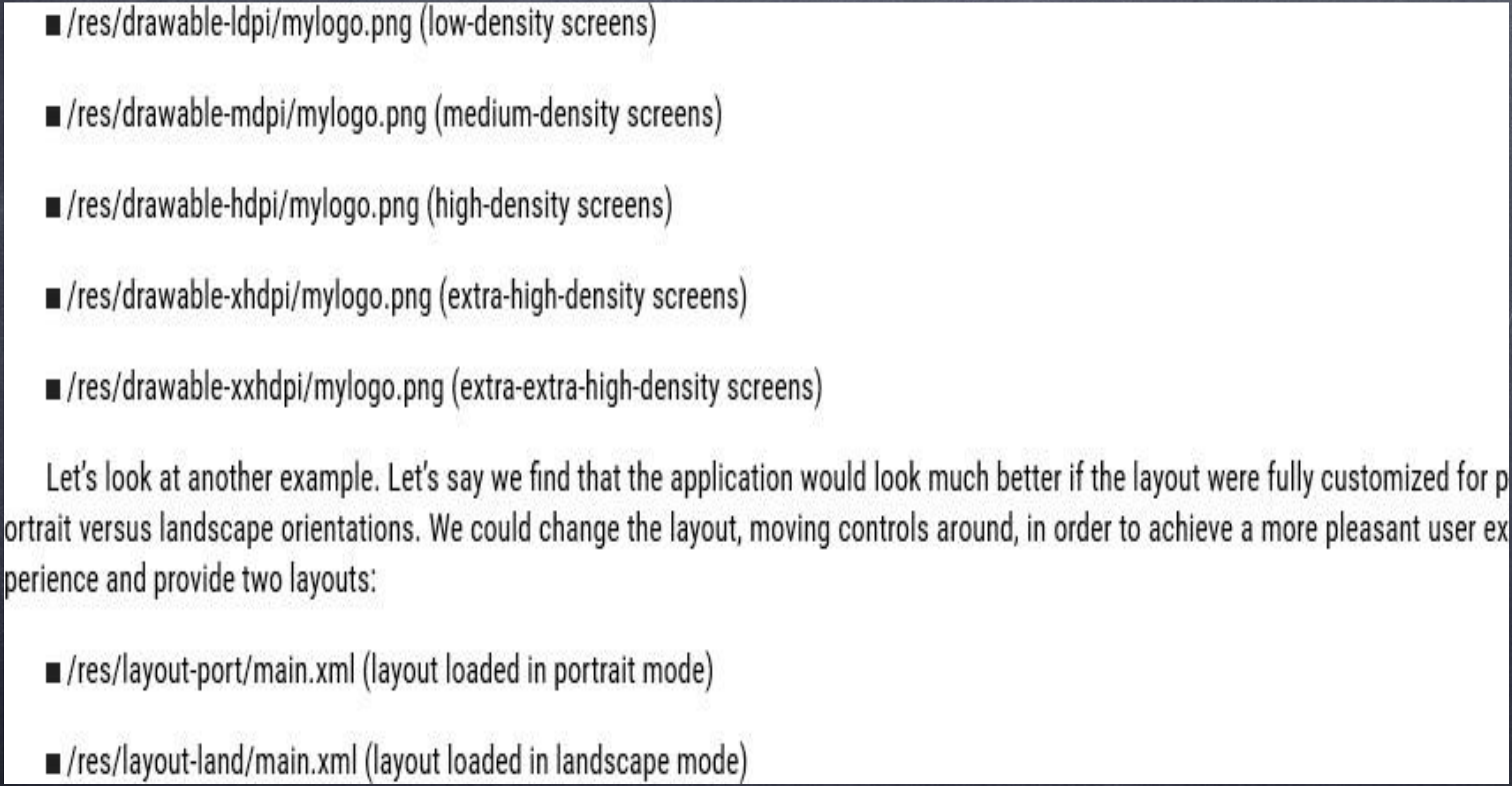
Storing Application Resources



Tweened animations	`/res/anim/`	Examples include `fadesequence.xml`, `spinsequence.xml`	`<set>`, `<alpha>`, `<scale>`, `<translate>`, `<rotate>`
Property animations	`/res/animator/`	`mypropanims.xml`	`<set>`, `<objectAnimator>`, `<valueAnimator>`
Frame-by-frame animations	`/res/drawable/`	Examples include `sequence1.xml`, `sequence2.xml`	`<animation-list>`, `<item>`
Menus	`/res/menu/`	Examples include `mainmenu.xml`, `helpmenu.xml`	`<menu>`
XML files	`/res/xml/`	Examples include `data.xml`, `data2.xml`	Defined by the developer
Raw files	`/res/raw/`	Examples include `jingle.mp3`, `somevideo.mp4`, `helptext.txt`	Defined by the developer
Layouts	`/res/layout/`	Examples include `main.xml`, `help.xml`	Varies; must be a layout control
Styles and themes	`/res/values/`	`styles.xml`, `themes.xml`	`<style>`

Table 6.2 How Common Resource Types Are Stored in the Project File Hierarchy

Storing Application Resources



■ /res/drawable-ldpi/mylogo.png (low-density screens)

■ /res/drawable-mdpi/mylogo.png (medium-density screens)

■ /res/drawable-hdpi/mylogo.png (high-density screens)

■ /res/drawable-xhdpi/mylogo.png (extra-high-density screens)

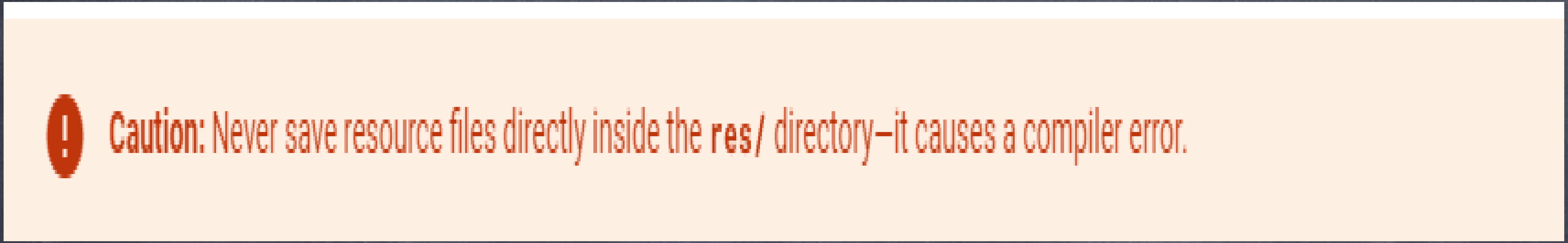
■ /res/drawable-xxhdpi/mylogo.png (extra-extra-high-density screens)

Let's look at another example. Let's say we find that the application would look much better if the layout were fully customized for p
ortrait versus landscape orientations. We could change the layout, moving controls around, in order to achieve a more pleasant user ex
perience and provide two layouts:

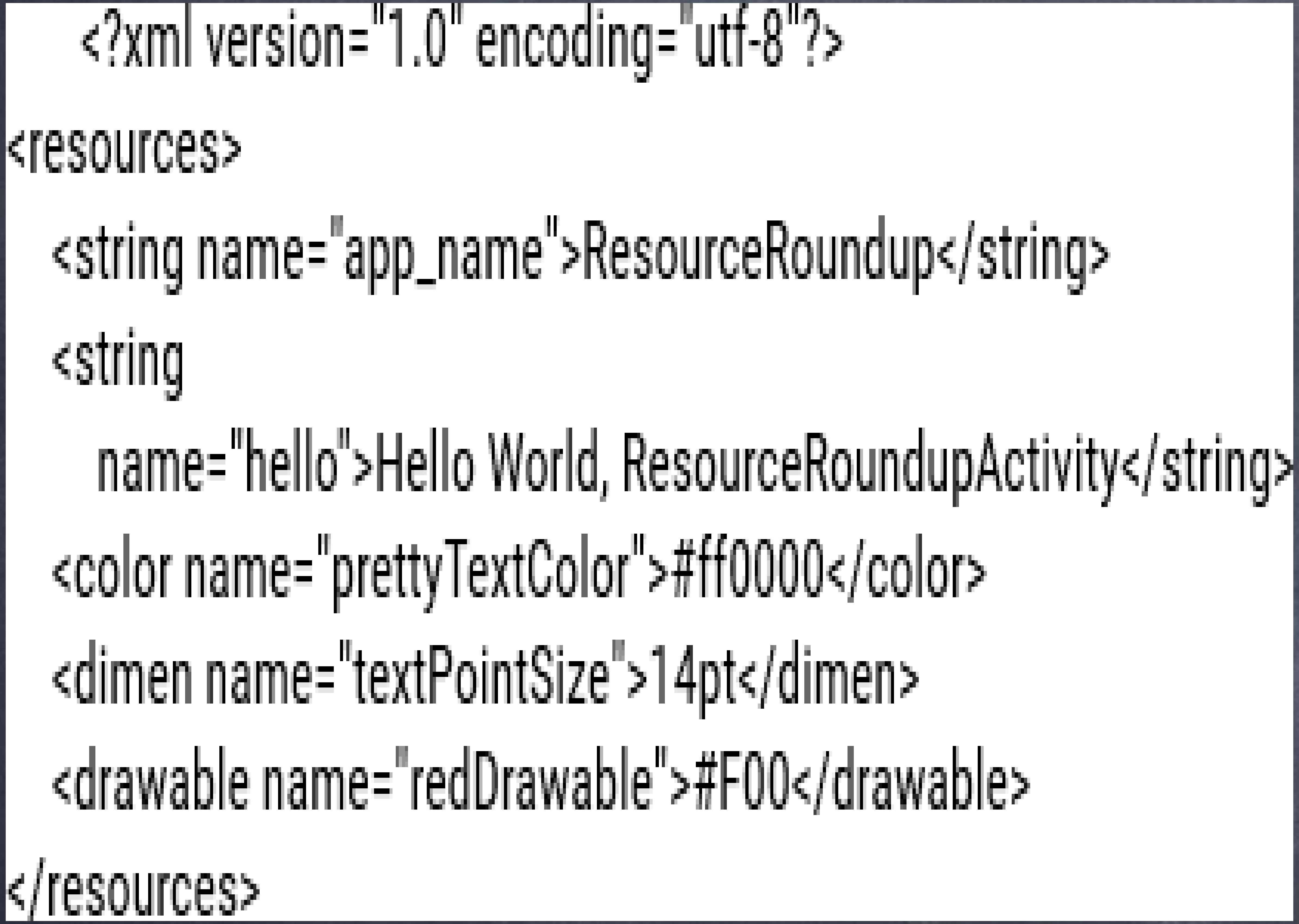
■ /res/layout-port/main.xml (layout loaded in portrait mode)

■ /res/layout-land/main.xml (layout loaded in landscape mode)

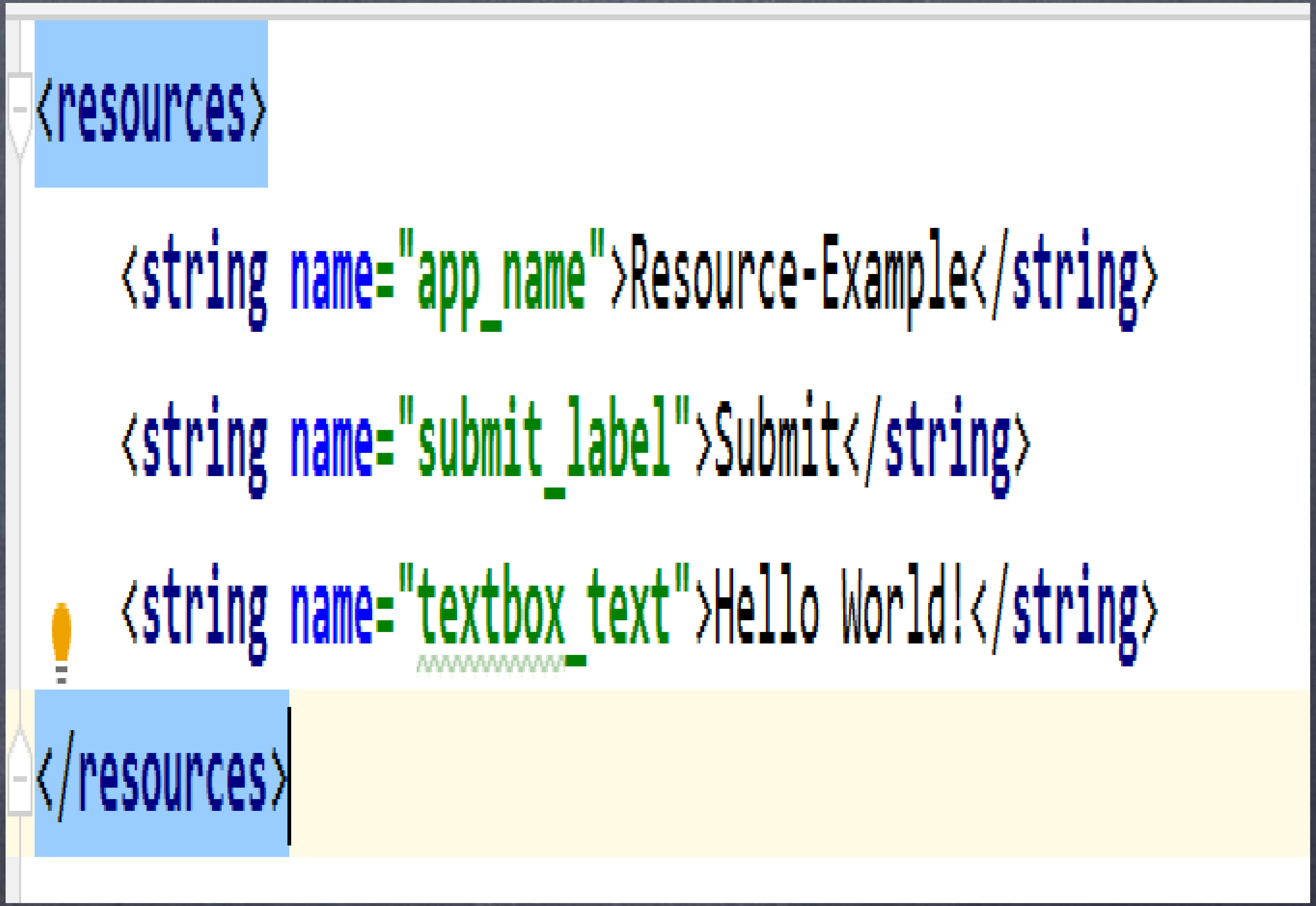
Storing Application Resources


Caution: Never save resource files directly inside the `res/` directory—it causes a compiler error.

String resources:

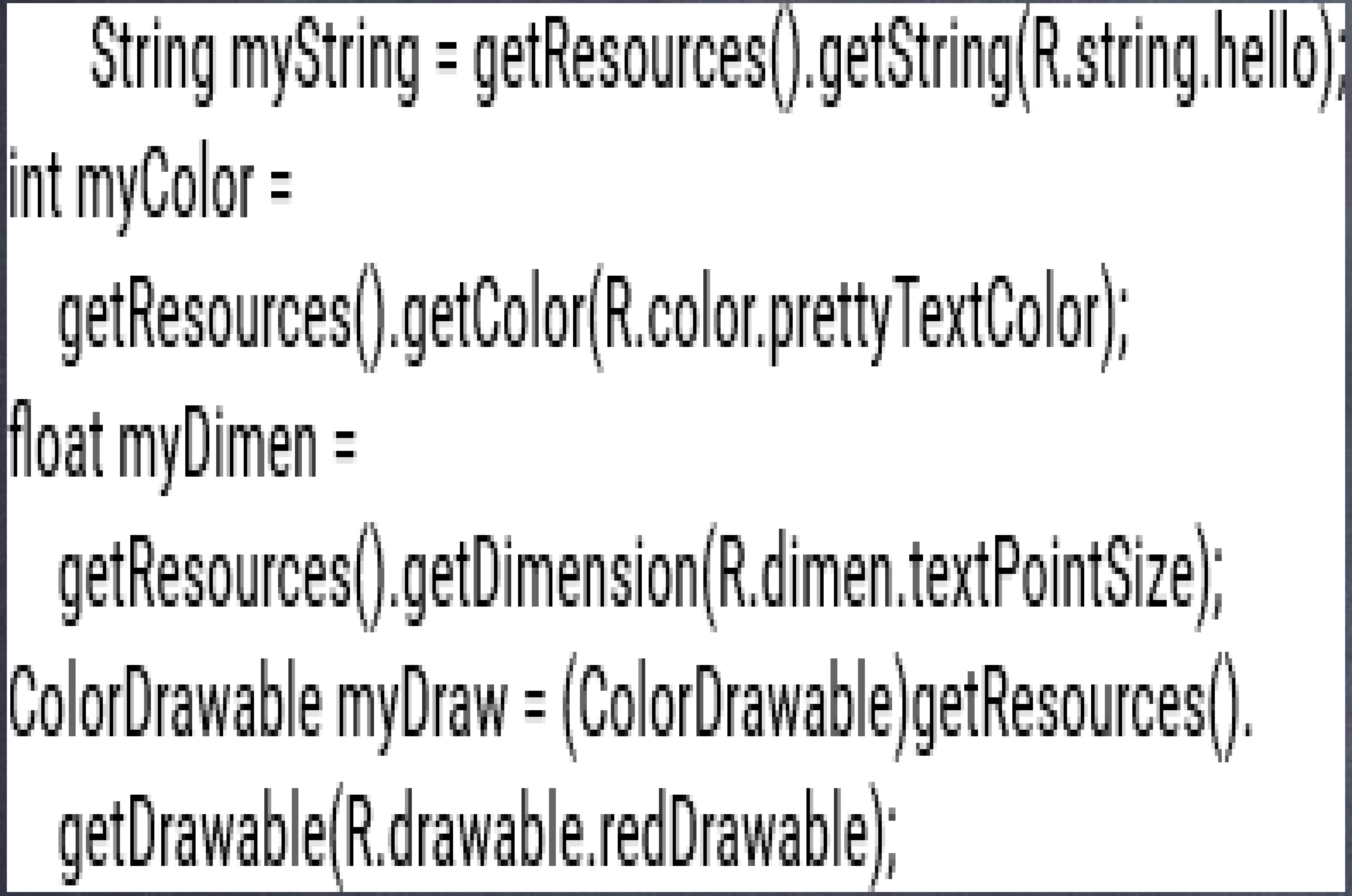


```
<?xml version="1.0" encoding="utf-8"?>
<resources>
  <string name="app_name">ResourceRoundup</string>
  <string
    name="hello">Hello World, ResourceRoundupActivity</string>
  <color name="prettyTextColor">#ff0000</color>
  <dimen name="textPointSize">14pt</dimen>
  <drawable name="redDrawable">#F00</drawable>
</resources>
```

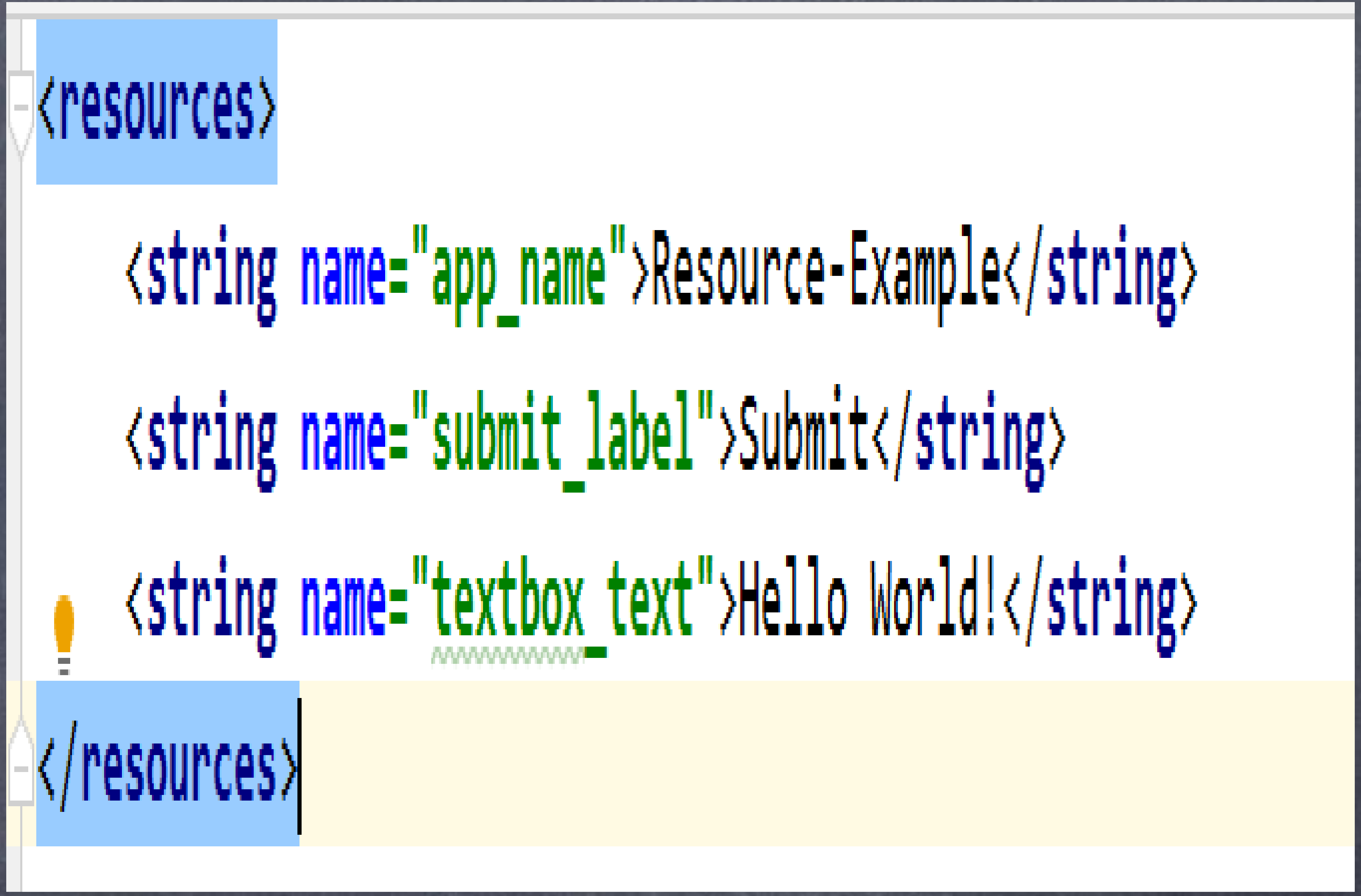


```
<resources>
    <string name="app_name">Resource-Example</string>
    <string name="submit_label">Submit</string>
    <string name="textbox_text">Hello World!</string>
</resources>
```

String resources:

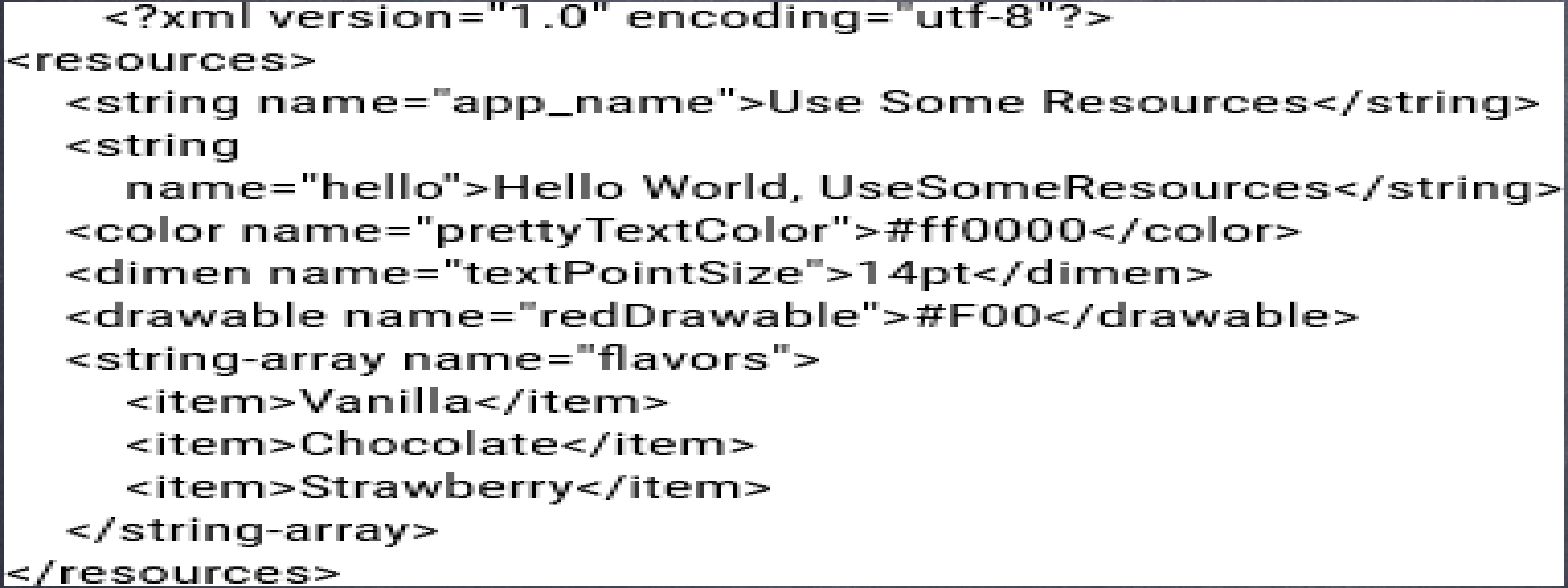


```
    String myString = getResources().getString(R.string.hello);
int myColor =
   getResources().getColor(R.color.prettyTextColor);
float myDimen =
   getResources().getDimension(R.dimen.textPointSize);
ColorDrawable myDraw = (ColorDrawable)getResources().
   getDrawable(R.drawable.redDrawable);
```

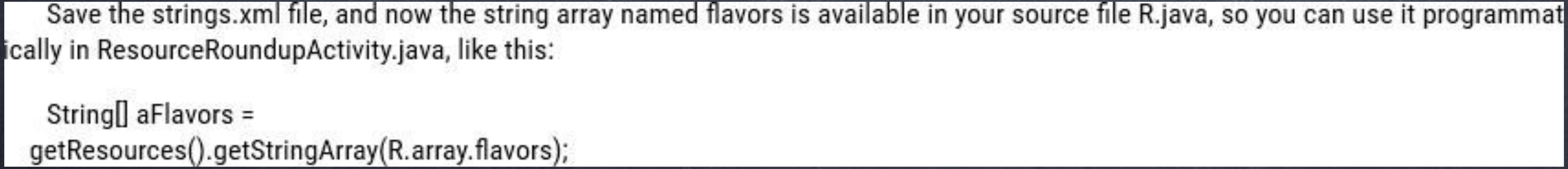


```
<resources>
    <string name="app_name">Resource-Example</string>
    <string name="submit_label">Submit</string>
    <string name="textbox_text">Hello World!</string>
</resources>
```

Providing alternative resources



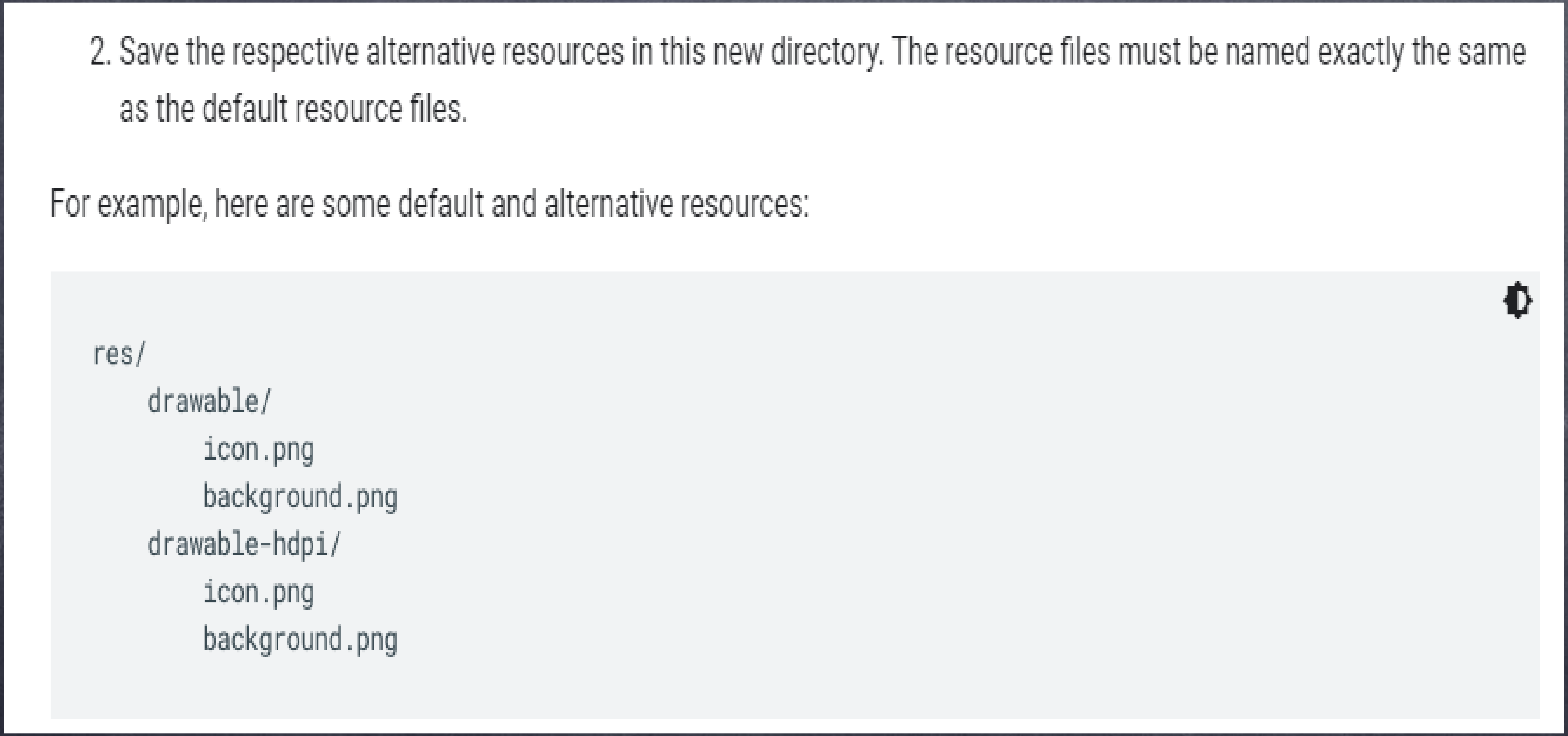
```
    <?xml version="1.0" encoding="utf-8"?>
<resources>
  <string name="app_name">Use Some Resources</string>
  <string
    name="hello">Hello World, UseSomeResources</string>
  <color name="prettyTextColor">#ff0000</color>
  <dimen name="textPointSize">14pt</dimen>
  <drawable name="redDrawable">#F00</drawable>
  <string-array name="flavors">
    <item>Vanilla</item>
    <item>Chocolate</item>
    <item>Strawberry</item>
  </string-array>
</resources>
```



Save the strings.xml file, and now the string array named flavors is available in your source file R.java, so you can use it programmat ically in ResourceRoundupActivity.java, like this:

```
  String[] aFlavors =
getResources().getStringArray(R.array.flavors);
```

Providing alternative resources

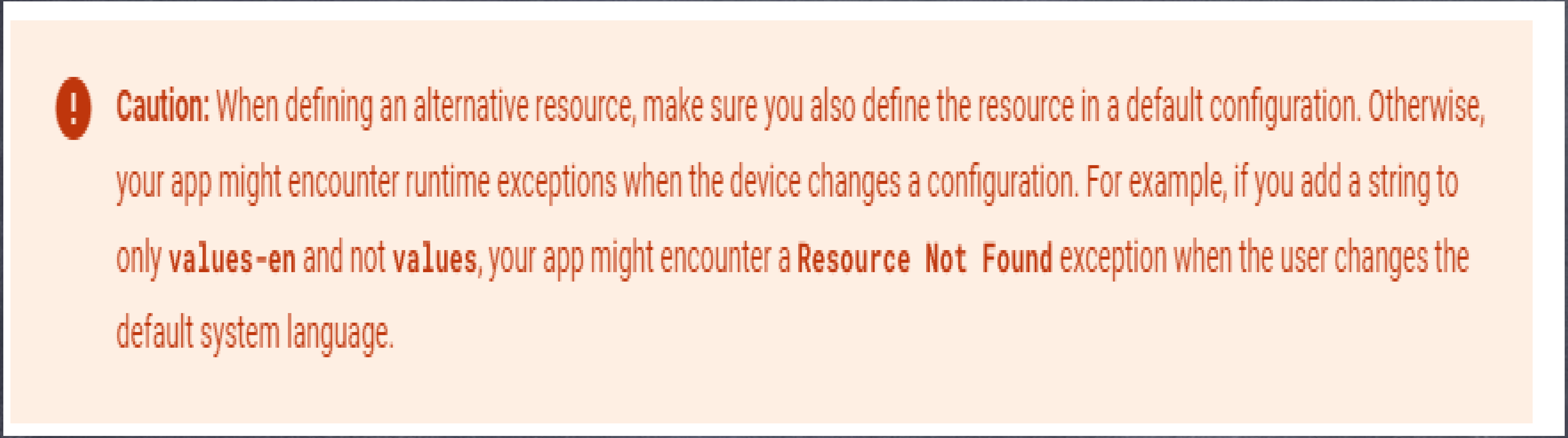


2. Save the respective alternative resources in this new directory. The resource files must be named exactly the same as the default resource files.

For example, here are some default and alternative resources:

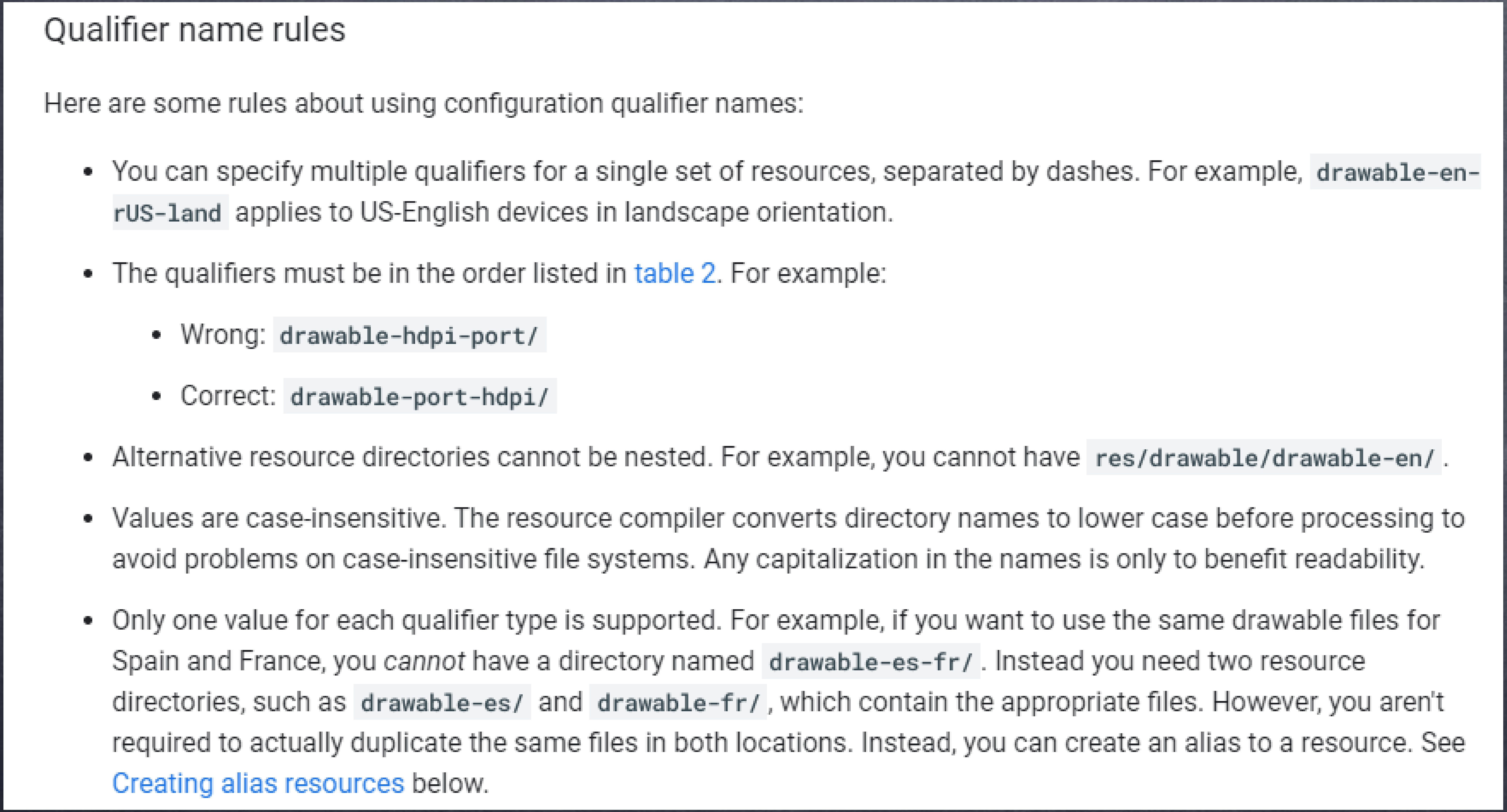
```
res/
    drawable/
        icon.png
        background.png
    drawable-hdpi/
        icon.png
        background.png
```

Providing alternative resources



Caution: When defining an alternative resource, make sure you also define the resource in a default configuration. Otherwise, your app might encounter runtime exceptions when the device changes a configuration. For example, if you add a string to only `values-en` and not `values`, your app might encounter a `Resource Not Found` exception when the user changes the default system language.

Qualifier name rules:

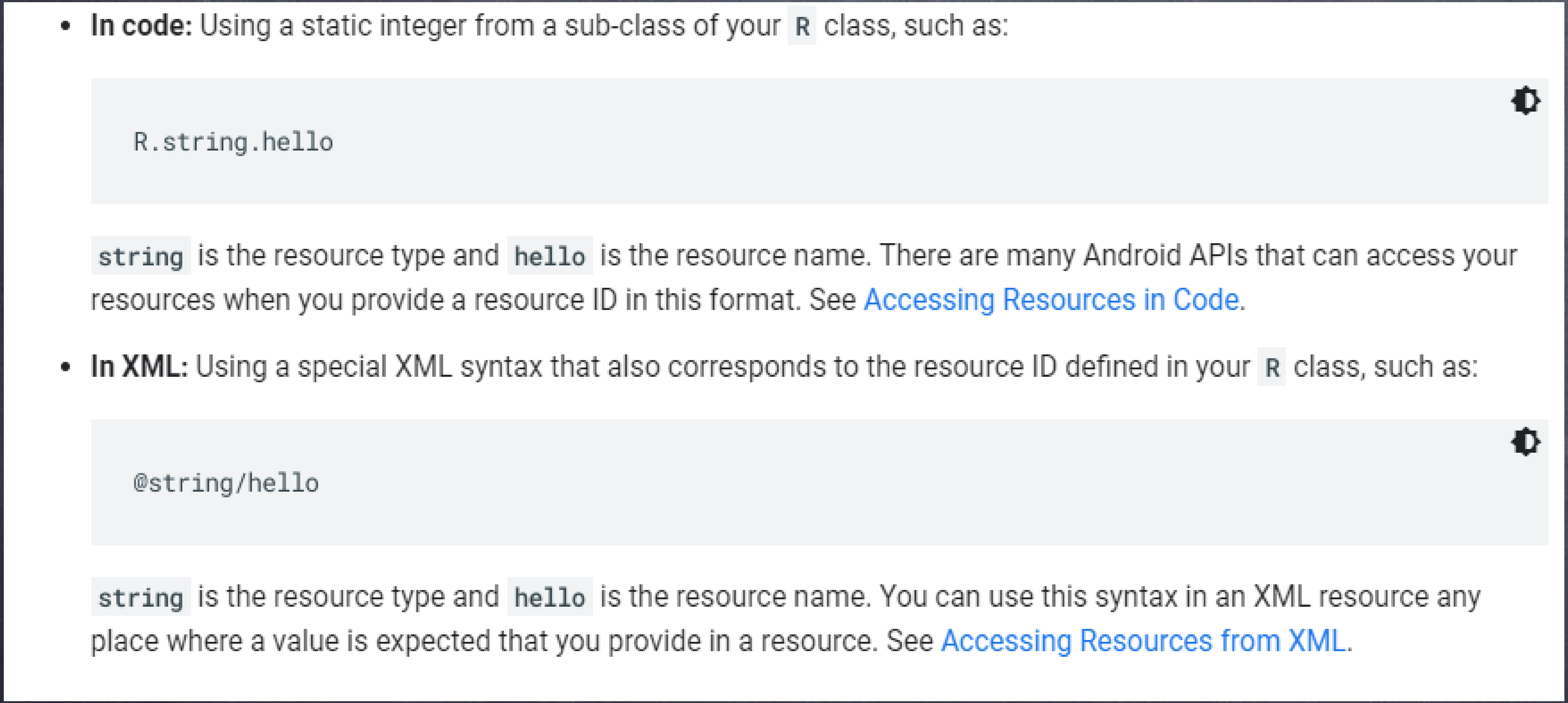


Qualifier name rules

Here are some rules about using configuration qualifier names:

- You can specify multiple qualifiers for a single set of resources, separated by dashes. For example, `drawable-en-rUS-land` applies to US-English devices in landscape orientation.
- The qualifiers must be in the order listed in table 2. For example:
  - Wrong: `drawable-hdpi-port/`
  - Correct: `drawable-port-hdpi/`
- Alternative resource directories cannot be nested. For example, you cannot have `res/drawable/drawable-en/`.
- Values are case-insensitive. The resource compiler converts directory names to lower case before processing to avoid problems on case-insensitive file systems. Any capitalization in the names is only to benefit readability.
- Only one value for each qualifier type is supported. For example, if you want to use the same drawable files for Spain and France, you *cannot* have a directory named `drawable-es-fr/`. Instead you need two resource directories, such as `drawable-es/` and `drawable-fr/`, which contain the appropriate files. However, you aren't required to actually duplicate the same files in both locations. Instead, you can create an alias to a resource. See Creating alias resources below.

Qualifier name rules:



- **In code:** Using a static integer from a sub-class of your `R` class, such as:

```
R.string.hello
```

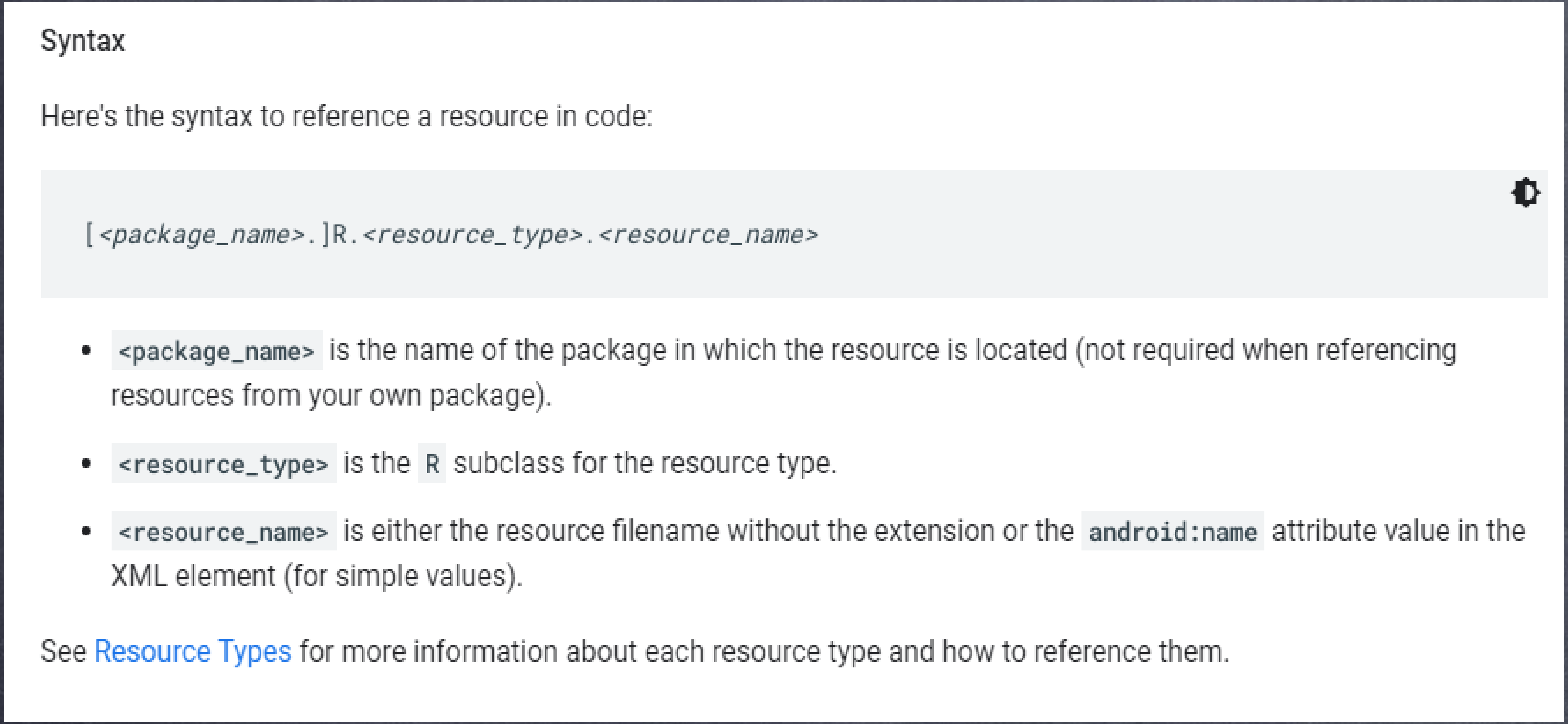
`string` is the resource type and `hello` is the resource name. There are many Android APIs that can access your resources when you provide a resource ID in this format. See Accessing Resources in Code.

- **In XML:** Using a special XML syntax that also corresponds to the resource ID defined in your `R` class, such as:

```
@string/hello
```

`string` is the resource type and `hello` is the resource name. You can use this syntax in an XML resource any place where a value is expected that you provide in a resource. See Accessing Resources from XML.

Qualifier name rules:



Syntax

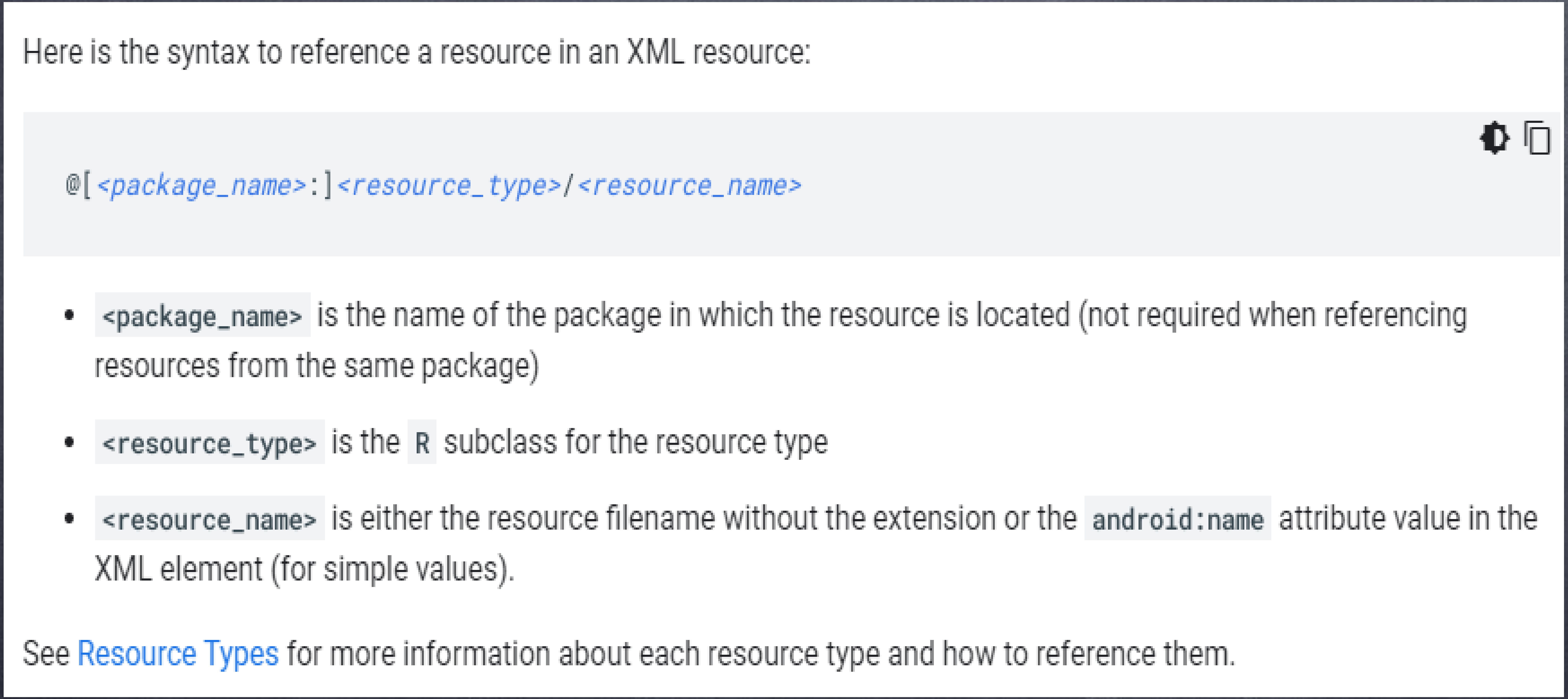
Here's the syntax to reference a resource in code:

```
[<package_name>.]R.<resource_type>.<resource_name>
```

- `<package_name>` is the name of the package in which the resource is located (not required when referencing resources from your own package).
- `<resource_type>` is the `R` subclass for the resource type.
- `<resource_name>` is either the resource filename without the extension or the `android:name` attribute value in the XML element (for simple values).

See Resource Types for more information about each resource type and how to reference them.

Qualifier name rules:



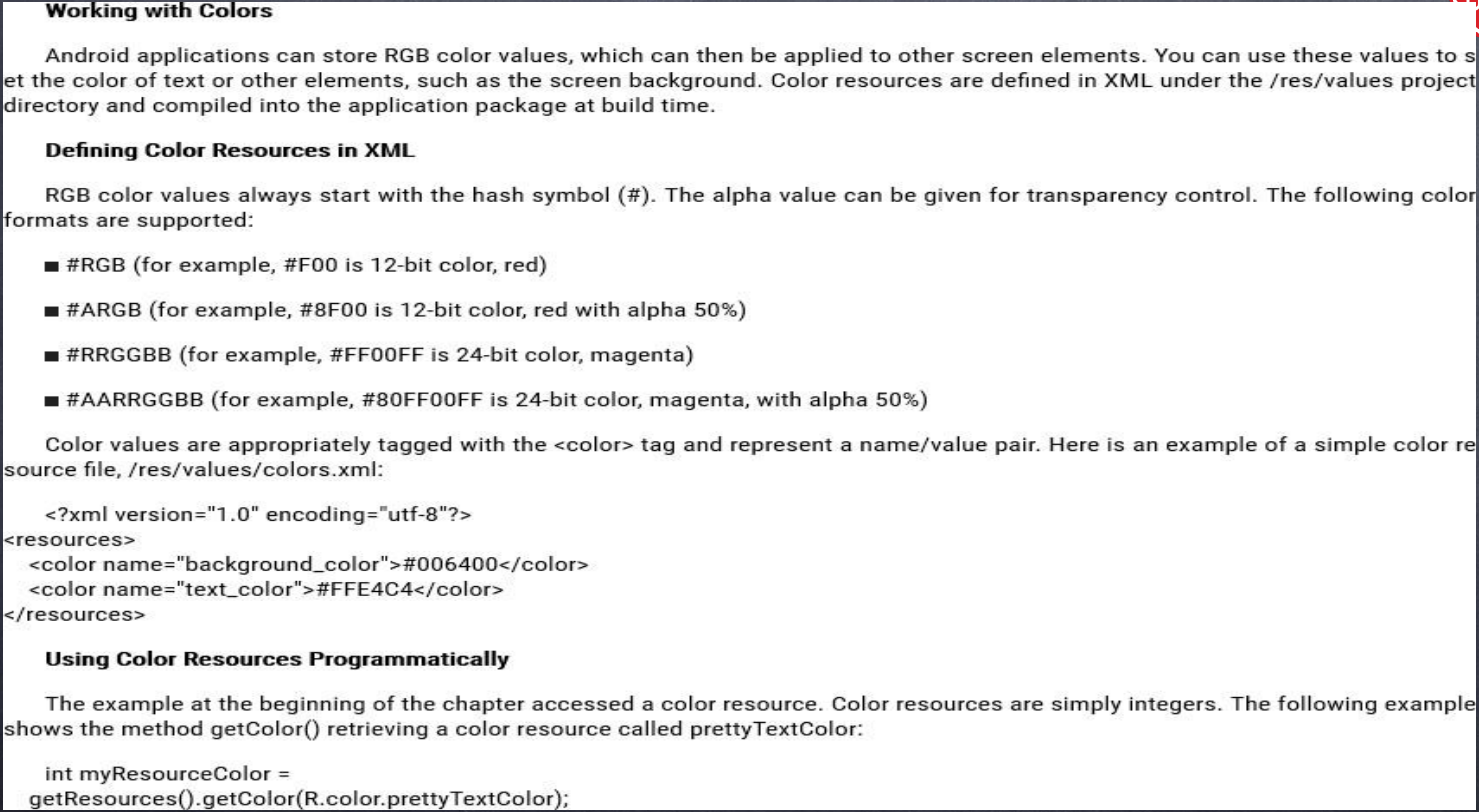
Here is the syntax to reference a resource in an XML resource:

```
@[<package_name>:]<resource_type>/<resource_name>
```

- `<package_name>` is the name of the package in which the resource is located (not required when referencing resources from the same package)
- `<resource_type>` is the `R` subclass for the resource type
- `<resource_name>` is either the resource filename without the extension or the `android:name` attribute value in the XML element (for simple values).

See Resource Types for more information about each resource type and how to reference them.

Color:



Working with Colors

Android applications can store RGB color values, which can then be applied to other screen elements. You can use these values to s
et the color of text or other elements, such as the screen background. Color resources are defined in XML under the /res/values project
directory and compiled into the application package at build time.

Defining Color Resources in XML

RGB color values always start with the hash symbol (#). The alpha value can be given for transparency control. The following color
formats are supported:

- #RGB (for example, #F00 is 12-bit color, red)
- #ARGB (for example, #8F00 is 12-bit color, red with alpha 50%)
- #RRGGBB (for example, #FF00FF is 24-bit color, magenta)
- #AARRGGBB (for example, #80FF00FF is 24-bit color, magenta, with alpha 50%)

Color values are appropriately tagged with the <color> tag and represent a name/value pair. Here is an example of a simple color re
source file, /res/values/colors.xml:

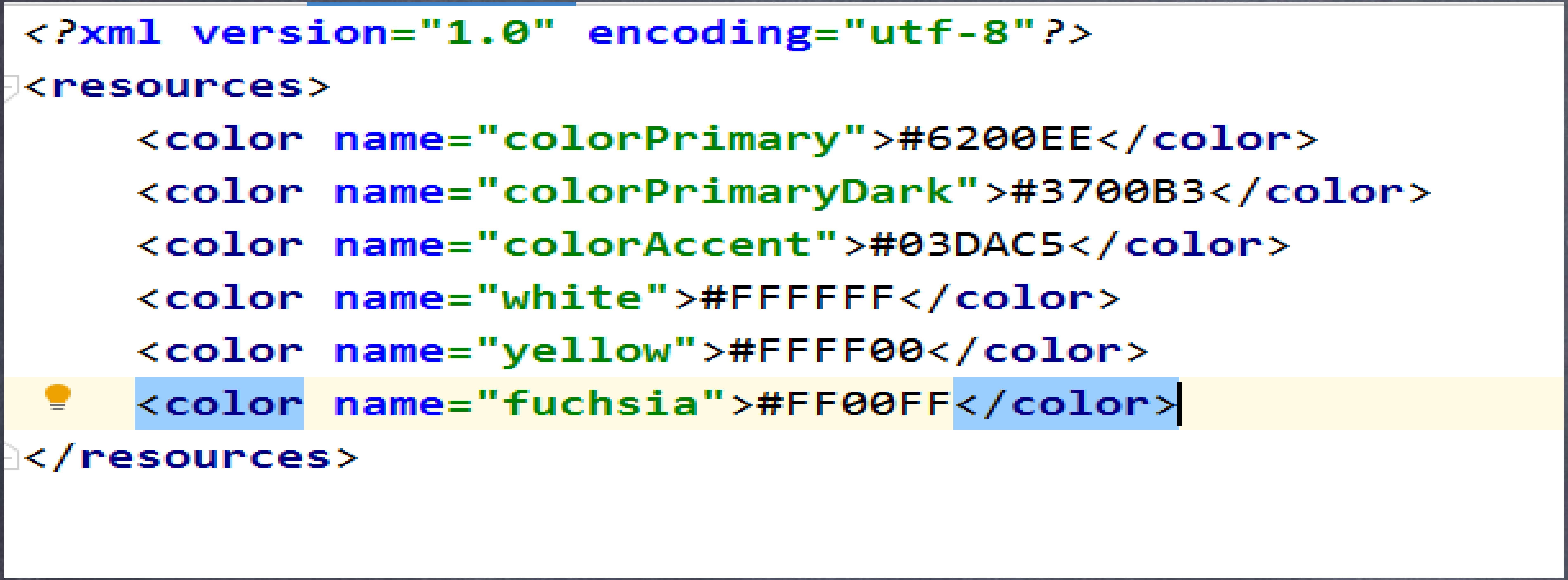
```
<?xml version="1.0" encoding="utf-8"?>
<resources>
  <color name="background_color">#006400</color>
  <color name="text_color">#FFE4C4</color>
</resources>
```

Using Color Resources Programmatically

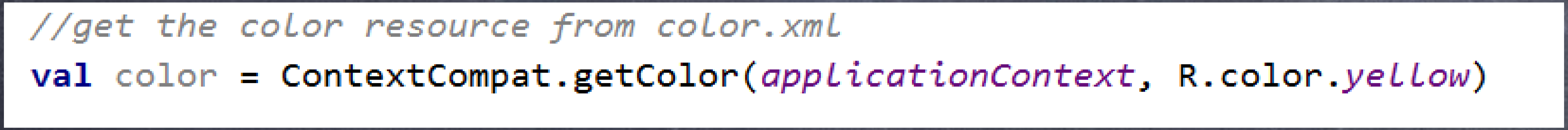
The example at the beginning of the chapter accessed a color resource. Color resources are simply integers. The following example
shows the method getColor() retrieving a color resource called prettyTextColor:

```
int myResourceColor =
  getResources().getColor(R.color.prettyTextColor);
```

Color:



```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <color name="colorPrimary">#6200EE</color>
    <color name="colorPrimaryDark">#3700B3</color>
    <color name="colorAccent">#03DAC5</color>
    <color name="white">#FFFFFF</color>
    <color name="yellow">#FFFF00</color>
    <color name="fuchsia">#FF00FF</color>
</resources>
```

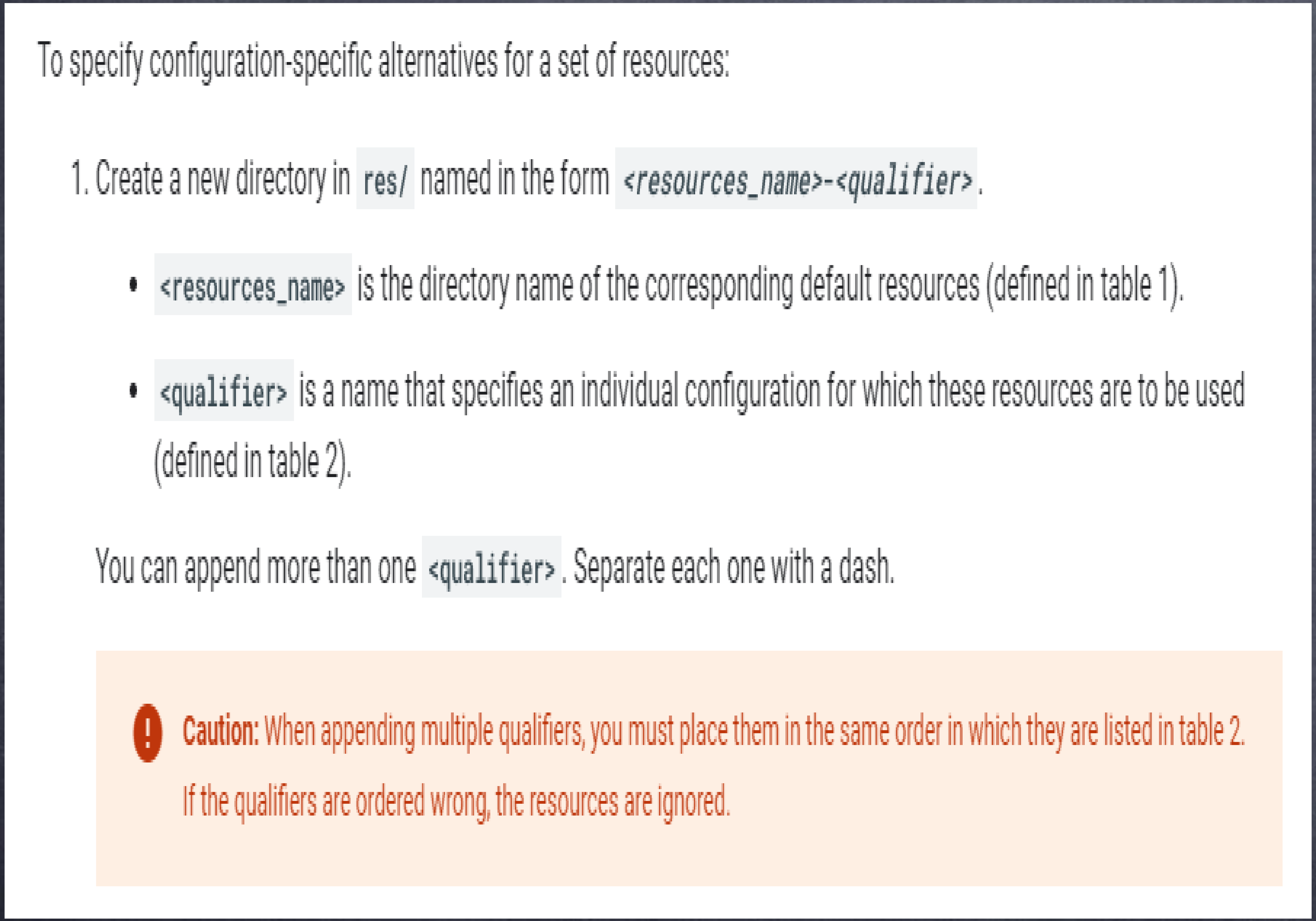


```
//get the color resource from color.xml
val color = ContextCompat.getColor(applicationContext, R.color.yellow)
```

It is important to note that the most current way of accessing color resources (since API 24) requires providing context in order to resolve any custom theme attributes.

https://html-color-codes.info/colors-from-image/

Providing alternative resources



To specify configuration-specific alternatives for a set of resources:

1. Create a new directory in `res/` named in the form `<resources_name>-<qualifier>`.

   - `<resources_name>` is the directory name of the corresponding default resources (defined in table 1).
   - `<qualifier>` is a name that specifies an individual configuration for which these resources are to be used (defined in table 2).

   You can append more than one `<qualifier>`. Separate each one with a dash.

   Caution: When appending multiple qualifiers, you must place them in the same order in which they are listed in table 2. If the qualifiers are ordered wrong, the resources are ignored.

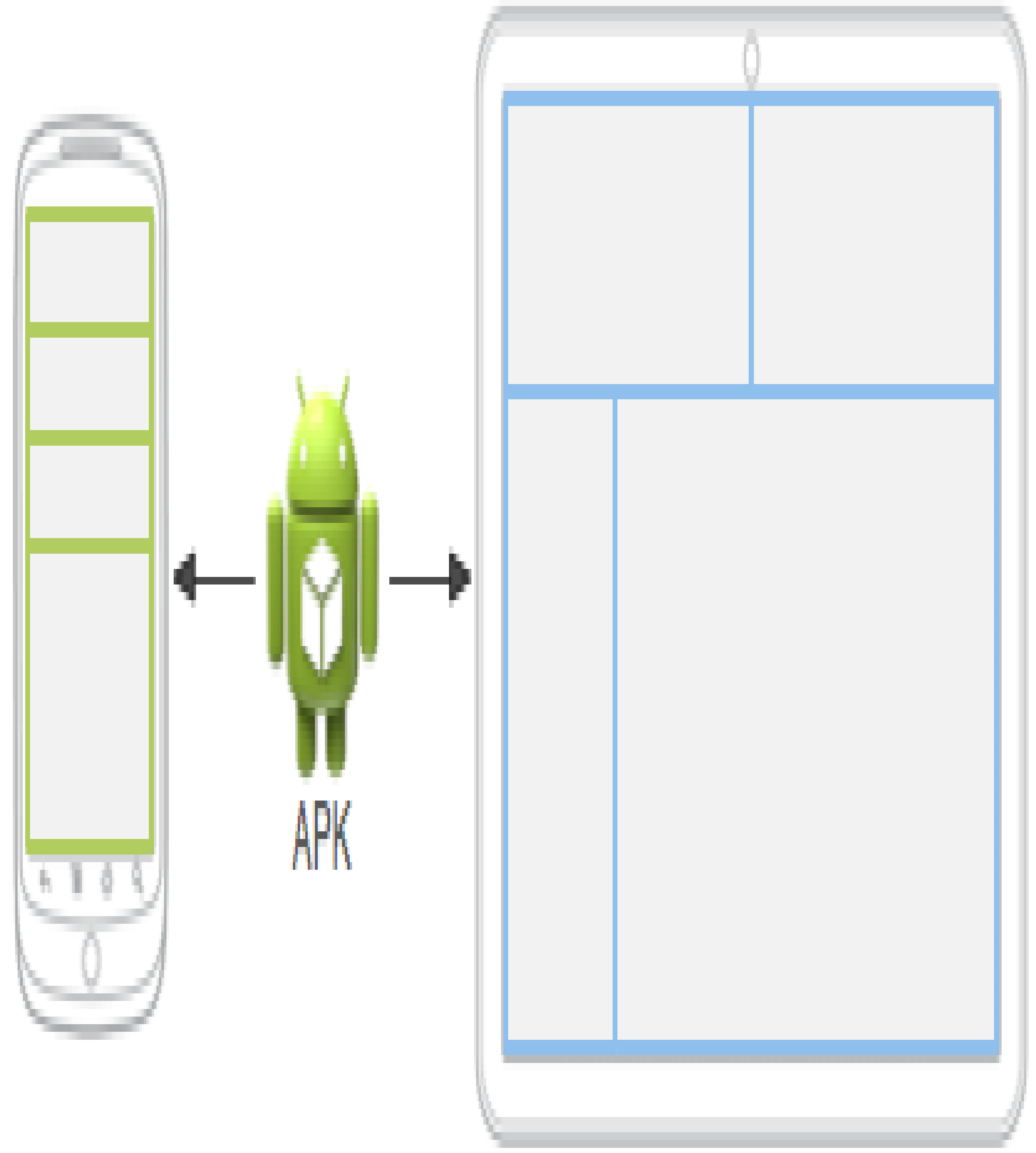
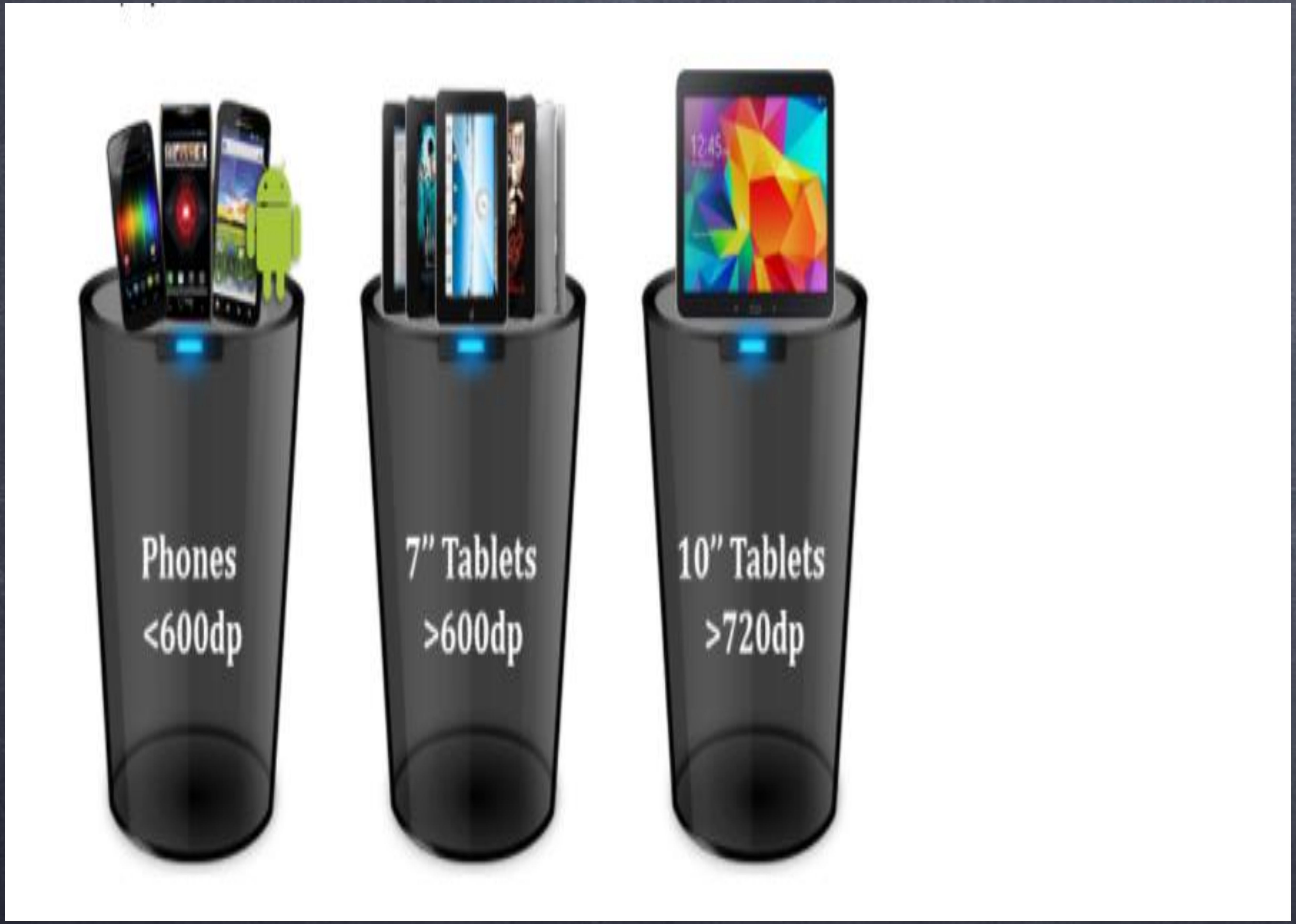


Figure 1. Two different devices, each using different layout resources.

Providing alternative resources



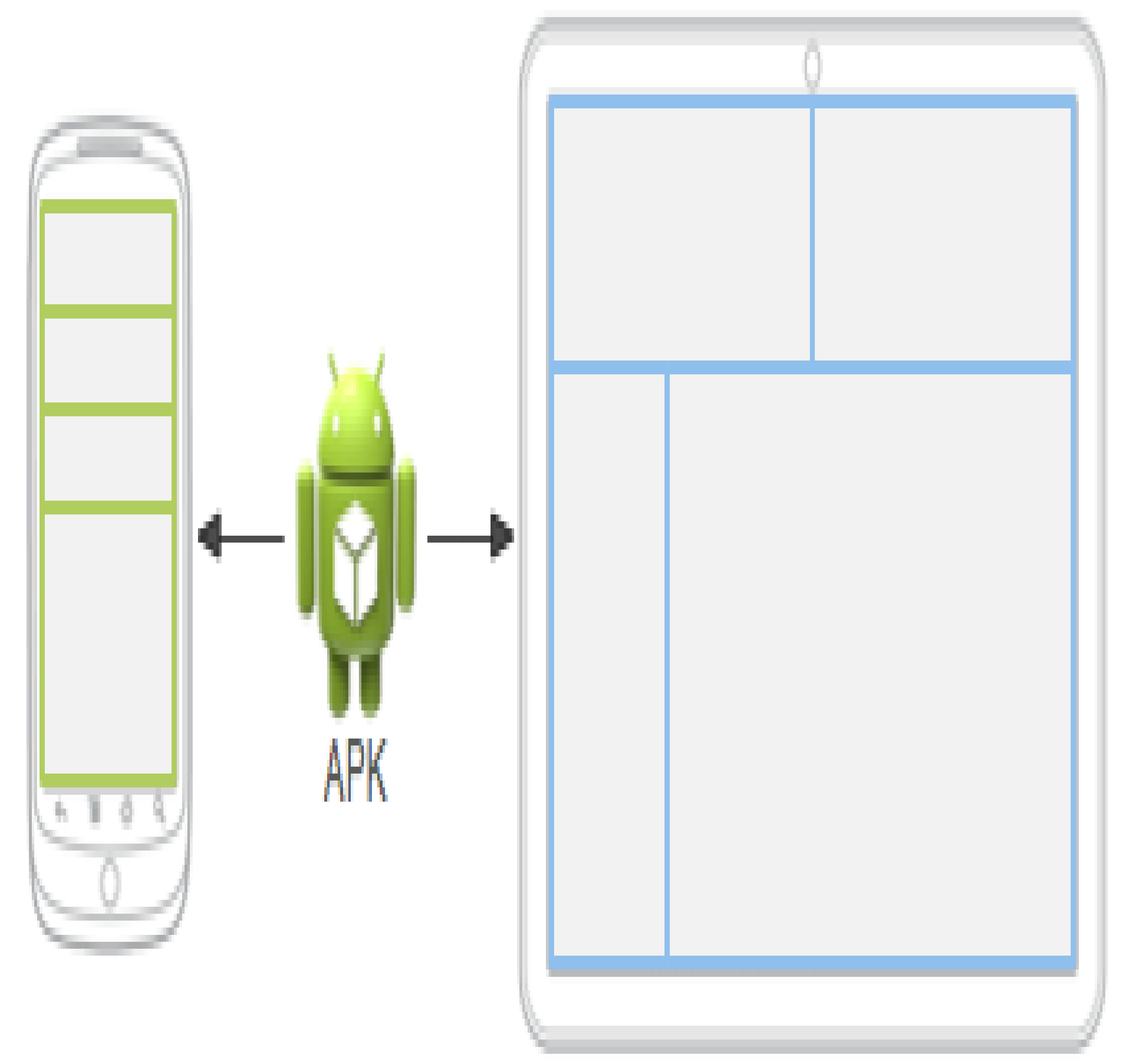
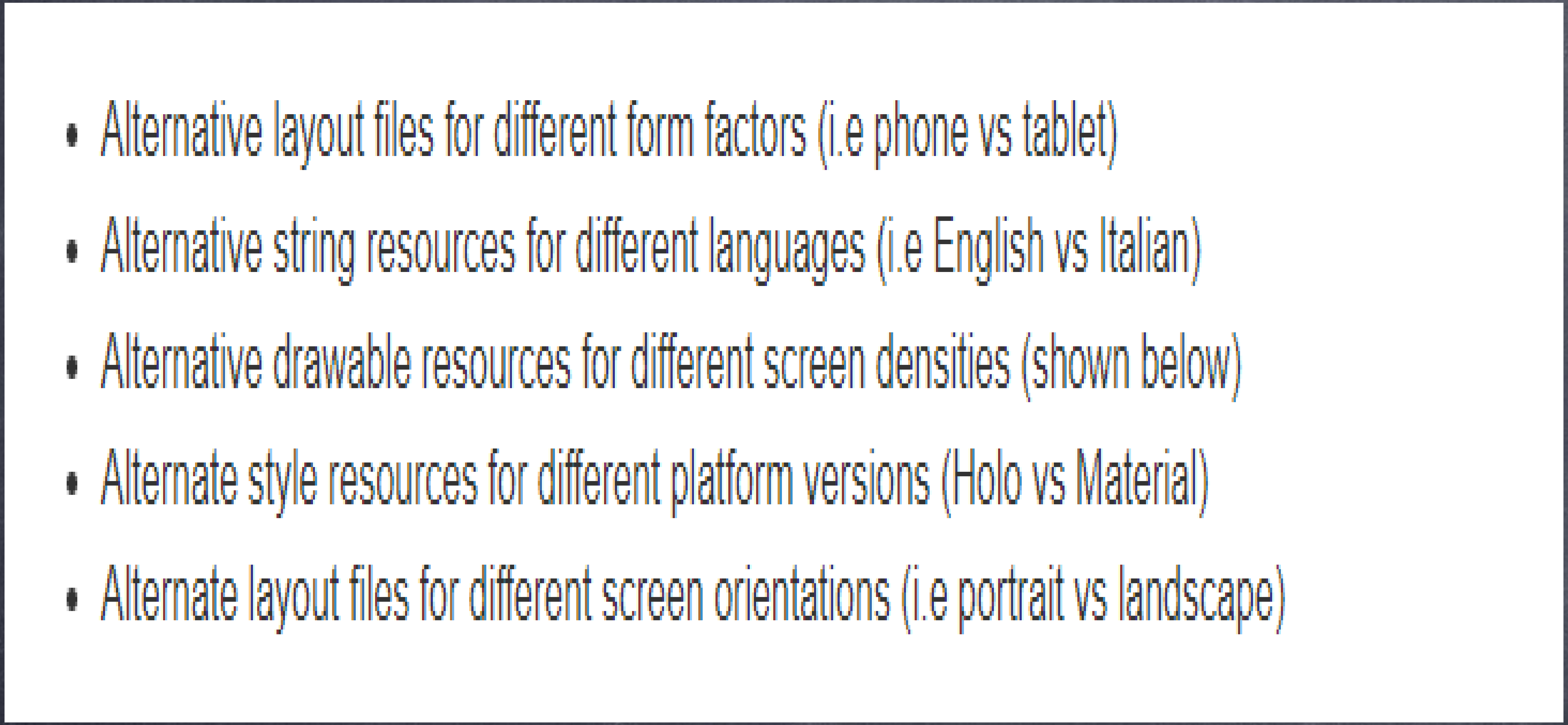


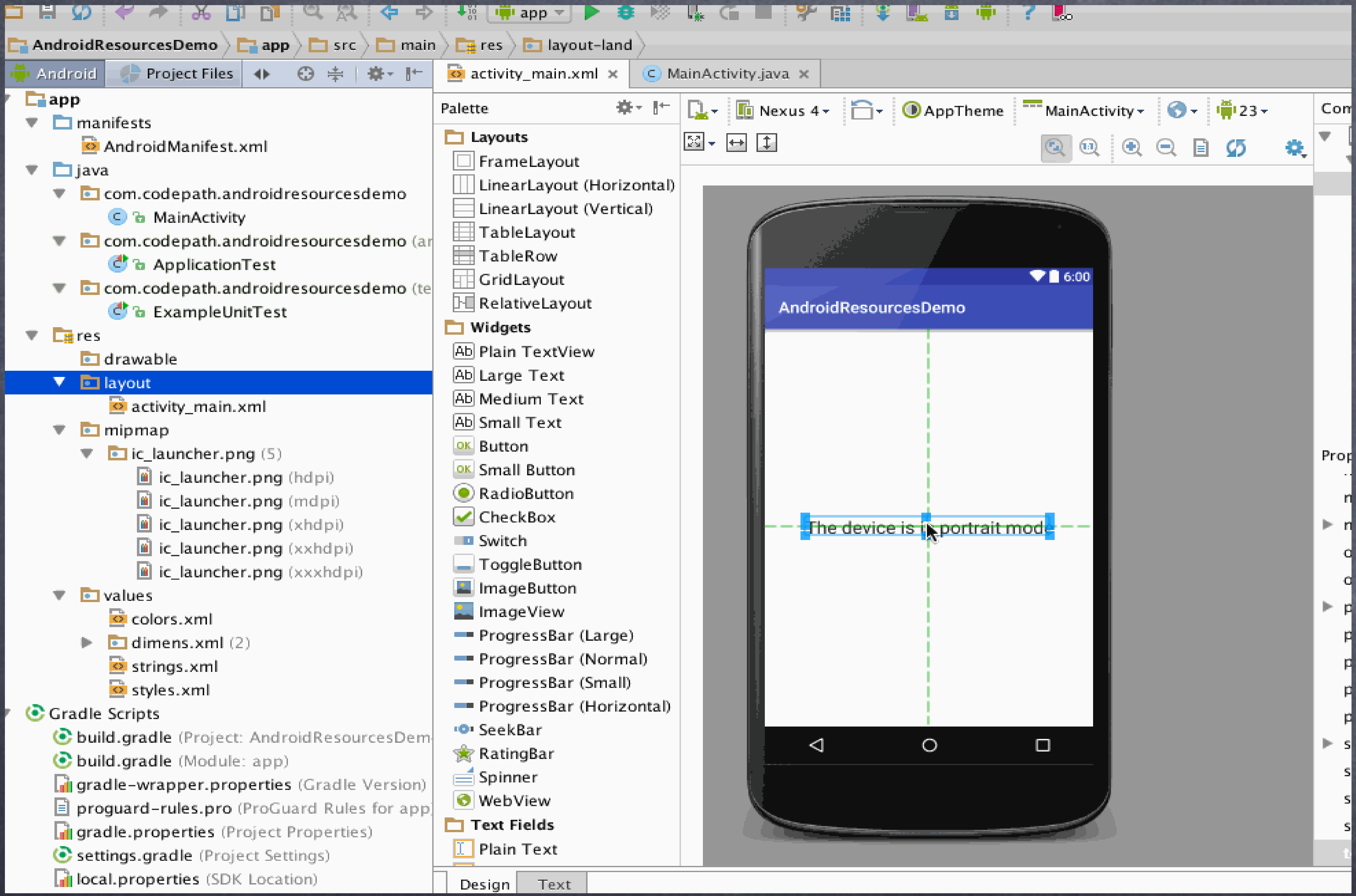
Figure 1. Two different devices, each using different layout resources.

Providing alternative resources

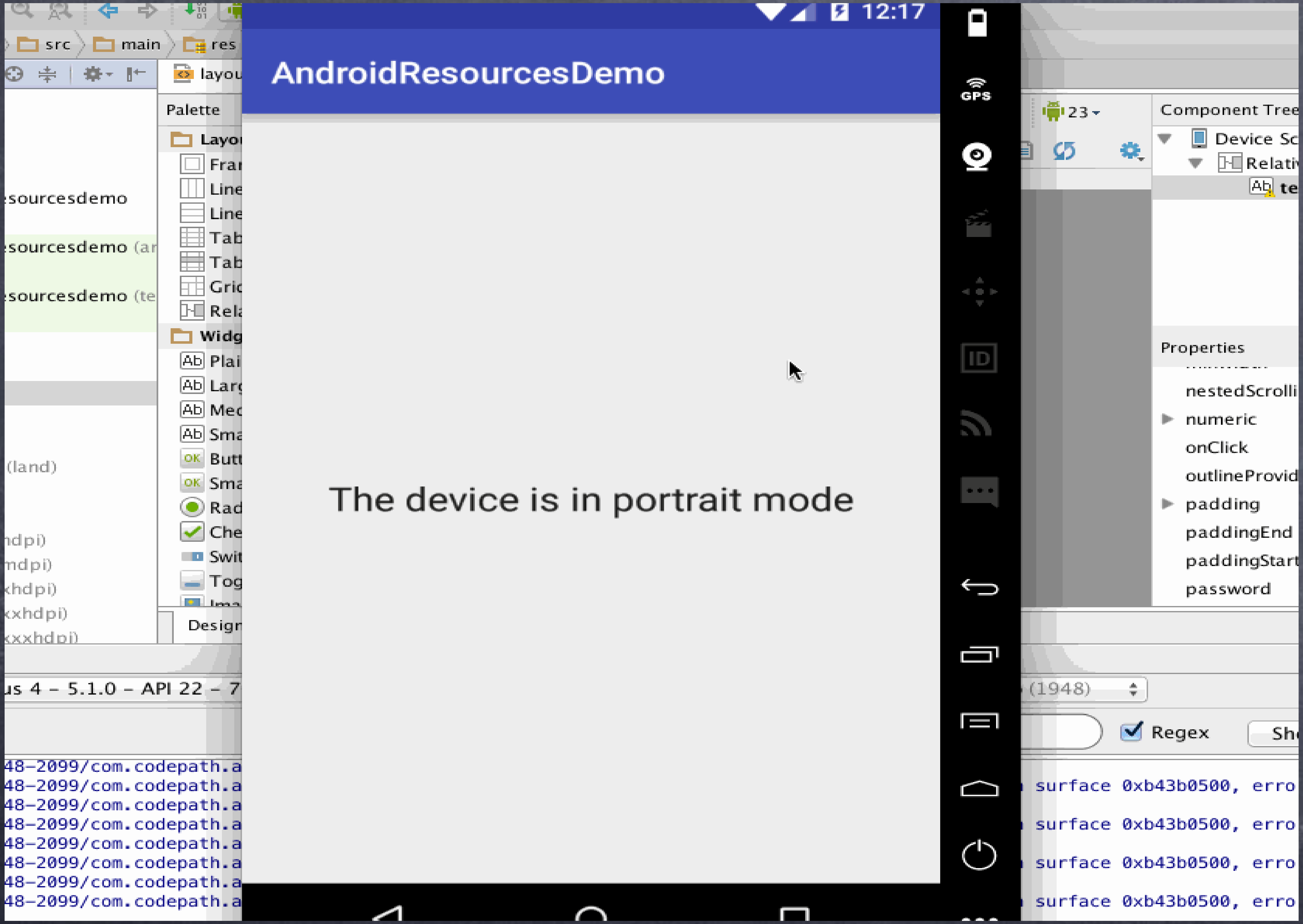


- Alternative layout files for different form factors (i.e phone vs tablet)
- Alternative string resources for different languages (i.e English vs Italian)
- Alternative drawable resources for different screen densities (shown below)
- Alternate style resources for different platform versions (Holo vs Material)
- Alternate layout files for different screen orientations (i.e portrait vs landscape)

Portrait vs landscape



Portrait vs landscape



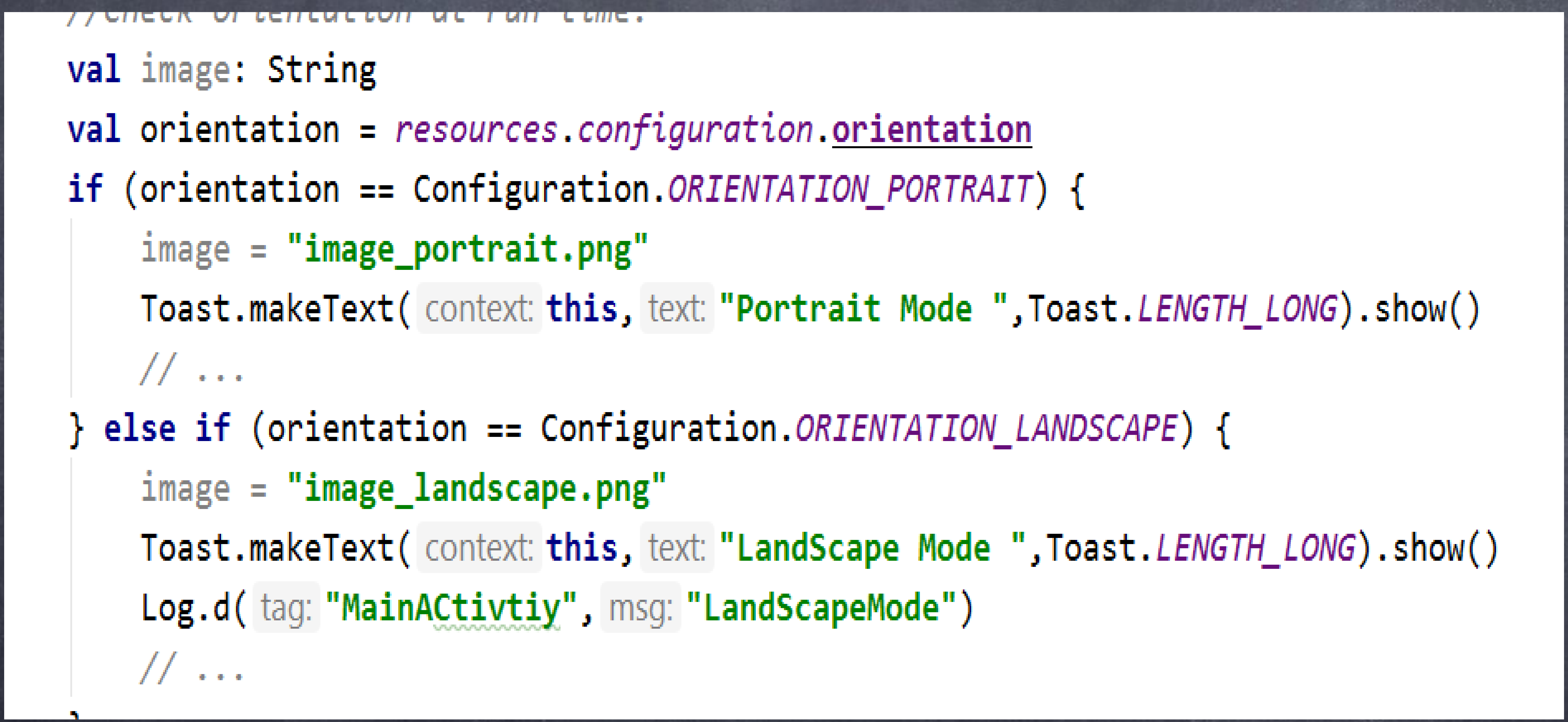
The device is in portrait mode

```
48-2099/com.codepath.a
48-2099/com.codepath.a
48-2099/com.codepath.a
48-2099/com.codepath.a
48-2099/com.codepath.a
48-2099/com.codepath.a
48-2099/com.codepath.a
48-2099/com.codepath.a
```

```
surface 0xb43b0500, erro
surface 0xb43b0500, erro
surface 0xb43b0500, erro
surface 0xb43b0500, erro
```

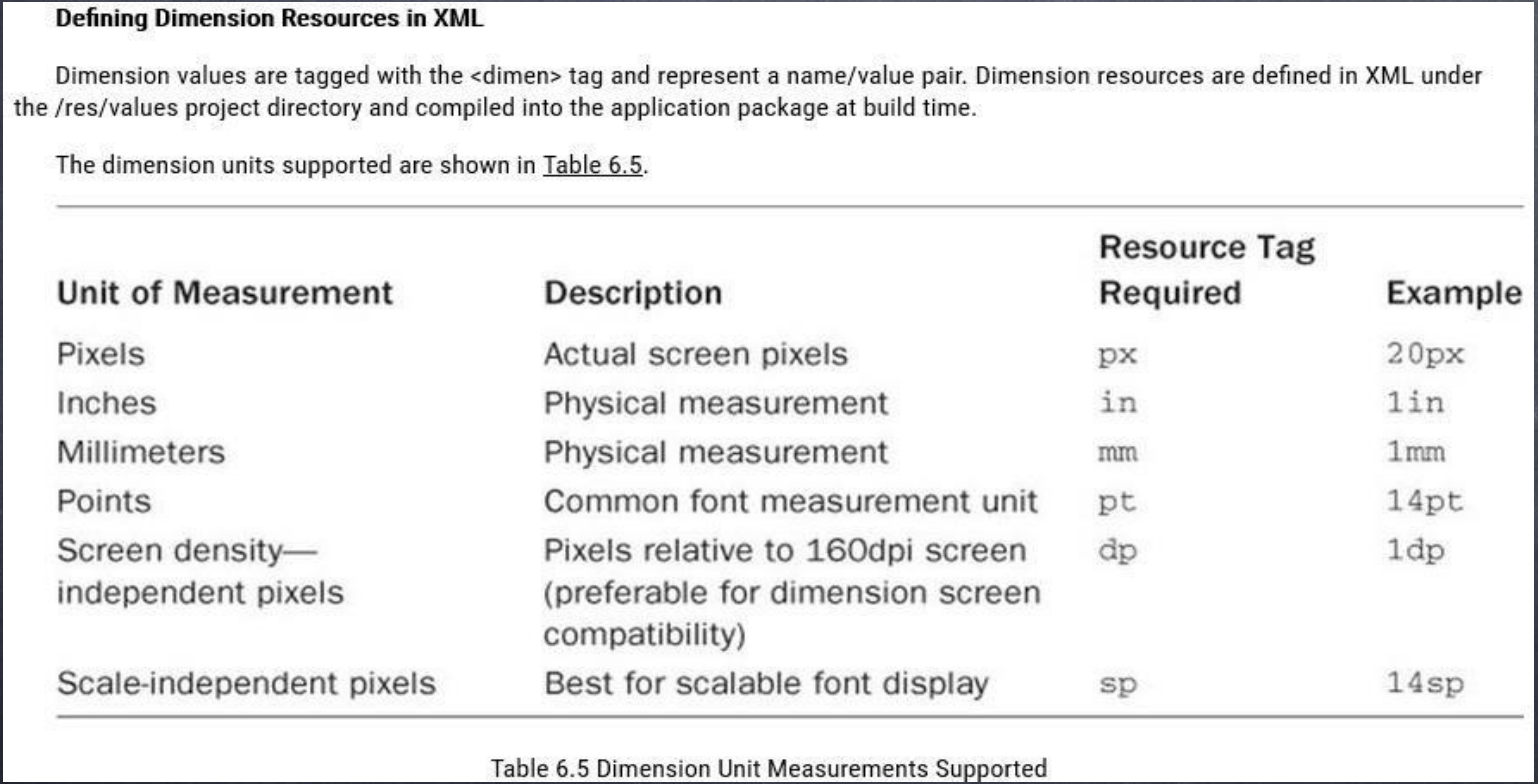
Portrait vs landscape

Determining Configuration at Runtime



```
//check orientation at run time.
val image: String
val orientation = resources.configuration.orientation
if (orientation == Configuration.ORIENTATION_PORTRAIT) {
    image = "image_portrait.png"
    Toast.makeText( context: this, text: "Portrait Mode ",Toast.LENGTH_LONG).show()
    // ...
} else if (orientation == Configuration.ORIENTATION_LANDSCAPE) {
    image = "image_landscape.png"
    Toast.makeText( context: this, text: "LandScape Mode ",Toast.LENGTH_LONG).show()
    Log.d( tag: "MainACtivtiy", msg: "LandScapeMode")
    // ...
}
```

Providing alternative resources-Dimension



Defining Dimension Resources in XML

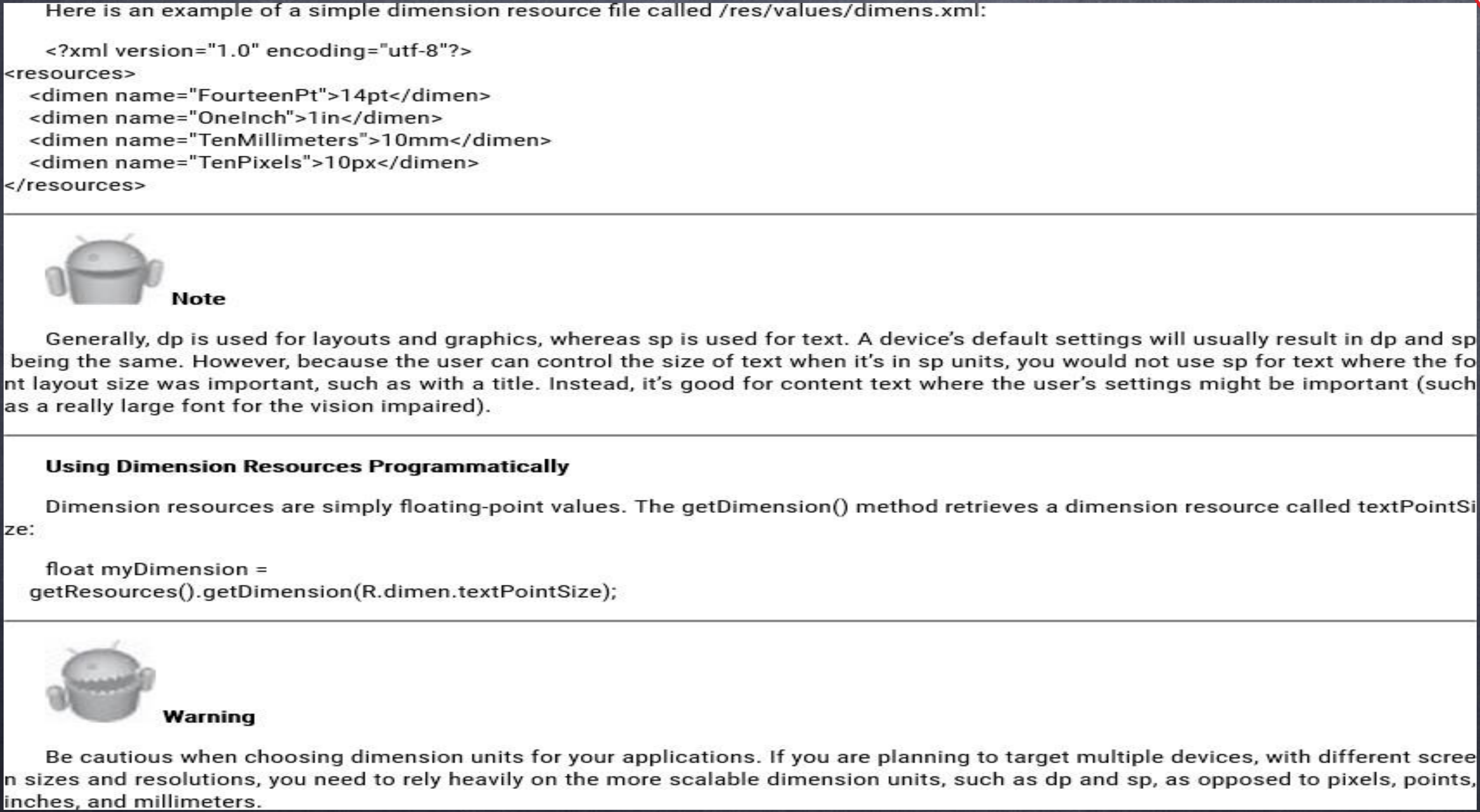
Dimension values are tagged with the <dimen> tag and represent a name/value pair. Dimension resources are defined in XML under the /res/values project directory and compiled into the application package at build time.

The dimension units supported are shown in Table 6.5.

Unit of Measurement	Description	Resource Tag Required	Example
Pixels	Actual screen pixels	px	20px
Inches	Physical measurement	in	1in
Millimeters	Physical measurement	mm	1mm
Points	Common font measurement unit	pt	14pt
Screen density—independent pixels	Pixels relative to 160dpi screen (preferable for dimension screen compatibility)	dp	1dp
Scale-independent pixels	Best for scalable font display	sp	14sp

Table 6.5 Dimension Unit Measurements Supported

Providing alternative resources-Dimension



Here is an example of a simple dimension resource file called /res/values/dimens.xml:

```
    <?xml version="1.0" encoding="utf-8"?>
<resources>
  <dimen name="FourteenPt">14pt</dimen>
  <dimen name="OneInch">1in</dimen>
  <dimen name="TenMillimeters">10mm</dimen>
  <dimen name="TenPixels">10px</dimen>
</resources>
```

Note

Generally, dp is used for layouts and graphics, whereas sp is used for text. A device's default settings will usually result in dp and sp
being the same. However, because the user can control the size of text when it's in sp units, you would not use sp for text where the fo
nt layout size was important, such as with a title. Instead, it's good for content text where the user's settings might be important (such
as a really large font for the vision impaired).

Using Dimension Resources Programmatically

Dimension resources are simply floating-point values. The getDimension() method retrieves a dimension resource called textPointSi
ze:

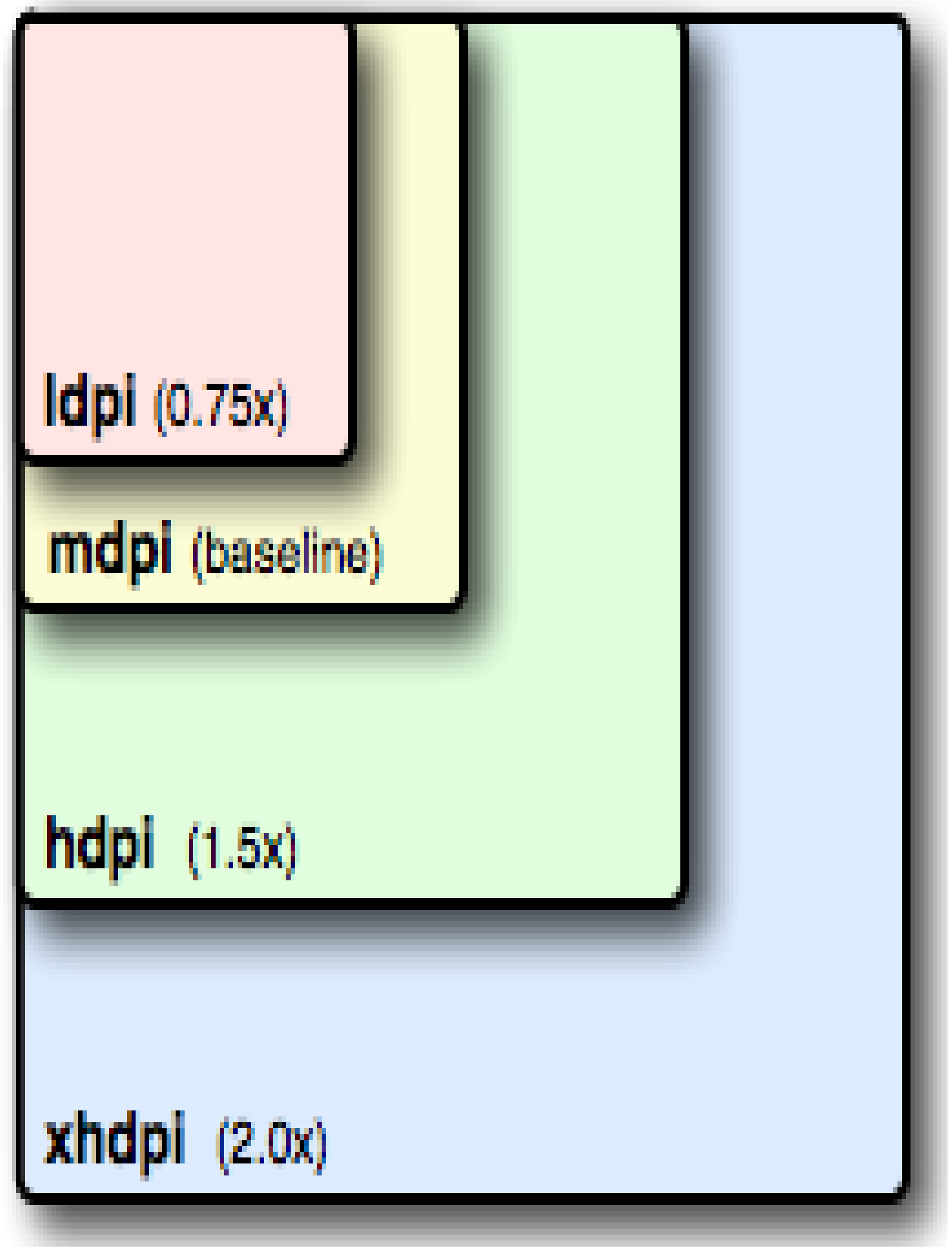
```
    float myDimension =
  getResources().getDimension(R.dimen.textPointSize);
```

Warning

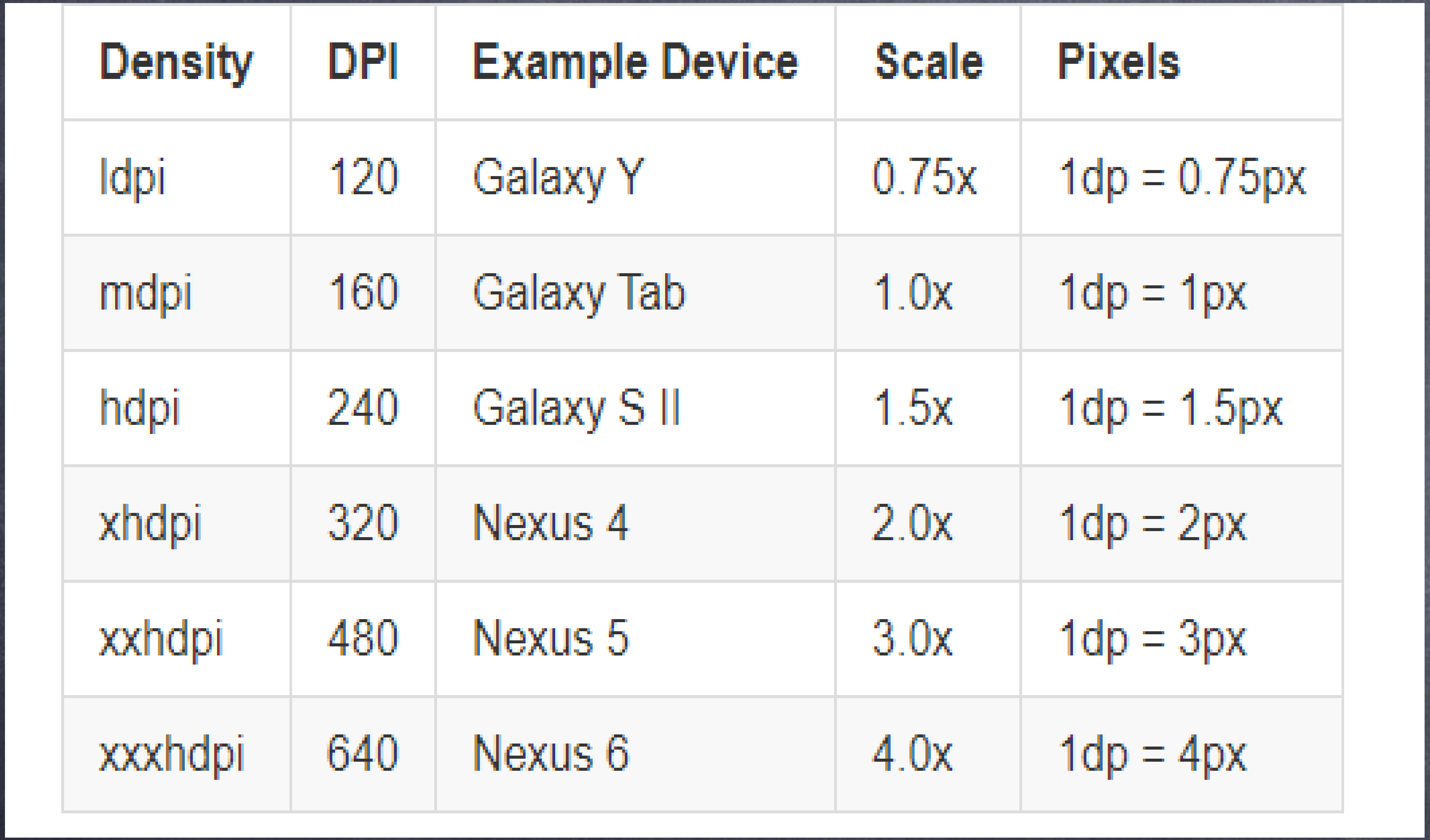
Be cautious when choosing dimension units for your applications. If you are planning to target multiple devices, with different scree
n sizes and resolutions, you need to rely heavily on the more scalable dimension units, such as dp and sp, as opposed to pixels, points,
inches, and millimeters.

Providing alternative resources-Dimension

To specify configuration-specific alternatives for a set of resources, we create a new directory in `res` in the form of `[resource]-[qualifiers]`. For example, one best practice is to ensure that all images are provided for multiple screen densities.

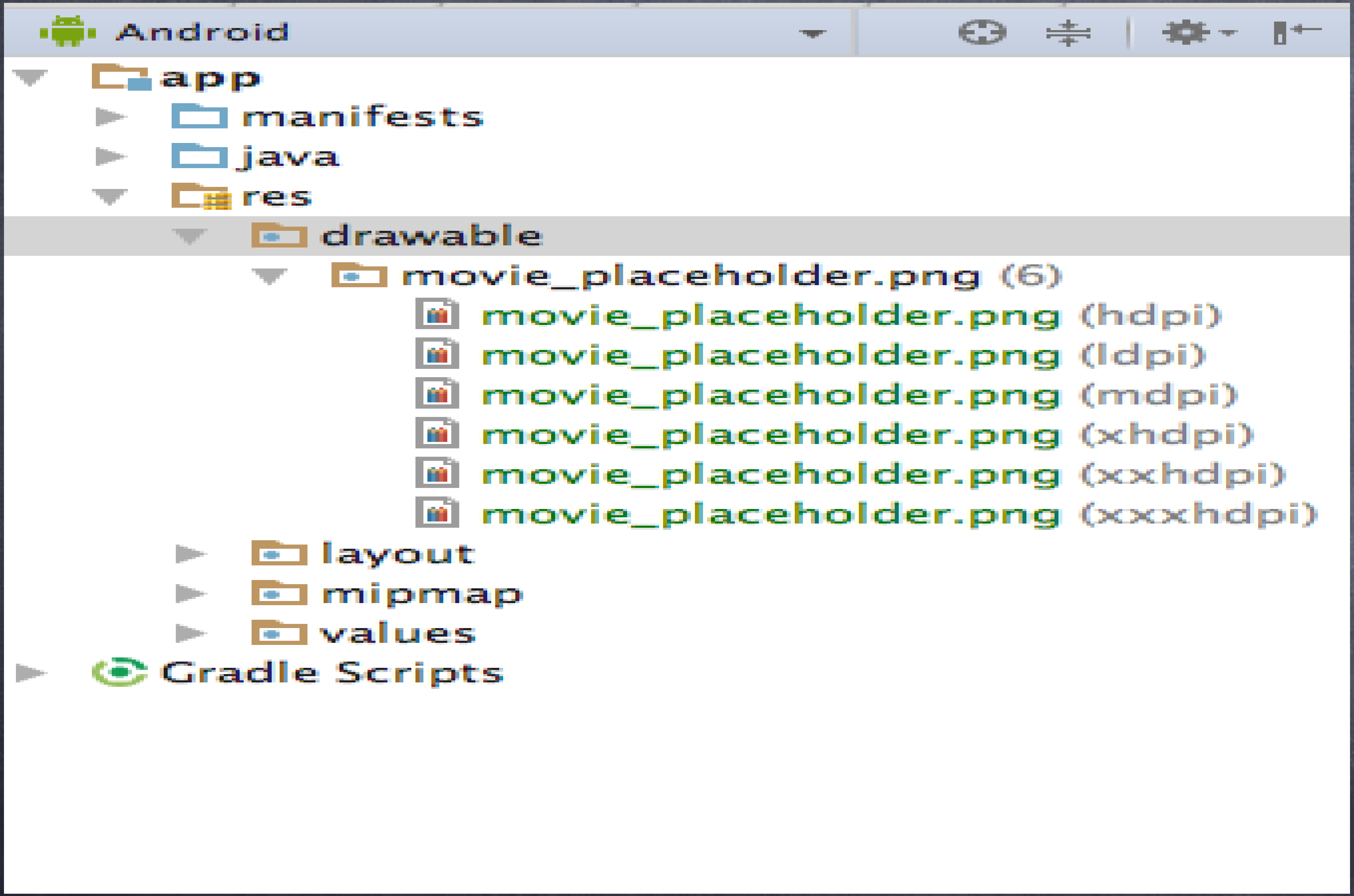


Providing alternative resources-Dimension



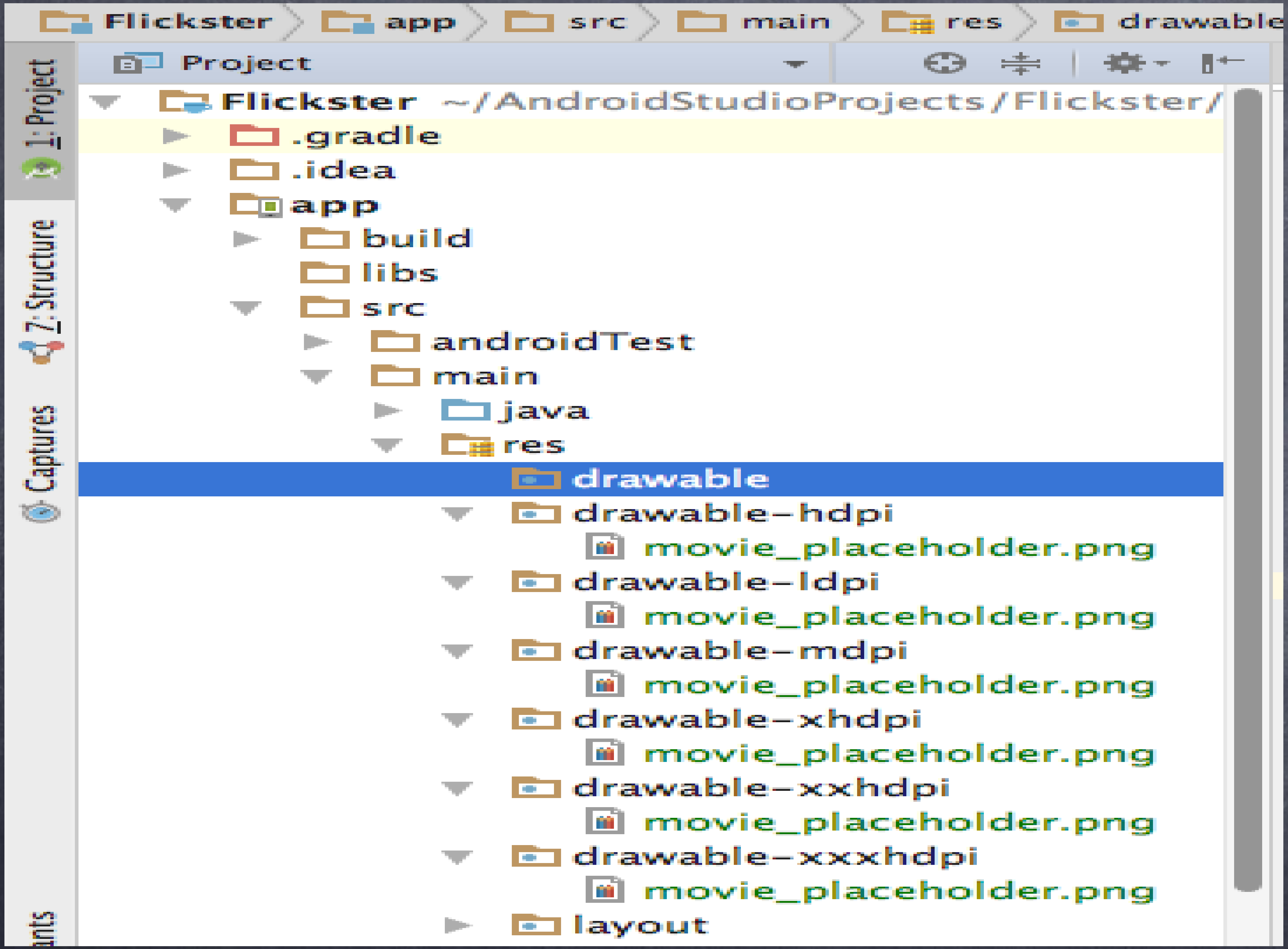
Density	DPI	Example Device	Scale	Pixels
ldpi	120	Galaxy Y	0.75x	1dp = 0.75px
mdpi	160	Galaxy Tab	1.0x	1dp = 1px
hdpi	240	Galaxy S II	1.5x	1dp = 1.5px
xhdpi	320	Nexus 4	2.0x	1dp = 2px
xxhdpi	480	Nexus 5	3.0x	1dp = 3px
xxxhdpi	640	Nexus 6	4.0x	1dp = 4px

Providing alternative resources-Dimension



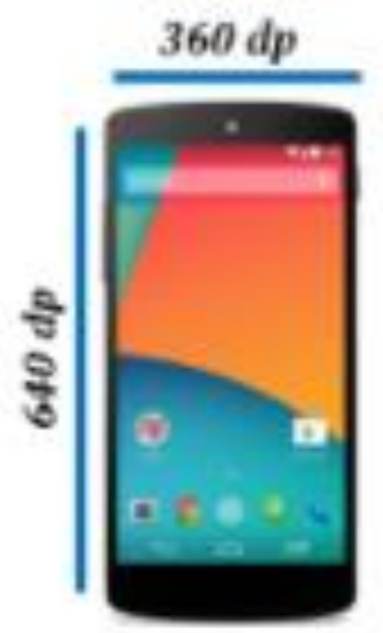
```
Android
▼ app
    ► manifests
    ► java
    ▼ res
        ▼ drawable
            ▼ movie_placeholder.png (6)
                movie_placeholder.png (hdpi)
                movie_placeholder.png (ldpi)
                movie_placeholder.png (mdpi)
                movie_placeholder.png (xhdpi)
                movie_placeholder.png (xxhdpi)
                movie_placeholder.png (xxxhdpi)
        ► layout
        ► mipmap
        ► values
► Gradle Scripts
```

Providing alternative resources-Dimension

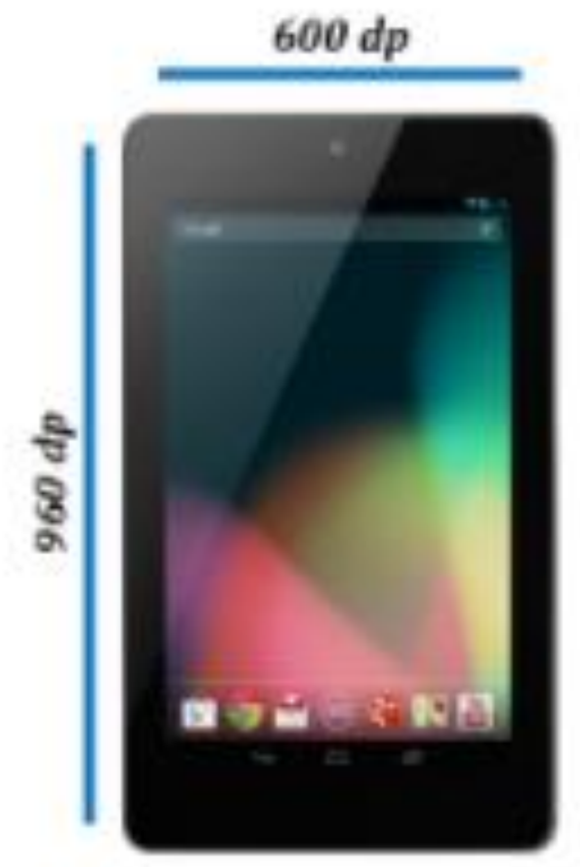


```
Flickster ~/AndroidStudioProjects/Flickster/
  .gradle
  .idea
  app
    build
    libs
    src
      androidTest
      main
        java
        res
          drawable
          drawable-hdpi
            movie_placeholder.png
          drawable-ldpi
            movie_placeholder.png
          drawable-mdpi
            movie_placeholder.png
          drawable-xhdpi
            movie_placeholder.png
          drawable-xxhdpi
            movie_placeholder.png
          drawable-xxxhdpi
            movie_placeholder.png
          layout
```

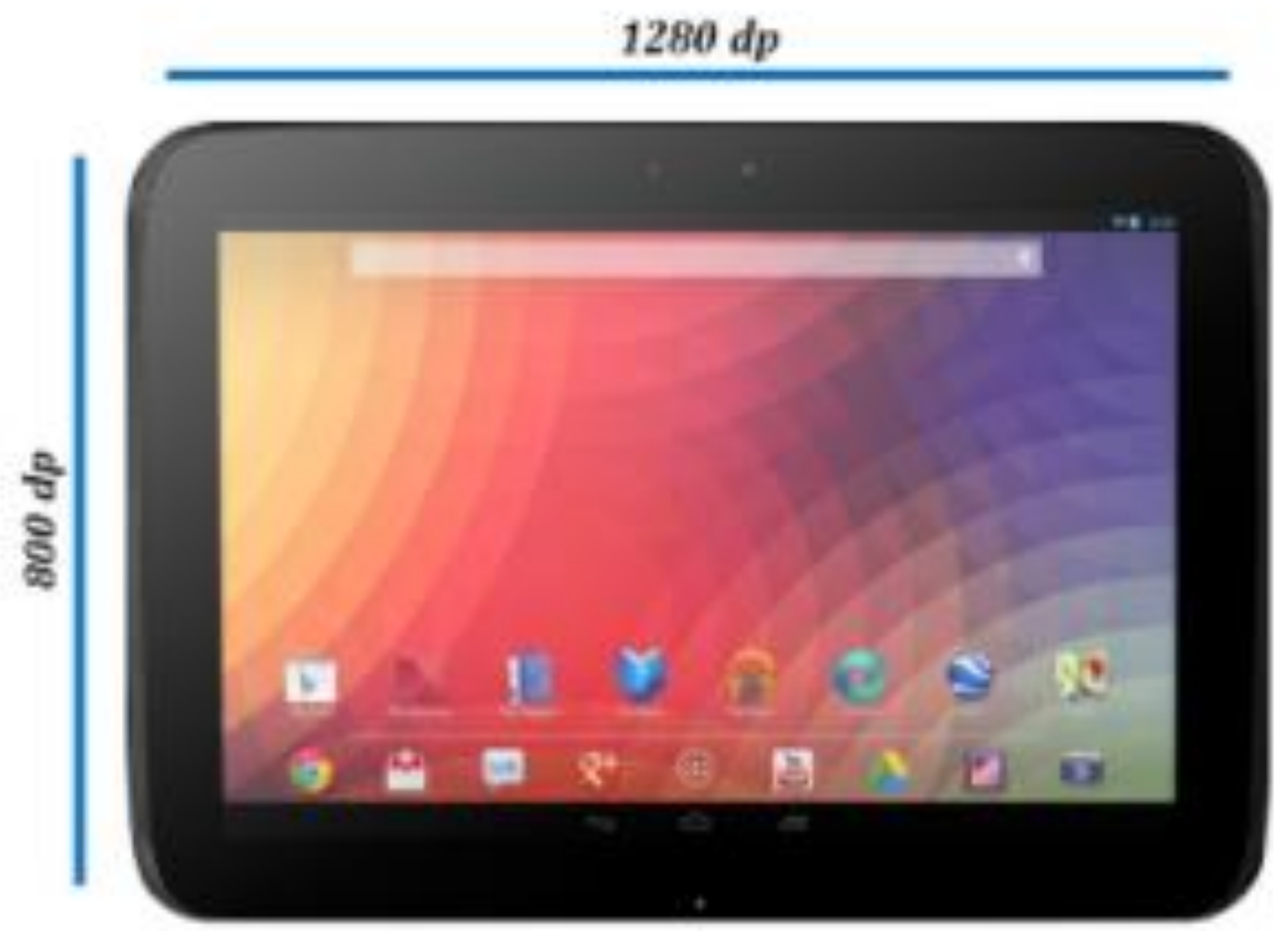
Providing alternative resources-Dimension



For this mobile the smallest width is 360, so it loads resources from res/layout.

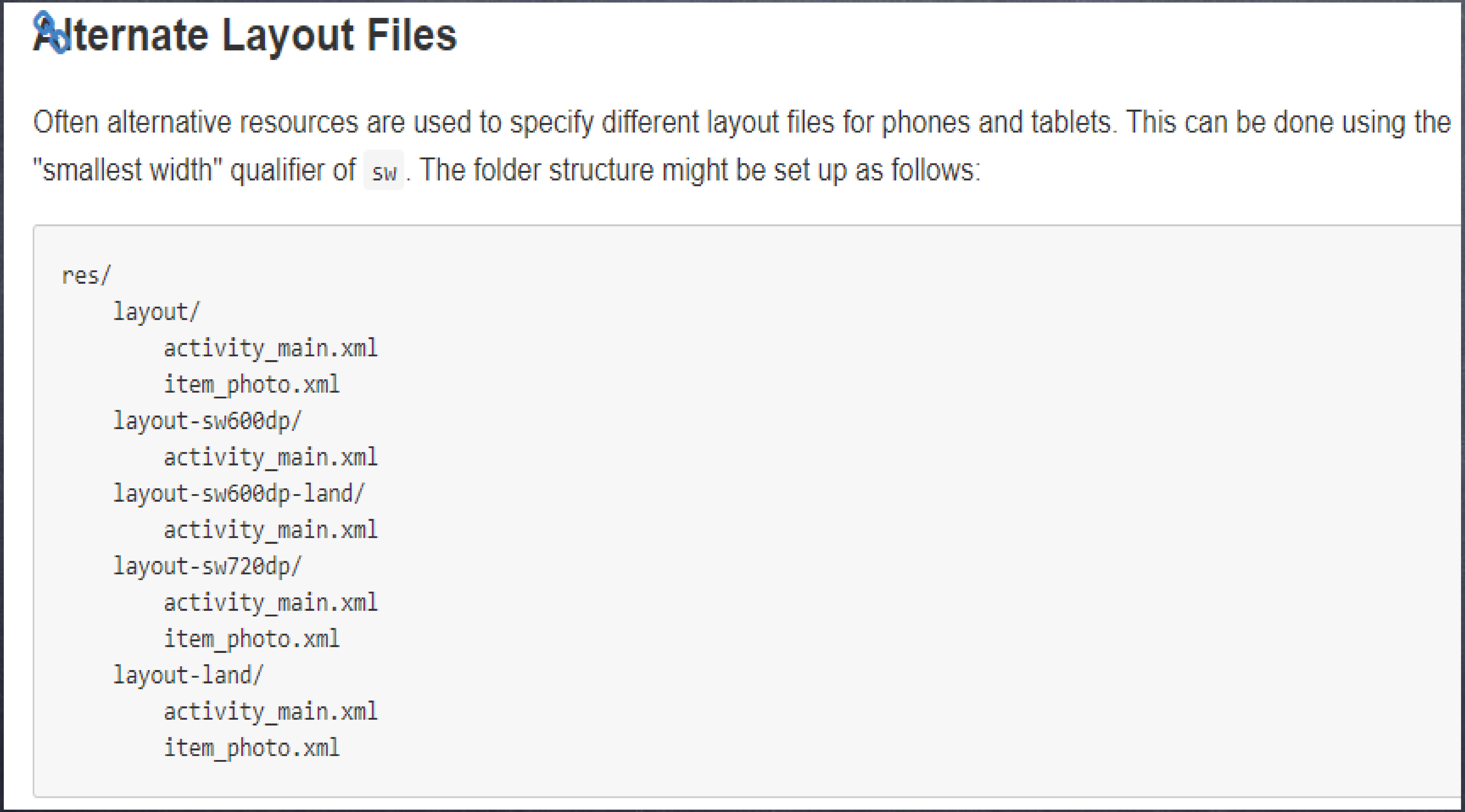


For this device the smallest width is 600, so android first checks res/layout-sw600dp. If it is not available it falls back to lower folder.



For this device, smallest width is 800dp, so first it checks for res/layout800dp, if it is not available it falls back to lower folder.

Providing alternative resources-Dimension

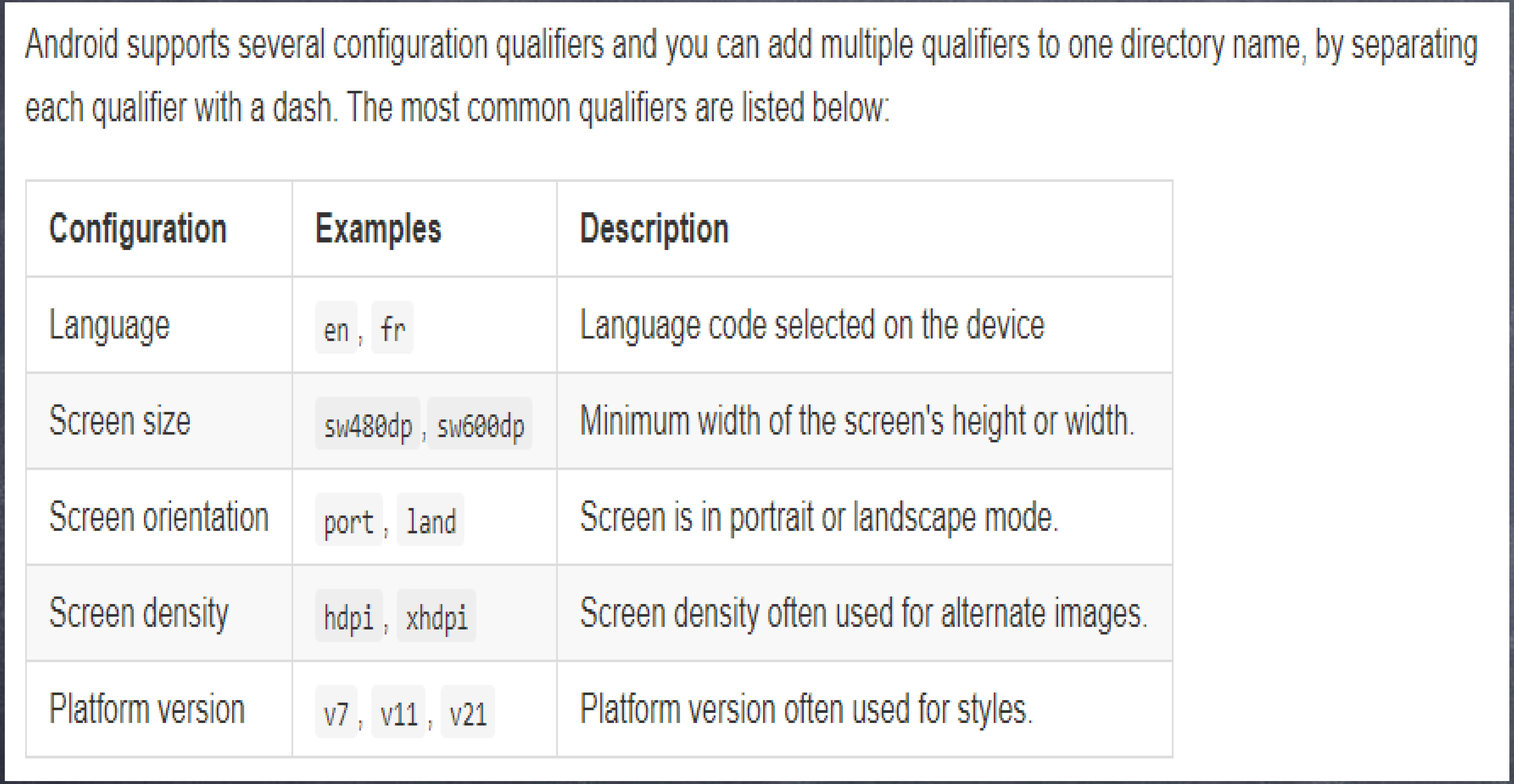


Alternate Layout Files

Often alternative resources are used to specify different layout files for phones and tablets. This can be done using the "smallest width" qualifier of `sw`. The folder structure might be set up as follows:

```
res/
    layout/
        activity_main.xml
        item_photo.xml
    layout-sw600dp/
        activity_main.xml
    layout-sw600dp-land/
        activity_main.xml
    layout-sw720dp/
        activity_main.xml
        item_photo.xml
    layout-land/
        activity_main.xml
        item_photo.xml
```

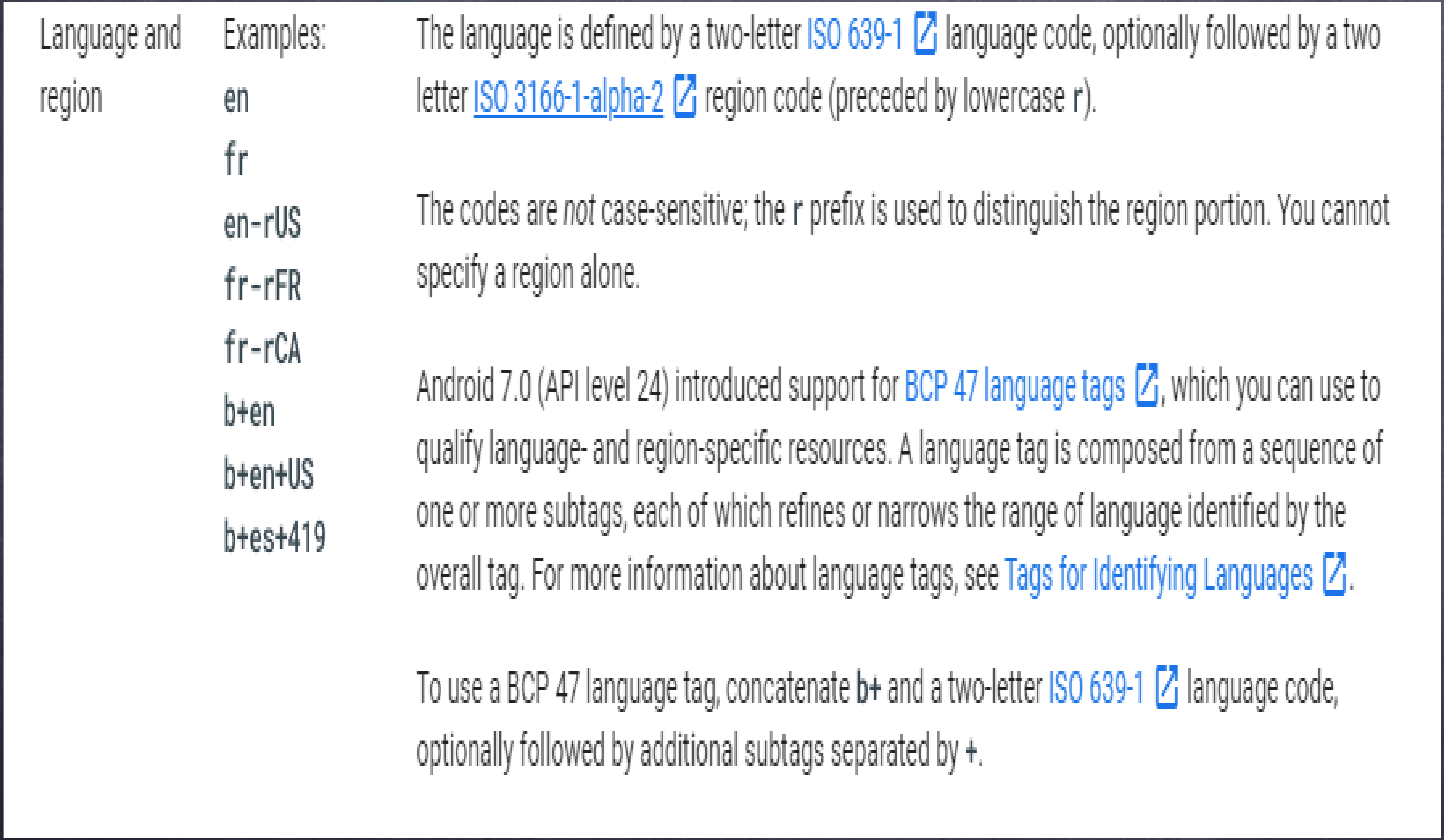
Providing alternative resources-Dimension



Android supports several configuration qualifiers and you can add multiple qualifiers to one directory name, by separating each qualifier with a dash. The most common qualifiers are listed below:

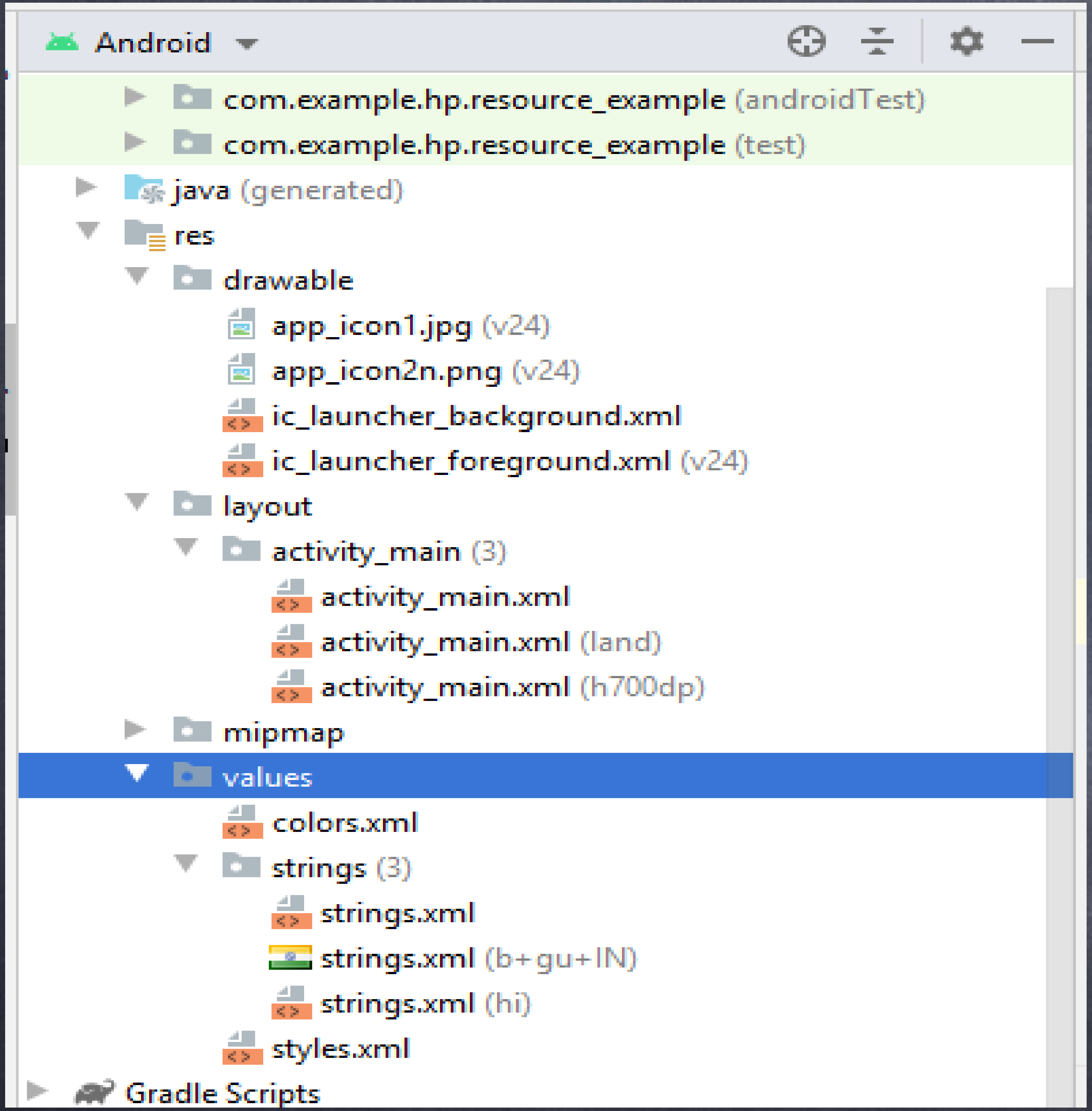
Configuration	Examples	Description
Language	`en`, `fr`	Language code selected on the device
Screen size	`sw480dp`, `sw600dp`	Minimum width of the screen's height or width.
Screen orientation	`port`, `land`	Screen is in portrait or landscape mode.
Screen density	`hdpi`, `xhdpi`	Screen density often used for alternate images.
Platform version	`v7`, `v11`, `v21`	Platform version often used for styles.

alternative resources-Multi Language

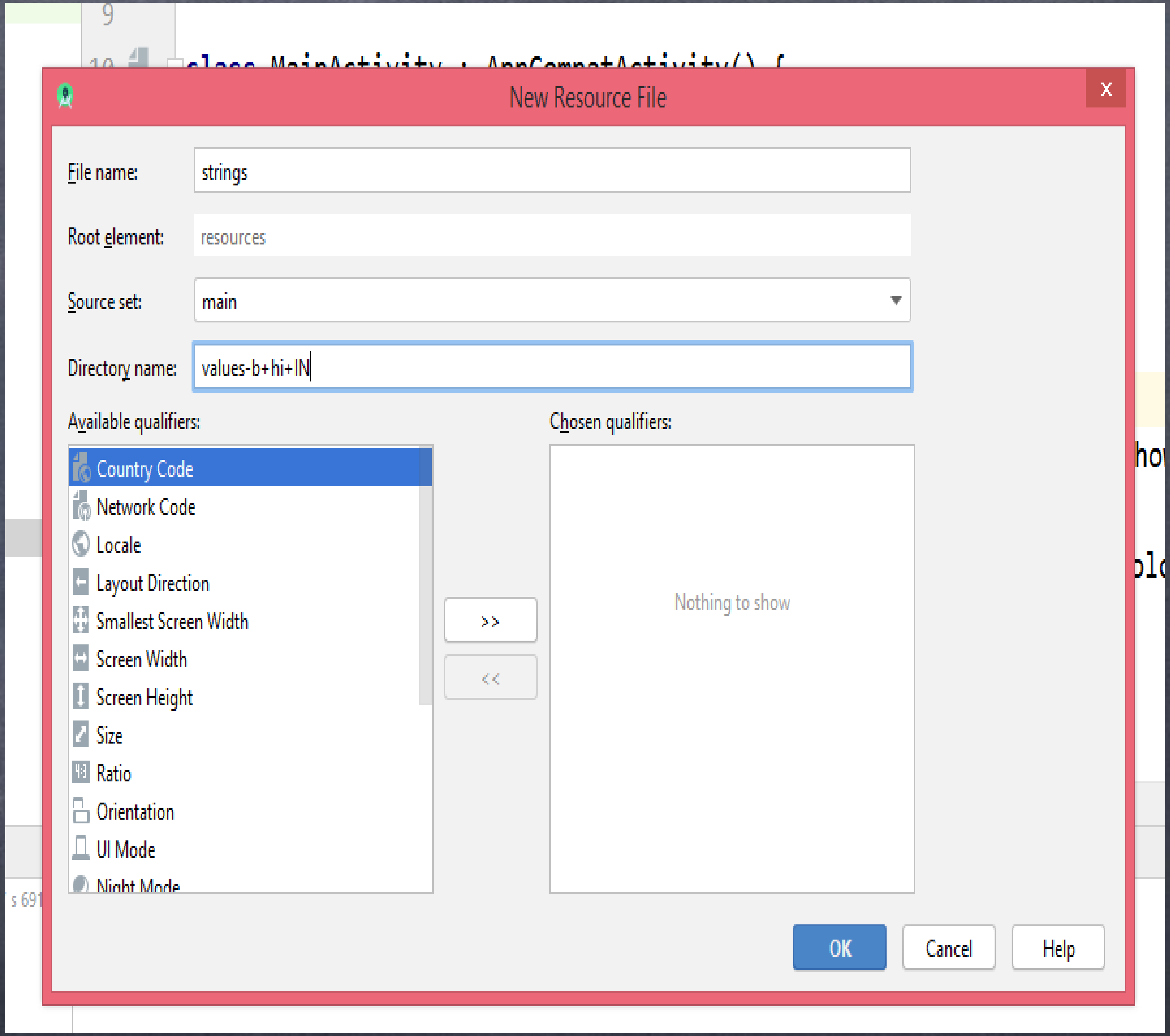


Language and region	Examples: en fr en-rUS fr-rFR fr-rCA b+en b+en+US b+es+419	The language is defined by a two-letter ISO 639-1 language code, optionally followed by a two letter ISO 3166-1-alpha-2 region code (preceded by lowercase r). The codes are *not* case-sensitive; the r prefix is used to distinguish the region portion. You cannot specify a region alone. Android 7.0 (API level 24) introduced support for BCP 47 language tags, which you can use to qualify language- and region-specific resources. A language tag is composed from a sequence of one or more subtags, each of which refines or narrows the range of language identified by the overall tag. For more information about language tags, see Tags for Identifying Languages. To use a BCP 47 language tag, concatenate b+ and a two-letter ISO 639-1 language code, optionally followed by additional subtags separated by +.

alternative resources-Multi Language



```
Android
  ▶ com.example.hp.resource_example (androidTest)
  ▶ com.example.hp.resource_example (test)
▶ java (generated)
▼ res
  ▼ drawable
      app_icon1.jpg (v24)
      app_icon2n.png (v24)
      ic_launcher_background.xml
      ic_launcher_foreground.xml (v24)
  ▼ layout
    ▼ activity_main (3)
        activity_main.xml
        activity_main.xml (land)
        activity_main.xml (h700dp)
  ▶ mipmap
  ▼ values
      colors.xml
    ▼ strings (3)
        strings.xml
        strings.xml (b+gu+IN)
        strings.xml (hi)
      styles.xml
▶ Gradle Scripts
```



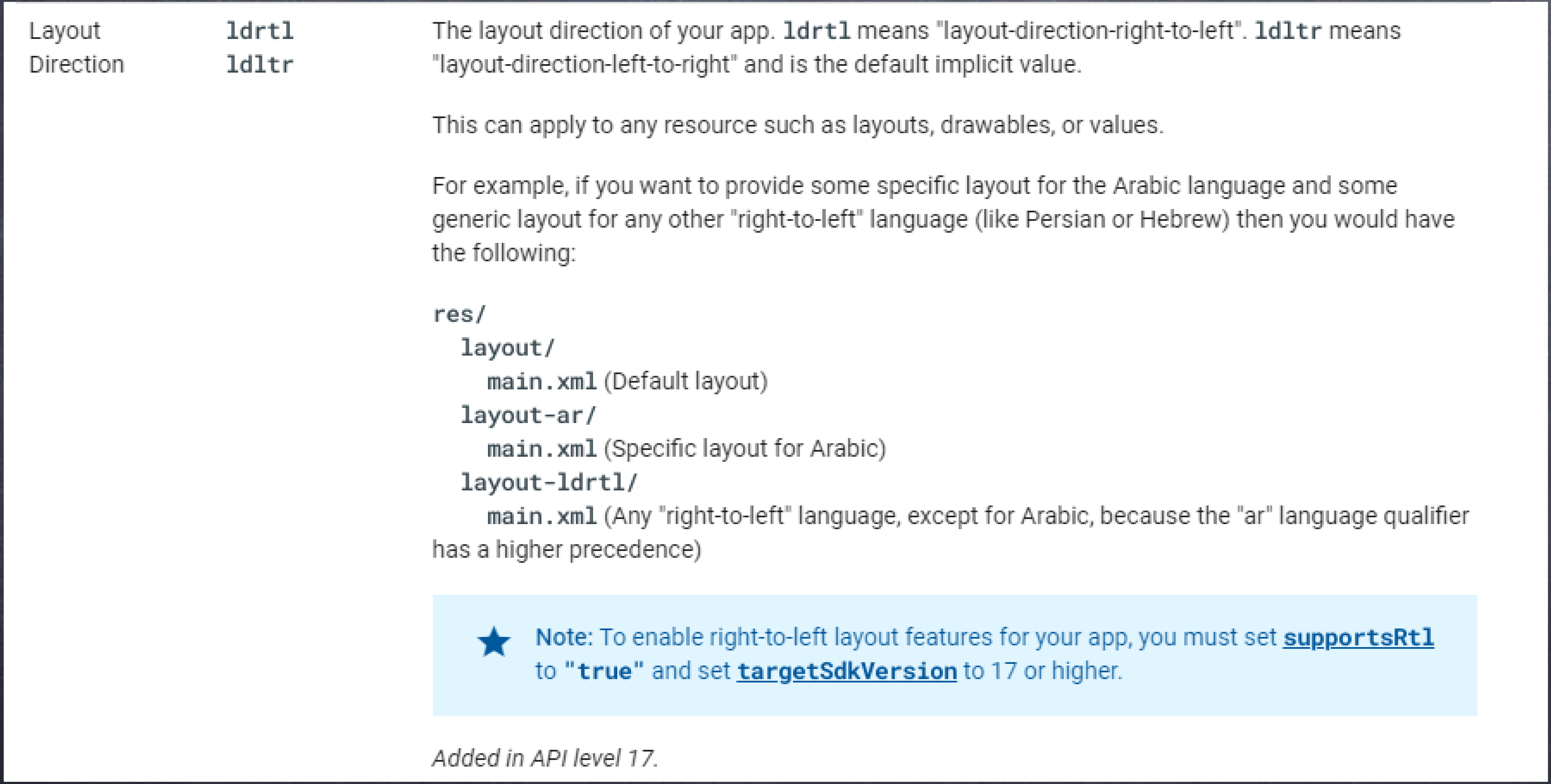
New Resource File

File name: strings
Root element: resources
Source set: main
Directory name: values-b+hi+IN

Available qualifiers:
- Country Code
- Network Code
- Locale
- Layout Direction
- Smallest Screen Width
- Screen Width
- Screen Height
- Size
- Ratio
- Orientation
- UI Mode
- Night Mode

Chosen qualifiers:
Nothing to show

Layout Direction



Layout Direction	`ldrtl` `ldltr`	The layout direction of your app. `ldrtl` means "layout-direction-right-to-left". `ldltr` means "layout-direction-left-to-right" and is the default implicit value.

This can apply to any resource such as layouts, drawables, or values.

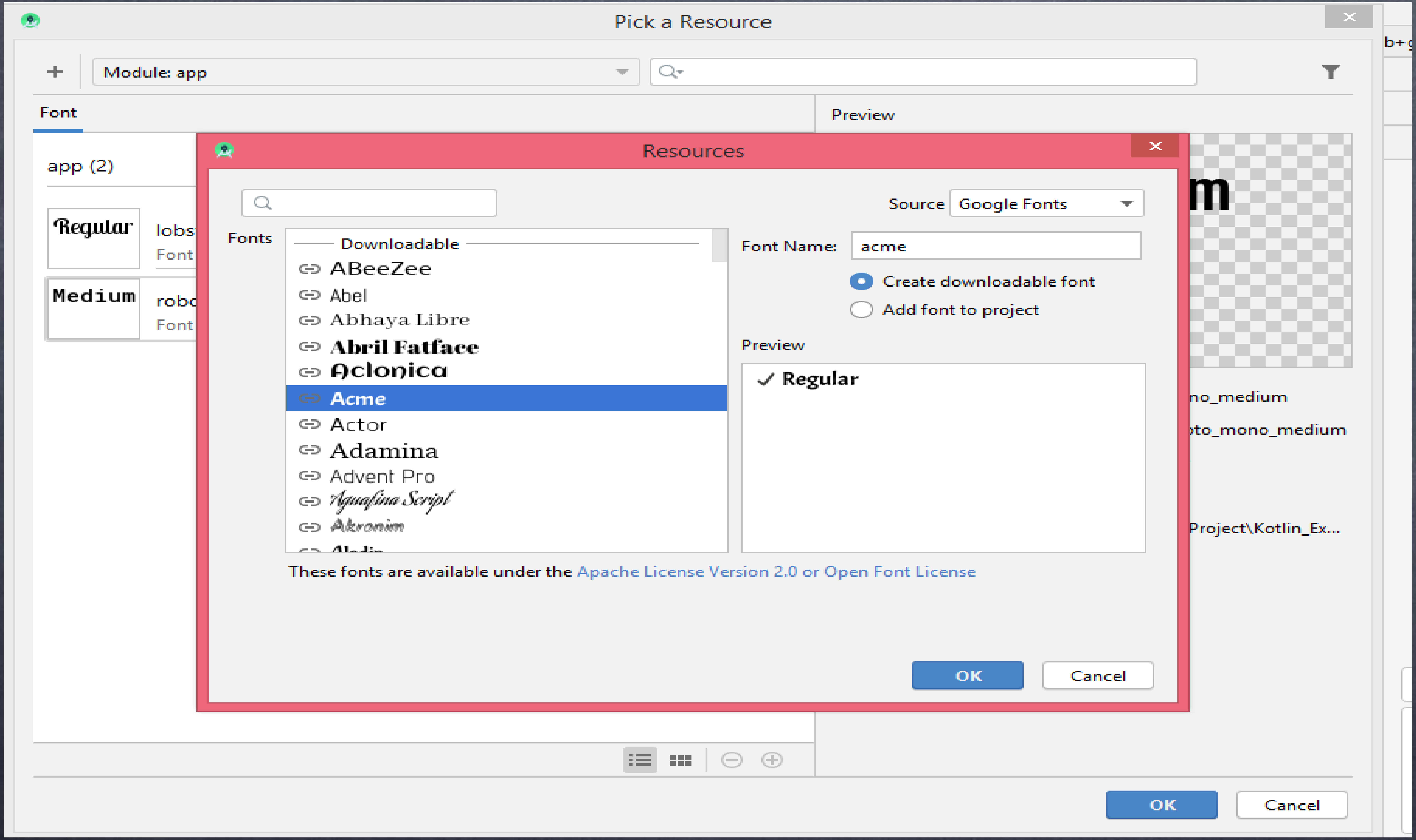
For example, if you want to provide some specific layout for the Arabic language and some generic layout for any other "right-to-left" language (like Persian or Hebrew) then you would have the following:

```
res/
  layout/
    main.xml (Default layout)
  layout-ar/
    main.xml (Specific layout for Arabic)
  layout-ldrtl/
    main.xml (Any "right-to-left" language, except for Arabic, because the "ar" language qualifier
has a higher precedence)
```

★ **Note:** To enable right-to-left layout features for your app, you must set `supportsRtl` to `"true"` and set `targetSdkVersion` to 17 or higher.

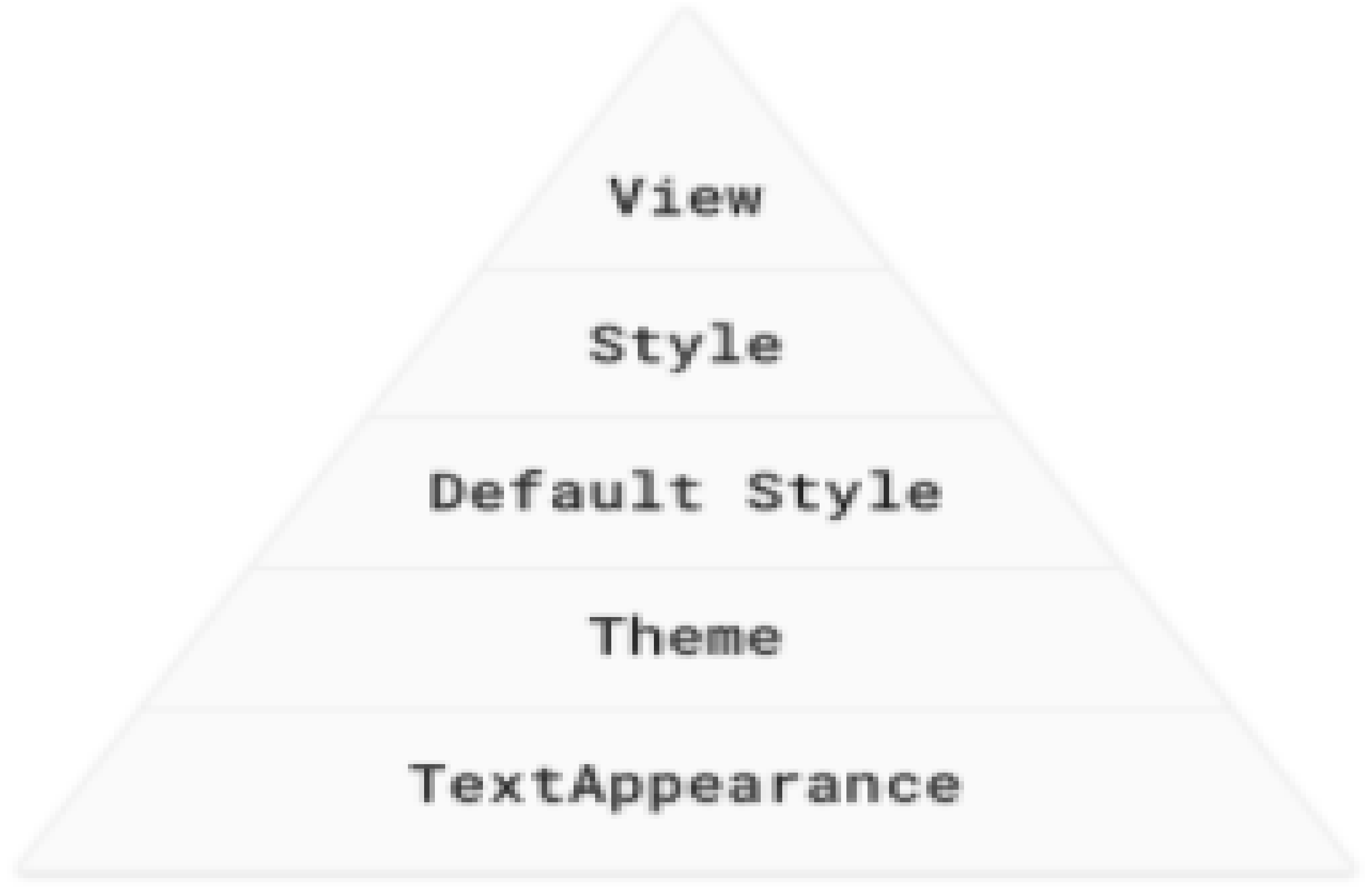
Added in API level 17.

Font:


Pick a Resource
Module: app
Font
Preview
app (2)
Regular
lobs
Font
Medium
robo
Font
Resources
Source
Google Fonts
Fonts
Downloadable
ABeeZee
Abel
Abhaya Libre
Abril Fatface
Aclonica
Acme
Actor
Adamina
Advent Pro
Aguafina Script
Akronim
Aladin
Font Name:
acme
Create downloadable font
Add font to project
Preview
✓ Regular
These fonts are available under the Apache License Version 2.0 or Open Font License
OK
Cancel
no_medium
oto_mono_medium
Project\Kotlin_Ex...
OK
Cancel

Android Style:

Android provides a rich styling system that lets you control the appearance of all the views in your app. You can use themes, styles, and view attributes to affect styling. The diagram shown below summarizes the precedence of each method of styling. The pyramid diagram shows the order in which styling methods are applied by the system, from the bottom up. For example, if you set the text size in the theme, and then set the text size differently in the view attributes, the view attributes will override the theme styling.

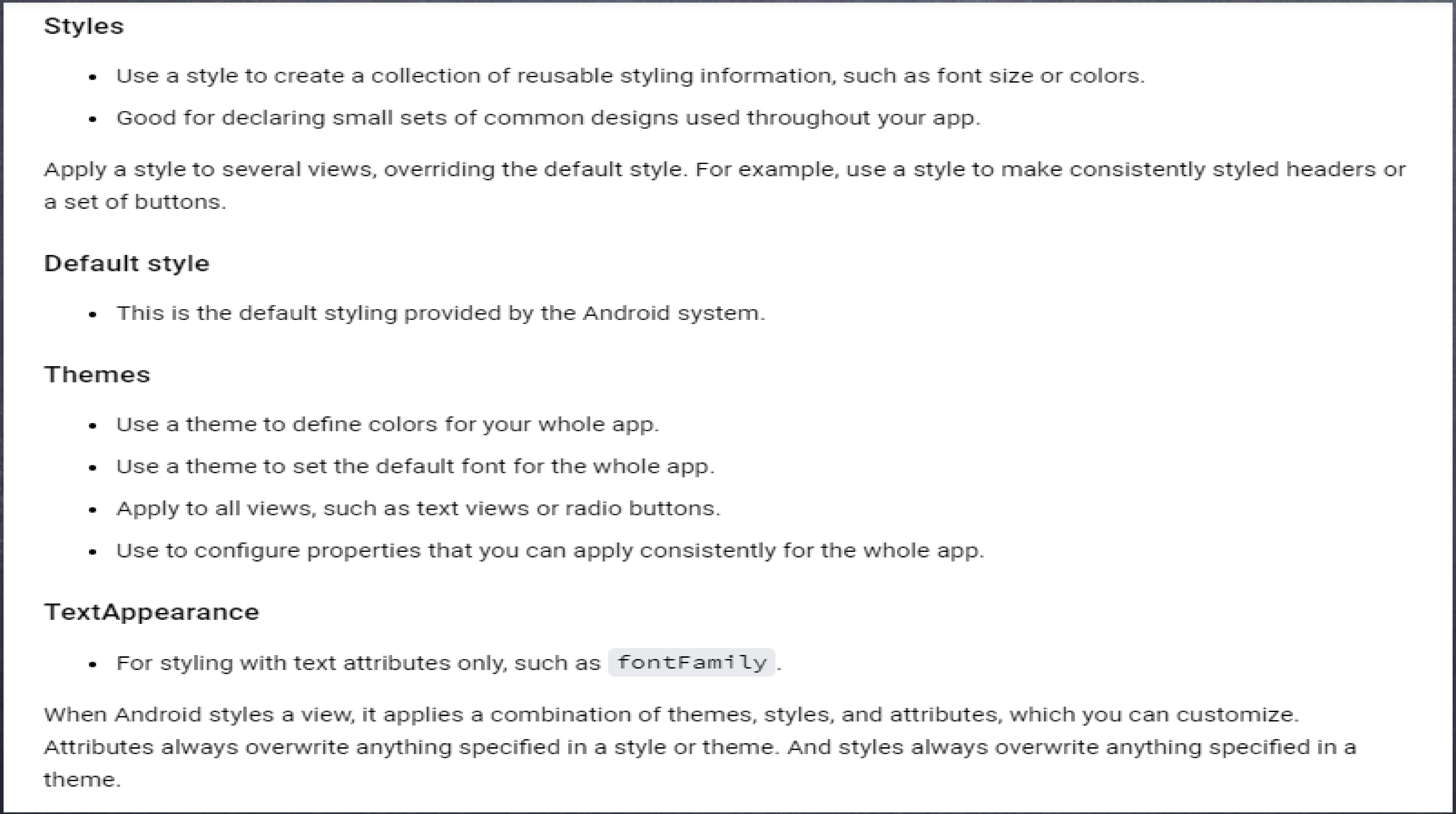


View attributes

- Use view attributes to set attributes explicitly for each view. (View attributes are not reusable, as styles are.)
- You can use every property that can be set via styles or themes.

Use for custom or one-off designs such as margins, paddings, or constraints.

Android Style:



Styles

- Use a style to create a collection of reusable styling information, such as font size or colors.
- Good for declaring small sets of common designs used throughout your app.

Apply a style to several views, overriding the default style. For example, use a style to make consistently styled headers or a set of buttons.

Default style

- This is the default styling provided by the Android system.

Themes

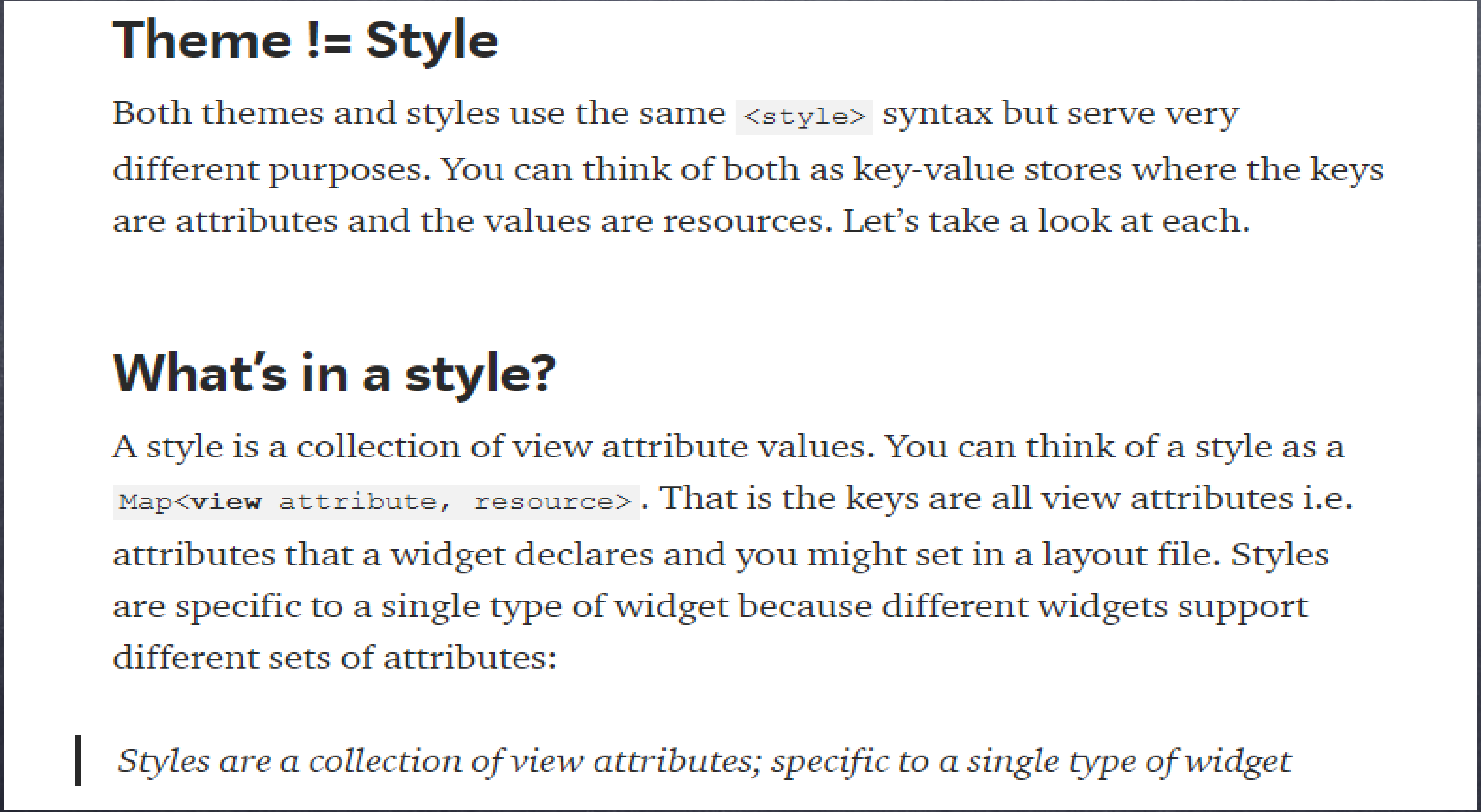
- Use a theme to define colors for your whole app.
- Use a theme to set the default font for the whole app.
- Apply to all views, such as text views or radio buttons.
- Use to configure properties that you can apply consistently for the whole app.

TextAppearance

- For styling with text attributes only, such as `fontFamily`.

When Android styles a view, it applies a combination of themes, styles, and attributes, which you can customize. Attributes always overwrite anything specified in a style or theme. And styles always overwrite anything specified in a theme.

Android styling: themes vs styles



Theme != Style

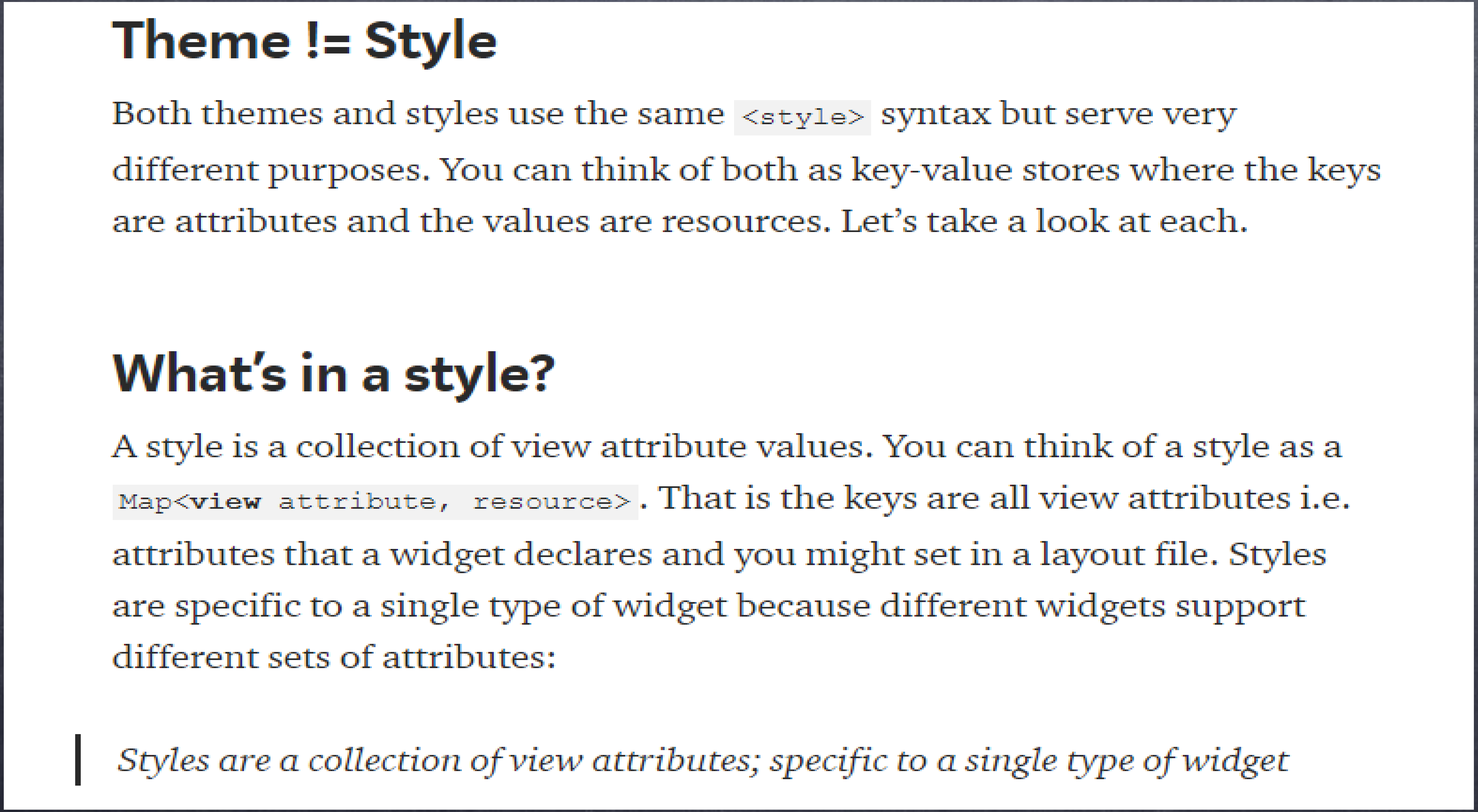
Both themes and styles use the same `<style>` syntax but serve very different purposes. You can think of both as key-value stores where the keys are attributes and the values are resources. Let's take a look at each.

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*

Android styling: themes vs styles



Theme != Style

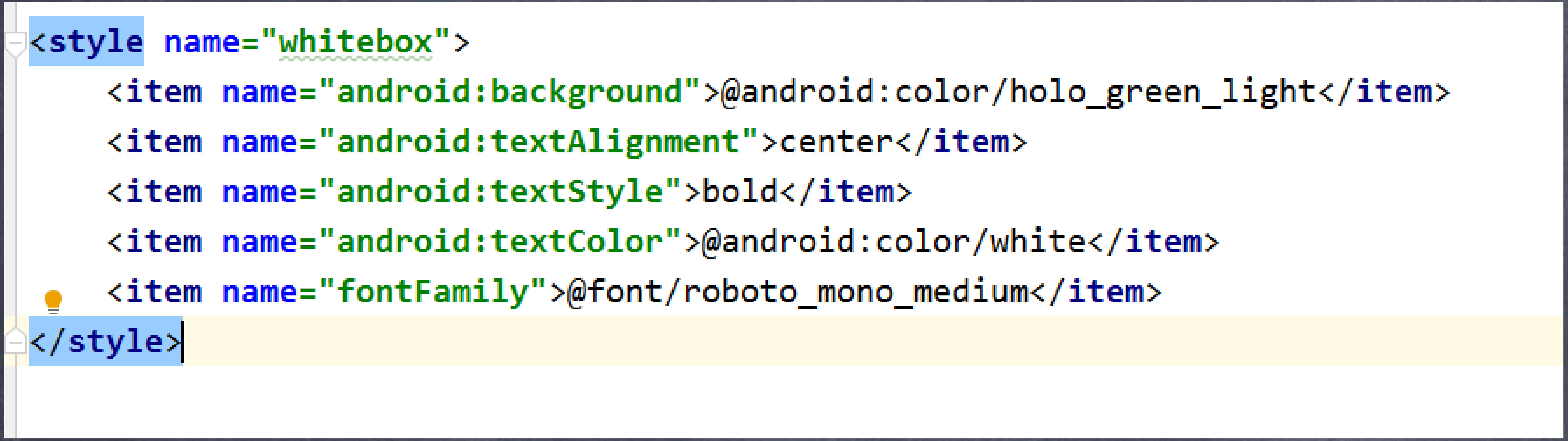
Both themes and styles use the same `<style>` syntax but serve very different purposes. You can think of both as key-value stores where the keys are attributes and the values are resources. Let's take a look at each.

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*

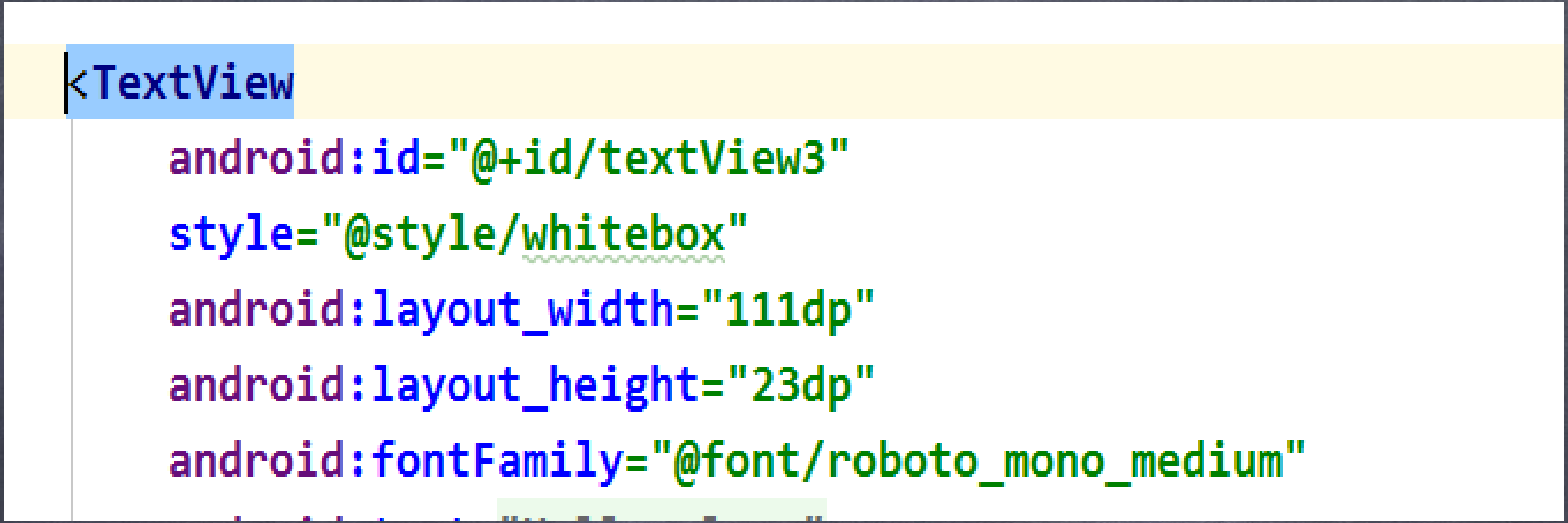
Android styling: themes vs styles



```
<style name="whitebox">
    <item name="android:background">@android:color/holo_green_light</item>
    <item name="android:textAlignment">center</item>
    <item name="android:textStyle">bold</item>
    <item name="android:textColor">@android:color/white</item>
    <item name="fontFamily">@font/roboto_mono_medium</item>
</style>
```

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

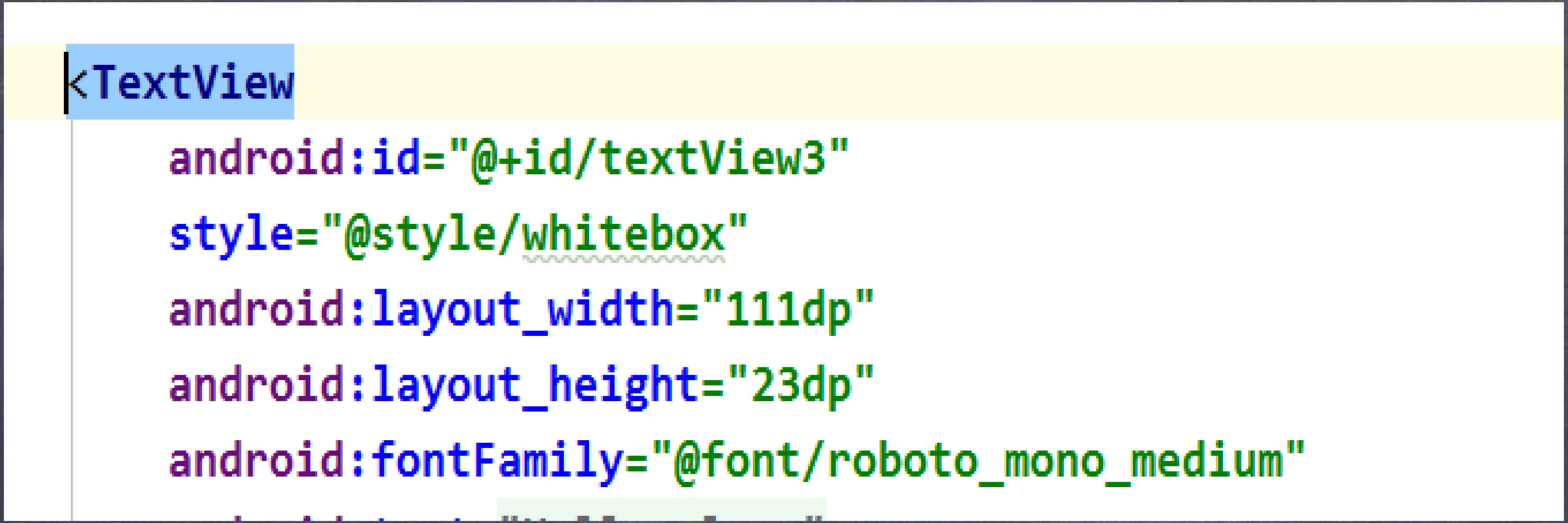
Android styling: themes vs styles



```
<TextView
    android:id="@+id/textView3"
    style="@style/whitebox"
    android:layout_width="111dp"
    android:layout_height="23dp"
    android:fontFamily="@font/roboto_mono_medium"
```

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

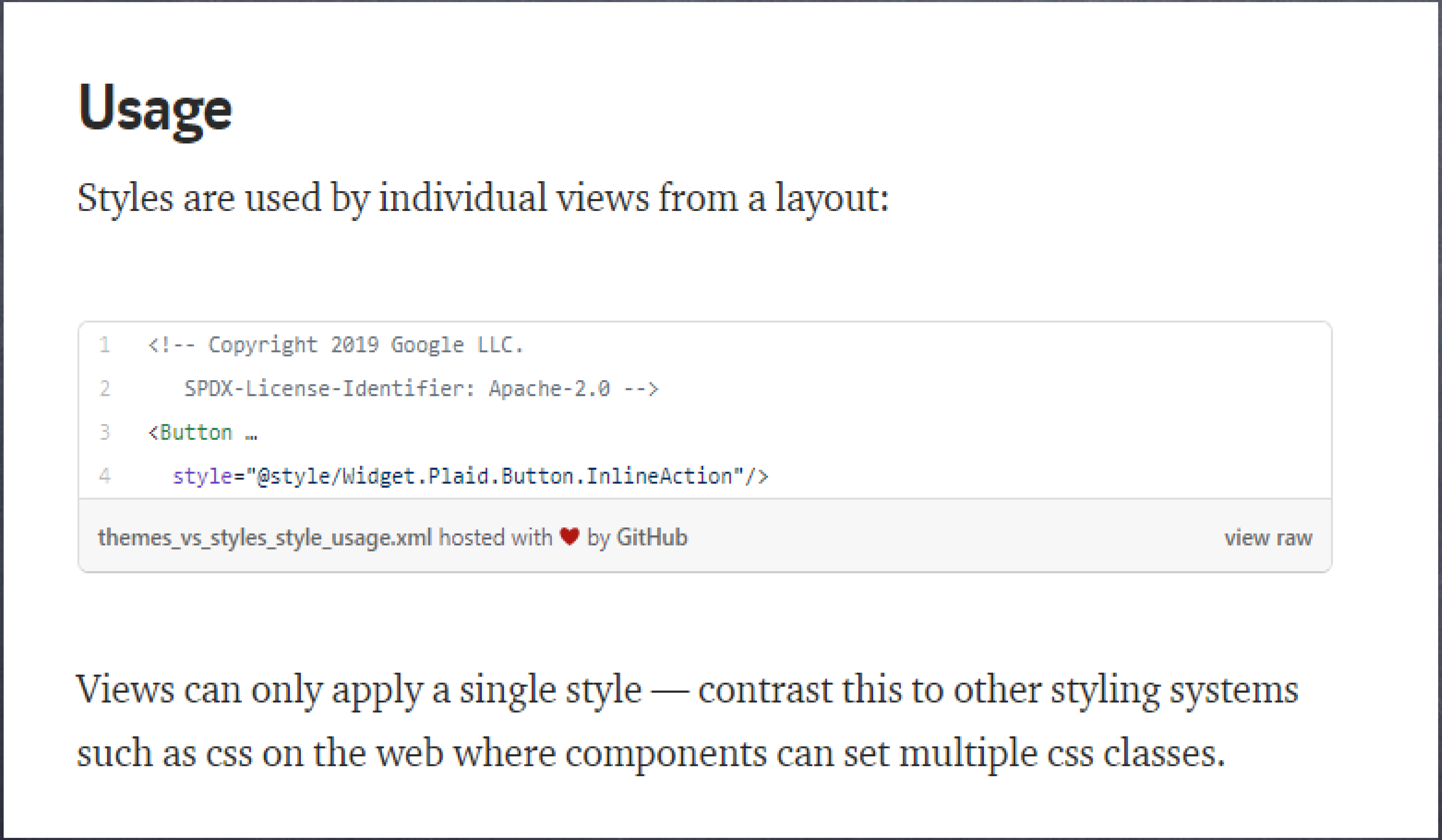
Android styling: themes vs styles



```
<TextView
    android:id="@+id/textView3"
    style="@style/whitebox"
    android:layout_width="111dp"
    android:layout_height="23dp"
    android:fontFamily="@font/roboto_mono_medium"
```

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

Android styling: themes vs styles



Usage

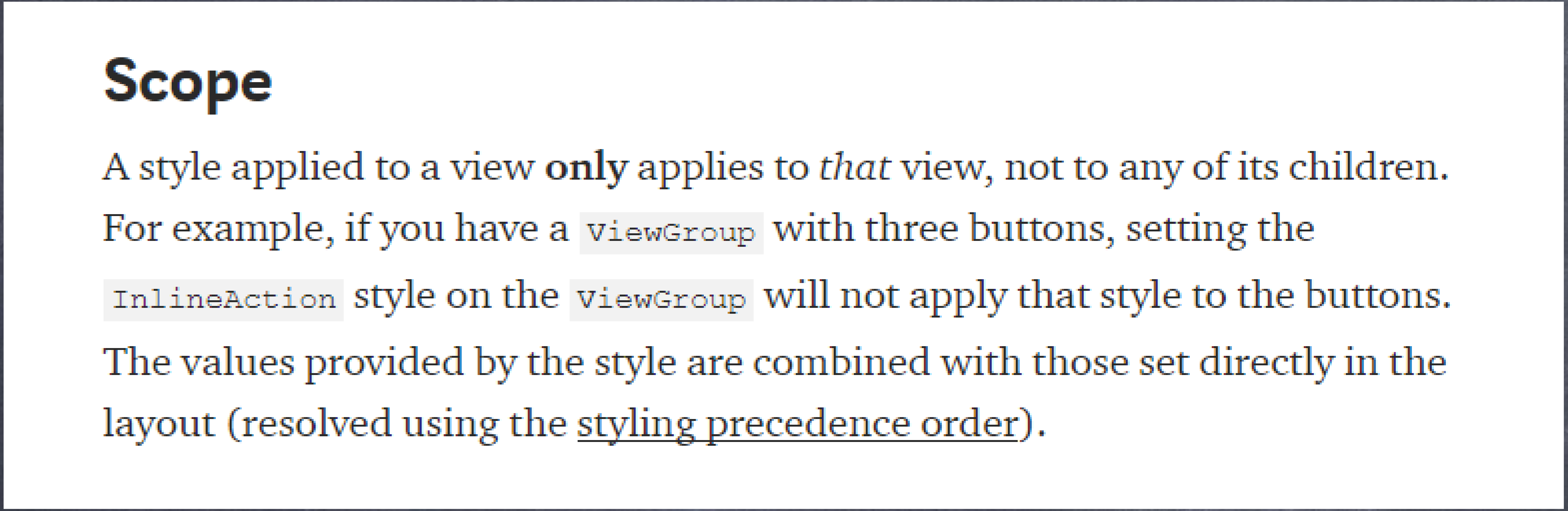
Styles are used by individual views from a layout:

```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.

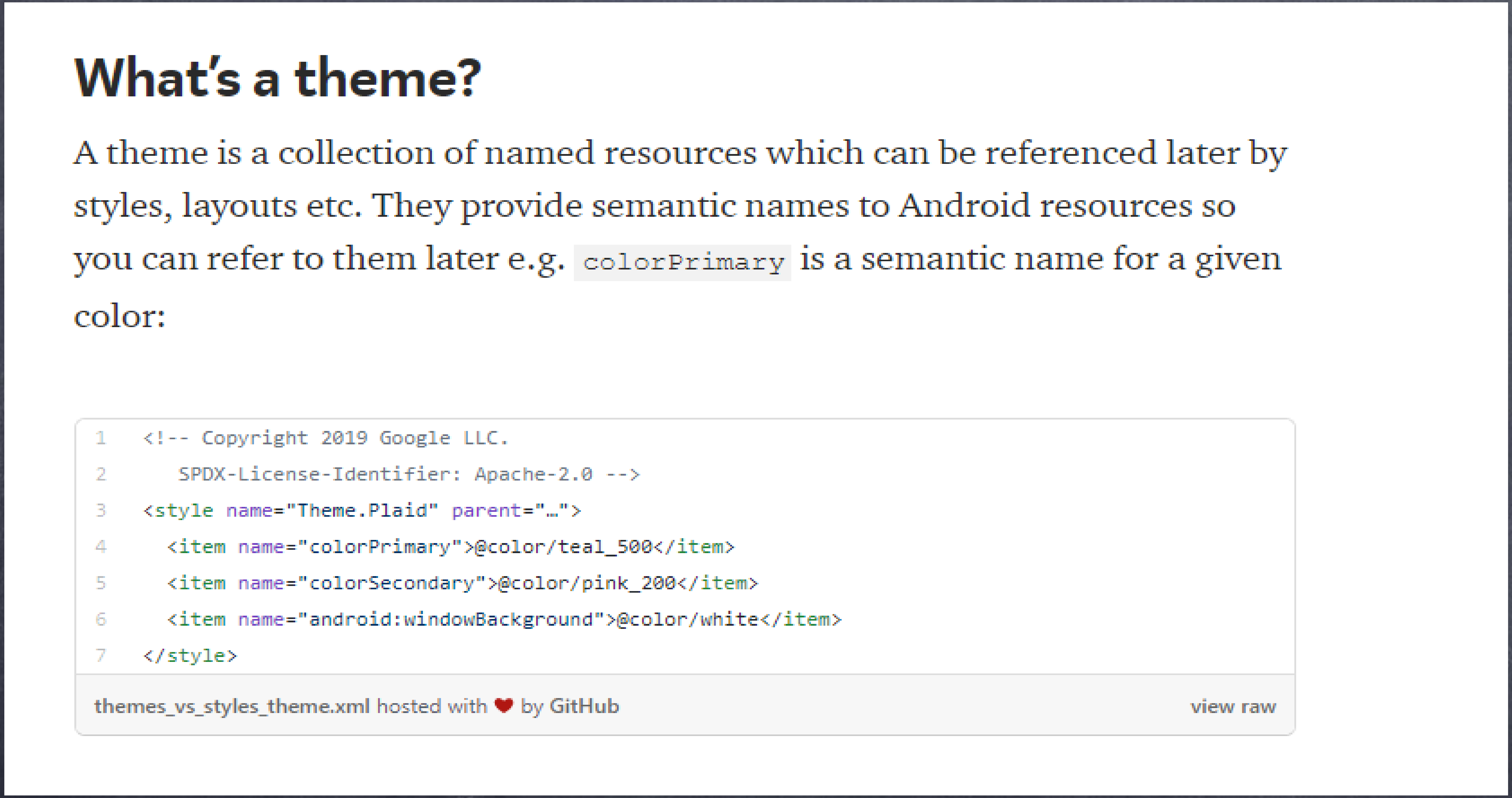
Android styling: themes vs styles



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).

Android styling: themes vs styles



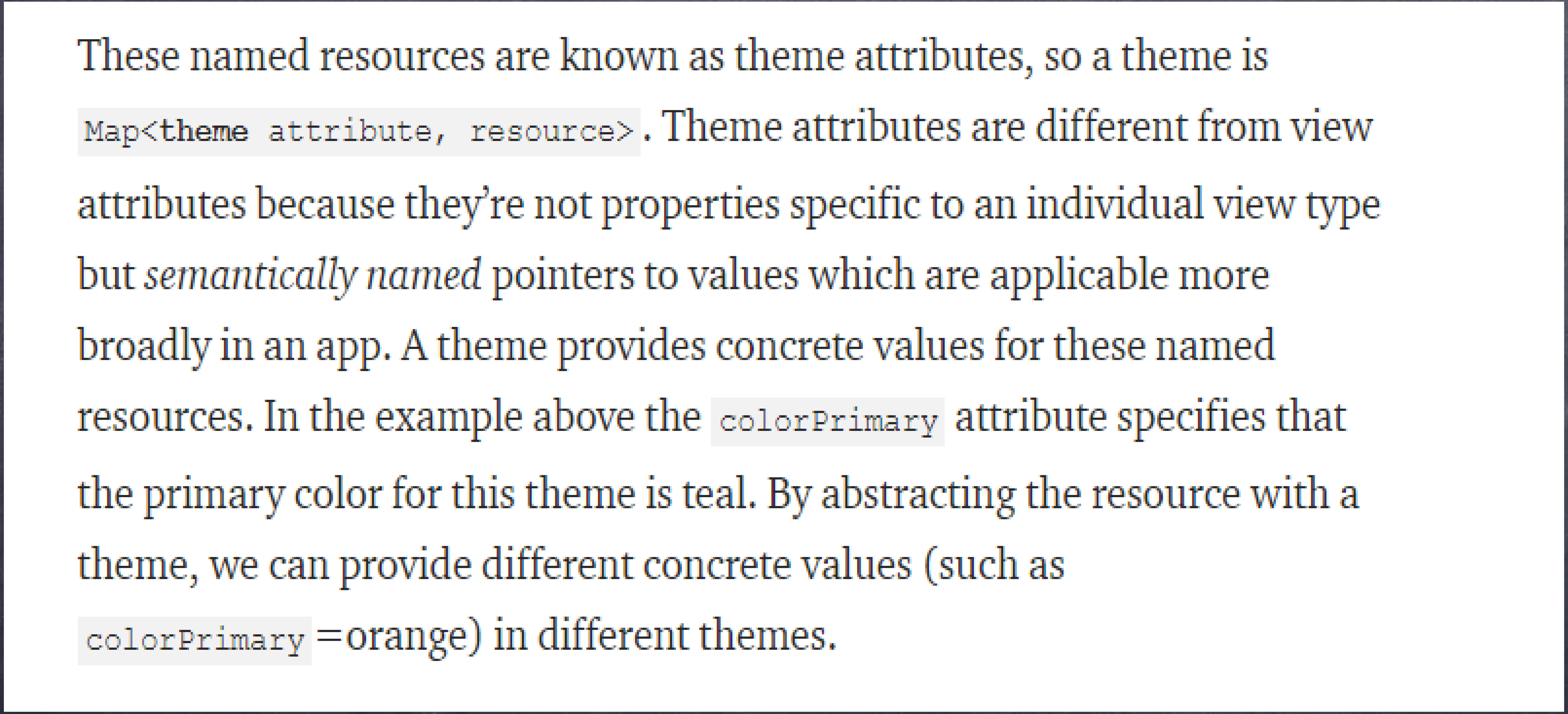
What's a theme?

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>
```

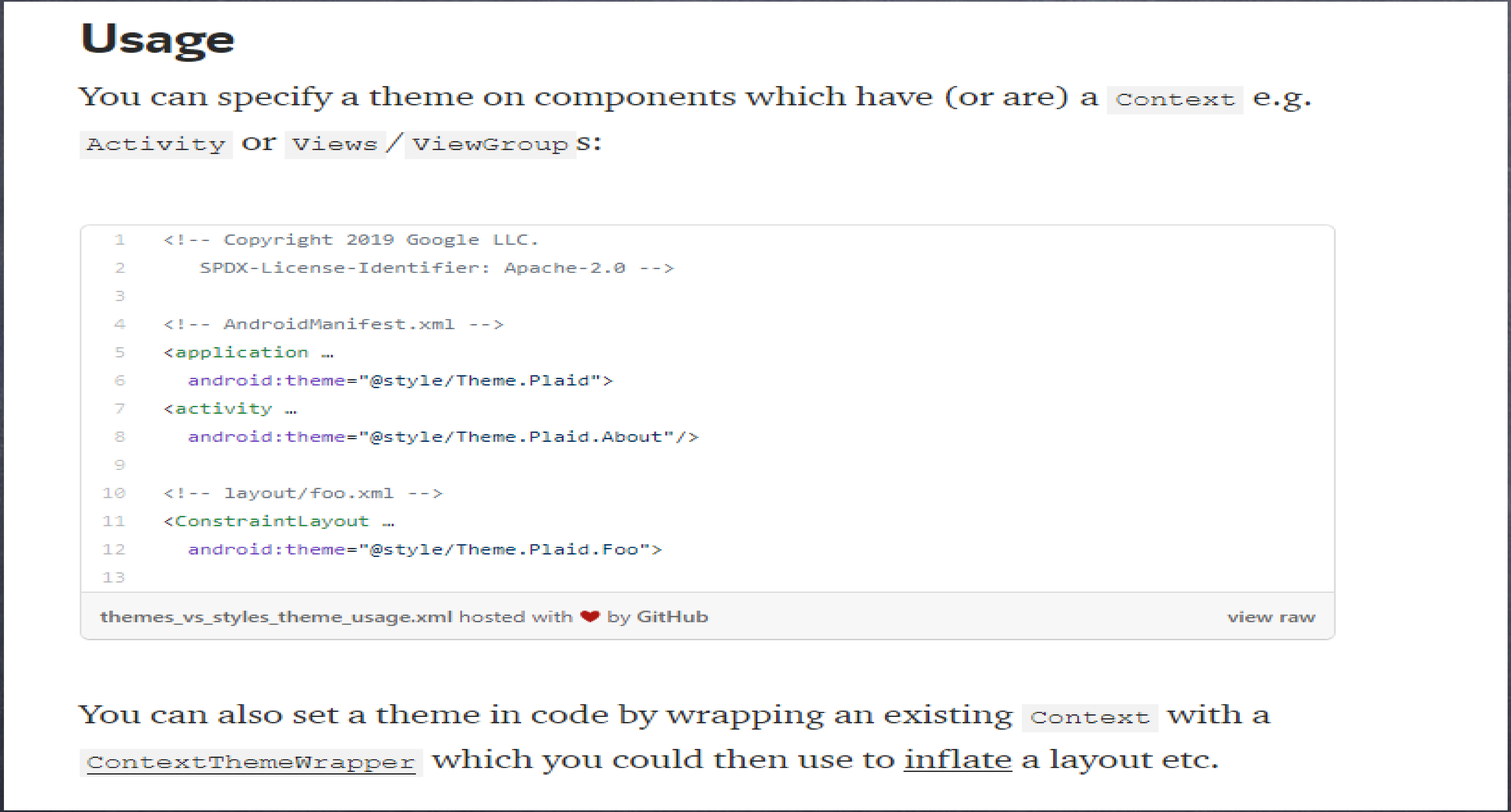
themes_vs_styles_theme.xml hosted with ❤ by GitHub view raw

Android styling: themes vs styles



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.

Android styling: themes vs styles



Usage

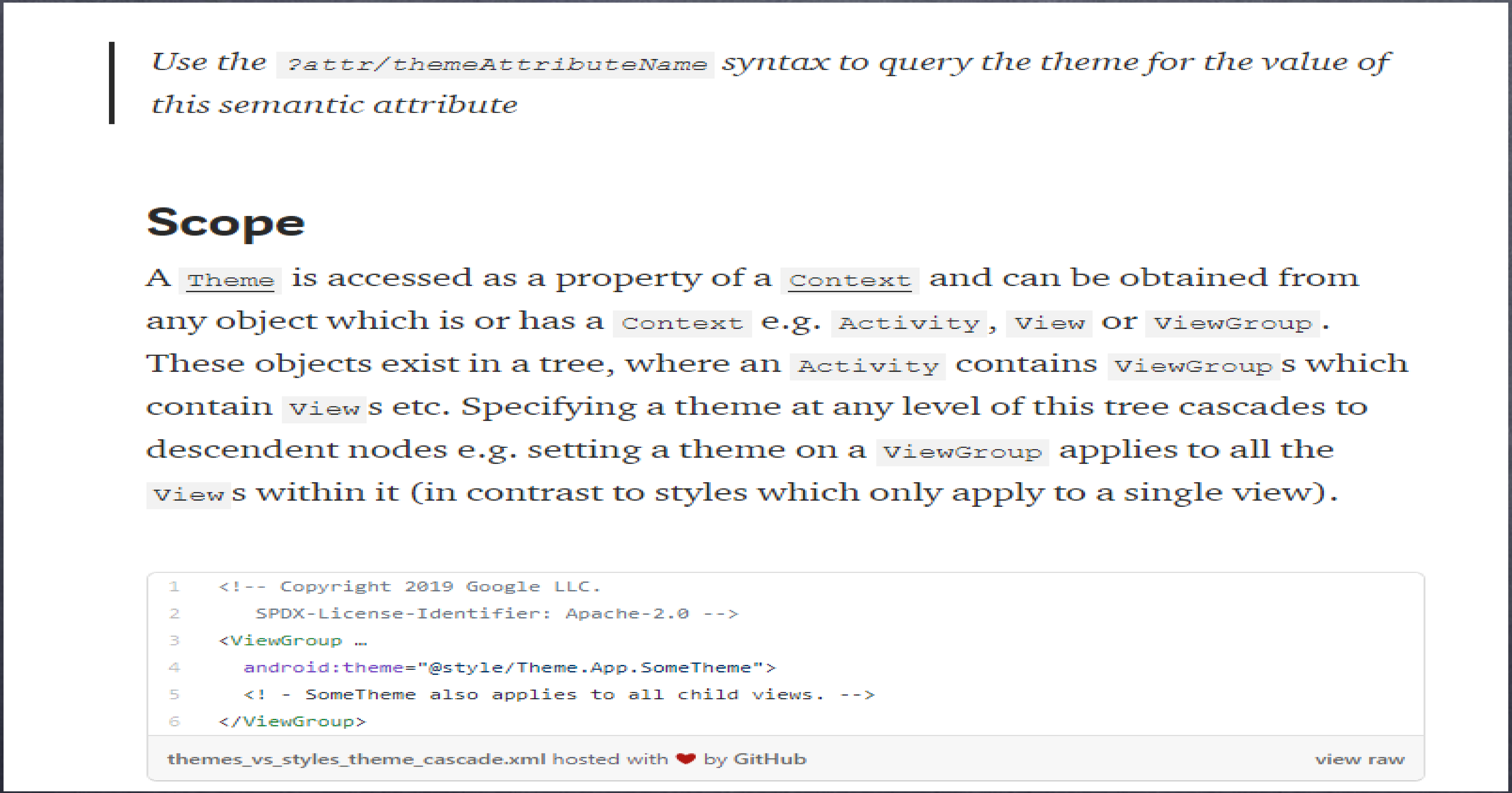
You can specify a theme on components which have (or are) a `Context` e.g. `Activity` or `Views` / `ViewGroup`s:

```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.

Android styling: themes vs styles



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

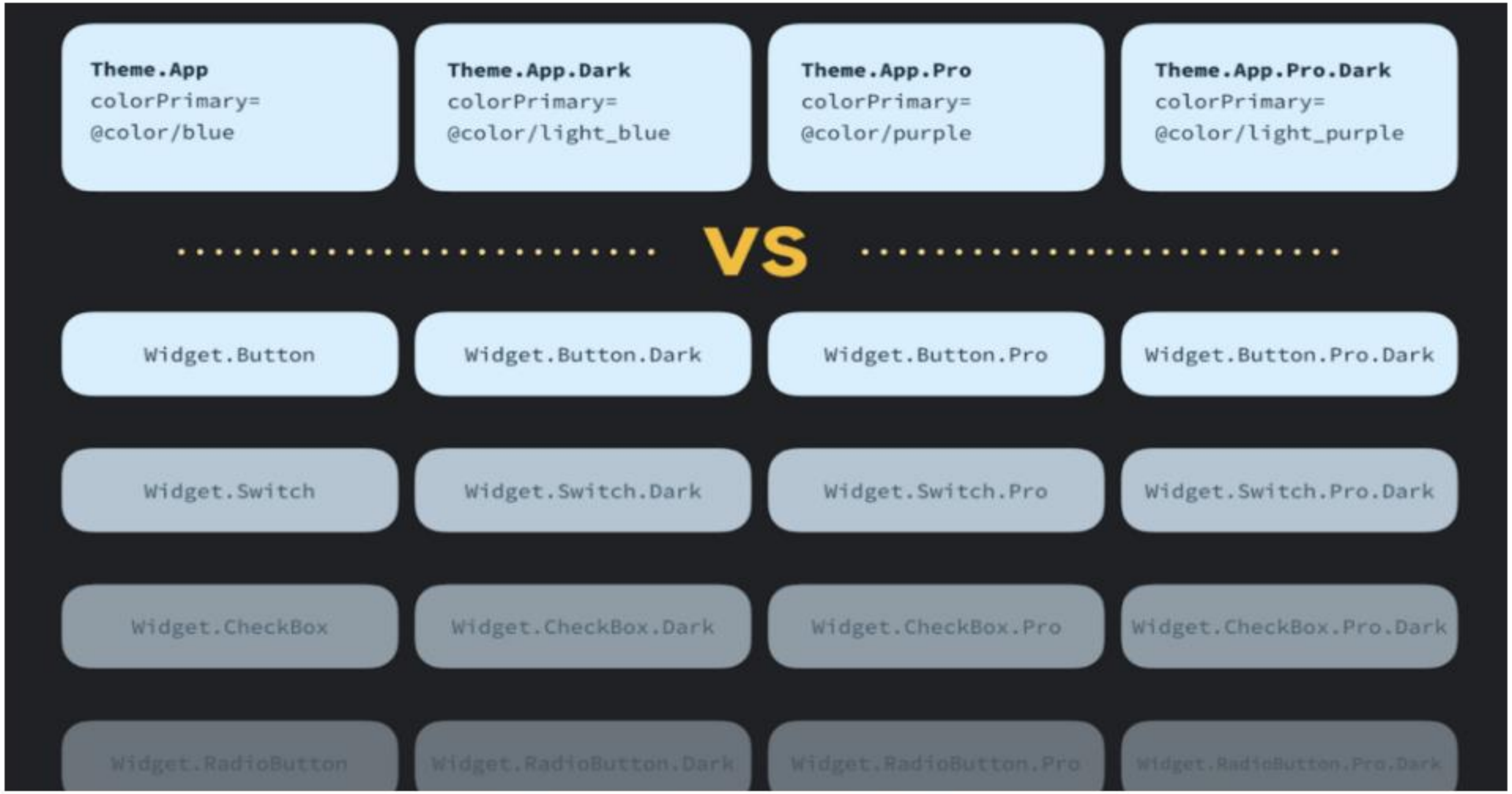
Scope

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>
```

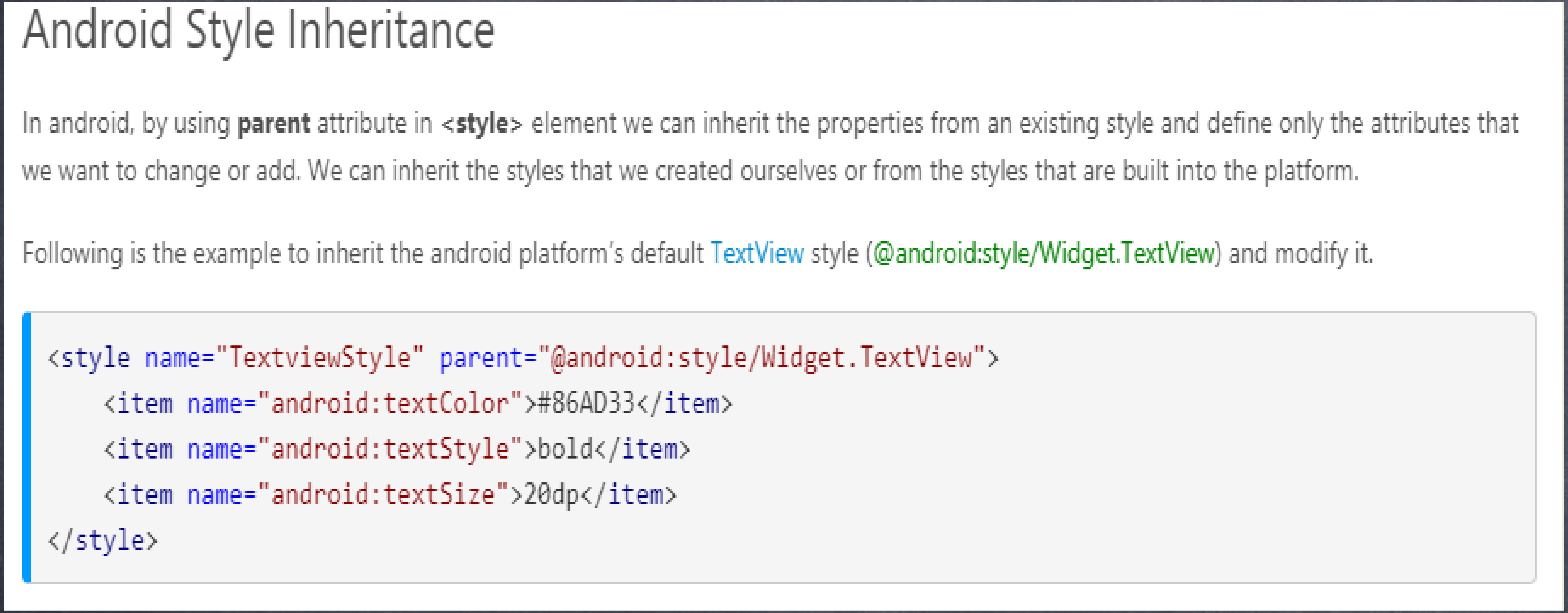
themes_vs_styles_theme_cascade.xml hosted with ❤ by GitHub view raw

Android styling: themes vs styles



Exploding permutations of widgets/styles without theming

Android style: inheritance



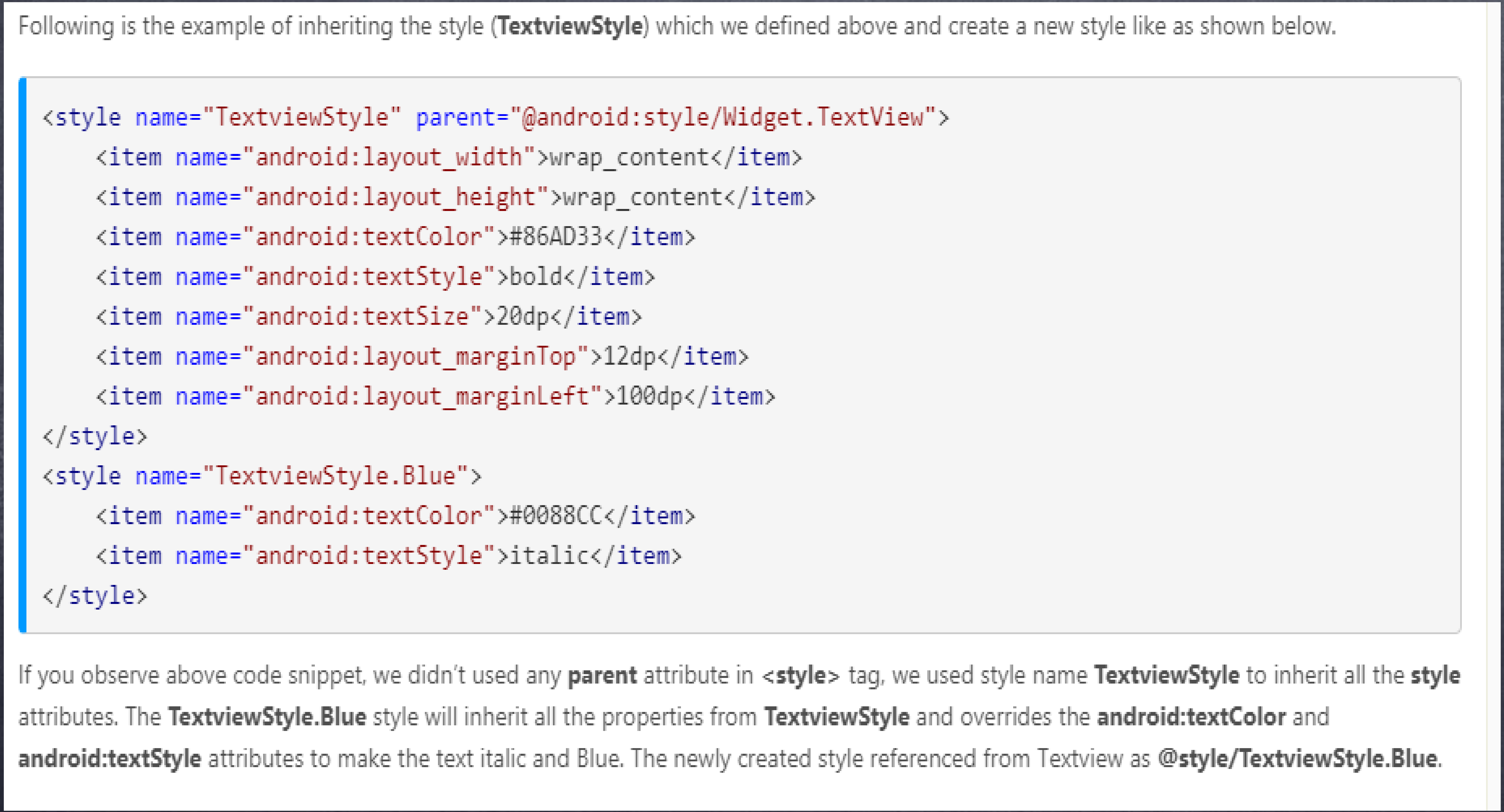
Android Style Inheritance

In android, by using **parent** attribute in **<style>** element we can inherit the properties from an existing style and define only the attributes that we want to change or add. We can inherit the styles that we created ourselves or from the styles that are built into the platform.

Following is the example to inherit the android platform's default TextView style (@android:style/Widget.TextView) and modify it.

```xml
<style name="TextviewStyle" parent="@android:style/Widget.TextView">
    <item name="android:textColor">#86AD33</item>
    <item name="android:textStyle">bold</item>
    <item name="android:textSize">20dp</item>
</style>
```

Android style: inheritance

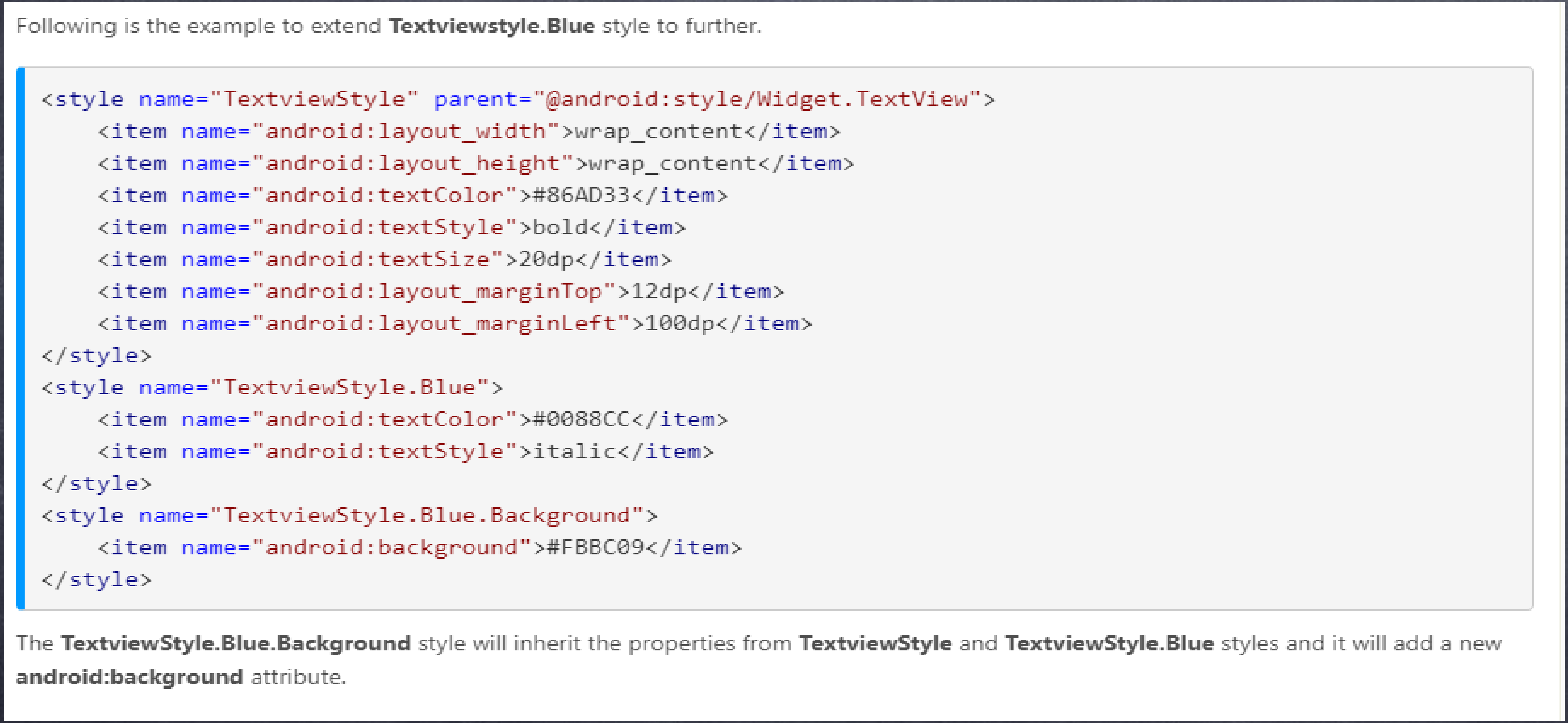


Following is the example of inheriting the style (**TextviewStyle**) which we defined above and create a new style like as shown below.

```xml
<style name="TextviewStyle" parent="@android:style/Widget.TextView">
    <item name="android:layout_width">wrap_content</item>
    <item name="android:layout_height">wrap_content</item>
    <item name="android:textColor">#86AD33</item>
    <item name="android:textStyle">bold</item>
    <item name="android:textSize">20dp</item>
    <item name="android:layout_marginTop">12dp</item>
    <item name="android:layout_marginLeft">100dp</item>
</style>
<style name="TextviewStyle.Blue">
    <item name="android:textColor">#0088CC</item>
    <item name="android:textStyle">italic</item>
</style>
```

If you observe above code snippet, we didn't used any **parent** attribute in **<style>** tag, we used style name **TextviewStyle** to inherit all the **style** attributes. The **TextviewStyle.Blue** style will inherit all the properties from **TextviewStyle** and overrides the **android:textColor** and **android:textStyle** attributes to make the text italic and Blue. The newly created style referenced from Textview as **@style/TextviewStyle.Blue**.

Android style: inheritance

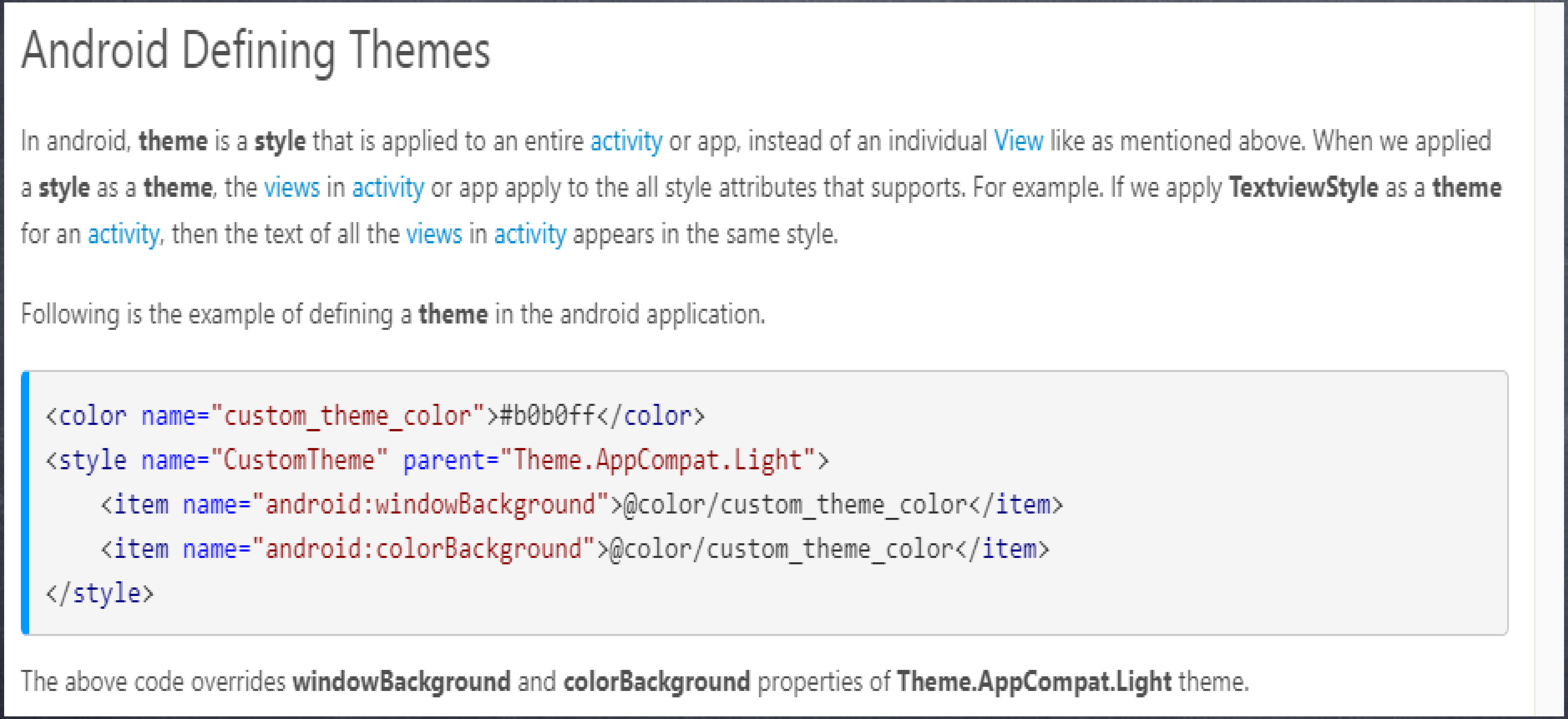


Following is the example to extend **Textviewstyle.Blue** style to further.

```xml
<style name="TextviewStyle"  parent="@android:style/Widget.TextView">
    <item name="android:layout_width">wrap_content</item>
    <item name="android:layout_height">wrap_content</item>
    <item name="android:textColor">#86AD33</item>
    <item name="android:textStyle">bold</item>
    <item name="android:textSize">20dp</item>
    <item name="android:layout_marginTop">12dp</item>
    <item name="android:layout_marginLeft">100dp</item>
</style>
<style name="TextviewStyle.Blue">
    <item name="android:textColor">#0088CC</item>
    <item name="android:textStyle">italic</item>
</style>
<style name="TextviewStyle.Blue.Background">
    <item name="android:background">#FBBC09</item>
</style>
```

The **TextviewStyle.Blue.Background** style will inherit the properties from **TextviewStyle** and **TextviewStyle.Blue** styles and it will add a new **android:background** attribute.

Android style: inheritance



Android Defining Themes

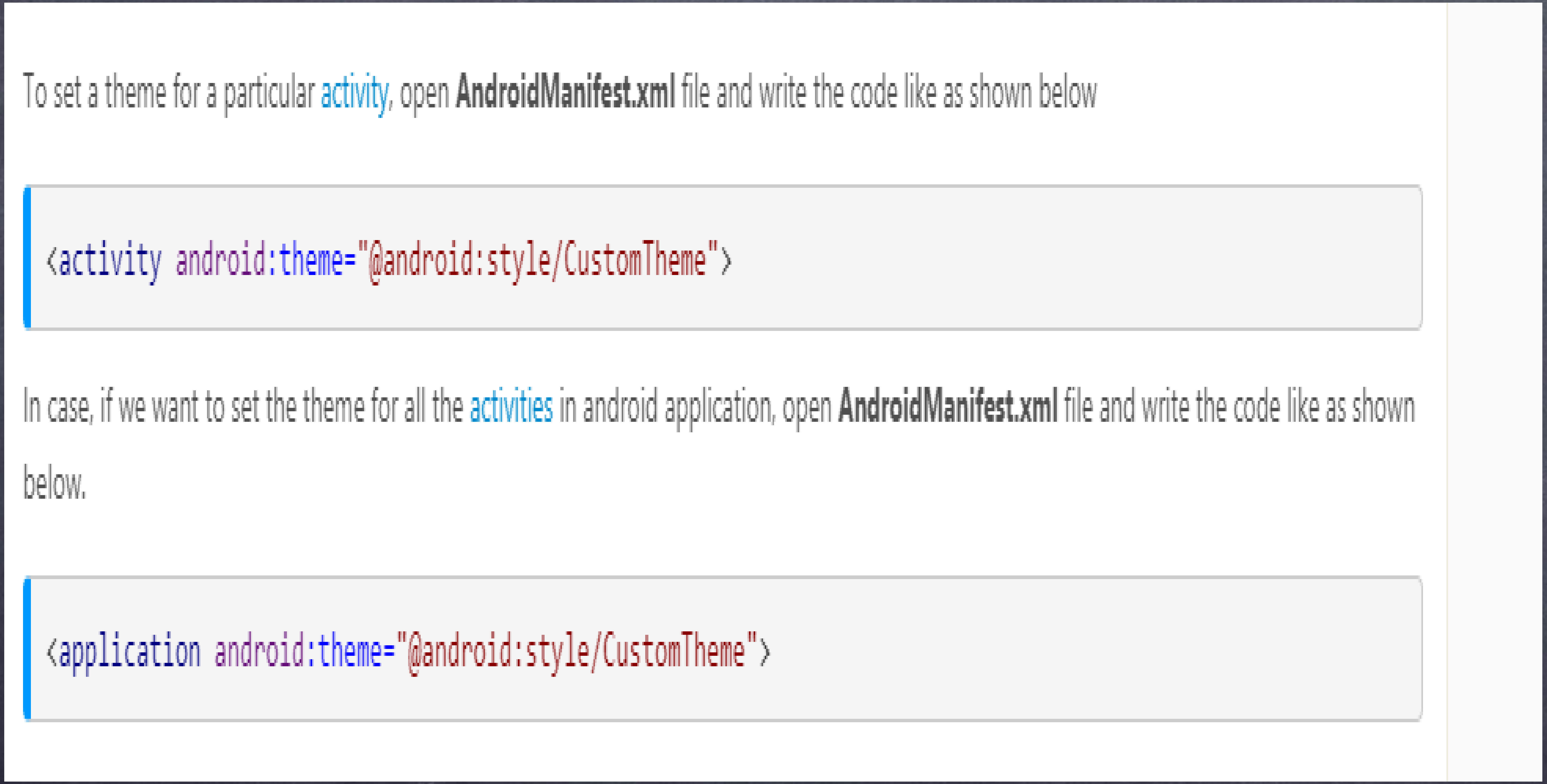
In android, **theme** is a **style** that is applied to an entire activity or app, instead of an individual View like as mentioned above. When we applied a **style** as a **theme**, the views in activity or app apply to the all style attributes that supports. For example. If we apply **TextviewStyle** as a **theme** for an activity, then the text of all the views in activity appears in the same style.

Following is the example of defining a **theme** in the android application.

```
<color name="custom_theme_color">#b0b0ff</color>
<style name="CustomTheme" parent="Theme.AppCompat.Light">
    <item name="android:windowBackground">@color/custom_theme_color</item>
    <item name="android:colorBackground">@color/custom_theme_color</item>
</style>
```

The above code overrides **windowBackground** and **colorBackground** properties of **Theme.AppCompat.Light** theme.

Android style: inheritance



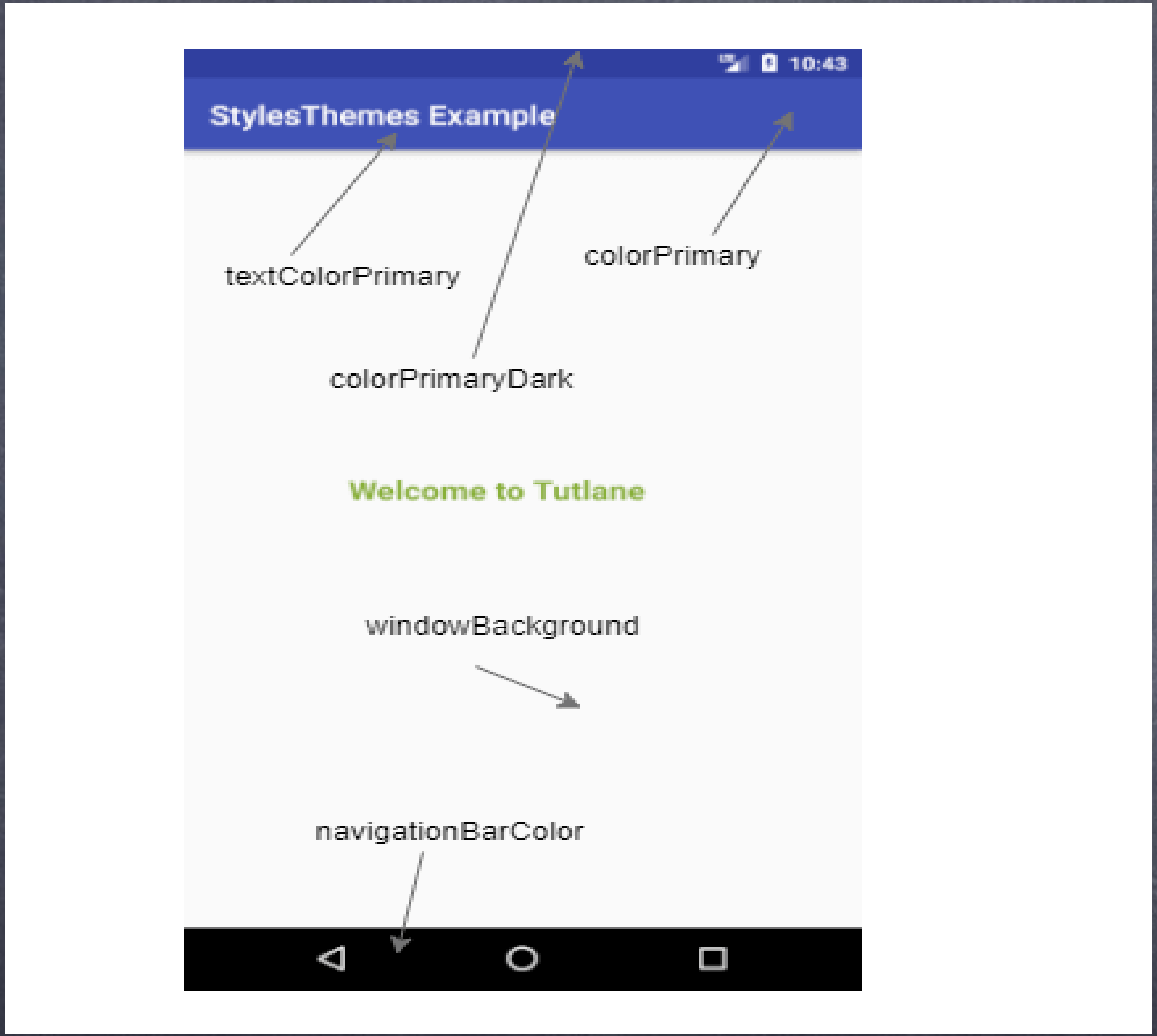
To set a theme for a particular activity, open **AndroidManifest.xml** file and write the code like as shown below

```
<activity android:theme="@android:style/CustomTheme">
```

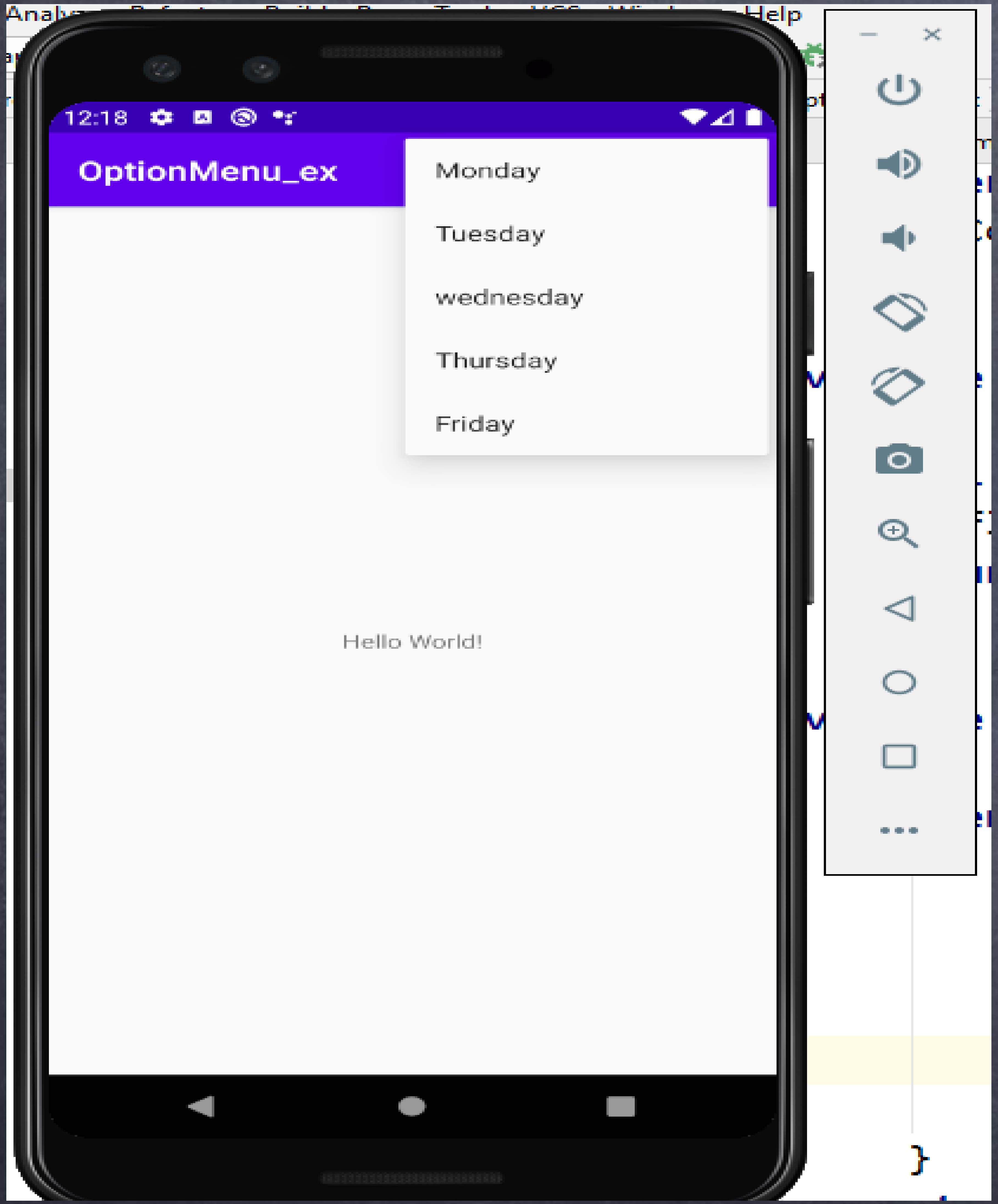
In case, if we want to set the theme for all the activities in android application, open **AndroidManifest.xml** file and write the code like as shown below.

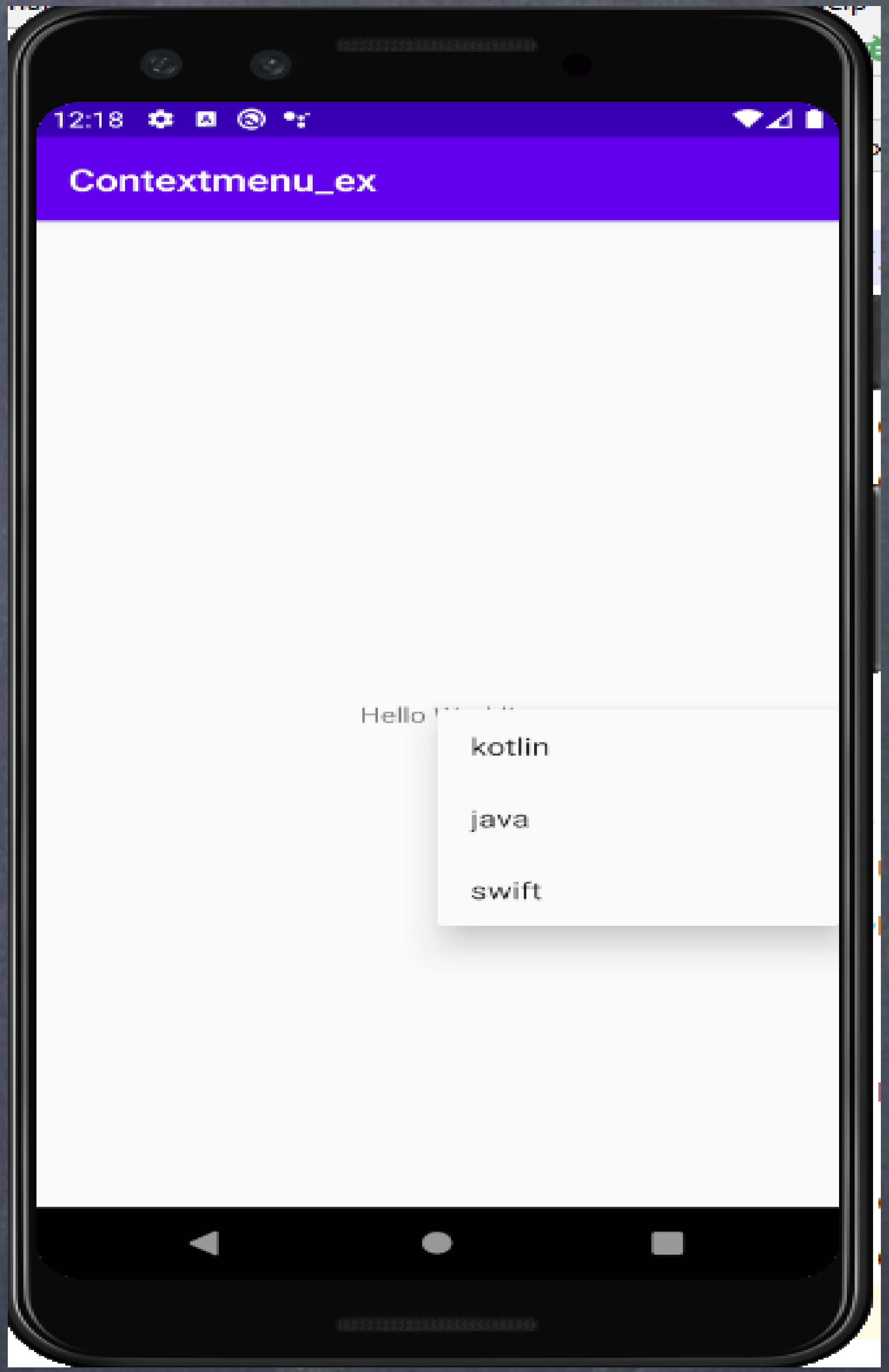
```
<application android:theme="@android:style/CustomTheme">
```

Material 2.0 Theme Android style: inheritance

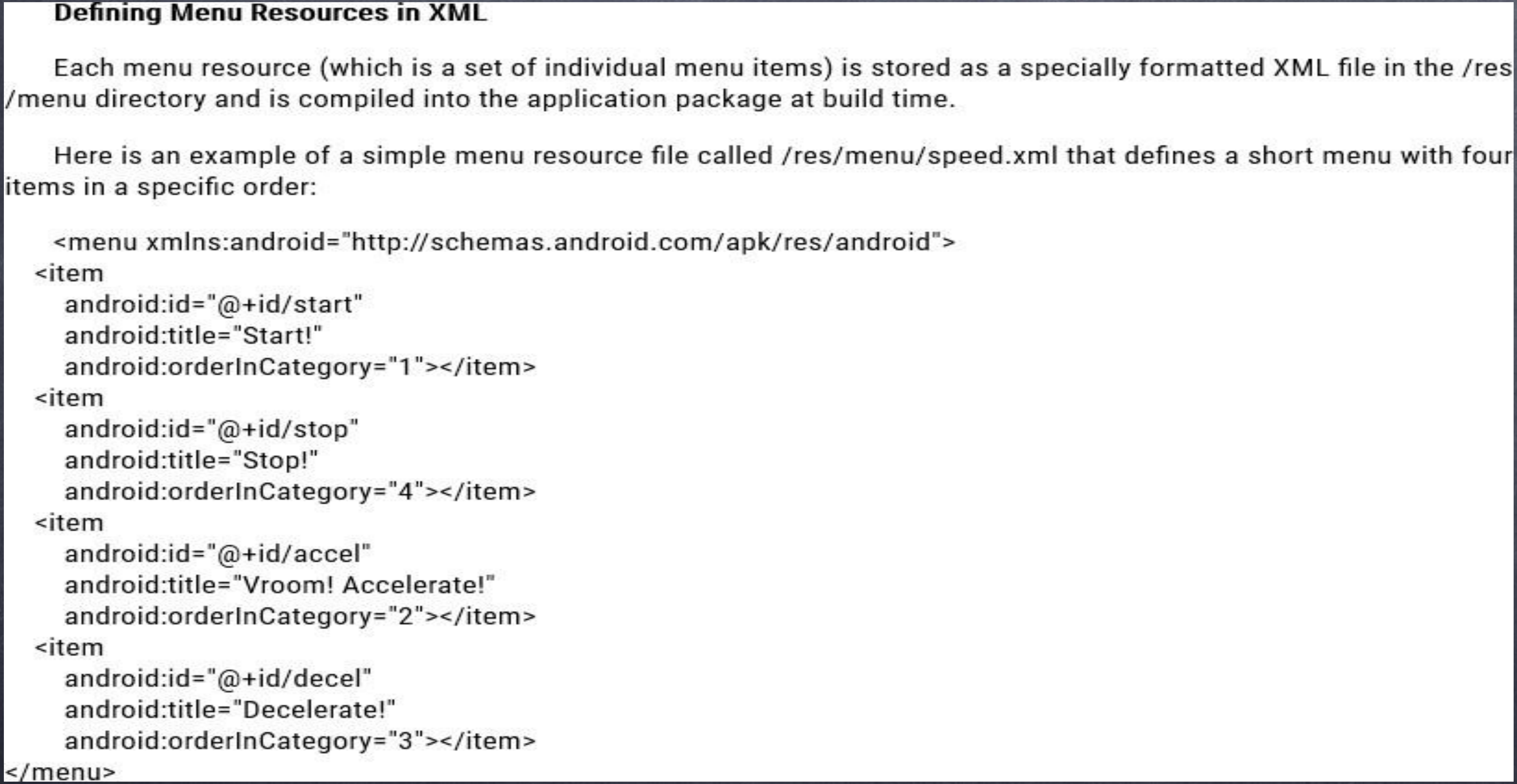


Menu:


12:18
OptionMenu_ex
Monday
Tuesday
wednesday
Thursday
Friday
Hello World!


12:18
Contextmenu_ex
Hello
kotlin
java
swift

Menu:



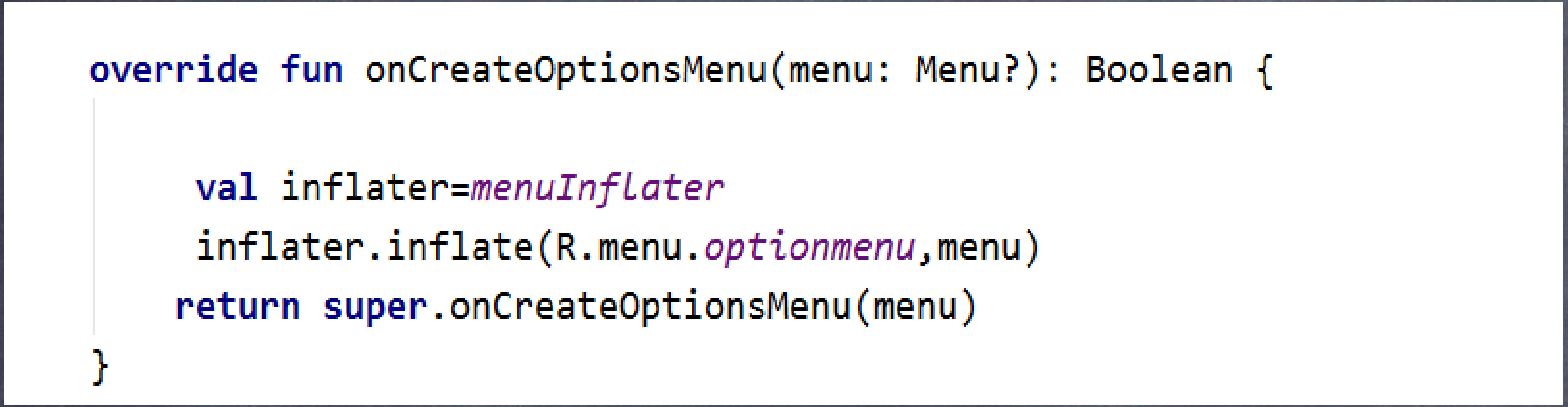
Defining Menu Resources in XML

Each menu resource (which is a set of individual menu items) is stored as a specially formatted XML file in the /res /menu directory and is compiled into the application package at build time.

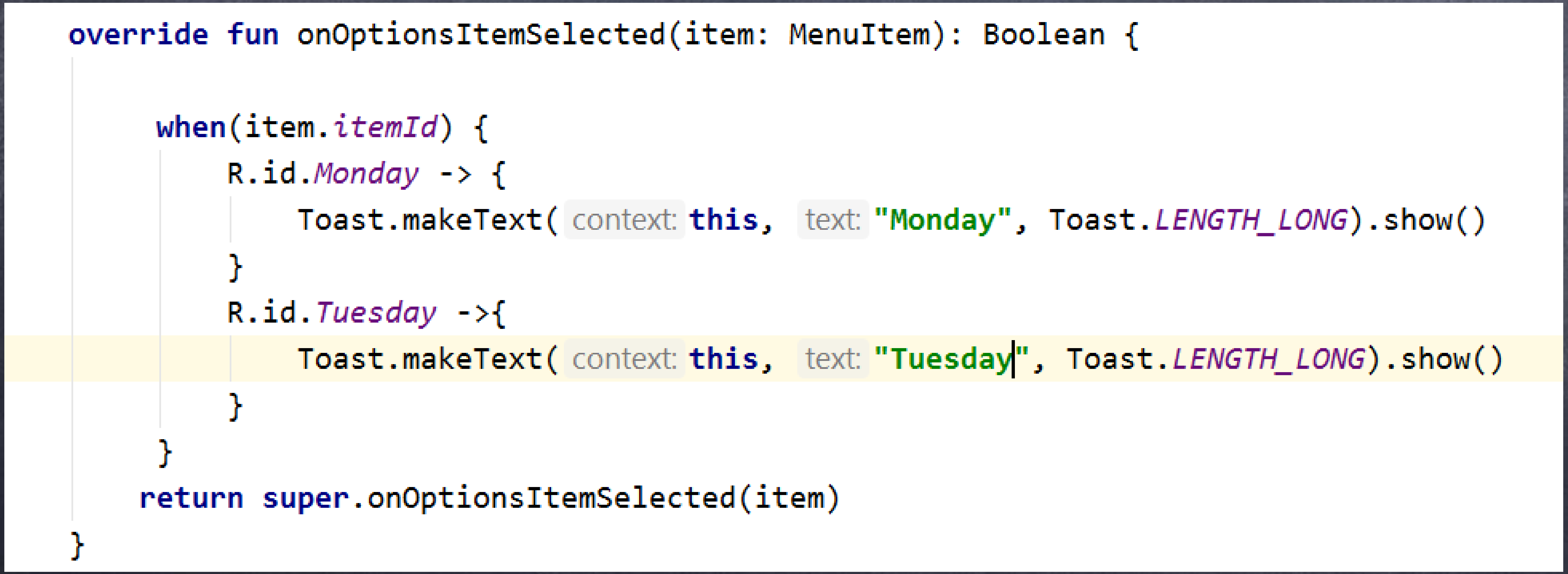
Here is an example of a simple menu resource file called /res/menu/speed.xml that defines a short menu with four items in a specific order:

```
  <menu xmlns:android="http://schemas.android.com/apk/res/android">
 <item
   android:id="@+id/start"
   android:title="Start!"
   android:orderInCategory="1"></item>
 <item
   android:id="@+id/stop"
   android:title="Stop!"
   android:orderInCategory="4"></item>
 <item
   android:id="@+id/accel"
   android:title="Vroom! Accelerate!"
   android:orderInCategory="2"></item>
 <item
   android:id="@+id/decel"
   android:title="Decelerate!"
   android:orderInCategory="3"></item>
</menu>
```

Menu:



```
override fun onCreateOptionsMenu(menu: Menu?): Boolean {

     val inflater=menuInflater
     inflater.inflate(R.menu.optionmenu,menu)
    return super.onCreateOptionsMenu(menu)
}
```



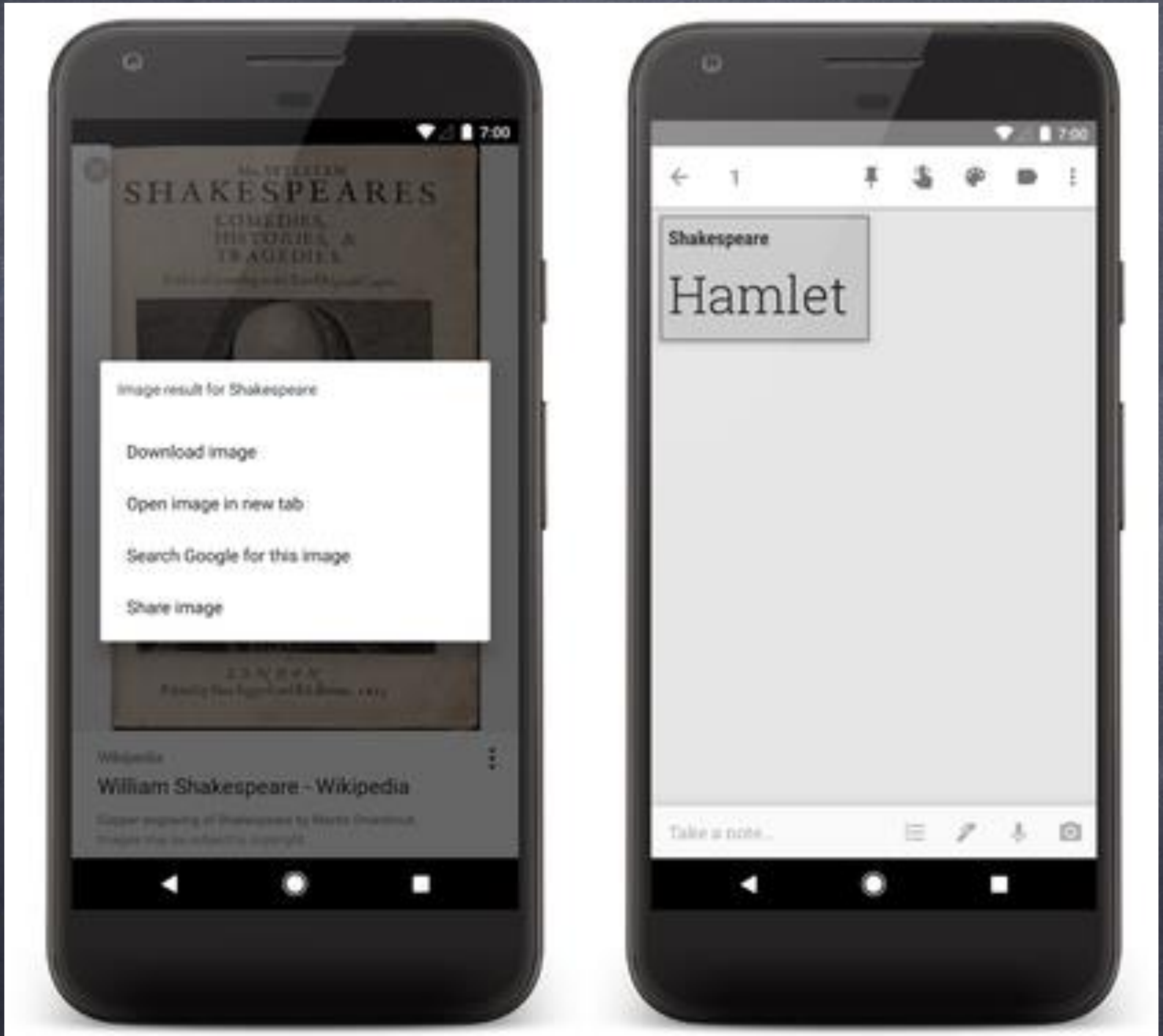
```
override fun onOptionsItemSelected(item: MenuItem): Boolean {

    when(item.itemId) {
        R.id.Monday -> {
            Toast.makeText( context: this,  text: "Monday", Toast.LENGTH_LONG).show()
        }
        R.id.Tuesday ->{
            Toast.makeText( context: this,  text: "Tuesday", Toast.LENGTH_LONG).show()
        }
    }
   return super.onOptionsItemSelected(item)
}
```

Contextual Menus

Context Menu

- A contextual menu offers actions that affect a specific item or context frame in the UI.
- You can provide a context menu for any view, but they are most often used for items in a ListView, GridView, or other view collections in which the user can perform direct actions on each item.
- There are two ways to provide contextual actions:
- floating context menu.
- contextual action mode.

Creating a floating context menu


SHAKESPEARES
COMEDIES, HISTORIES, & TRAGEDIES
Image result for Shakespeare
Download image
Open image in new tab
Search Google for this image
Share image
William Shakespeare - Wikipedia
Shakespeare
Hamlet
Take a note...

Creating a floating context menu

- To provide a floating context menu:
- **Step 1:**
- Register the View to which the context menu should be associated by calling registerForContextMenu() and pass it the View.
- If your activity uses a ListView or GridView and you want each item to provide the same context menu, register all items for a context menu by passing the ListView or GridView to registerForContextMenu().
- **Step 2:**
- Implement the onCreateContextMenu() method in your Activity or Fragment.
- When the registered view receives a long-click event, the system calls your onCreateContextMenu() method. This is where you define the menu items, usually by inflating a menu resource. For example:

Creating a floating context menu

```
override fun onCreateContextMenu(menu: ContextMenu, v: View,
                                 menuInfo: ContextMenu.ContextMenuInfo) {
    super.onCreateContextMenu(menu, v, menuInfo)
    val inflater: MenuInflater = menuInflater
    inflater.inflate(R.menu.context_menu, menu)
}
```

Creating a floating context menu

- **Step 3:**
- Implement onContextItemSelected().
- When the user selects a menu item, the system calls this method so you can perform the appropriate action. For example:

```
override fun onContextItemSelected(item: MenuItem): Boolean {
    val info = item.menuInfo as AdapterView.AdapterContextMenuInfo
    return when (item.itemId) {
        R.id.edit -> {
            editNote(info.id)
            true
        }
        R.id.delete -> {
            deleteNote(info.id)
            true
        }
        else -> super.onContextItemSelected(item)
    }
}
```

alternative resources-Site references

https://developer.android.com/develop/ui/views/components/menus

https://appicon.co/#app-icon

https://www.canva.com/

https://colorhunt.co/

https://material.io/resources/devices/