https://google-developer-training.github.io/android-developer-fundamentals-course-concepts-v2/

Lesson 5: Delightful user experience

5.1: Drawables, styles, and themes

Contents:

- Introduction
- Drawables
- Images
- Styles
- Themes
- Related practical
- Learn more

In this chapter you learn how to use *drawables*, which are compiled images that you can use in your app. Android provides classes and resources to help you include rich images in your application with a minimal impact to your app's performance.

You also learn how to use styles and themes to provide a consistent appearance to all the elements in your app while reducing the amount of code.

Drawables

A `Drawable` is a graphic that can be drawn to the screen. You retrieve a `Drawable` using APIs such as `getDrawable(int)`, and you apply a `Drawable` to an XML resource using attributes such as `android:drawable` and `android:icon`.
Android includes several types of drawables, most of which are covered in this chapter.

Covered in this chapter:

- Image files
- Nine-patch files
- Layer lists
- Shape drawables
- State lists
- Level lists
- Transition drawables
- Vector drawables

Not covered in this chapter:

- Scale drawables
- Inset drawables
- Clip drawables

Using drawables

To display a `Drawable`, use the `ImageView` class to create a `View`. In the `<ImageView>` element in your XML file, define how the `Drawable` is displayed and where the `Drawable` file is located. For example, this `ImageView` displays an image called "birthdaycake.png":

```
<ImageView
      android:id="@+id/tiles"
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:src="@drawable/birthdaycake" />
```

About the `<ImageView>` attributes:

- The `android:id` attribute sets a shortcut name that you use to call the image later.
- The `android:layout_width` and `android:layout_height` attributes specify the size of the `View`. In the above example the height and width are set to `wrap_content`, which means the `View` is only big enough to enclose the image within it, plus padding.
- The `android:src` attribute gives the location where this image is stored. If you have versions of the image that are appropriate for different screen resolutions, store them in folders named `res/drawable-`*density*`/`. For example, store a version of birthdaycake.png appropriate for hdpi screens in `res/drawable-hdpi/birthdaycake.png`. For more information, see Screen Compatibility Overview.
- `<ImageView>` also has attributes that you can use to crop your image if it is too large or has a different aspect ratio than the layout or the View. For complete details, see `ImageView`.

To represent a drawable in your app, use the `Drawable` class or one of its subclasses. For example, this code retrieves the birthdaycake.png image as a `Drawable`:

```
Resources res = getResources();
Drawable drawable = res.getDrawable(R.drawable.birthdaycake);
```

Image files

An *image file* is a generic bitmap file. Android supports image files in several formats: WebP (preferred), PNG (preferred), and JPG (acceptable). GIF and BMP formats are supported, but discouraged.

The WebP format is fully supported from Android 4.2. WebP compresses better than other formats for lossless and lossy compression, potentially resulting in images more than 25% smaller than JPEG formats. You can convert existing PNG and JPEG images into WebP format before upload. For more about WebP, see the WebP documentation.

Store image files in the `res/drawable` folder. Use them with the `android:src` attribute for an `ImageView` and its descendants, or to create a `BitmapDrawable` class in Java code.
Be aware that images look different on screens with different pixel densities and aspect ratios. For information on supporting different screen sizes, see Speeding up your app (below) and Support Different Screen Sizes.

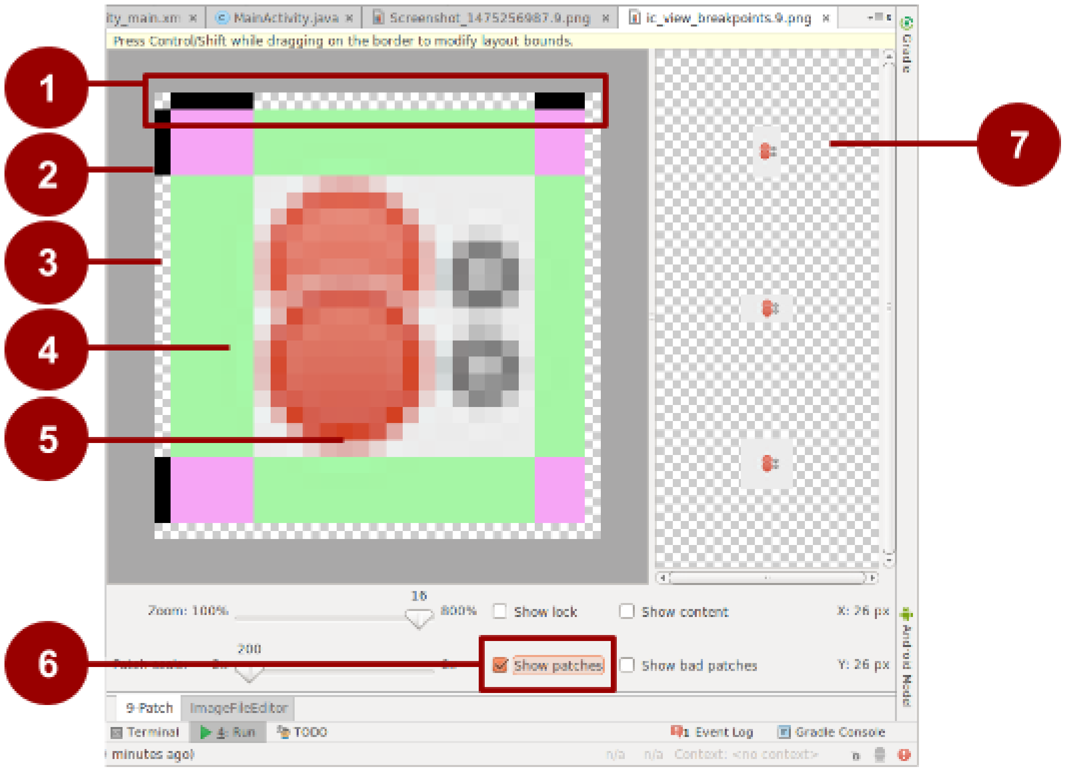
Note: Always use appropriately sized images, because images can use up a lot of disk space and affect your app's performance.

Nine-patch files

A *9-patch* is a PNG image in which you define stretchable regions. Use a 9-patch as the background image for a `View` to make sure the `View` looks correct for different screen sizes and orientations.
For example, in a `View` that has `layout_width` set to `"wrap_content"`, the `View` stays big enough to enclose its content (plus padding). If you use a normal PNG image as the background image for the `View`, the image might be too small for the `View` on some devices, because the `View` stretches to accommodate the content inside it. If you use a 9-patch image instead, the 9-patch stretches as the `View` stretches.
The Android standard `Button` is an example of a `View` that uses a 9-patch as its background image. The 9-patch stretches to accommodate the text or image inside the `Button`.
Save 9-patch files with a `.9.png` extension and store them in the `res/drawable` folder. Use them with the `android:src` attribute for an `ImageView` and its descendants, or to create a `NinePatchDrawable` class in Java code.
To create a 9-patch, use the Draw 9-Patch tool in Android Studio. The tool lets you start with a regular PNG and define a 1-pixel border around the image in places where it's okay for the Android system to stretch the image if needed. To use the tool:

1. Put a PNG file into the `res/drawable` folder. (To do this, copy the image file into the **app/src/main/res/drawable** folder of your project.)
2. In Android Studio, **right-click** (or **Control-click**) the file and choose **Create 9-Patch file**. Android Studio saves the file with a `.9.png` extension.
3. In Android Studio, double-click the **.9.png** file to open the editor.

4. Specify which regions of the image are okay to stretch.



In this figure:

1. Border to indicate which regions are okay to stretch for width (horizontally). For example, in a `View` that is wider than the image, the green stripes on the left- and right-hand sides of this 9-patch can be stretched to fill the `View`. Places that can stretch are marked with black. Click to turn pixels black.
2. Border to indicate regions that are okay to stretch for height (vertically). For example, in a `View` that is taller than the image, the green stripes on the top and bottom of this 9-patch can be stretched to fill the `View`.
3. Turn off pixels by pressing **Shift-click** (**Control-click** in macOS).
4. Stretchable area.
5. Not stretchable.
6. Check **Show patches** to preview the stretchable patches in the drawing area.
7. Previews of stretched image.

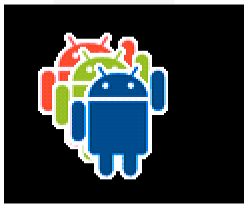
Tip: Make sure that stretchable regions are at least 2x2 pixels in size. Otherwise, they may disappear when the image is scaled down.

For a more detailed discussion about how to create a 9-patch file with stretchable regions, see the 9-patch drawables.

https://google-developer-training.github.io/android-developer-fundamentals-course-concepts-v2/

Layer list drawables

In Android you can build up an image by layering other images together, just as you can in Gimp and other image-manipulation programs. Each layer is represented by an individual `Drawable`. The drawables that make up a single image are organized and managed in a `<layer-list>` element in XML. Within the `<layer-list>`, each `Drawable` is represented by an `<item>` element.
Layers are drawn on top of each other in the order defined in the XML file, which means that the last `Drawable` in the list is drawn on top. For example, this layer list `Drawable` is made up of three drawables superimposed on each other:



In the following XML, which defines this layer list, the `android_blue` image is defined last, so it's drawn last and shown on top:

```
<layer-list xmlns:android="http://schemas.android.com/apk/res/android">
    <item>
      <bitmap android:src="@drawable/android_red"
        android:gravity="center" />
    </item>
    <item android:top="10dp" android:left="10dp">
      <bitmap android:src="@drawable/android_green"
        android:gravity="center" />
    </item>
    <item android:top="20dp" android:left="20dp">
      <bitmap android:src="@drawable/android_blue"
        android:gravity="center" />
    </item>
</layer-list>
```

A `LayerDrawable` is a `Drawable` object that manages an array of other drawables. For more information about how to use a layer list `Drawable`, see the Layer list in Drawable Resources.

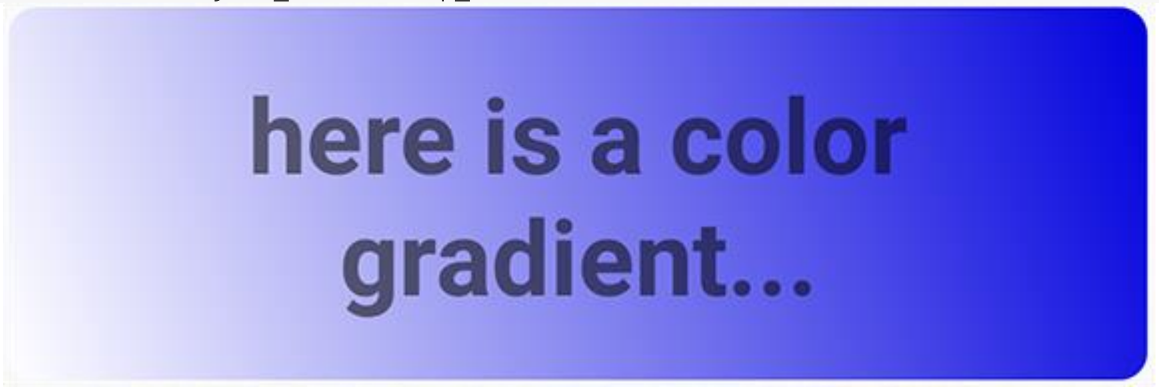
Shape drawables

A shape `Drawable` is a rectangle, oval, line, or ring that you define in XML. You specify the size and style of the shape using XML attributes.
For example, this XML file creates a rectangle with rounded corners and a color gradient. The rectangle's fill color shifts from white (`#000000`) in the lower left corner to blue (`#0000dd`) in the upper right corner. The `angle` attribute determines how the gradient is tilted:

```
<shape xmlns:android="http://schemas.android.com/apk/res/android"
    android:shape="rectangle">
    <corners android:radius="8dp" />
    <gradient
        android:startColor="#000000"
        android:endColor="#0000dd"
        android:angle="45"/>
    <padding android:left="7dp"
        android:top="7dp"
        android:right="7dp"
        android:bottom="7dp" />
</shape>
```

Assuming that the shape `Drawable` XML file is saved at `res/drawable/gradient_box.xml`, the following layout XML applies the shape `Drawable` as the background to a `View`:

```
<TextView
    android:background="@drawable/gradient_box"
    android:layout_height="wrap_content"
    android:layout_width="wrap_content" />
```



The following code shows how to programmatically get the shape `Drawable` and use it as the background for a `View`, as an alternative to defining the background attribute in XML:

```
Resources res = getResources();
Drawable shape = res. getDrawable(R.drawable.gradient_box);

TextView tv = (TextView)findViewByID(R.id.textview);
tv.setBackground(shape);
```

You can set other attributes for a shape `Drawable`. The complete syntax is as follows:

```
<shape
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:shape=["rectangle" | "oval" | "line" | "ring"] >
    <!-- If it's a line, the stroke element is required. -->
    <corners
        android:radius="integer"
        android:topLeftRadius="integer"
        android:topRightRadius="integer"
        android:bottomLeftRadius="integer"
        android:bottomRightRadius="integer" />
    <gradient
        android:angle="integer"
        <!-- The angle must be 0 or a multiple of 45 -->
        android:centerX="float"
        android:centerY="float"
        android:centerColor="integer"
        android:endColor="color"
        android:gradientRadius="integer"
        android:startColor="color"
        android:type=["linear" | "radial" | "sweep"]
        android:useLevel=["true" | "false"] />
    <padding
        android:left="integer"
        android:top="integer"
        android:right="integer"
        android:bottom="integer" />
    <size
        android:width="integer"
        android:height="integer" />
    <solid
        android:color="color" />
    <stroke
        android:width="integer"
        android:color="color"
        android:dashWidth="integer"
        android:dashGap="integer" />
</shape>
```

For details about these attributes, see Shape drawable in Drawable Resources.

State list drawables

A `StateListDrawable` is a `Drawable` object that uses a different image to represent the same object, depending on what state the object is in. For example, a `Button` can exist in one of several states (pressed, focused on, hovered over, or none of these). Using a state list `Drawable`, you can provide a different background image for each state.

You describe the state list in an XML file. Each graphic is represented by an `<item>` element inside a single `<selector>` element. Each `<item>` uses a `state_` attribute to indicate the situation in which the graphic is used.

During each state change, Android traverses the state list from top to bottom. The first item that matches the current state is used, which means that the selection is not based on the best match, but is simply the first item that meets the minimum criteria of the state.

The state list in the following example defines which image is shown for a `Button` when the `Button` is in different states. When the `Button` is pressed—that is, when `state_pressed="true"`—the app shows an image named `button_pressed`. When the `Button` is in focus (`state_focused="true"`), or when the `Button` is being hovered over (`state_hovered="true"`), the app shows a different `Drawable` for the `Button`.

```
<selector xmlns:android="http://schemas.android.com/apk/res/android">
    <item android:state_pressed="true"
          android:drawable="@drawable/button_pressed" /> <!-- pressed -->
    <item android:state_focused="true"
          android:drawable="@drawable/button_focused" /> <!-- focused -->
    <item android:state_hovered="true"
          android:drawable="@drawable/button_focused" /> <!-- hovered -->
    <item android:drawable="@drawable/button_normal" /> <!-- default -->
</selector>
```

Other available states include `android:state_selected`, `android:state_checkable`, `android:state_checked`, and others. For details about all the options, see State list in Drawable Resources.

Level list drawables

A *level list drawable* defines alternate drawables, each assigned a maximum numerical value. To select which drawable to use, call the `setLevel()` method, passing in an integer that is matched against the maximum level integer defined in XML. The resource with the lowest maximum level greater than or equal to the integer passed into `setLevel()` is selected.
For example, the following XML defines a level list that includes two alternate drawables, `status_off` and `status_on`:

```
<level-list xmlns:android="http://schemas.android.com/apk/res/android" >
    <item
        android:drawable="@drawable/status_off"
        android:maxLevel="0" />
    <item
        android:drawable="@drawable/status_on"
        android:maxLevel="1" />
</level-list>
```

To select the `status_off` `Drawable`, call `setLevel(0)`. To select the `status_on` `Drawable`, call `setLevel(1)`.

An example use of a `LevelListDrawable` is a battery level indicator icon that uses different images to indicate different current battery levels. #WIDTH: 24.00[IMAGEINFO]: ic_battery_charging_icon.png, Battery icon

Transition drawables

A `TransitionDrawable` is a `Drawable` that crossfades between two other drawables. To define a `TransitionDrawable` in XML, use the `<transition>` element. Each `Drawable` is represented by an `<item>` element inside the `<transition>` element. No more than two `<item>` elements are supported.
For example, this drawable crossfades between an "on" state and an "off" state drawable:

```
<transition xmlns:android="http://schemas.android.com/apk/res/android">
    <item android:drawable="@drawable/on" />
    <item android:drawable="@drawable/off" />
</transition>
```

To transition forward, meaning to shift from the first `Drawable` to the second, call `startTransition()`. To transition in the other direction, call `reverseTransition()`. Each of these methods takes an argument of type `int`, representing the number of milliseconds for the transition.

Vector drawables

In Android 5.0 (API Level 21) and above, you can define *vector drawables*, which are images that are defined by a path. A vector `Drawable` scales without losing definition. Most vector drawables use SVG files, which are plain text files or compressed binary files that include two-dimensional coordinates for how the image is drawn on the screen.
Because SVG files are text, they are more space efficient than most other image files. Also, you only need one file for a vector image instead of a file for each screen density, as is the case for bitmap images.

To bring an existing vector image or a Material Design icon into your Android Studio project as a vector `Drawable`:

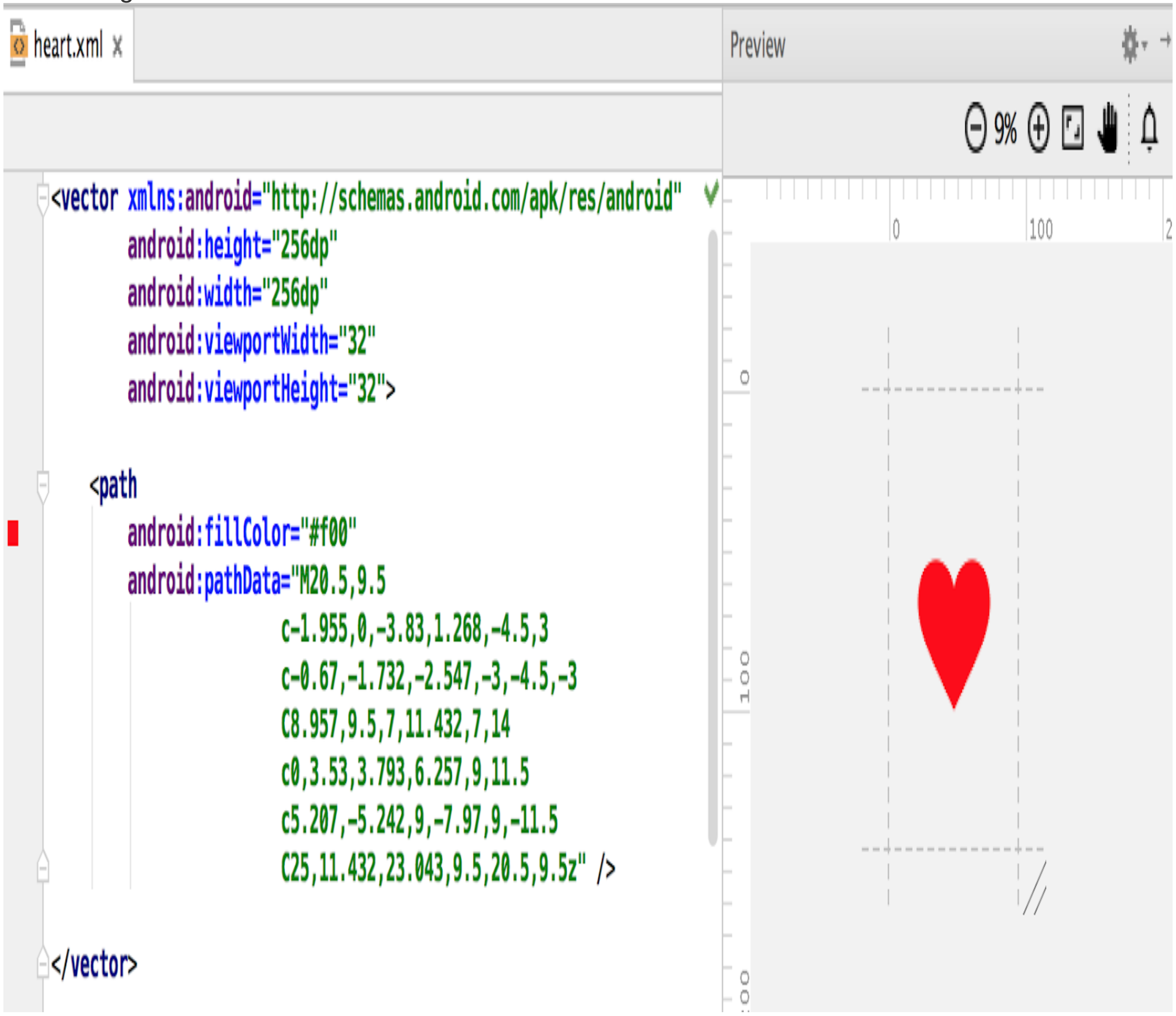
1. **Right-click** (or **Control-click**) on the **drawable** folder in the **Project > Android** pane.
2. Select **New > Vector Asset**. The Vector Asset Studio opens and guides you through the process.

To create a vector image, define the details of the shape inside a `<vector>` XML element. For example, the following code defines the shape of a heart and fills it with a red color (`#f00`):

```
<vector xmlns:android="http://schemas.android.com/apk/res/android"
    <!-- intrinsic size of the drawable -->
    android:height="256dp"
    android:width="256dp"
    <!-- size of the virtual canvas -->
    android:viewportWidth="32"
    android:viewportHeight="32">

  <!-- draw a path -->
  <path android:fillColor="#f00"
      android:pathData="M20.5,9.5
                        c-1.955,0,-3.83,1.268,-4.5,3
                        c-0.67,-1.732,-2.547,-3,-4.5,-3
                        C8.957,9.5,7,11.432,7,14
                        c0,3.53,3.793,6.257,9,11.5
                        c5.207,-5.242,9,-7.97,9,-11.5
                        C25,11.432,23.043,9.5,20.5,9.5z" />
</vector>
```

Android Studio shows a preview of vector drawables. For example, here's the result of creating the XML file described above:



```
<vector xmlns:android="http://schemas.android.com/apk/res/android"
        android:height="256dp"
        android:width="256dp"
        android:viewportWidth="32"
        android:viewportHeight="32">

    <path
        android:fillColor="#f00"
        android:pathData="M20.5,9.5
                        c-1.955,0,-3.83,1.268,-4.5,3
                        c-0.67,-1.732,-2.547,-3,-4.5,-3
                        C8.957,9.5,7,11.432,7,14
                        c0,3.53,3.793,6.257,9,11.5
                        c5.207,-5.242,9,-7.97,9,-11.5
                        C25,11.432,23.043,9.5,20.5,9.5z" />

</vector>
```

If you already have an image in SVG format, there are several ways to get the image's `pathData` information:

- In Android Studio, **right-click** (or **Control-click**) on the **drawable** folder and select **New > Vector Asset** to open the Vector Asset Studio tool. Use the tool to import a local SVG file.
- Use a file-conversion tool such as svg2android.
- Open the image in a text editor, or if you're viewing the image in a browser, view the page source. Look for the `d=` information, which is equivalent to the `pathData` in your XML.

Vector images are represented in Android as `VectorDrawable` objects. For details about the `pathData` syntax, see the SVG Path reference. To learn how to animate the properties of vector drawables, see Animate Drawable Graphics.

Images

Images, from launcher icons to banner images, are used in many ways in Android. Each use case has different requirements for image resolution, scalability and simplicity. In this section you learn about the different ways to generate images and include them in your app.

Creating icons

Every app requires at least a launcher icon, and apps often include icons for action bar actions, notifications, and other use cases.

There are two approaches to creating icons:

- Create a set of image files of the same icon in different resolutions and sizes so that the icon looks the same across devices with different screen densities. You can use Image Asset Studio to do this.
- Use vector drawables, which scale automatically without the image becoming pixelated or blurry. You can use Vector Asset Studio to do this.

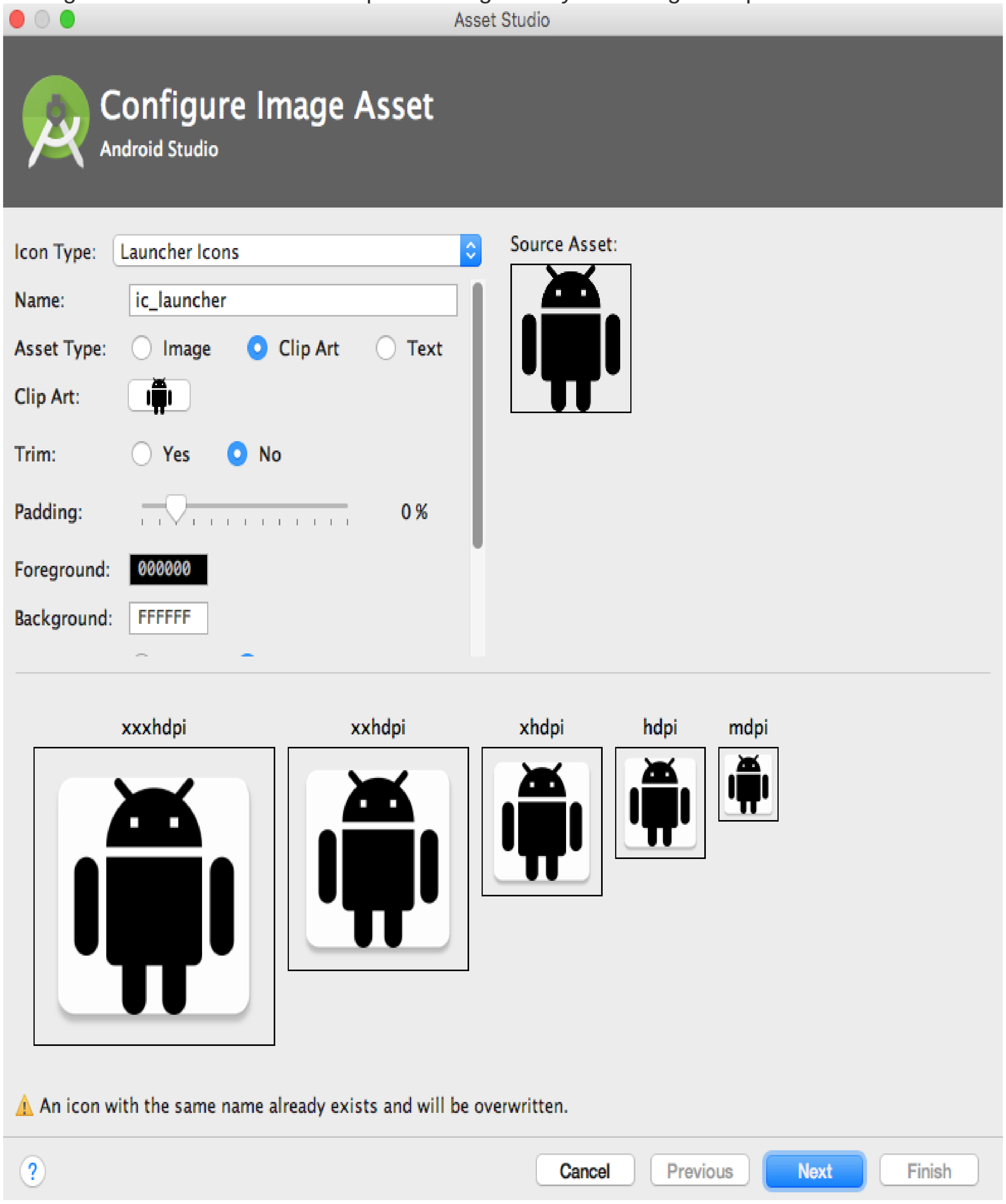
Image Asset Studio

Android Studio includes a tool called Image Asset Studio that helps you generate your own app icons from Material Design icons, custom images, and text strings. It generates a set of icons at the appropriate resolution for each generalized screen density that your app supports. Image Asset Studio places the newly generated icons in density-specific folders under the res/ folder in your project. At runtime, Android uses the appropriate resource based on the screen density of the device your app is running on.

Image Asset Studio helps you generate the following icon types:

- Launcher icons
- Action bar and tab icons
- Notification icons

To use Image Asset Studio, **right-click** (or **Control-click**) on the **res** folder in the **Project > Android** pane of Android Studio, and select **New > Image Asset**. The Configure Asset Studio wizard opens and guides you through the process.



Icon Type: Launcher Icons

Name: ic_launcher

Asset Type: Image [✓ Clip Art] Text

Clip Art:

Trim: Yes [✓ No]

Padding: 0 %

Foreground: 000000

Background: FFFFFF

An icon with the same name already exists and will be overwritten.

For more about Image Asset Studio, see Image Asset Studio.

Vector Asset Studio

Starting with API 21, you can use vector drawables instead of image files for your icons.

Advantages of using vector drawables as icons:

- Vector drawables can reduce your APK file size dramatically, because you don't have to include multiple versions of each icon image. You can use one vector image to scale seamlessly to any resolution.
- Users might be more likely to download an app that has smaller files and a smaller package size.

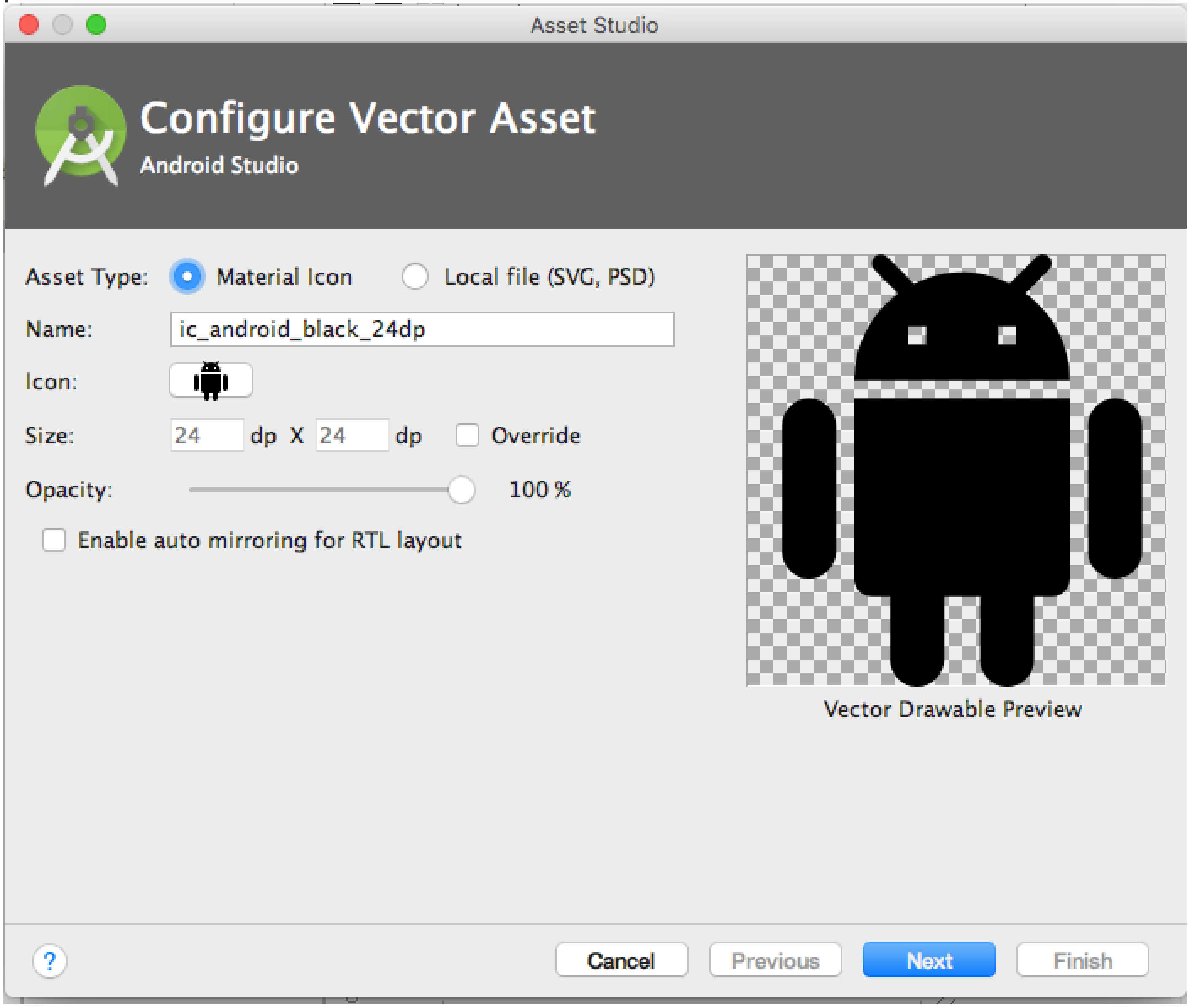
Disadvantages of using vector drawables as icons:

- A vector drawable can include only a limited amount of detail. Vector drawables are mostly used for less detailed icons such as the Material Design icons. Icons with more detail usually need image files.
- Vector drawables are not supported on devices running API level 20 or below.

To use vector drawables on devices running API level 20 or below, you have to decide between two methods of backward-compatibility:

- By default, at build time the system creates bitmap versions of your vector drawables in different resolutions. This allows the icons to run on devices that aren't able to draw vector drawables.
- The `VectorDrawableCompat` class in the Android Support Library allows you to support vector drawables in Android 2.1 (API level 7) and higher.

Vector Asset Studio is a tool that helps you add Material Design icons and vector drawables to your Android project. To use it, **right-click** (or **Control-click**) on the **res** folder in the **Project > Android** pane of Android Studio, and select **New > Vector Asset**. The Configure Asset Studio wizard opens and guides you through the process.



Asset Type: [✓ Material Icon] Local file (SVG, PSD)
Name: ic_android_black_24dp
Icon:
Size: 24 dp X 24 dp ☐ Override
Opacity: 100 %
☐ Enable auto mirroring for RTL layout

For more information on using the Vector Asset Studio and supporting backward compatibility, see Vector Asset Studio.

Creating other images

Banner images, user profile pictures, and other images come in all shapes and sizes. In many cases they are larger than they need to be for a typical app UI. For example, the system Gallery app displays photos taken using an Android device's camera, and these photos are typically much higher resolution than the screen density of the device.

Android devices have finite memory, so ideally, you want to load only a lower resolution version of a photo in memory. The lower resolution version should match the size of the UI component that displays it. An image with a higher resolution doesn't provide any visible benefit, but still takes up precious memory and adds additional performance overhead due to additional on-the-fly scaling.

You can load resized images manually (see Loading Large Bitmaps Efficiently), but several third party libraries have been created to help with loading, scaling and caching images.

Using image-loading libraries

Image-loading libraries like Glide and Picasso can handle image sizing, caching, and display. These third-party libraries are optimized for mobile, and they are well-documented.

Glide supports fetching, decoding, and displaying video stills, images, and animated GIFs. You can use Glide to load images from Web APIs, as well as ones located in your resource files. Glide includes features such as loading placeholder images (for loading more detailed images), cross-fade animations, and automatic caching.

To use Glide:

1. Download the library.
2. Include the dependency in your app-level build.gradle file, replacing `n.n.n` with the latest version of Glide:

compile 'com.github.bumptech.glide:glide:n.n.n'

You can use Glide to load any image into a UI element. The following example loads an image from a URL into an `ImageView`:

```
ImageView imageView = (ImageView) findViewById(R.id.my_image_view);
Glide.with(this).load("URL").into(imageView);
```

In the code snippet, `this` refers to the context of the application. Replace `URL` with the URL of the image's location. By default, the image is stored in a local cache and accessed from there the next time it's called.

- For more examples, see the Glide wiki.
- For more about Glide, see the Glide documentation and the [glide] tag on stack overflow.
- For more about Picasso, see the Picasso documentation and the [picasso] tag on stack overflow.
- To compare the features of different libraries, search on stack overflow, for example Glide vs. Picasso.

Testing image rendering

Images render differently on different devices. To avoid surprises, use the Android Virtual Device (AVD) manager to create virtual devices that simulate screens of different sizes and densities. Use these AVDs to test all your images.

Speeding up your app

Fetching and caching images

When your app fetches an image, it can use a lot of data. To conserve data, make sure your request starts out as small as possible. Define and store pre-sized images on the server side, then request images that are already sized to the View.

Cache your images so that each image only needs to travel over the network once. When an image is requested, check your cache first. Only request the image over the network if the image is not in the cache. Use an image-loading library like Glide or Picasso to handle caching. These libraries also manage the size of the cache, getting rid of old or unused images. For more about libraries, see the libraries section of this chapter.

To maximize performance in different contexts, set conditional rules for how your app handles images, depending on connection type and stability.
Use `ConnectivityManager` to determine the connection type and status, then set conditional rules accordingly. For example, when a user is on a data connection (not WiFi), downgrade the requested image resolution to less than screen resolution. Upgrade the requested screen resolution again when the user is on WiFi.
When your app is fetching images over a network, a slow connection might leave your user waiting. Here are ways to keep your app feeling fast, even if images load slowly:

- Prioritize more important images so that they load first. Libraries like Glide and Picasso let you order requests by image priority.
- Prioritize requests for text before requests for images. If your app is usable without images, for example if it's a news feed app, letting a user scroll past your image can make the app functional and might even render the image request obsolete.
- Display placeholder colors while fetching images.

If you display placeholder colors, you want the look of your app to stay consistent while the app loads images. Use the Palette library to select a placeholder color based on the requested image's color balance. First, include the Palette library in your `build.gradle (Module:app)` file:

```
dependencies: {
    compile 'com.android.support:palette-v7:26.1.0'
}
```

Pull the dominant color for the image you want and set it as the background color in your `ImageView`. If you fetch the image using a library, put the following code after you've defined the URL to load into the `ImageView`:

```
Palette palette = Palette.from(tiles).generate(new PaletteAsyncListener(){
   Public void onGenerated(Pallet pallette) {
      Palette.Swatch background = palette.getDominantSwatch();
         if (background != null) {
            ImageView.setBackgroundColor(background.getRgb());
          }
     }
}
```

Serving images over a network

To save bandwidth and keep your app moving fast, use WebP formats to serve and send images.

Another way to save bandwidth is to serve and cache custom-sized images. To do this, allow clients to specify the resolution and size required for their device and `View`, then generate and cache the needed image on the server side before you send it. For example, a news feed landing page might request only a thumbnail image. Instead of sending a full-sized image, send only the thumbnail specified by that `ImageView`. You can further reduce the size of the thumbnail by producing images at different resolutions.

Tip: Use the `Activity.isLowRamDevice()` method to find out whether a device defines itself as "low RAM." If the method returns `true`, send low-resolution images so that your app uses less on-device memory.

Styles

In Android, a *style* is a collection of attributes that define the look and format of a `View`. You can apply the same style to any number of `View` elements in your app; for example, several `TextView` elements might have the same text size and layout. Using styles allows you to keep these common attributes in one location and apply them to each `TextView` using a single line of code in XML.

You can define styles yourself or use one of the platform styles that Android provides.

Defining and applying styles

To create a `style`, add a `<style>` element inside a `<resources>` element in any XML file located in the `values` folder inside the `res` folder in the **Project > Android** pane. When you create a project in Android Studio, a `styles.xml` file is created for you.
A `<style>` element includes the following:

- A `name` attribute. Use the style's name when you apply the `style` to a `View`.
- An optional `parent` attribute. You learn about using `parent` attributes in the Inheritance section below.
- Any number of `<item>` elements as child elements of `<style>`.
  Each `<item>` element includes one `style` attribute.

This example creates a `style` that formats text to use a light gray monospace typeface so it looks like code:

```
<resources>
    <style name="CodeFont">
        <item name="android:typeface">monospace</item>
        <item name="android:textColor">#D7D6D7</item>
    </style>
</resources>
```

The following XML applies the new `CodeFont` style to a `TextView`:

```
<TextView
    style="@style/CodeFont"
    android:text="@string/code_string" />
```

Inheritance

A new style can inherit the properties of an existing style. When you create a style that inherits properties, you define only the properties that you want to change or add. You can inherit properties from platform styles and from styles that you create yourself.

To inherit a platform style, use the `parent` attribute to specify the resource ID of the style you want to inherit. For example, here's how to inherit the Android platform's default text appearance (the `TextAppearance` style) and change its color:

```
<style name="GreenText" parent="@android:style/TextAppearance">
    <item name="android:textColor">#00FF00</item>
</style>
```

To apply this style, use `@style/GreenText`.

To inherit a style that you created yourself, use the name of the style you want to inherit as the first part of the new style's name, and separate the parts with a period:

`name="`*StyleToInherit.Qualifier*`"`

For example, to create a style that inherits the `CodeFont` style defined above, use `CodeFont` as the first part of the new style's name:

```
<style name="CodeFont.RedLarge">
    <item name="android:textColor">#FF0000</item>
    <item name="android:textSize">34sp</item>
</style>
```

This example includes the `typeface` attribute from the original `CodeFont` style, overrides the original `textColor` attribute with red, and adds a new attribute, `textSize`. To apply this style, use `@style/CodeFont.RedLarge`.

Themes

You create a theme the same way you create a style, which is by adding a `<style>` element inside a `<resources>` element in any XML file located in the `values` folder inside the `res` folder in the **Project > Android** pane.
What's the difference between a style and a theme?

- A *style* applies to a `View`. In XML, you apply a style using the `style` attribute.
- A *theme* applies to an `Activity` or an entire app, rather than to an individual `View`. In XML, you apply a theme using the `android:theme` attribute.

Any style can be used as a theme. For example, you could apply the `CodeFont` style as a theme for an `Activity`, and all the text inside the `Activity` would use gray monospace font.

Applying themes

To apply a theme to your app, declare it inside an `<application>` element inside the `AndroidManifest.xml` file. This example applies the `AppTheme` theme to the entire application:

```
android:theme="@style/AppTheme"
```

To apply a theme to an `Activity`, declare it inside an `<activity>` element in the `AndroidManifest.xml` file. In this example, the `android:theme` attribute applies the Theme_Dialog platform theme to the `Activity`:

```
<activity android:theme="@android:style/Theme.Dialog">
```

Default theme

When you create a new project in Android Studio, a default theme is defined for you within the `styles.xml` file. For example, this code might be in your `styles.xml` file:

```
<style name="AppTheme" parent="Theme.AppCompat.Light.DarkActionBar">
   <!-- Customize your theme here. -->
   <item name="colorPrimary">@color/colorPrimary</item>
   <item name="colorPrimaryDark">@color/colorPrimaryDark</item>
   <item name="colorAccent">@color/colorAccent</item>
</style>
```

In this example, `AppTheme` inherits from `Theme.AppCompat.Light.DarkActionBar`, which is one of the many Android platform themes available to you. (You'll learn about the color attributes in the unit on Material Design.)

Platform styles and themes

The Android platform provides a collection of styles and themes that you can use in your app. To find a list of all of them, you need to look in two places:

- The R.style class lists most of the available platform styles and themes.
- The support.v7.appcompat.R.style class lists more of them. These styles and themes have "`AppCompat`" in their names, and they are supported by the v7 appcompat library.

The style and theme names include underscores. To use them in your code, replace the underscores with periods. For example, here's how to apply the Theme_NoTitleBar theme to an activity:

```
<activity android:theme="@android:style/Theme.NoTitleBar">
```

And here's how to apply the AlertDialog_AppCompat style to a `View`:

```
<TextView
    style="@style/AlertDialog.AppCompat"
    android:text="@string/code_string" />
```

The documentation doesn't describe all the styles and themes in detail, but you can infer things about them from their names. For example, in `Theme.AppCompat.Light.DarkActionBar`:

- "Theme" indicates that this style is meant to be used as a theme.
- "AppCompat" indicates that this theme is supported by the v7 appcompat library.
- "Light" indicates that the theme consists of light background, white by default. All the text colors in this theme are dark, to contrast with the light background. (If you wanted a dark background and light text, your theme could inherit from a theme such as `Theme.AppCompat`, without "Light" in the name.)
- "DarkActionBar" indicates that a dark color is used for the action bar, so any text or icons in the action bar are a light color.

Another useful theme is `Theme.AppCompat.DayNight`, which enables the user to browse in a low-contrast "night mode" at night. It automatically changes the theme from `Theme.AppCompat.Light` to `Theme.AppCompat`, based on the time of day. To learn more about the `DayNight` theme, read Chris Banes's blog post.
To learn more about using platform styles and themes, visit the styles and themes guide.

Related practical

The related practical is 5.1: Drawables, styles, and themes.

Learn more

Android Studio documentation:

- Android Studio User Guide
- Add Multi-Density Vector Graphics
- Create App Icons with Image Asset Studio

Android developer documentation:

- Best Practices for User Interface
- Linear Layout
- Drawable Resources
- Styles and Themes
- Buttons
- Layouts
- Support Library
- Screen Compatibility Overview
- Support Different Screen Sizes
- Animate Drawable Graphics
- Loading Large Bitmaps Efficiently
- R.style class of styles and themes
- support.v7.appcompat.R.style class of styles and themes

Material Design:

- Design - Patterns - Navigation
- App Bar

Android Developers Blog: Android Design Support Library

Other:

- Medium: DayNight Theme Guide
- Roman Nurik's Android Asset Studio
- Glide documentation

5.2: Material Design

Contents:

- Introduction
- Principles of Material Design
- Colors
- Typography
- Layout
- Components and patterns
- Motion
- Related practical
- Learn more

Material Design is a visual design philosophy that Google created in 2014. The aim of Material Design is a unified user experience across platforms and device sizes. Material Design includes a set of guidelines for style, layout, motion, and other aspects of app design. The complete guidelines are available in the Material Design spec.

Material Design is for desktop web applications as well as for mobile apps. This chapter focuses only on Material Design for mobile apps on Android.

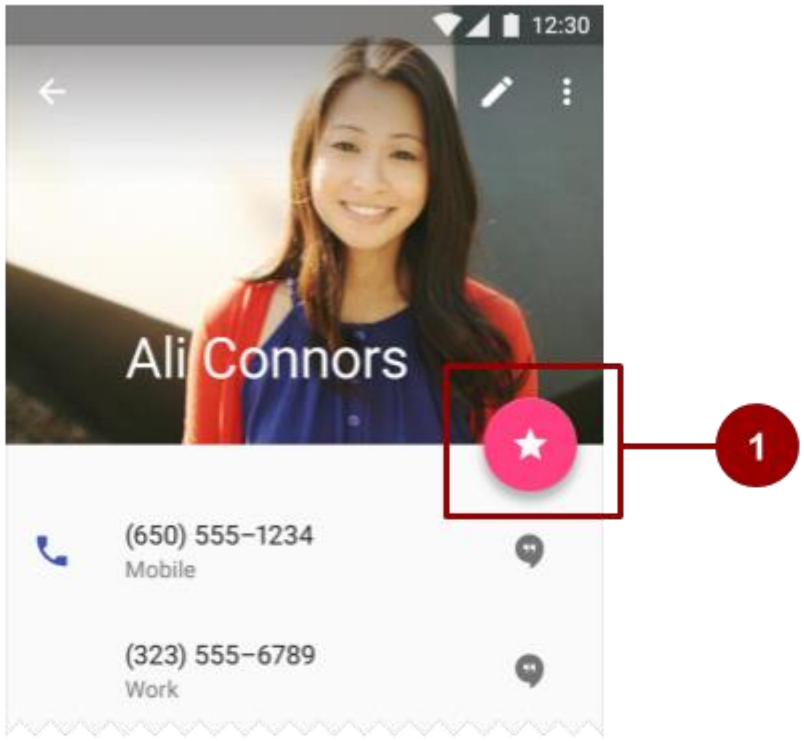
Principles of Material Design

In Material Design, elements in your Android app behave like real world materials: they cast shadows, occupy space, and interact with each other.

Bold, graphic, intentional

Material Design involves deliberate color choices, edge-to-edge imagery, large-scale typography, and intentional white space that create a bold and graphic interface.

Emphasize user actions in your app so that the user knows right away what to do, and how to do it. For example, highlight things that users can interact with, such as buttons, EditText fields, and switches.



In the above figure, #1 is a `FloatingActionButton` with a pink accent color.

Meaningful motion

Make animations and other motions in your app meaningful, so they don't happen at random. Use motions to reinforce the idea that the user is the app's primary mover. For example, design your app so that most motions are initiated by the user's actions, not by events outside the user's control. You can also use motion to focus the user's attention, give the user subtle feedback, or highlight an element of your app.

When your app presents an object to the user, make sure the motion doesn't break the continuity of the user's experience. For example, the user shouldn't have to wait for an animation or transition to complete.

The Motion section in this chapter goes into more detail about how to use motion in your app.

Colors

Material Design principles include the use of bold color.

Material Design color palette

The Material Design color palette contains colors to choose from, each with a primary color and shades labeled from 50 to 900:

- Choose a color labeled "500" as the primary color for your brand. Use that color and shades of that color in your app.
- Choose a contrasting color as your accent color and use it to create highlights in your app. Select any color that starts with "A".

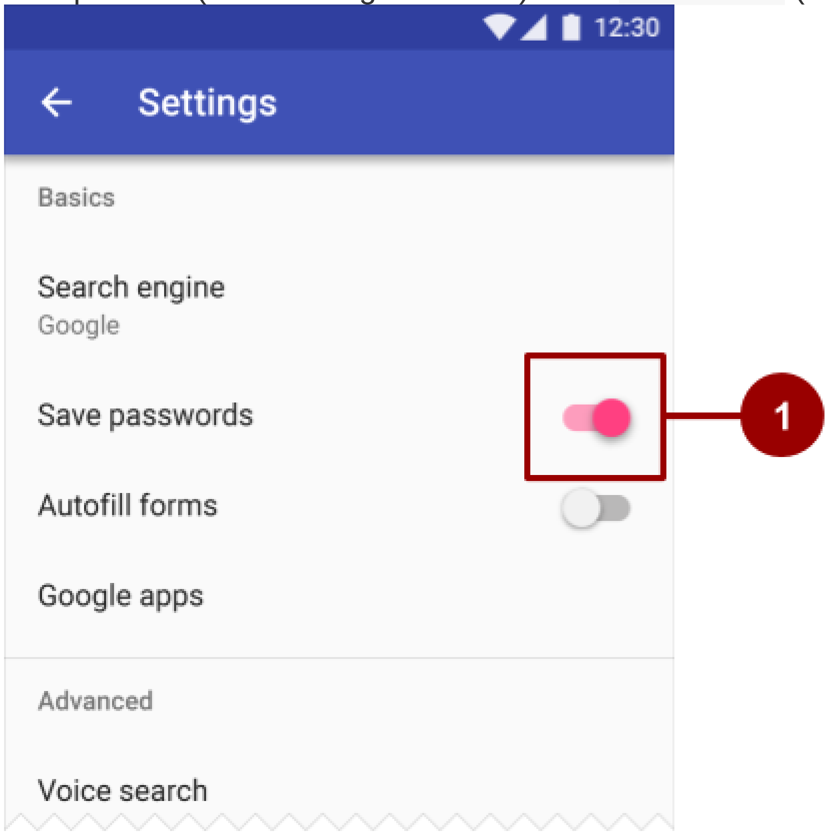
When you create an Android project in Android Studio, a sample Material Design color scheme is selected for you and applied to your theme. In `colors.xml` in the `values` folder, three `<color>` elements are defined, `colorPrimary`, `colorPrimaryDark`, and `colorAccent`:

```
<resources>
    <color name="colorPrimary">#3F51B5</color>
    <!-- Indigo. -->
    <color name="colorPrimaryDark">#303F9F</color>
    <!-- A darker shade of indigo. -->
    <color name="colorAccent">#FF4081</color>
    <!-- A shade of pink. -->
</resources>
```

In `styles.xml` in the `values` folder, the three defined colors are applied to the default theme, which applies the colors to some app elements by default:

- `colorPrimary` is used by several `View` elements by default. For example, in the `AppTheme` theme, `colorPrimary` is used as the background color for the action bar. Change this value to the "500" color that you select as your brand's primary color.
- `colorPrimaryDark` is used in areas that need to slightly contrast with your primary color, for example the status bar above the app bar. Set this value to a slightly darker version of your primary color.
- `colorAccent` is used as the highlight color for several `View` elements. It's also used for switches in the "on" position, `FloatingActionButton`, and more.

In the screenshot below, the background of the action bar uses `colorPrimary` (indigo), the status bar uses `colorPrimaryDark` (a darker shade of indigo), and the switch in the "on" position (#1 in the figure below) uses `colorAccent` (a shade of pink).



In summary, here's how to use the Material Design color palette in your Android app:

1. Pick a primary color for your app from Material Design color palette and copy its hex value into the `colorPrimary` item in `colors.xml`.
2. Pick a darker shade of this color and copy its hex value into the `colorPrimaryDark` item.
3. Pick an accent color from the shades starting with an "A" and copy its hex value into the `colorAccent` item.
4. If you need more colors, create additional `<color>` elements in the `colors.xml` file. For example, you could pick a lighter version of indigo and create an additional `<color>` element named `colorPrimaryLight`. (The name is up to you.)

```
<color name="colorPrimaryLight">#9FA8DA</color>
<!-- A lighter shade of indigo. -->
```

To use this color, reference it as `@color/colorPrimaryLight`.

Changing the values in `colors.xml` automatically changes the colors of the `View` elements in your app, because the colors are applied to the theme in `styles.xml`.

Contrast

Make sure all the text in your app's UI contrasts with its background. Where you have a dark background, make the text on top of it a light color, and vice versa. This kind of contrast is important for readability and accessibility, because not all people see colors the same way.

If you use a platform theme such as `Theme.AppCompat`, contrast between text and its background is handled for you. For example:

- If your theme inherits from `Theme.AppCompat`, the system assumes you are using a dark background. Therefore all of the text is near white by default.
- If your theme inherits from `Theme.AppCompat.Light`, the text is near black, because the theme has a light background.
- If you use the `Theme.AppCompat.Light.DarkActionBar` theme, the text in the action bar is near white, to contrast with the action bar's dark background. The rest of the text in the app is near black, to contrast with the light background.

Use color contrast to create visual separation among the elements in your app. Use your `colorAccent` color to call attention to key UI elements such as `FloatingActionButton` and switches in the "on" position.

Opacity

Your app can display text with different degrees of opacity to convey the relative importance of information. For example, text that's less important might be nearly transparent (low opacity).

Set the `android:textColor` attribute using any of these formats: `"#rgb"`, `"#rrggbb"`, `"#argb"`, or `"#aarrggbb"`. To set the opacity of text, use the `"#argb"` or `"#aarrggbb"` format and include a value for the *alpha channel*. The alpha channel is the `a` or the `aa` at the start of the `textColor` value.
The maximum opacity value, `FF` in hex, makes the color completely opaque. The minimum value, `00` in hex, makes the color complete transparent.
To determine what hex number to use in the alpha channel:

1. Decide what level of opacity you want to use, as a percentage. The level of opacity used for text depends on whether your background is dark or light. To find out what level of opacity to use in different situations, see the Text color portion of the Material Design guide.
2. Multiply that percentage, as a decimal value, by 255. For example, if you need primary text that's 87% opaque, multiply 0.87 x 255. The result is 221.85.
3. Round the result to the nearest whole number: 222.
4. Use a hex converter to convert the result to hex: `DE`. If the result is a single value, prefix it with `0`.

In the following XML code, the background of the text is dark, and the color of the primary text is 87% white (`deffffff`). The first two numbers of the color code (`de`) indicate the opacity.

```
<TextView
   android:layout_width="wrap_content"
   android:layout_height="wrap_content"
   android:text="Hello World!"
   android:textSize="45dp"
   android:background="@color/colorPrimaryDark"
   android:textColor="#deffffff"/>
```

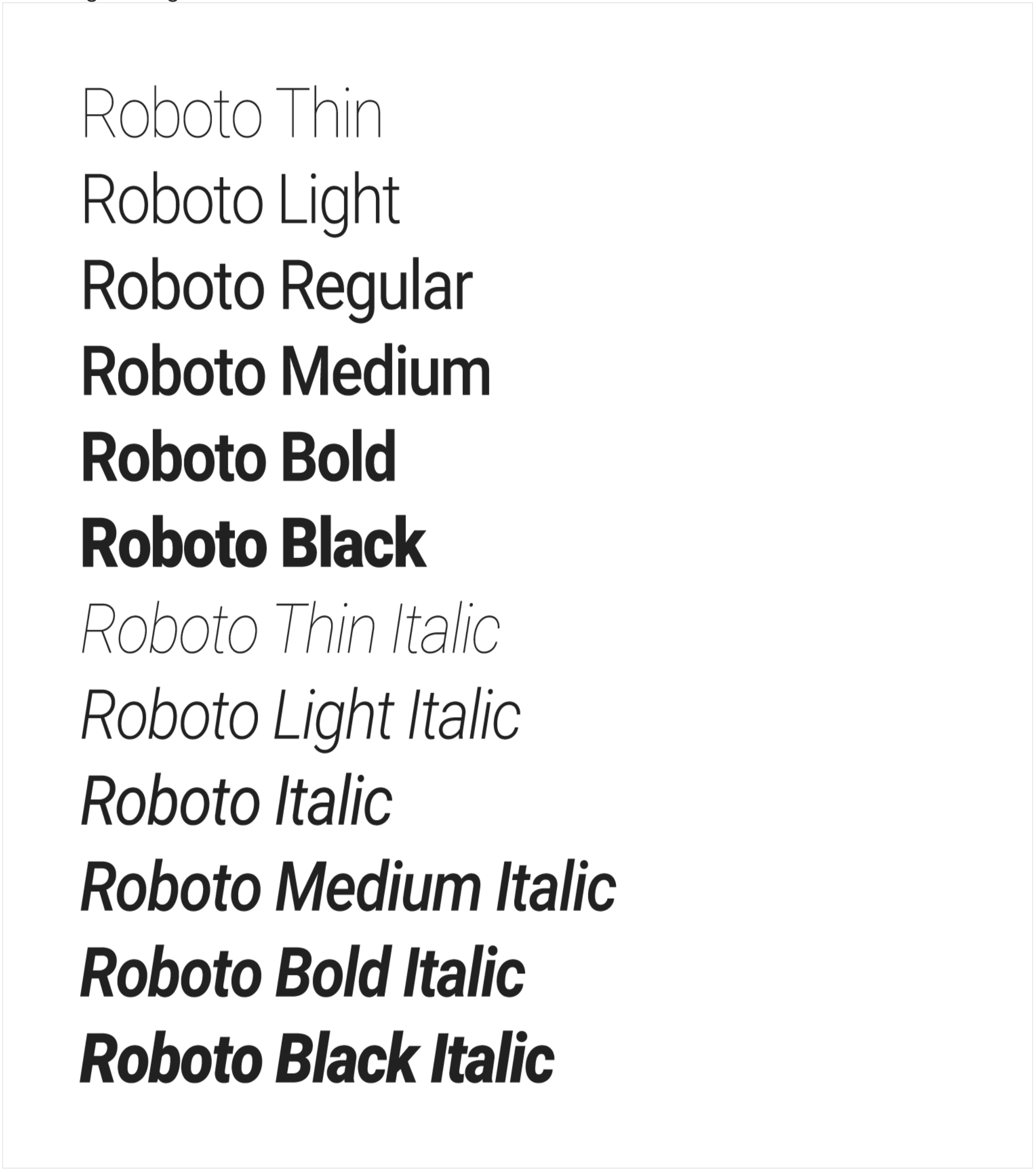
Typography

The Android design language relies on traditional typographic tools such as scale, space, rhythm, and alignment with an underlying grid. Successful deployment of these tools is essential to help users quickly understand a screen of information. To support such use of typography, Android supplies a type family named Roboto, created specifically for the requirements of UI and high-resolution screens.

With Android 8.0 (API level 26), you can also choose to provide a font as a resource in XML that is bundled with the app package (APK), or download a font from a font provider app. These features are available on devices running Android API versions 14 and higher through the Support Library 26.

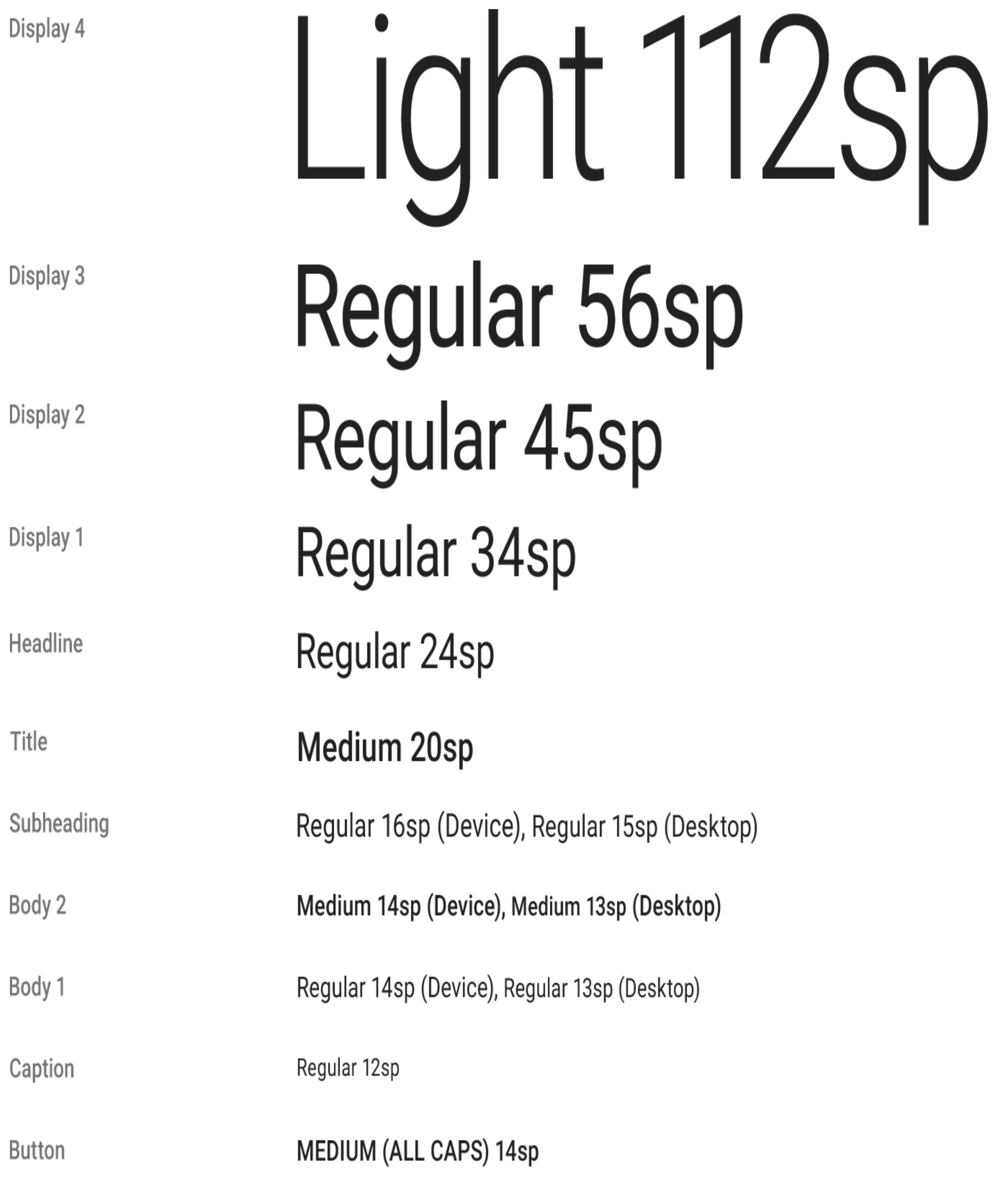
Typeface

Roboto is the standard Material Design typeface on Android. Roboto has six weights: Thin, Light, Regular, Medium, Bold, and Black.



Font styles

The Android platform provides predefined font styles and sizes that you can use in your app. These styles and sizes were developed to balance content density and reading comfort under typical conditions. Type sizes are specified with `sp` (scaleable pixels) to enable large type modes for accessibility.
Be careful not to use too many different type sizes and styles together in your layout.



Display 4	Light 112sp
Display 3	Regular 56sp
Display 2	Regular 45sp
Display 1	Regular 34sp
Headline	Regular 24sp
Title	Medium 20sp
Subheading	Regular 16sp (Device), Regular 15sp (Desktop)
Body 2	Medium 14sp (Device), Medium 13sp (Desktop)
Body 1	Regular 14sp (Device), Regular 13sp (Desktop)
Caption	Regular 12sp
Button	MEDIUM (ALL CAPS) 14sp

To use one of these predefined styles in a `View`, set the `android:textAppearance` attribute. This attribute defines the default appearance of the text: its color, typeface, size, and style. Use the backward-compatible `TextAppearance.AppCompat` style.
For example, to make a `TextView` appear in the Display 3 style, add the following attribute to the `TextView` in XML:

```
android:textAppearance="@style/TextAppearance.AppCompat.Display3"
```

For more information on styling text, view the Typography Material Design guidelines.

Fonts as resources

Android 8.0 (API level 26) introduces Fonts in XML, which lets you bundle fonts as resources in your app package (APK). You can create a `font` folder within the `res` folder as a resource directory using Android Studio, and then add a font XML file to the font folder. The fonts are compiled in your `R` file and are automatically available in Android Studio. To access a font resource, use `@font/myfont`, or `R.font.myfont`.
To use the Fonts in XML feature, the device that runs your app must run Android 8.0 (API level 26). To use the feature on devices running Android 4.1 (API level 16) and higher, use the Support Library 26. For more information on using the support library, refer to the Using the support library section.

To learn how to add fonts as XML resources, see Fonts in XML.

Downloadable fonts

An alternative to bundling fonts with the app package (APK) is to download the fonts from a provider app. Android 8.0 (API level 26) enables APIs to request fonts from a provider app, and the feature is available on devices running Android API versions 14 and higher through the Support Library 26. A font provider app is an app that retrieves fonts and caches them locally so that other apps can request and share fonts.

Downloadable fonts offer the following benefits:

- Reduces the APK size.
- Increases the app installation success rate.
- Improves the overall system health, as multiple APKs can share the same font through a provider. This saves users cellular data, phone memory, and disk space. In this model, the font is fetched over the network when needed.

You can set your app to download fonts by using the layout editor in Android Studio 3.0. For detailed instructions, see Downloadable Fonts.

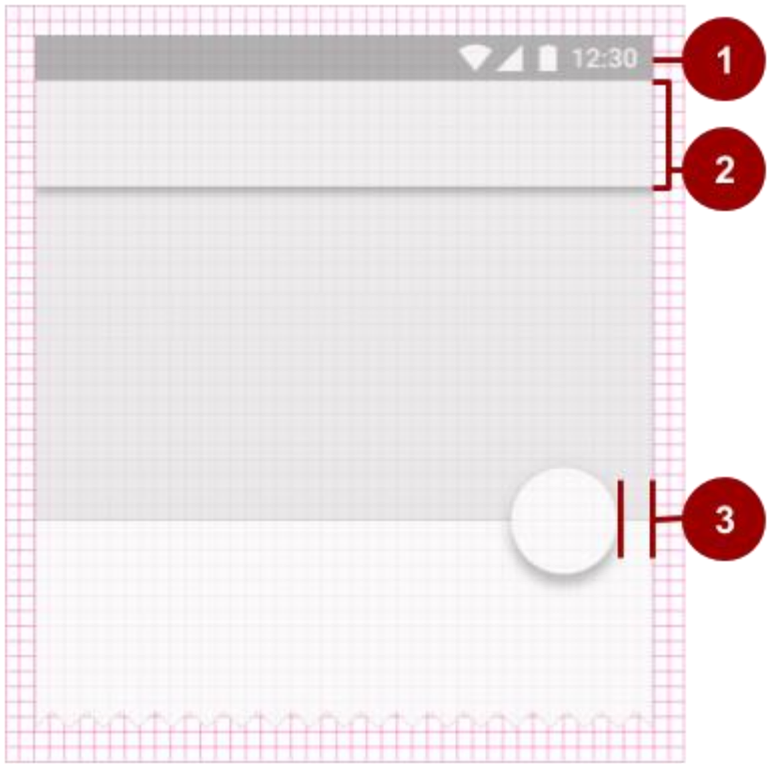
Layout

You specify `View` elements for your app's UI in layout resource files. Layout resources are written in XML and listed within the **layout** folder in the **res** folder in the **Project > Android** pane. The following guide explains some of the best practices for designing a layout.

Metrics

Components in the Material Design templates that are meant for mobile, tablet, and desktop devices align to an 8dp square grid. A *dp* is a density-independent pixel, an abstract unit based on screen density. A *dp* is similar to an *sp*, but *sp* is also scaled by the user's font size preference. That's why *sp* is preferred for accessibility.

The 8dp square grid guides the placement of elements in your layout. Every square in the grid is 8dp x 8dp, so the height and width of every element in the layout is a



multiple of 8dp.

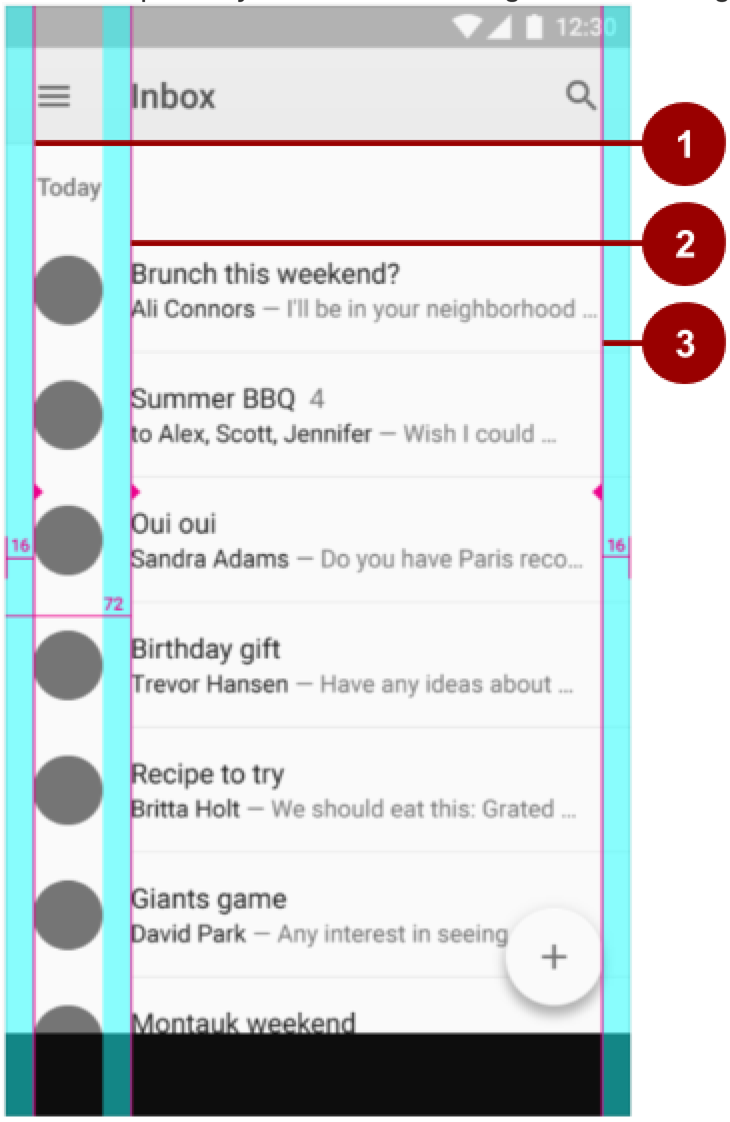
In the above figure:

1. The status bar in this layout is 24dp tall, the height of three grid squares.
2. The toolbar is 56dp tall, the height of seven grid squares.
3. One of the right-hand content margins is 16dp from the edge of the screen, the width of two grid squares.

Iconography in toolbars aligns to a 4dp square grid instead of an 8dp square grid, so the dimensions of icons in the toolbar are multiples of 4dp.

Keylines

Keylines are outlines in a layout grid that determine the placement of text and icons. For example, keylines mark the edges of the margins in a layout.


12:30
Inbox
Today
1
2
3
Brunch this weekend?
Ali Connors — I'll be in your neighborhood ...
Summer BBQ 4
to Alex, Scott, Jennifer — Wish I could ...
Oui oui
Sandra Adams — Do you have Paris reco...
Birthday gift
Trevor Hansen — Have any ideas about ...
Recipe to try
Britta Holt — We should eat this: Grated ...
Giants game
David Park — Any interest in seeing
Montauk weekend
16
72
16

In the above figure:

1. Keyline showing the left margin for the screen edge, which in this case is 16dp.
2. Keyline showing the left margin for content associated with an icon or avatar, 72dp.
3. Keyline showing the right margin for the screen edge, 16dp.

Material Design typography aligns to a 4dp *baseline grid*, which is a grid made up only of horizontal lines.

To learn more about metrics and keylines in Material Design, visit the metrics and keylines guide.

Components and patterns

`Button` elements and many other `View` elements used in Android conform by default to Material Design principles. The Material Design guide includes *components and patterns* that you can build on to help your users intuit how the elements in your UI work, even if users are new to your app.
Use Material Design components to guide the specs and behavior of buttons, chips, cards, and many other UI elements. Use Material Design patterns to guide how you format dates and times, gestures, the navigation drawer, and many other aspects of your UI.

This section teaches you about the Design Support Library and some of the components and patterns that are available to you. For complete documentation about all the components and patterns that you can use, see the Material Design guide.

Design Support Library

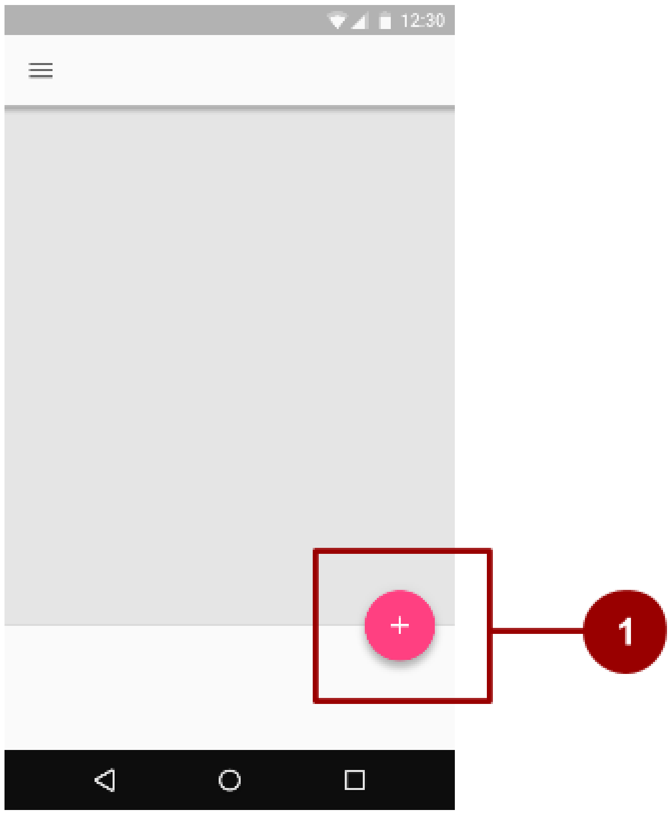
The Design package provides APIs to support adding Material Design components and patterns to your apps. The Design Support Library adds support for various Material Design components and patterns for you to build on. To use the library, include the following dependency in your `build.gradle (Module: app)` file:

```
compile 'com.android.support:design:26.1.0'
```

To make sure you have the most recent version number for the Design Support Library, check the Support Library page.

Floating action buttons (FABs)

Use a *floating action button* (FAB) for actions you want to encourage users to take. A FAB is a circled icon that floats "above" the UI. On focus it changes color slightly, and it appears to lift up when selected. When tapped, it can contain related actions.



In this figure:

1. A normal-sized FAB

To implement a FAB, use `FloatingActionButton` and set the FAB's attributes in your layout XML. For example:

```
<android.support.design.widget.FloatingActionButton
    android:id="@+id/addNewItemFAB"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:src="@drawable/ic_plus_sign"
    app:fabSize="normal"
    app:elevation="10%" />
```

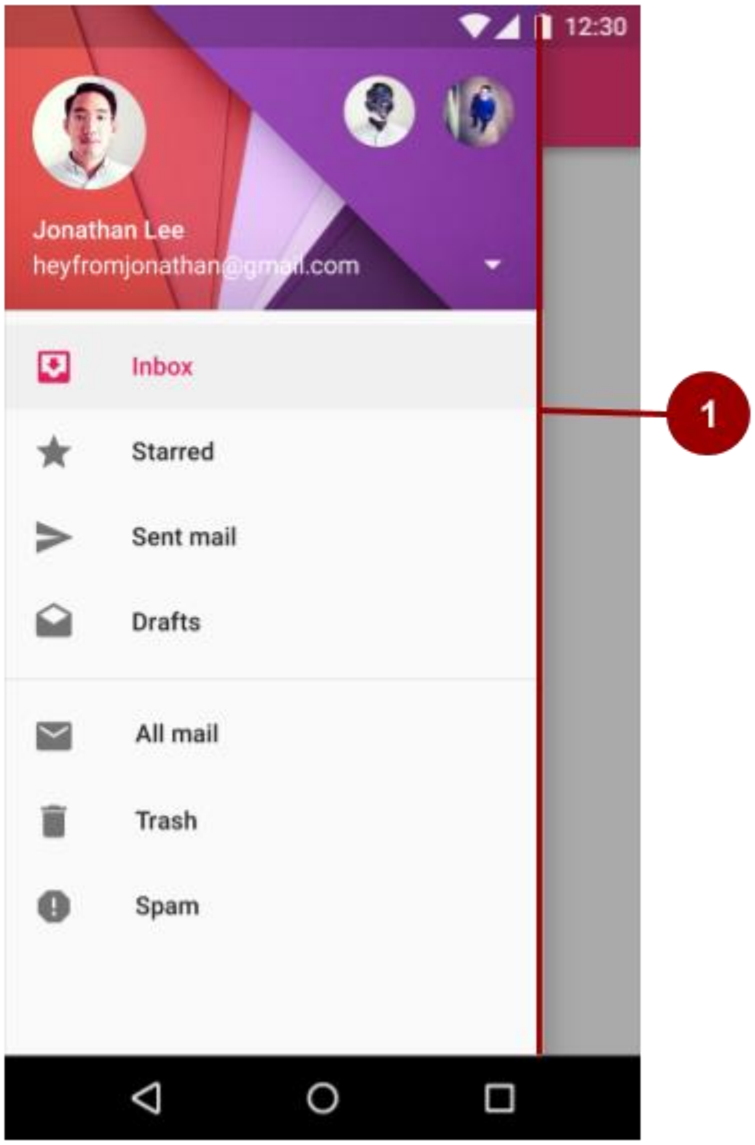
The `fabSize` attribute sets the FAB's size. It can be `"normal"` (56dp), `"mini"` (40dp), or `"auto"`, which changes based on the window size.

The FAB's *elevation* is the distance between its surface and the depth of its shadow. You can set the `elevation` attribute as a reference to another resource, a string, a boolean, or several other ways.

To learn about all the attributes you can set for a FAB including `clickable`, `rippleColor`, and `backgroundTint`, see `FloatingActionButton`. To make sure you're using FABs as intended, check the extensive FAB usage information in the Material Design guide.

Navigation drawers

A *navigation drawer* is a panel that slides in from the left and contains navigation destinations for your app. A navigation drawer (shown as #1 in the figure below) spans the height of the screen, and everything behind it is visible, but darkened.



To implement a navigation drawer, use the `DrawerLayout` APIs available in the Support Library.

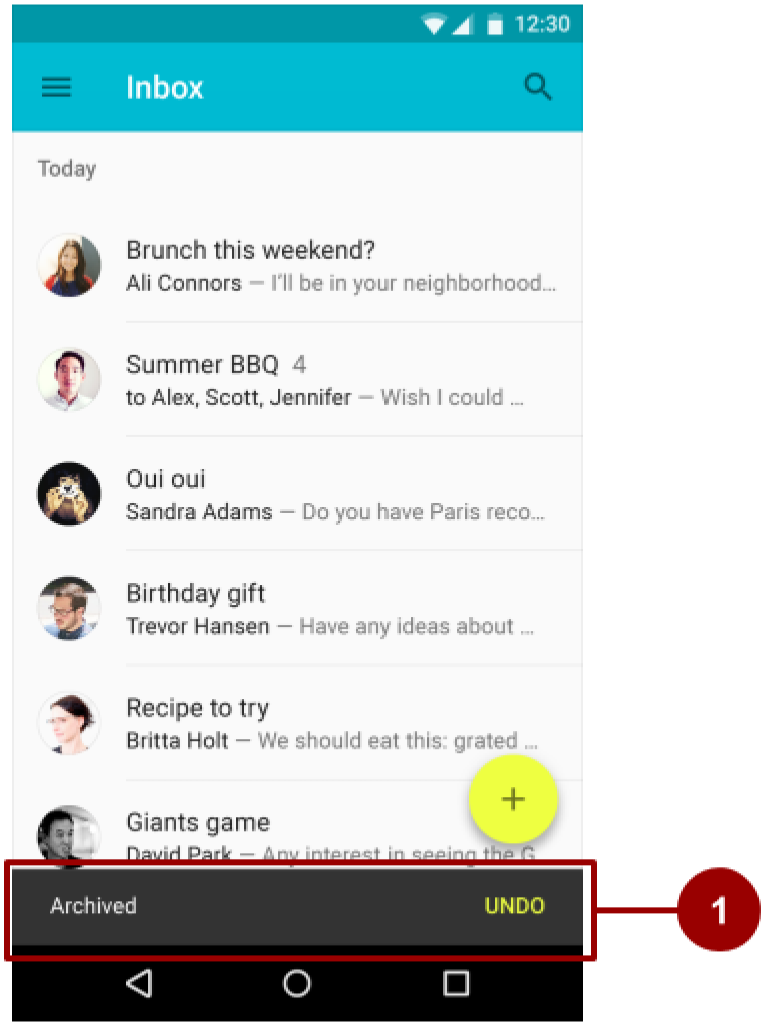
In your XML, use a `DrawerLayout` object as the root view of your layout. Inside it, add two views, one for your primary layout when the drawer is hidden, and one for the contents of the drawer.
For example, the following layout has two child views: a `FrameLayout` to contain the main content (populated by a `Fragment` at runtime), and a `ListView` for the navigation drawer.

```
<android.support.v4.widget.DrawerLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:id="@+id/drawer_layout"
    android:layout_width="match_parent"
    android:layout_height="match_parent">
    <!-- The main content view -->
    <FrameLayout
        android:id="@+id/content_frame"
        android:layout_width="match_parent"
        android:layout_height="match_parent" />
    <!-- The navigation drawer -->
    <ListView android:id="@+id/left_drawer"
        android:layout_width="240dp"
        android:layout_height="match_parent"
        android:layout_gravity="start"
        android:choiceMode="singleChoice"
        android:divider="@android:color/transparent"
        android:dividerHeight="0dp"
        android:background="#111"/>
</android.support.v4.widget.DrawerLayout>
```

The lesson on user navigation in this course provides a complete example of using a `DrawerLayout`. For more information, see Creating a Navigation Drawer and the usage information in the Material Design guide.

Snackbars

A *snackbar* provides brief feedback about an operation through a message in a horizontal bar on the screen. It contains a single line of text directly related to the operation performed. A snackbar (shown as #1 in the figure below) can contain a text


12:30
Inbox
Today
Brunch this weekend?
Ali Connors — I'll be in your neighborhood...
Summer BBQ 4
to Alex, Scott, Jennifer — Wish I could ...
Oui oui
Sandra Adams — Do you have Paris reco...
Birthday gift
Trevor Hansen — Have any ideas about ...
Recipe to try
Britta Holt — We should eat this: grated ...
Giants game
David Park — Any interest in seeing the G
Archived
UNDO
1

action, but no icons.

Snackbars automatically disappear after a timeout or after a user interaction elsewhere on the screen. You can associate a snackbar with any kind of view (any object derived from the `View` class). However, if you associate the snackbar with a `CoordinatorLayout`, the snackbar gains additional features:

- The user can dismiss the snackbar by swiping it away.
- The layout moves some other UI elements when the snackbar appears. For example, if the layout has a FAB, the layout moves the FAB up when it shows the snackbar, instead of drawing the snackbar on top of the FAB.

To create a `Snackbar` object, use the `Snackbar.make()` method. Specify the ID of the `CoordinatorLayout` to use for the snackbar, the message that the snackbar displays, and the length of time to show the message. For example, this statement creates the snackbar and calls `show()` to show the snackbar to the user:

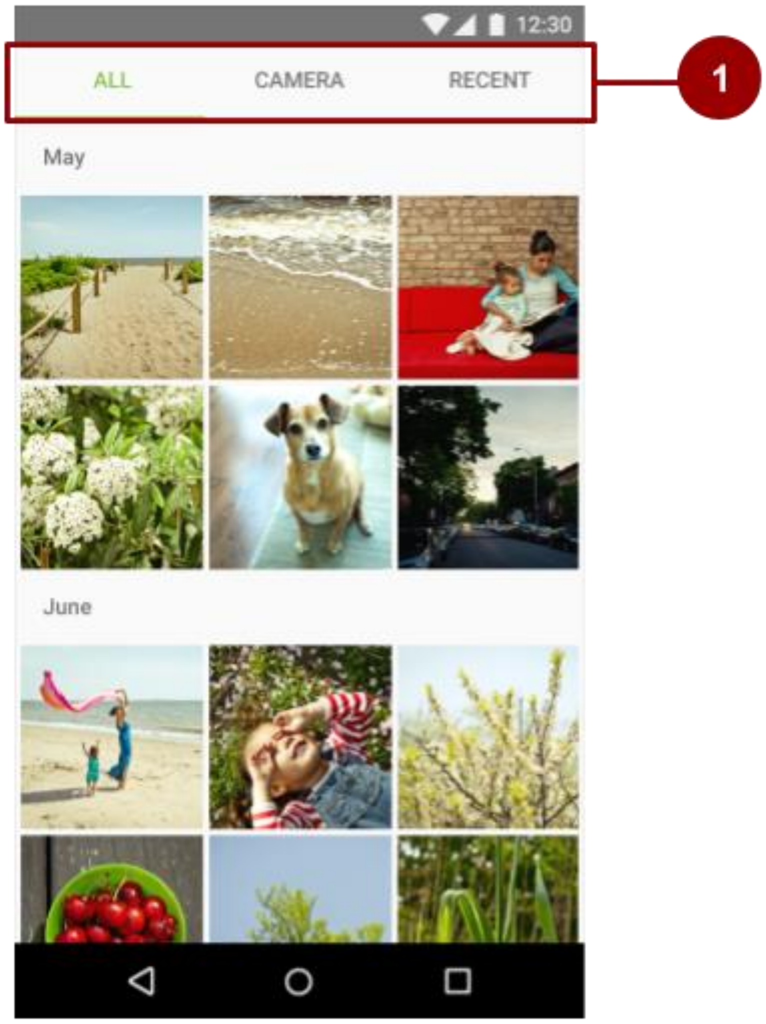
```
Snackbar.make(findViewById(R.id.myCoordinatorLayout),
               R.string.email_sent,Snackbar.LENGTH_SHORT).show;
```

For more information, see Building and Displaying a Pop-Up Message and the `Snackbar` reference. To make sure you're using snackbars as intended, see the snackbar usage information in the Material Design guide.

Tip: A `Toast` is similar to a `Snackbar`, except that a `Toast` is usually used for a system message, and a `Toast` can't be swiped off the screen.

Tabs

Use *tabs* to organize content at a high level. For example, the user might use tabs to switch between `View` elements, data sets, or functional aspects of an app. Present tabs as a single row above their associated content. Make tab labels short and informative. For example, in the figure below, the app shows three tabs (marked by



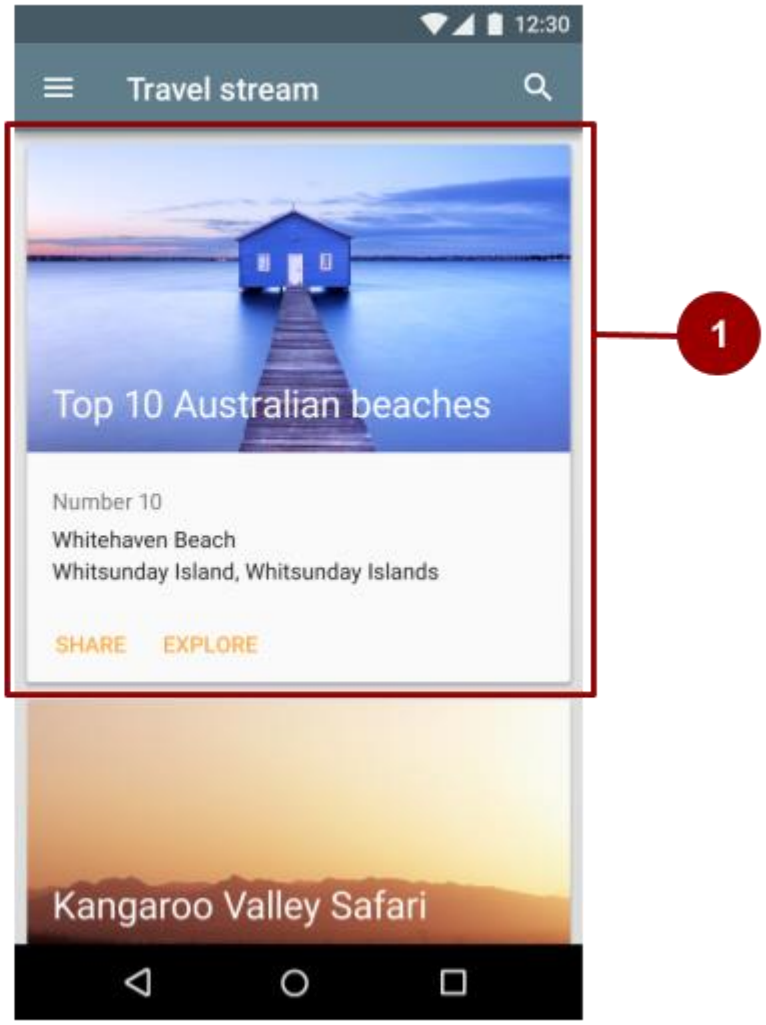
#1), with the **All** tab selected.
You can you use tabs with *swipe views* in which users navigate between tabs with a horizontal finger gesture (horizontal paging). If your tabs use swipe views, don't pair the tabs with content that also supports swiping. For an example, see the lesson in this course on providing user navigation.

For information on implementing tabs, see Creating Swipe Views with Tabs. To make sure you're using tabs as intended, see the extensive tab usage information in the Material Design guide.

Cards

A *card* is a sheet of material that serves as an entry point to more detailed information. Each card covers only one subject. A card may contain a photo, text, and a link. It can display content containing elements of varying size, such as photos with captions of variable length.

A *card collection* is a layout of cards on the same plane. The figure below shows one card in a card collection (marked by #1).

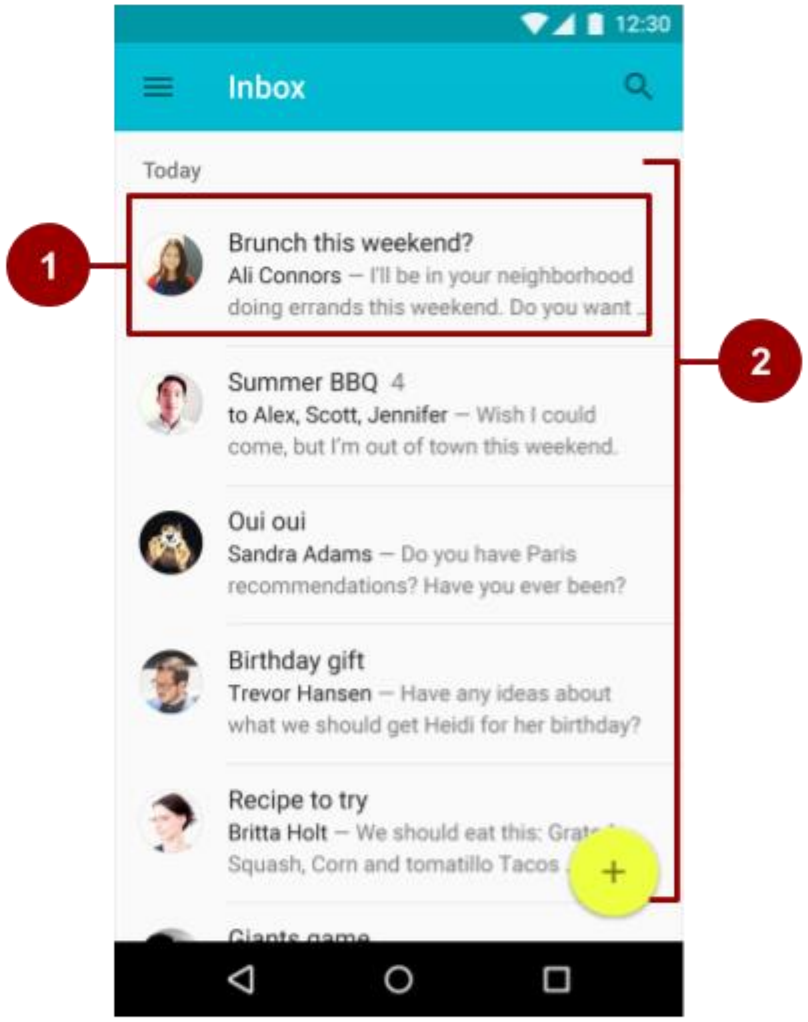


`CardView` is included as part of the v7 support library. To use the library, include the following dependency in your `build.gradle (Module: app)` file:

```
compile 'com.android.support:cardview-v7:26.1.0'
```

Lists

A *list* is a single continuous column of rows of equal width. Each row functions as a container for a tile. *Tiles* hold content, and can vary in height within a list.


12:30
Inbox
Today
Brunch this weekend?
Ali Connors — I'll be in your neighborhood doing errands this weekend. Do you want ...
Summer BBQ 4
to Alex, Scott, Jennifer — Wish I could come, but I'm out of town this weekend.
Oui oui
Sandra Adams — Do you have Paris recommendations? Have you ever been?
Birthday gift
Trevor Hansen — Have any ideas about what we should get Heidi for her birthday?
Recipe to try
Britta Holt — We should eat this: Grated Squash, Corn and tomatillo Tacos ...
Giants game
1
2

In the figure above:

1. A tile within the list
2. A list with rows of equal width, each containing a tile

To create a list, use the `RecyclerView` widget. include the following dependency in your `build.gradle (Module: app)` file:

```
compile 'com.android.support:recyclerview-v7:26.1.0'
```

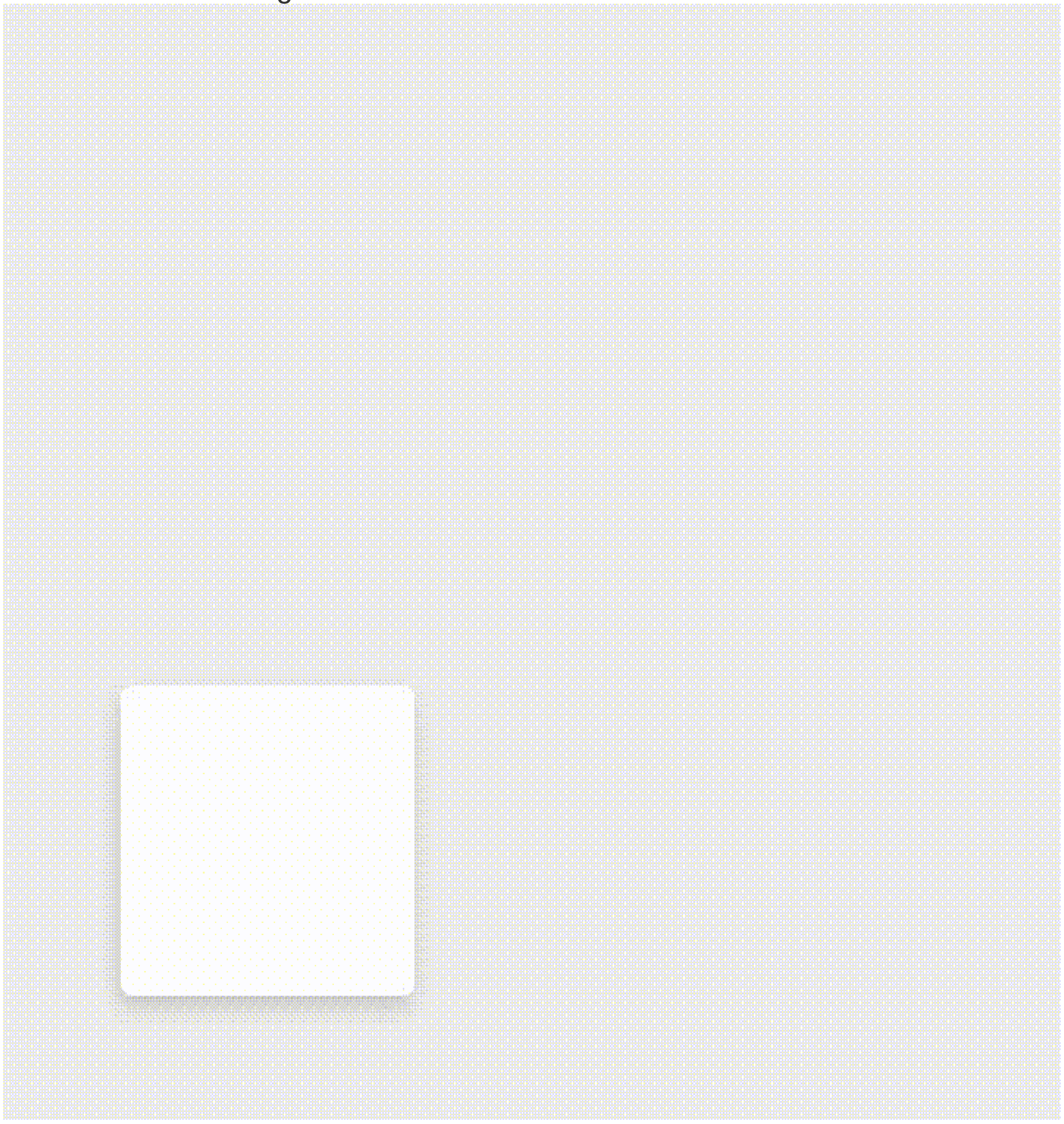
For more information on creating lists in Android, see Create a List with RecyclerView.

Motion

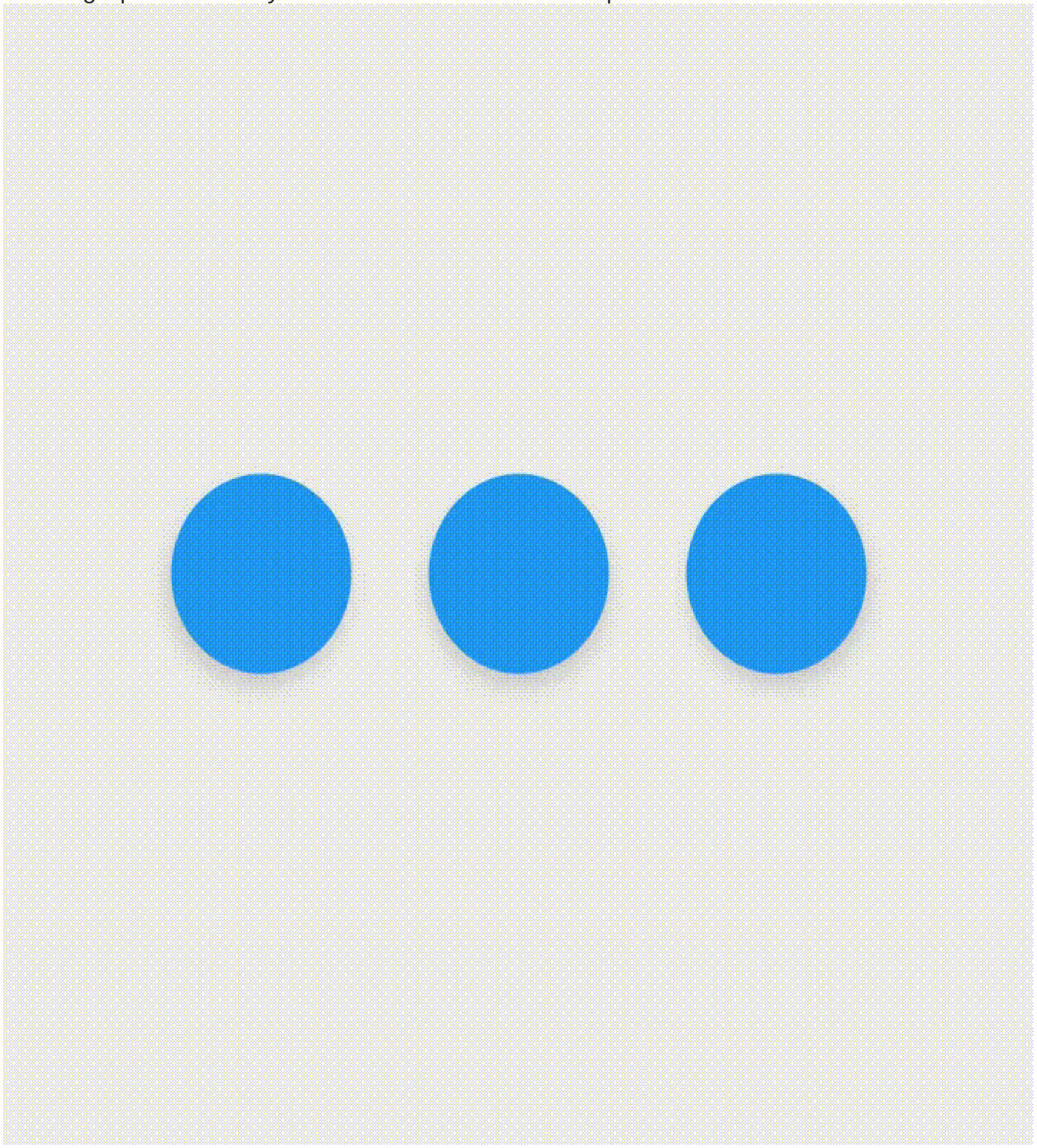
Motion in the world of Material Design is used to describe spatial relationships, functionality, and intention with beauty and fluidity. Motion shows how an app is organized and what it can do.

Motion in Material Design must be:

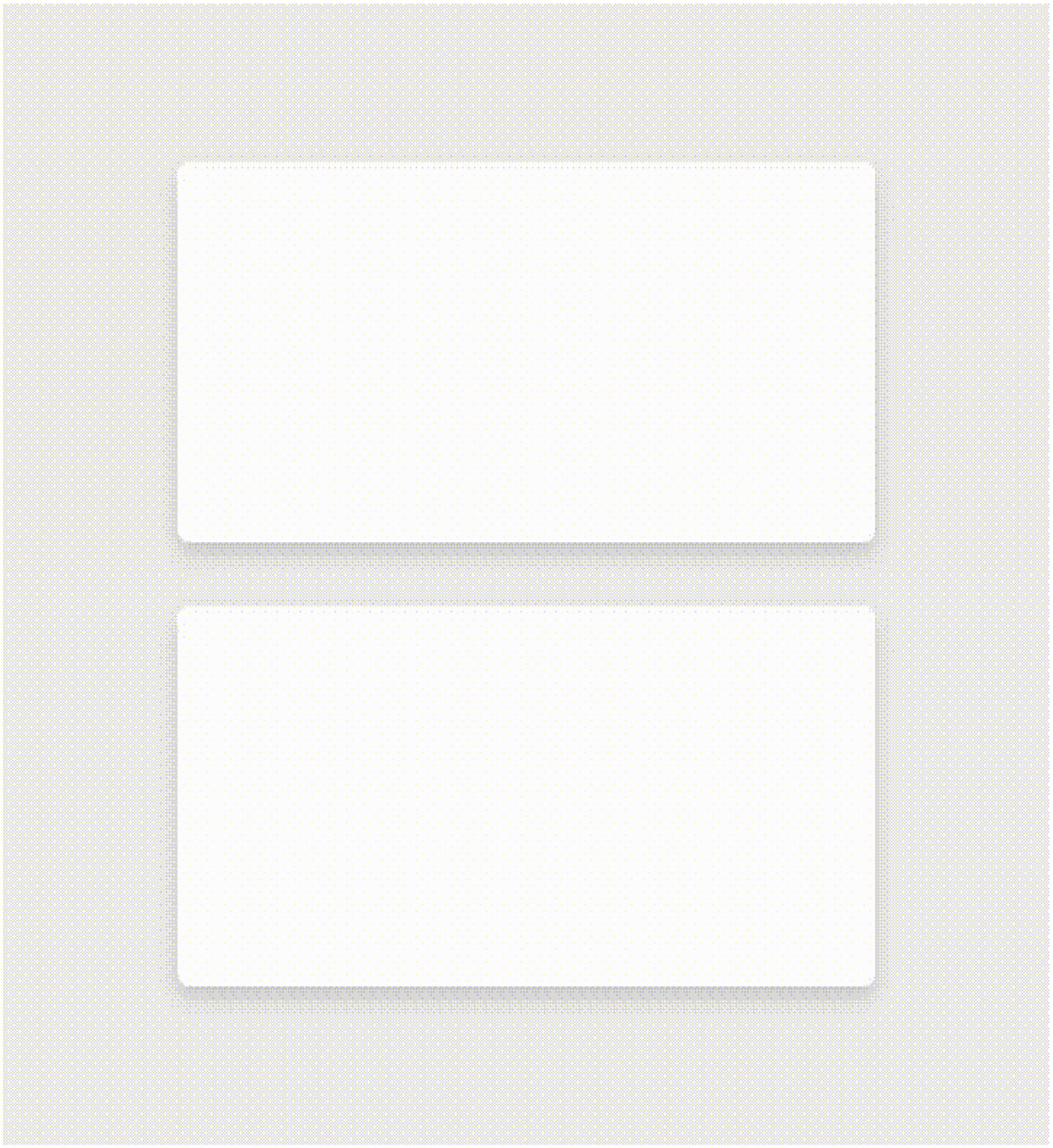
1. **Responsive**. It quickly responds to user input precisely where the user triggers it.
2. **Natural**. Movement is inspired by forces in the natural world. For example, real-world forces like gravity inspire an element's movement along an arc rather than in a straight line.



3. **Aware**. Material is aware of its surroundings, including the user and other material around it. Objects can be attracted to other objects in the UI, and they respond appropriately to user intent. As elements transition into view, their movement is choreographed in a way that defines their relationships.



4. **Intentional**. Movement guides the user's focus to the right place at the right time. Movement can communicate different signals, such as whether an action is unavailable.



To put these principles into practice in Android, use animations and transitions.

Animations

There are three ways you can create animation in your app:

- Property animation changes an object's properties over a specified period of time.The property animation system was introduced in Android 3.0 (API level 11). Property animation is more flexible than `View` animation, and it offers more features.
- `View` animation calculates animation using start points, endpoints, rotation, and other aspects of animation. The Android view animation system is older than the property animation system and can only be used for `View` elements. It's relatively easy to set up and offers enough capabilities for many use cases.
- Drawable animation lets you load a series of drawable resources one after another to create an animation. Drawable animation is useful if you want to animate things that are easier to represent with drawable resources, such as a progression of bitmap images.

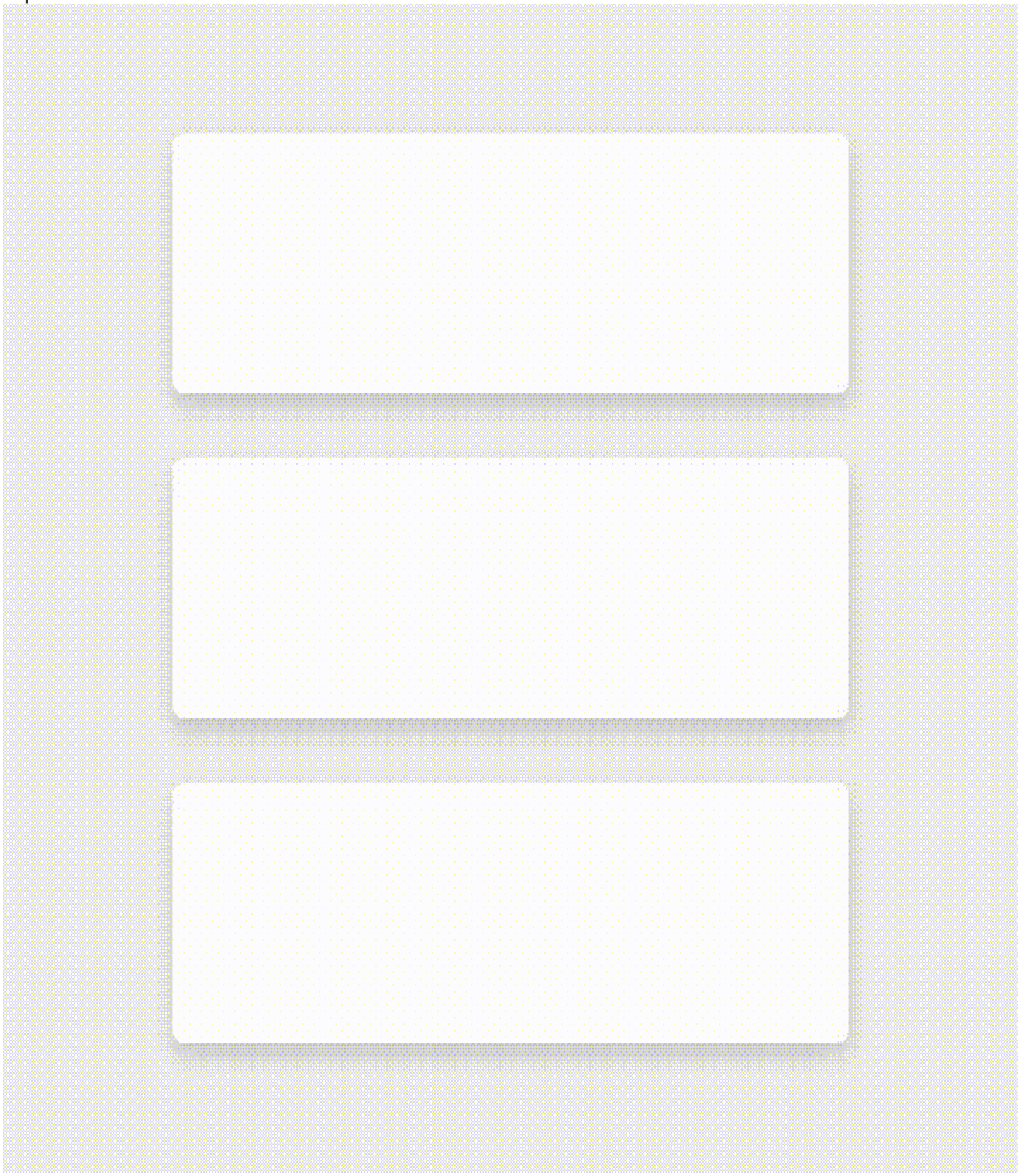
For complete details about these three types for animation, see the Animation and Graphics Overview.

The Material Design theme provides some default animations for touch feedback and activity transitions. The animation APIs let you create custom animations for touch feedback in UI controls, changes in view state, and activity transitions.

Touch feedback

Touch feedback provides instant visual confirmation at the point of contact when a user interacts with a UI element. The default touch feedback animation for a `Button` uses the `RippleDrawable` class, which transitions between different states with a ripple effect.

In this example, ripples of ink expand outward from the point of touch to confirm user input. The card "lifts" and casts a shadow to indicate an active state:



In most cases, you apply ripple functionality in your view XML by specifying the view background as follows:

- `?android:attr/selectableItemBackground` for a bounded ripple.
- `?android:attr/selectableItemBackgroundBorderless` for a ripple that extends beyond the `View`. It is drawn upon, and bounded by, the nearest parent of the `View` with a non-null background.

Note: The `selectableItemBackgroundBorderless` attribute was introduced in API level 21.

Alternatively, you can define a `RippleDrawable` as an XML resource using the `<ripple>` element.
You can assign a color to `RippleDrawable` objects. To change the default touch feedback color, use the theme's `android:colorControlHighlight` attribute.

Circular reveal

A *reveal animation* shows or hides a group of UI elements by animating the clipping boundaries for a `View`. In *circular reveal*, you reveal or hide a `View` by animating a clipping circle. (A *clipping circle* is a circle that crops or hides the part of an image that's outside the circle.)
To animate a clipping circle, use the `ViewAnimationUtils.createCircularReveal()` method. For example, here's how to reveal a previously invisible view using circular reveal:

```
// Previously invisible view
View myView = findViewById(R.id.my_view);

// Get the center for the clipping circle.
int cx = myView.getWidth() / 2;
int cy = myView.getHeight() / 2;

// Get the final radius for the clipping circle.
float finalRadius = (float) Math.hypot(cx, cy);

// Create the animator for this view (the start radius is zero).
Animator anim = ViewAnimationUtils
               .createCircularReveal(myView, cx, cy, 0, finalRadius);

// Make the view visible and start the animation.
myView.setVisibility(View.VISIBLE);
anim.start();
```

Here's how to hide a previously visible `View` using circular reveal:

```
// Previously visible view
final View myView = findViewById(R.id.my_view);

// Get the center for the clipping circle.
int cx = myView.getWidth() / 2;
int cy = myView.getHeight() / 2;
```

```
// Get the initial radius for the clipping circle.
float initialRadius = (float) Math.hypot(cx, cy);

// Create the animation (the final radius is zero.
Animator anim = ViewAnimationUtils.
              createCircularReveal(myView, cx, cy, initialRadius, 0);

// Make the view invisible when the animation is done.
anim.addListener(new AnimatorListenerAdapter() {
    @Override
    public void onAnimationEnd(Animator animation) {
        super.onAnimationEnd(animation);
        myView.setVisibility(View.INVISIBLE);
    }
});

// Start the animation.
anim.start();
```

Activity transitions

Activity transitions are animations that provide visual connections between different states in your UI. You can specify custom animations for *enter* and *exit* transitions, and for transitions of shared elements between activities.

- An *enter transition* determines how `View` elements in an `Activity` enter the scene. For example in an *explode enter transition*, `View` elements enter the scene from the outside and fly towards the center of the screen.
- An *exit transition* determines how `View` elements in an `Activity` exit the scene. For example in an *explode exit transition*, `View` elements exit the scene by moving away from the center.
- A *shared elements transition* determines how `View` elements that are shared between two activities transition between these activities. For example, if two activities have the same image in different positions and sizes, the `changeImageTransform` shared element transition translates and scales the image smoothly between these activities.

To use these transitions, set transition attributes in a `<style>` element in your XML. The following example creates a theme named `BaseAppTheme` that inherits one of the Material Design themes. The `BaseAppTheme` theme uses all three types of `Activity` transitions:

```
<style name="BaseAppTheme" parent="android:Theme.Material">
  <!-- enable window content transitions -->
  <item name="android:windowActivityTransitions">true</item>

  <!-- specify enter and exit transitions -->
  <item name="android:windowEnterTransition">@transition/explode</item>
  <item name="android:windowExitTransition">@transition/explode</item>

  <!-- specify shared element transitions -->
  <item name="android:windowSharedElementEnterTransition">
    @transition/change_image_transform</item>
  <item name="android:windowSharedElementExitTransition">
    @transition/change_image_transform</item>
</style>
```

The `change_image_transform` transition in this example is defined as follows:

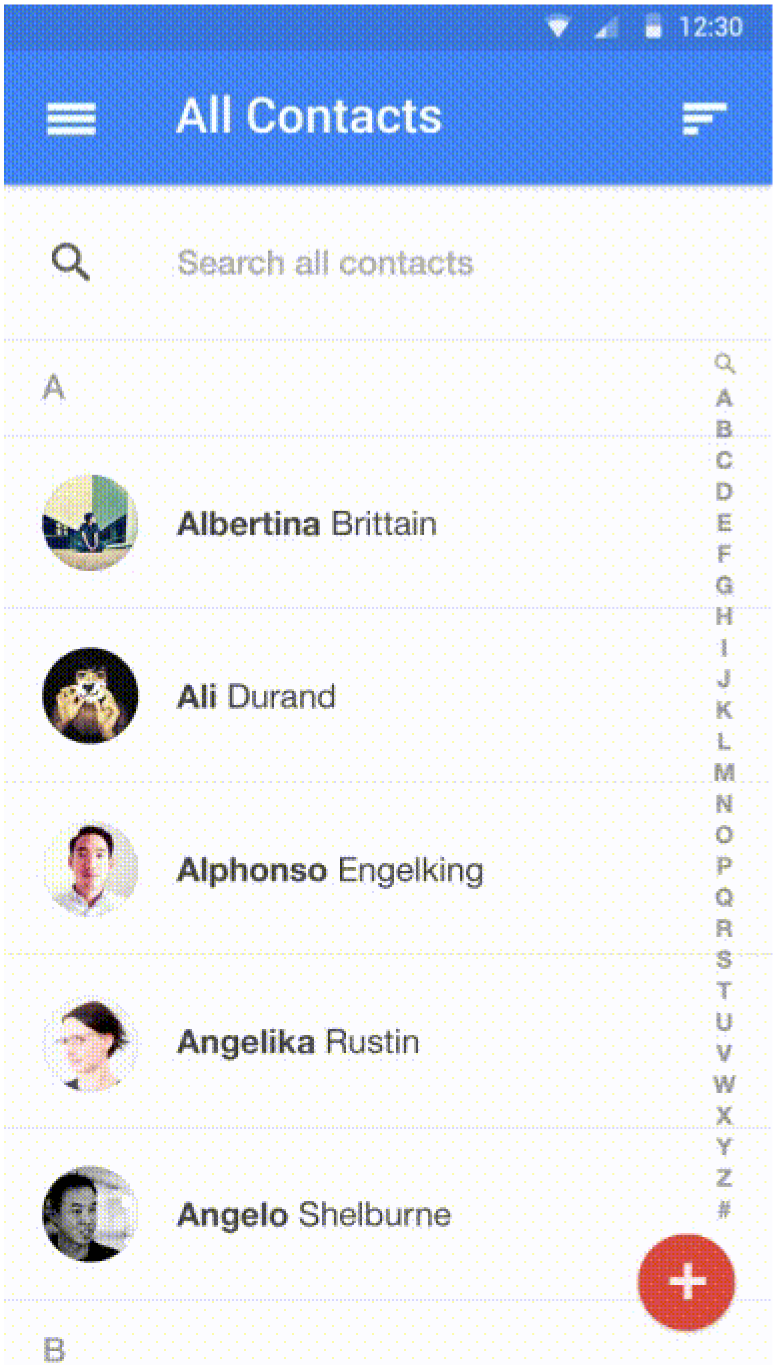
```
<!-- res/transition/change_image_transform.xml -->
<!-- (see also Shared Transitions below) -->
<transitionSet xmlns:android="http://schemas.android.com/apk/res/android">
  <changeImageTransform/>
</transitionSet>
```

The `changeImageTransform` element corresponds to the `ChangeImageTransform` class. For more information, see the API reference for `Transition`.

To enable window content transitions in your Java code instead, call the `Window.requestFeature()` method:

```
// Inside your activity
// (if you did not enable transitions in your theme)
getWindow().requestFeature(Window.FEATURE_CONTENT_TRANSITIONS);

// Set an exit transition
getWindow().setExitTransition(new Explode());
```



To specify transitions in your code, call the following methods with a `Transition` object:

- `Window.setEnterTransition()`
- `Window.setExitTransition()`
- `Window.setSharedElementEnterTransition()`
- `Window.setSharedElementExitTransition()`

For details about these methods, see `Window`.

To start an activity that uses transitions, use the `ActivityOptions.makeSceneTransitionAnimation()` method.
For more about implementing transitions in your app, see Start an Activity with an Animation.

Curved motion

In Android 5.0 (API level 21) and newer, you can define custom timing curves and curved motion patterns for animations. To do this, use the `PathInterpolator` class, which interpolates an object's path based on a Bézier curve or a `Path` object. The interpolator specifies a motion curve in a 1x1 square, with anchor points at (0,0) and (1,1) and control points that you specify using the constructor arguments. You can also define a path interpolator as an XML resource:

```
<pathInterpolator xmlns:android="http://schemas.android.com/apk/res/android"
    android:controlX1="0.4"
    android:controlY1="0"
    android:controlX2="1"
    android:controlY2="1"/
```

The system provides XML resources for the three basic curves in the Material Design specification:

- `@interpolator/fast_out_linear_in.xml`
- `@interpolator/fast_out_slow_in.xml`
- `@interpolator/linear_out_slow_in.xml`

To use a `PathInterpolator` object, pass it to the `Animator.setInterpolator()` method.
The `ObjectAnimator` class has constructors you can use to animate coordinates along a path using two or more properties at once. For example, the following code uses a `Path` object to animate the X and Y properties of a `View`:

```
ObjectAnimator mAnimator;
mAnimator = ObjectAnimator.ofFloat(view, View.X, View.Y, path);
// ... Rest of code
mAnimator.start();
```

Other custom animations

Other custom animations are possible, including animated state changes (using the `StateListAnimator` class) and animated vector drawables (using the `AnimatedVectorDrawable` class). For complete details, see Defining Custom Animations.

Related practical

The related practical is 5.2: Cards and colors.

Learn more

Android Studio documentation:

- Android Studio User Guide
- Add Multi-Density Vector Graphics
- Create App Icons with Image Asset Studio

Android developer documentation:

- Images and graphics
- Creating swipe views with tabs
- Create a List with RecyclerView
- Start an activity using an animation
- Defining Custom Animations
- Fonts in XML
- Downloadable Fonts
- `FloatingActionButton`
- Drawable Resources
- View Animation
- Support Library
- Animate drawable graphics
- Loading Large Bitmaps Efficiently

Material Design:

- Material Design for Android
- Material Design guidelines
- Material Design palette generator
- Create a List with RecyclerView
- App Bar
- Defining Custom Animations

Android Developers Blog: Android Design Support Library

Material Design

5.3: Resources for adaptive layouts

Contents:

- Introduction
- Externalizing resources
- Grouping resources
- Alternative resources
- Creating alternative resources
- Common alternative-resource qualifiers
- Providing default resources
- Related practical
- Learn more

An *adaptive layout* is a layout that works well for different screen sizes and orientations, different devices, different locales and languages, and different versions of Android.

In this chapter you learn how to create an adaptive layout by externalizing and grouping resources, providing alternative resources, and providing default resources in your app.

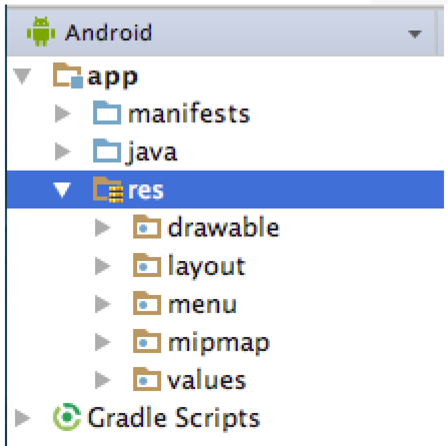
Externalizing resources

When you *externalize* resources, you keep them separate from your app code. For example, instead of hard-coding a string into your code, you name the string and add it to the `strings.xml` file, which appears in Android Studio's **Project > Android** pane inside the `values` folder in the `res` folder.
Always externalize resources such as drawables, icons, layouts, and strings. Here's why it's important:

- You can maintain externalized resources separately from your other code. If a resource is used in several places in your code and you need to change the resource, you only need to change it in one place.
- You can provide alternative resources that support specific device configurations, for example devices with different languages or screen sizes. This becomes increasingly important as more Android-powered devices become available.

Grouping resources

Store all your resources in the `res` folder. Organize resources by type into folders within `res`. You must use standardized names for these folders.
For example, the screenshot below shows the file hierarchy for a small project, as seen in the **Project > Android** pane. The folders that contain this project's default resources use standardized names: `drawable`, `layout`, `menu`, `mipmap` (for icons),



and `values`.

Folder name	Resource type
`animator`	XML files that define property animations.
`anim`	XML files that define tween animations.
`color`	XML files that define "state lists" of colors. (This is different from the `colors.xml` file in the `values` folder.) See Color State List Resource.
`drawable`	Bitmap files (WebP, PNG, 9-patch, JPG, GIF) and XML files that are compiled into drawables. See Drawable Resources.
`mipmap`	Drawable files for different launcher icon densities. See Projects Overview.
`layout`	XML files that define UI layouts. See Layout Resource.
`menu`	XML files that define app menus such as the options menu. See Menu Resource.
`raw`	Arbitrary files saved in raw form. To open these resources with a raw InputStream, call Resources.openRawResource() with the resource ID, which is `R.raw.filename`. If you need access to original file names and file hierarchy, consider saving resources in the `assets` folder instead of the `raw` folder within `res`. Files in `assets` are not given a resource ID, so you can read them only using AssetManager.
`values`	XML files that contain simple values, such as strings, integers, and colors. For clarity, place unique resource types in different files. For example, here are some filename conventions for resources you can create in this folder: * arrays.xml for resource arrays (typed arrays) * dimens.xml for dimension values * strings.xml, colors.xml, styles.xml See String Resources, Style Resource, and More Resource Types.
`xml`	Arbitrary XML files that can be read at runtime by calling Resources.getXml(). Various XML configuration files, such as a searchable configuration, must be saved here, along with preference settings.

Table 1 lists the standard resource folder names. The types are described more fully in Providing Resources.

Table 1: Standard resource folder names

Alternative resources

Most apps provide *alternative resources* to support specific device configurations. For example, your app should include alternative `drawable` resources for different screen densities, and alternative string resources for different languages. At runtime, Android detects the current device configuration and loads the appropriate resources.
If no resources are available for the device's specific configuration, Android uses the default resources that you include in your app—the default `drawable` elements, which are in the `drawable` folder, the default text strings, which are in the `strings.xml` file, and so on.
Like default resources, alternative resources are kept in folders inside `res`.
Alternative-resource folders use the following naming convention:
resource_name-*config_qualifier*

- *resource_name* is the folder name for this type of resource, as shown in Table 1. For example, **drawable** or **values**.
- *config_qualifier* specifies a device configuration for which these resources are used. All the possible qualifiers are shown in App Resources Overview (Table 2).

To add multiple qualifiers to one folder name, separate the qualifiers with a dash. If you use multiple qualifiers for a resource folder, you must list them in the order they are listed in the table.

Examples with one qualifier:

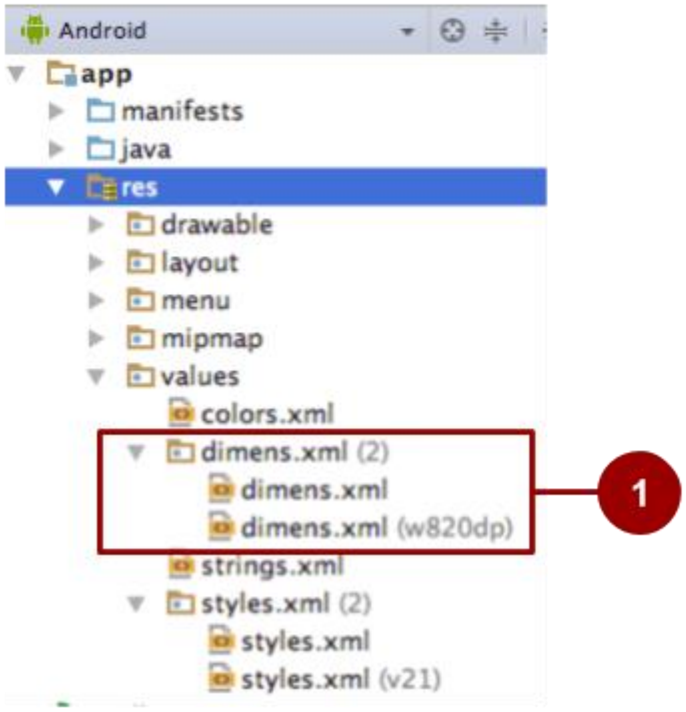
- String resources localized to Japanese would be in a `strings.xml` file inside the `values-ja` folder in the `res` folder (abbreviated as `res/values-ja/strings.xml`). Default string resources (resources to be used when no language-specific resources are found) would be in `res/values/strings.xml`. Notice that the XML files have identical names, in this case "strings.xml".
- Style resources for API level 21 and higher would be in a `res/values-v21/styles.xml` file. Default style resources would be in `res/values/styles.xml`.

Example with multiple qualifiers:

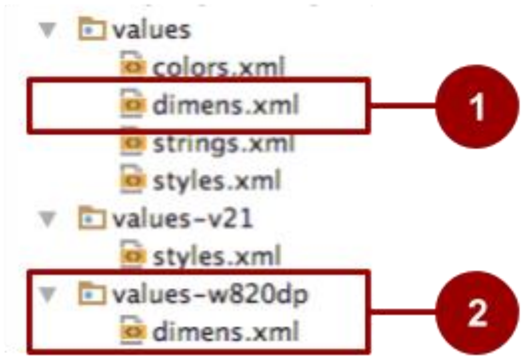
- Layout resources for a right-to-left layout running in "night" mode would be in a `res/layout-ldrtl-night/` folder.

In the **Project > Android** pane in Android Studio, the qualifier is not appended to the end of the folder. Instead, the qualifier is shown as a label on the right side of the file in parentheses. For example, in the **Project > Android** pane shown below, the `res/values/dimens.xml/` folder shows two files marked by #1 in the figure:

- The `dimens.xml` file, which includes default dimension resources.
- The `dimens.xml (w820dp)` file, which includes dimension resources for devices that have a minimum available screen width of 820dp.



In the **Project > Project Files** pane, the same information is presented differently, as shown in the screenshot below.



In the figure above:

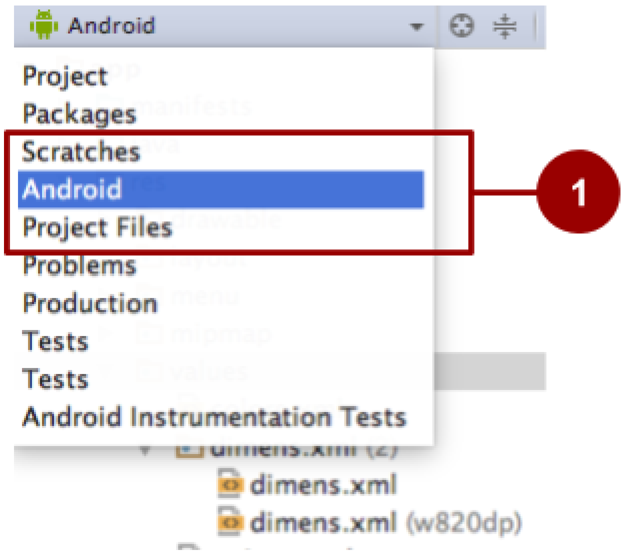
1. In the **Project > Project Files** pane in Android Studio, default resources for dimensions are shown in the `res/values` folder.
2. Alternative resources for dimensions are shown in `res/values-`*qualifier* folders.

Android supports many configuration qualifiers, and they are described in detail in Providing alternative resources. The table on that page lists the qualifiers in the order you must use when you combine multiple qualifiers in one folder name.

For example, `res/layout-ldrtl-night/` is a correct folder name, because the table lists the qualifier for layout direction before it lists the qualifier for night mode. A folder with the qualifier names in reverse order (`res/layout-night-ldrtl`) would be incorrect.

Creating alternative resources

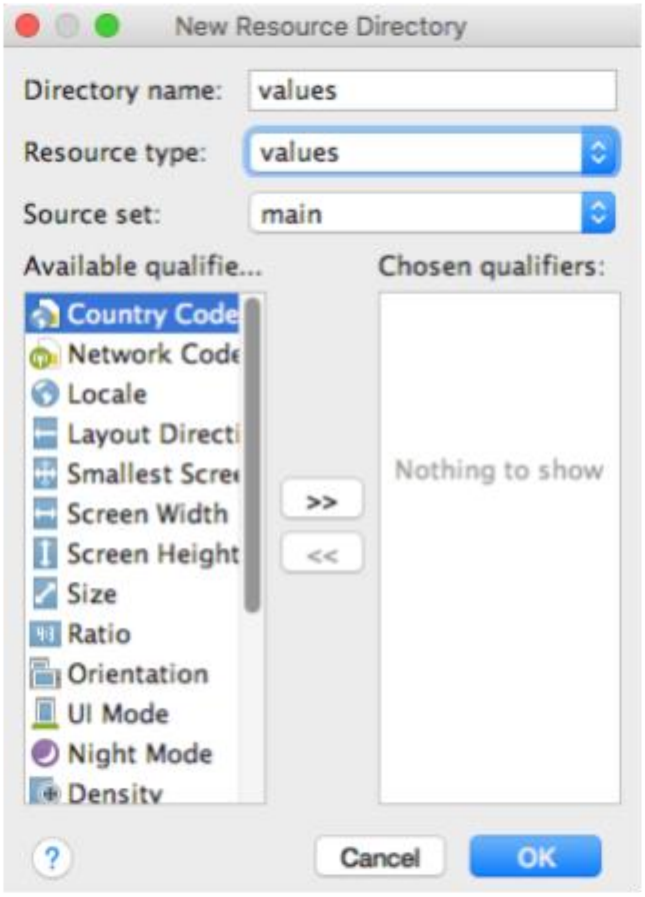
To create alternative resource folders most easily in Android Studio, choose **Android** in the **Project** pane (abbreviated as **Project > Android** pane) as shown by #1 in the figure below. If you don't see these options, make sure the **Project** pane is visible by selecting **View > Tool Windows > Project**.



To use Android Studio to create a new configuration-specific alternative resource folder in the `res` folder:

1. Be sure you are using the **Project > Android** pane, as shown above.

2. **Right-click** (or **Control-click**) on the **res** folder and select **New > Android resource directory**. The New Resource Directory dialog appears.



Directory name: values
Resource type: values
Source set: main

3. Select the type of resource (described in Table 1) and the qualifiers (described in App Resources Overview (Table 2) that apply to this set of alternative resources.

4. Click **OK**.

Save alternative resources in the new folder. The alternative resource files must be named exactly the same as the default resource files, for example "styles.xml" or "dimens.xml".

For the complete documentation about alternative resources, see Providing Alternative Resources.

Common alternative-resource qualifiers

This section describes a few commonly used qualifiers. App Resources Overview (Table 2)

gives the complete list.

Screen orientation

The screen-orientation qualifier has two possible values:

- `port`: The device is in portrait mode (vertical). For example, `res/layout-port/` would contain layout files to use when the device is in portrait mode.
- `land`: The device is in landscape mode (horizontal). For example, `res/layout-land/` would contain layout files to use when the device is in landscape mode.

If the user rotates the screen while your app is running, and if alternative resources are available, Android automatically reloads your app with alternative resources that match the new device configuration. For information about controlling how your app behaves during a configuration change, see Handling Runtime Changes.

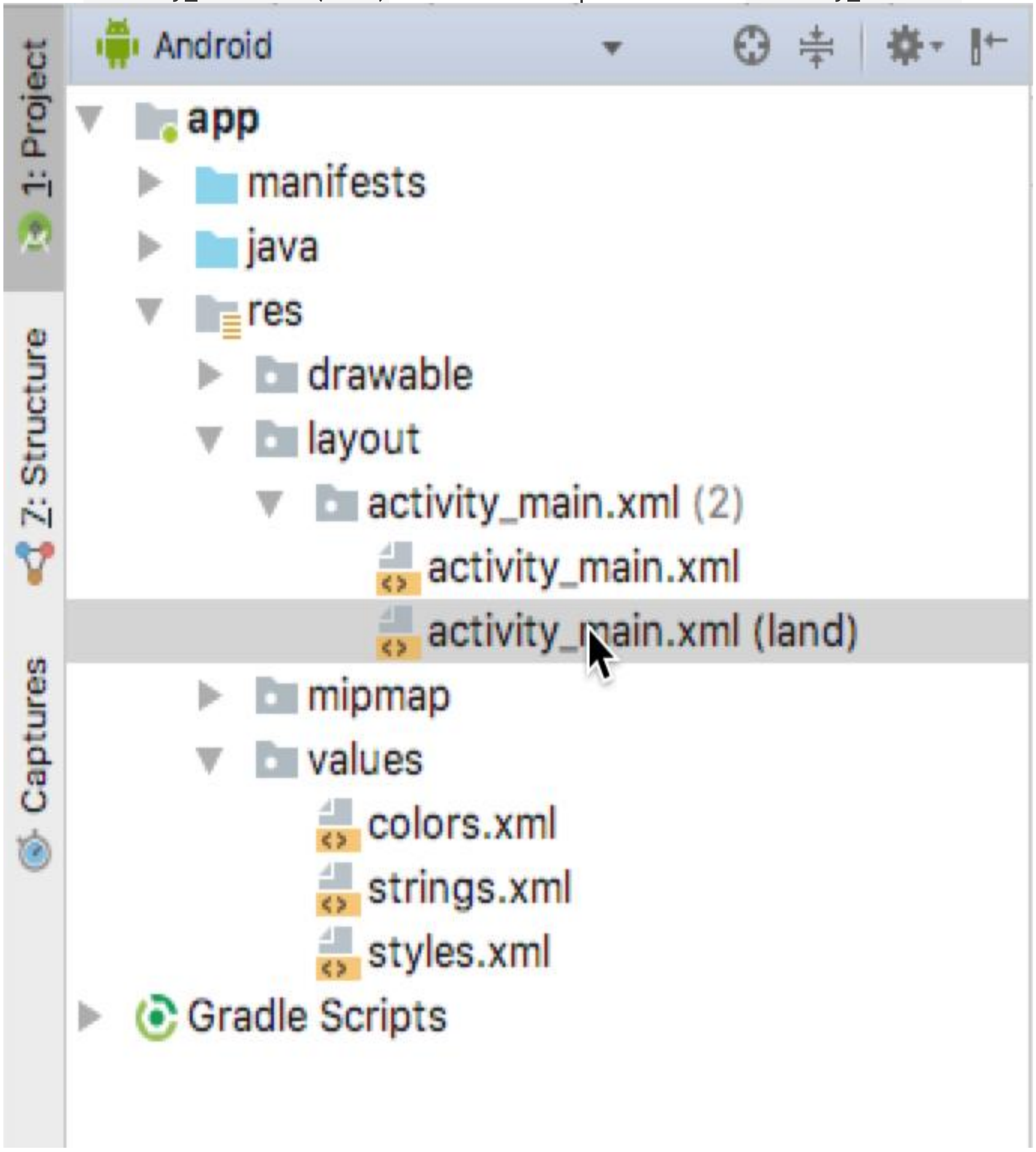
To create variants of your layout XML file for landscape orientation and larger displays, use the layout editor. To use the layout editor:

1. In Android Studio, open the XML file (such as **activity_main.xml**). The layout editor appears.
2. Click the **Design** tab at the bottom of the layout editor (if it is not already selected).
3. Click the **Orientation in Editor** button in the top toolbar.
4. Choose an option such as **Create Landscape Variation**.

For a landscape (horizontal) variation, a new editor window opens with the **land/activity_main.xml** tab showing the layout. You can change this layout, which is specifically for landscape orientation, without changing the original portrait (vertical) orientation.

5. In the **Project > Android** pane, look inside the **res > layout** directory, and you will see that Android Studio automatically created the variant for you,

called `activity_main.xml (land)` for the landscape version of `activity_main.xml`.



See the practical about using the layout editor for more details on using the layout editor.

Smallest width

The smallest-width qualifier, specified in the New Resource File dialog for Android Studio as **Smallest Screen Width**, specifies the minimum width of the device. It is the shortest of the screen's available height and width, the "smallest possible width" for the screen. The smallest width is a fixed screen-size characteristic of the device, and it does not change when the screen's orientation changes.

Specify smallest width in dp units, using the following format:

`sw`*n*`dp`
where *n* is the minimum width. For example, resources in a file named `res/values-sw320dp/styles.xml` are used if the device's screen is always at least 320dp wide. You can use this qualifier to ensure that a certain layout won't be used unless it has at least *n*dp of width available to it, regardless of the screen's current orientation.

Some values for common screen sizes:

- 320, for devices with screen configurations such as 240x320 ldpi (QVGA handset), 320x480 mdpi (handset), and 480x800 hdpi (high-density handset)
- 480, for screens such as 480x800 mdpi (tablet/handset)
- 600, for screens such as 600x1024 mdpi (7" tablet)
- 720, for screens such as 720x1280 mdpi (10" tablet)

If your app provides multiple resource folders with different values for the smallest-width qualifier, the system uses the one closest to (without exceeding) the device's smallest width.

For example, `res/values-sw600dp/dimens.xml` contains dimensions for images. When the app runs on a device with a smallest width of 600dp or higher (such as a tablet), Android uses the images in this folder.

Platform version

The platform-version qualifier specifies the minimum API level supported by the device. For example, use `v11` for API level 11 (devices with Android 3.0 or higher). See the Android API levels document for more information about these values. Use the platform-version qualifier when you use resources for functionality that's unavailable in prior versions of Android.

For example, WebP images require API level 14 (Android 4.0) or higher, and for full support they require API level 17 (Android 4.2) or higher. If you use WebP images:

- Put default versions of the images in a `res/drawable` folder. These images must use an image format that's supported for all API levels, for example PNG.
- Put WebP versions of the images in a `res/drawable-v17` folder. If the device uses API level 17 or greater, Android will select these resources at runtime.

Localization

The localization qualifier specifies a language and, optionally, a region. This qualifier is 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`).
You can specify a language alone, but not a region alone. Examples:

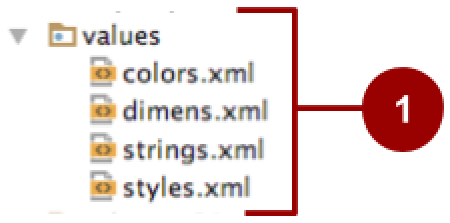
- `res/values-fr-rFR/strings.xml`
  Strings in this file are used on devices that are configured for the French language and have their region set to France.
- `res/mipmap-fr-rCA/`
  Icons in this folder are used on devices that are configured for the French language and have their region set to Canada.
- `res/layout-ja/content_main.xml`
  This layout is used on devices that are configured for the Japanese language.

If the user changes the language or region in the device's system settings while your app is running, and if alternative resources are available, Android automatically reloads your app with alternative resources that match the new device configuration. For information about controlling how your app behaves during a configuration change, see Handle configuration changes.

For a full guide on localization, see Localizing with Resources.

Providing default resources

Default resources specify the default design and content for your application. For example, when the app runs in a locale for which you have not provided locale-specific text, Android loads the default strings from `res/values/strings.xml`. If this default file is absent, or if it is missing even one string that your application needs, then your app doesn't run and shows an error.
Default resources have standard resource folder names (`values`, for example) without any qualifiers in the folder name or in parentheses after the file names.



Tip: Always provide default resources, because your app might run on a device configuration that you don't anticipate.

Sometimes new versions of Android add configuration qualifiers that older versions don't support. If you use a new resource qualifier and maintain code compatibility with older versions of Android, then when an older version of Android runs your app, the app crashes unless default resources are available. This is because the older version of Android can't use the alternative resources that are named with the new qualifier.

For example:

- Assume your `minSdkVersion` is set to `4` and you qualify all your drawable resources using night mode, meaning that you put all your `drawable` resources in `res/drawable-night/` and `res/drawable-notnight/`.
- When an API level 4 device runs the app, the device can't access your `drawable` resources. The Android version doesn't know about `night` and `notnight`, because these qualifiers weren't added until API level 8. The app crashes, because it doesn't include any default resources to fall back on.

In this example, you probably want `notnight` to be your default case. To solve the problem:

- Exclude the `notnight` qualifier and put your `drawable` resources in `res/drawable/` and `res/drawable-night/`.
- When an API level 4 device runs the app, it always uses the resources in the default `res/drawable/` folder.
- When a device at API level 8 or above uses the app, it uses the resources in the `res/drawable-night/` folder whenever the device is in night mode. At all other times, it uses the default (`notnight`) resources.

To provide the best device compatibility, provide default resources for every resource that your application needs. After your default resources are in place, create alternative resources for specific device configurations using the alternative-resource configuration qualifiers shown in App Resources Overview (Table 2).

Related practical

The related practical is 5.3: Adaptive layouts.

Learn more

Android Studio documentation: Android Studio User Guide

Android developer documentation:

- Resources Overview
- Providing Resources
- Localizing with Resources
- `LinearLayoutManager`
- `GridLayoutManager`
- Supporting Multiple Screens

Material Design:

- Material Design for Android
- Material Design Guidelines