App resources overview

Resources are the additional files and static content that your code uses, such as bitmaps, layout definitions, user interface strings, animation instructions, and more.

You should always externalize app resources such as images and strings from your code, so that you can maintain them independently. You should also provide alternative resources for specific device configurations, by grouping them in specially-named resource directories. At runtime, Android uses the appropriate resource based on the current configuration. For example, you might want to provide a different UI layout depending on the screen size or different strings depending on the language setting.

Once you externalize your app resources, you can access them using resource IDs that are generated in your project's `R` class. This document shows you how to group your resources in your Android project and provide alternative resources for specific device configurations, and then access them from your app code or other XML files.

Grouping resource types

You should place each type of resource in a specific subdirectory of your project's `res/` directory. For example,

here's the file hierarchy for a simple project:

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

As you can see in this example, the **`res/`** directory contains all the resources (in subdirectories): an image resource, two layout resources, **`mipmap/`** directories for launcher icons, and a string resource file. The resource directory names are important and are described in table 1.

Note: For more information about using the mipmap folders, see Managing Projects Overview.

Table 1. Resource directories supported inside project **`res/`** directory.

Directory	Resource Type
`animator/`	XML files that define property animations.

anim/	XML files that define tween animations. (Property animations can also be saved in this directory, but the **animator/** directory is preferred for property animations to distinguish between the two types.)
color/	XML files that define a state list of colors. See Color State List Resource
drawable/	Bitmap files (**.png**, **.9.png**, **.jpg**, **.gif**) or XML files that are compiled into the following drawable resource subtypes: • Bitmap files • Nine-Patches (re-sizable bitmaps) • State lists • Shapes • Animation drawables • Other drawables See Drawable Resources.
mipmap/	Drawable files for different launcher icon densities. For more information on managing launcher icons with **mipmap/** folders, see Managing Projects Overview.
layout/	XML files that define a user interface layout. See Layout Resource.
menu/	XML files that define app menus, such as an Options Menu, Context Menu, or Sub Menu. See Menu Resource.
raw/	Arbitrary files to save in their raw form. To open these resources with a raw **InputStream**, call **Resources.openRawResource()** with the resource ID, which is **R.raw.*filename***. However, if you need access to original file names and file hierarchy, you might consider saving some resources in the **assets/** directory (instead of **res/raw/**). Files in **assets/** aren't

	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. Whereas XML resource files in other **`res/`** subdirectories define a single resource based on the XML filename, files in the **`values/`** directory describe multiple resources. For a file in this directory, each child of the **`<resources>`** element defines a single resource. For example, a **`<string>`** element creates an **`R.string`** resource and a **`<color>`** element creates an **`R.color`** resource. Because each resource is defined with its own XML element, you can name the file whatever you want and place different resource types in one file. However, for clarity, you might want to place unique resource types in different files. For example, here are some filename conventions for resources you can create in this directory: • arrays.xml for resource arrays (typed arrays). • colors.xml for color values • dimens.xml for dimension values. • strings.xml for string values. • styles.xml for styles. 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 must be saved here, such as a searchable configuration.
	Font files with extensions such as **`.ttf`**, **`.otf`**, or **`.ttc`**, or XML files that include a **`<font-family>`**

`font/`	element. For more information about fonts as resources, go to Fonts in XML.

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

For more information about certain types of resources, see the Resource Types documentation.

The resources that you save in the subdirectories defined in table 1 are your "default" resources. That is, these resources define the default design and content for your app. However, different types of Android-powered devices might call for different types of resources. For example, if a device has a larger than normal screen, then you should provide different layout resources that take advantage of the extra screen space. Or, if a device has a different language setting, then you should provide different string resources that translate the text in your user interface. To provide these different resources for different device configurations, you need to provide alternative resources, in addition to your default resources.

Providing alternative resources

Almost every app should provide alternative resources to support specific device configurations. For instance, you 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 for your app.

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

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

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

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

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

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

For example, here are some default and alternative resources:

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

The **`hdpi`** qualifier indicates that the resources in that directory are for devices with a high-density screen. The images in each of these drawable directories are sized for a specific screen density, but the filenames are exactly the same. This way, the resource ID that you use to reference the **`icon.png`** or **`background.png`** image is always the same, but Android selects the version of each resource that best matches the current device, by comparing the device configuration information with the qualifiers in the resource directory name.

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

the default system language.

Android supports several configuration qualifiers and you can add multiple qualifiers to one directory name, by separating each qualifier with a dash. Table 2 lists the valid configuration qualifiers, in order of precedence—if you use multiple qualifiers for a resource directory, you must add them to the directory name in the order they are listed in the table.

Table 2. Configuration qualifier names.

Configuration	Qualifier Values	Description

MCC and MNC	Examples:
	`mcc310`
	`mcc310-mnc004`
	`mcc208-mnc00`
	etc.
		The mobile country code (MC optionally followed by mobile network code (MNC) from the SIM card in the device. For example, `mcc310` is U.S. on any carrier, `mcc310-mnc004` is U.S. Verizon, and `mcc208-mnc00` is France on Orange.
		
		If the device uses a radio connection (GSM phone), the MCC and MNC values come f the SIM card.
		
		You can also use the MCC alc (for example, to include count specific legal resources in you app). If you need to specify based on the language only, t use the *language and region* qualifier instead (discussed n If you decide to use the MCC

		MNC qualifier, you should do with care and test that it work expected. Also see the configuration fie **mcc**, and **mnc**, which indicate tl current mobile country code a mobile network code, respectively.
Language and region	Examples: **en** **fr** **en-rUS** **fr-rFR** **fr-rCA** **b+en** **b+en+US** **b+es+419**	The language is defined by a two-letter ISO 639-1 languag code, optionally followed by a two letter ISO 3166-1-alpha-2 region code (preceded by lowercase **r**). The codes are *not* case-sensitive; the **r** prefix is used distinguish the region portion You cannot specify a region alone. Android 7.0 (API level 24) introduced support for BCP 4 language tags, which you can to qualify language- and regio specific resources. A languag tag is composed from a sequence of one or more subtags, each of which refine narrows the range of languag identified by the overall tag. F more information about langu tags, see Tags for Identifying Languages. To use a BCP 47 language tag concatenate **b+** and a two-lett ISO 639-1 language code,

		optionally followed by additio subtags separated by +. The language tag can change during the life of your app if th users change their language i the system settings. See Handling Runtime Changes fo information about how this ca affect your app during runtim See Localization for a comple guide to localizing your app fo other languages. Also see the `getLocales()` method, which provides the defined list of locales. This lis includes the primary locale.

Layout Direction	`ldrtl`
	`ldltr`
		The layout direction of your a
		`ldrtl` means "layout-directio
		right-to-left". `ldltr` means
		"layout-direction-left-to-right
		and is the default implicit valu
		
		This can apply to any resourc
		such as layouts, drawables, o
		values.
		
		For example, if you want to
		provide some specific layout
		the Arabic language and som
		generic layout for any other
		"right-to-left" language (like
		Persian or Hebrew) then you
		would have the following:
		
		`res/`
		`layout/`
		`main.xml` (Default layout)

		`layout-ar/`
		`main.xml` (Specific layout Arabic)
		`layout-ldrtl/`
		`main.xml` (Any "right-to-l language, except for Arabic, because the "ar" language qualifier has a higher precedence)
		
		Note: To enable right-to-left layout features for your app, must set `supportsRtl` to `"true` and set `targetSdkVersion` to 1 higher.
		
		Added in API level 17.

		The fundamental size of a screen, as indicated by the shortest dimension of the available screen area. Specifically, the device's smallestWidth is the shortest the screen's available height width (you may also think of i the "smallest possible width" the screen). You can use this qualifier to ensure that, regardless of the screen's current orientation, your app' has at least `<N>` dps of width available for its UI. For example, if your layout requires that its smallest dimension of screen area be least 600 dp at all times, then you can use this qualifier to create the layout resources,

smallestWidth	**`sw<N>dp`** Examples: **`sw320dp`** **`sw600dp`** **`sw720dp`** etc.	**`res/layout-sw600dp/`**. The sys uses these resources only wh the smallest dimension of available screen is at least 600dp, regardless of whether 600dp side is the user-perce height or width. The smallest width is a fixed screen size characteristic of the device; **t device's smallest width doesn't change when the screen's orientation change** Using smallest width to determine the general screen size is useful because width i often the driving factor in designing a layout. A UI will o scroll vertically, but have fairly hard constraints on the minim space it needs horizontally. T available width is also the key factor in determining whether use a one-pane layout for handsets or multi-pane layou tablets. Thus, you likely care most about what the smallest possible width will be on each device. The smallest width of a devic takes into account screen decorations and system UI. F example, if the device has so persistent UI elements on the screen that account for space along the axis of the smallest width, the system declares th smallest width to be smaller t

the actual screen size, becau
those are screen pixels not
available for your UI.

Some values you might use h
for common screen sizes:

- 320, for devices with scr
configurations such as:
    - 240x320 ldpi (QVG
handset)
    - 320x480 mdpi
(handset)
    - 480x800 hdpi (hig
density handset)
- 480, for screens such as
480x800 mdpi
(tablet/handset).
- 600, for screens such as
600x1024 mdpi (7" table
- 720, for screens such as
720x1280 mdpi (10" tab

When your app provides mult
resource directories with
different values for the
smallestWidth qualifier, the
system uses the one closest t
(without exceeding) the devic
smallestWidth.

Added in API level 13.

Also see the
android:requiresSmallestWid
attribute, which declares the
minimum smallestWidth with
which your app is compatible
and the **smallestScreenWidthI**
configuration field, which hold

		the device's smallestWidth va For more information about designing for different screen and using this qualifier, see th Supporting Multiple Screens developer guide.
Available width	**`w<N>dp`** Examples: **`w720dp`** **`w1024dp`** etc.	Specifies a minimum available screen width, in **`dp`** units at wh the resource should be used- defined by the **`<N>`** value. This configuration value changes when the orientation changes between landscape and portr to match the current actual width. This is often useful to determ whether to use a multi-pane layout, because even on a tab device, you often won't want same multi-pane layout for portrait orientation as you do landscape. Thus, you can use this to specify the minimum w required for the layout, instea using both the screen size an orientation qualifiers together When your app provides mult resource directories with different values for this configuration, the system use the one closest to (without exceeding) the device's curre screen width. The value here takes into account screen decorations, so if the device h some persistent UI elements

		the left or right edge of the display, it uses a value for the width that is smaller than the screen size, accounting for th UI elements and reducing the app's available space. *Added in API level 13*. Also see the **`screenWidthDp`** configuration field, which hol the current screen width. For more information about designing for different screen and using this qualifier, see th Supporting Multiple Screens developer guide.
		Specifies a minimum available screen height, in "dp" units at which the resource should be used—defined by the **`<N>`** valu This configuration value chan when the orientation changes between landscape and portr to match the current actual height. Using this to define the heigh required by your layout is use in the same way as **`w<N>dp`** is f defining the required width, instead of using both the scre size and orientation qualifiers However, most apps won't ne this qualifier, considering that often scroll vertically and are more flexible with how much height is available, whereas th

Available height	**`h<N>dp`** Examples: **`h720dp`** **`h1024dp`** etc.	width is more rigid. When your app provides mult resource directories with different values for this configuration, the system use the one closest to (without exceeding) the device's curre screen height. The value here takes into account screen decorations, so if the device h some persistent UI elements the top or bottom edge of the display, it uses a value for the height that is smaller than the real screen size, accounting f these UI elements and reduci the app's available space. Scr decorations that aren't fixed (such as a phone status bar th can be hidden when full scree are *not* accounted for here, n are window decorations like th title bar or action bar, so apps must be prepared to deal with somewhat smaller space than they specify. *Added in API level 13*. Also see the **`screenHeightDp`** configuration field, which hold the current screen height. For more information about designing for different screen and using this qualifier, see th Supporting Multiple Screens developer guide.

Screen size	`small`
	`normal`
	`large`
	`xlarge`
		• **`small`**: Screens that are c similar size to a low-den QVGA screen. The minin layout size for a small screen is approximately 320x426 dp units. Exam are QVGA low-density ar VGA high density.
		• **`normal`**: Screens that are similar size to a medium density HVGA screen. Th minimum layout size for normal screen is approximately 320x470 units. Examples of such screens a WQVGA low-density, HVGA medium-density, WVGA high-den
		• **`large`**: Screens that are c similar size to a medium density VGA screen. The minimum layout size for large screen is approximately 480x640 units. Examples are VGA and WVGA medium-den screens.
		• **`xlarge`**: Screens that are considerably larger than traditional medium-dens HVGA screen. The minin layout size for an xlarge screen is approximately 720x960 dp units. In mo cases, devices with extra large screens would be t large to carry in a pocke and would most likely be tablet-style devices. *Ado*

		in API level 9. **Note:** Using a size qualifier d not imply that the resources a *only* for screens of that size. I you do not provide alternative resources with qualifiers that better match the current devi configuration, the system ma use whichever resources are best match. **Caution:** If all your resources a size qualifier that is *larger* th the current screen, the syster will **not** use them and your ap will crash at runtime (for example, if all layout resource are tagged with the `xlarge` qualifier, but the device is a normal-size screen). *Added in API level 4*. See Supporting Multiple Scre for more information. Also see the `screenLayout` configuration field, which indicates whether the screen small, normal, or large.
		• `long`: Long screens, such WQVGA, WVGA, FWVGA • `notlong`: Not long screen such as QVGA, HVGA, a VGA *Added in API level 4*.

Screen aspect	**`long`** **`notlong`**	This is based purely on the aspect ratio of the screen (a "long" screen is wider). This i related to the screen orientati Also see the **`screenLayout`** configuration field, which indicates whether the screen long.
Round screen	**`round`** **`notround`**	• **`round`**: Round screens, su as a round wearable dev • **`notround`**: Rectangular screens, such as phones tablets *Added in API level 23.* Also see the **`isScreenRound()`** configuration method, which indicates whether the screen round.
Wide Color Gamut	**`widecg`** **`nowidecg`**	• **`widecg`**: Displays with a w color gamut such as Dis P3 or AdobeRGB • **`nowidecg`**: Displays with a narrow color gamut such sRGB *Added in API level 26.* Also see the **`isScreenWideColorGamut()`** configuration method, which indicates whether the screen a wide color gamut.
		• **`highdr`**: Displays with a h dynamic range • **`lowdr`**: Displays with a

High Dynamic Range (HDR)	**`highdr`** **`lowdr`**	low/standard dynamic ra *Added in API level 26.* Also see the **`isScreenHdr()`** configuration method, which indicates whether the screen a HDR capabilities.
Screen orientation	**`port`** **`land`**	• **`port`**: Device is in portrai orientation (vertical) • **`land`**: Device is in landsc orientation (horizontal) This can change during the lif your app if the user rotates th screen. See Handling Runtime Changes for information abou how this affects your app dur runtime. Also see the **`orientation`** configuration field, which indicates the current device orientation.
		• **`car`**: Device is displaying car dock • **`desk`**: Device is displayin a desk dock • **`television`**: Device is displaying on a televisio providing a "ten foot" experience where its UI on a large screen that th user is far away from, primarily oriented aroun DPAD or other non-poin interaction • **`appliance`**: Device is ser

UI mode	**`car`** **`desk`** **`television`** **`appliance`** **`watch`** **`vrheadset`**	as an appliance, with no display • **`watch`**: Device has a disp and is worn on the wrist • **`vrheadset`**: Device is displaying in a virtual rea headset *Added in API level 8, televisio added in API 13, watch addec API 20.* For information about how yo app can respond when the de is inserted into or removed fr dock, read Determining and Monitoring the Docking State Type. This can change during the lif your app if the user places th device in a dock. You can ena or disable some of these moc using **`UiModeManager`**. See Handling Runtime Changes fc information about how this affects your app during runtin
Night mode	**`night`** **`notnight`**	• **`night`**: Night time • **`notnight`**: Day time *Added in API level 8.* This can change during the lif your app if night mode is left i auto mode (default), in which case the mode changes base the time of day. You can enab or disable this mode using **`UiModeManager`**. See Handling

		Runtime Changes for informa about how this affects your a during runtime.
		• **ldpi**: Low-density scree approximately 120dpi. • **mdpi**: Medium-density (c traditional HVGA) screer approximately 160dpi. • **hdpi**: High-density scree approximately 240dpi. • **xhdpi**: Extra-high-densit screens; approximately 320dpi. *Added in API Le 8* • **xxhdpi**: Extra-extra-high density screens; approximately 480dpi. *Added in API Level 16* • **xxxhdpi**: Extra-extra-ext high-density uses (launc icon only, see the note ir *Supporting Multiple Screens*); approximately 640dpi. *Added in API Le 18* • **nodpi**: This can be used bitmap resources that yc don't want to be scaled match the device densit • **tvdpi**: Screens somewh between mdpi and hdpi; approximately 213dpi. T isn't considered a "prim density group. It is most intended for televisions most apps shouldn't nee —providing mdpi and hd resources is sufficient fc

Screen pixel density (dpi)	**`ldpi`** **`mdpi`** **`hdpi`** **`xhdpi`** **`xxhdpi`** **`xxxhdpi`** **`nodpi`** **`tvdpi`** **`anydpi`** ***`nnn`*`dpi`**	most apps and the syste scales them as appropri *Added in API Level 13* • **`anydpi`**: This qualifier matches all screen dens and takes precedence o other qualifiers. This is useful for vector drawab *Added in API Level 21* • ***`nnn`*`dpi`**: Used to represe non-standard densities, where ***`nnn`*** is a positive integer screen density. T shouldn't be used in mo cases. Use standard der buckets, which greatly reduces the overhead of supporting the various device screen densities the market. There is a 3:4:6:8:12:16 scalir ratio between the six primary densities (ignoring the tvdpi density). So, a 9x9 bitmap in is 12x12 in mdpi, 18x18 in hdp 24x24 in xhdpi and so on. If you decide that your image resources don't look good enough on a television or othe certain devices and want to t tvdpi resources, the scaling factor is 1.33*mdpi. For exam a 100px x 100px image for mc screens should be 133px x 13 for tvdpi. **Note:** Using a density qualifie

		doesn't imply that the resourc are *only* for screens of that density. If you don't provide alternative resources with qualifiers that better match th current device configuration, system may use whichever resources are the best match See Supporting Multiple Scre for more information about hc to handle different screen densities and how Android mi scale your bitmaps to fit the current density.
Touchscreen type	**`notouch`** **`finger`**	• **`notouch`**: Device doesn't have a touchscreen. • **`finger`**: Device has a touchscreen that is inter to be used through direc interaction of the user's finger. Also see the **`touchscreen`** configuration field, which indicates the type of touchsc on the device.
		• **`keysexposed`**: Device has keyboard available. If the device has a software keyboard enabled (whic likely), this may be used even when the hardware keyboard *isn't* exposed t the user, even if the devi has no hardware keyboa If no software keyboard provided or it's disabled

		then this is only used wh a hardware keyboard is exposed. • **`keyshidden`**: Device has a hardware keyboard avail but it is hidden *and* the device does *not* have a software keyboard enab • **`keyssoft`**: Device has a software keyboard enab whether it's visible or no
Keyboard availability	**`keysexposed`** **`keyshidden`** **`keyssoft`**	If you provide **`keysexposed`** resources, but not **`keyssoft`** resources, the system uses th **`keysexposed`** resources regard of whether a keyboard is visib as long as the system has a software keyboard enabled. This can change during the lif your app if the user opens a hardware keyboard. See Hanc Runtime Changes for informa about how this affects your ap during runtime. Also see the configuration fie **`hardKeyboardHidden`** and **`keyboardHidden`**, which indicat the visibility of a hardware keyboard and the visibility of kind of keyboard (including software), respectively.
		• **`nokeys`**: Device has no hardware keys for text in • **`qwerty`**: Device has a hardware qwerty keyboa whether it's visible to the

Primary text input method	**`nokeys`** **`qwerty`** **`12key`**	user or not. • **`12key`**: Device has a hardware 12-key keyboa whether it's visible to th user or not. Also see the **`keyboard`** configuration field, which indicates the primary text inpu method available.
Navigation key availability	**`navexposed`** **`navhidden`**	• **`navexposed`**: Navigation k are available to the user. • **`navhidden`**: Navigation ke aren't available (such as behind a closed lid). This can change during the lif your app if the user reveals th navigation keys. See Handling Runtime Changes for informa about how this affects your ap during runtime. Also see the **`navigationHidde`** configuration field, which indicates whether navigation are hidden.
Primary non-touch navigation method	**`nonav`** **`dpad`** **`trackball`** **`wheel`**	• **`nonav`**: Device has no navigation facility other t using the touchscreen. • **`dpad`**: Device has a directional-pad (d-pad) navigation. • **`trackball`**: Device has a trackball for navigation. • **`wheel`**: Device has a directional wheel(s) for navigation (uncommon).

		Also see the `navigation` configuration field, which indicates the type of navigatic method available.

Platform Version (API level)	Examples:
	`v3`
	`v4`
	`v7`
	etc.
		The API level supported by th device. For example, `v1` for AI level 1 (devices with Android 1 or higher) and `v4` for API level (devices with Android 1.6 or higher). See the Android API levels document for more information about these value

Note: Some configuration qualifiers have been added since Android 1.0, so not all versions of Android support all the qualifiers. Using a new qualifier implicitly adds the platform version qualifier so that older devices are sure to ignore it. For example, using a `w600dp` qualifier automatically includes the `v13` qualifier, because the available-width qualifier was new in API level 13. To avoid any issues, always include a set of default resources (a set of resources with *no qualifiers*). For more information, see the section about Providing the Best Device Compatibility with Resources.

Qualifier name rules

Here are some rules about using configuration qualifier names:

- You can specify multiple qualifiers for a single set of

resources, separated by dashes. For example, **`drawable-en-rUS-land`** applies to US-English devices in landscape orientation.

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

After you save alternative resources into directories named with these qualifiers, Android automatically applies the resources in your app based on the current device

configuration. Each time a resource is requested, Android checks for alternative resource directories that contain the requested resource file, then finds the best-matching resource (discussed below). If there are no alternative resources that match a particular device configuration, then Android uses the corresponding default resources (the set of resources for a particular resource type that doesn't include a configuration qualifier).

Creating alias resources

When you have a resource that you'd like to use for more than one device configuration (but don't want to provide as a default resource), you don't need to put the same resource in more than one alternative resource directory. Instead, you can (in some cases) create an alternative resource that acts as an alias for a resource saved in your default resource directory.

Note: Not all resources offer a mechanism by which you can create an alias to another resource. In particular, animation, menu, raw, and other unspecified resources in the `xml/` directory don't offer this feature.

For example, imagine you have an app icon, `icon.png`, and need unique version of it for different locales. However, two locales, English-Canadian and French-Canadian, need to use the same version. You might assume that you need to copy the same image into the resource directory for both English-Canadian and French-Canadian, but it's

not true. Instead, you can save the image that's used for both as **`icon_ca.png`** (any name other than **`icon.png`**) and put it in the default **`res/drawable/`** directory. Then create an **`icon.xml`** file in **`res/drawable-en-rCA/`** and **`res/drawable-fr-rCA/`** that refers to the **`icon_ca.png`** resource using the **`<bitmap>`** element. This allows you to store just one version of the PNG file and two small XML files that point to it. (An example XML file is shown below.)

Drawable

To create an alias to an existing drawable, use the **`<drawable>`** element. For example:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <drawable name="icon">@drawable/icon_ca</drav
</resources>
```

If you save this file as **`icon.xml`** (in an alternative resource directory, such as **`res/values-en-rCA/`**), it is compiled into a resource that you can reference as **`R.drawable.icon`**, but is actually an alias for the **`R.drawable.icon_ca`** resource (which is saved in **`res/drawable/`**).

Layout

To create an alias to an existing layout, use the **`<include>`** element, wrapped in a **`<merge>`**. For example:

```
<?xml version="1.0" encoding="utf-8"?>
<merge>
    <include layout="@layout/main_ltr"/>
</merge>
```

If you save this file as **`main.xml`**, it is compiled into a resource you can reference as **`R.layout.main`**, but is actually an alias for the **`R.layout.main_ltr`** resource.

Strings and other simple values

To create an alias to an existing string, simply use the resource ID of the desired string as the value for the new string. For example:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <string name="hello">Hello</string>
    <string name="hi">@string/hello</string>
</resources>
```

The **`R.string.hi`** resource is now an alias for the **`R.string.hello`**.

Other simple values work the same way. For example, a color:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <color name="red">#f00</color>
```

```
    <color name="highlight">@color/red</color>
</resources>
```

Accessing your app resources

Once you provide a resource in your application, you can apply it by referencing its resource ID. All resource IDs are defined in your project's **`R`** class, which the **`aapt`** tool automatically generates.

When your application is compiled, **`aapt`** generates the **`R`** class, which contains resource IDs for all the resources in your **`res/`** directory. For each type of resource, there is an **`R`** subclass (for example, **`R.drawable`** for all drawable resources), and for each resource of that type, there is a static integer (for example, **`R.drawable.icon`**). This integer is the resource ID that you can use to retrieve your resource.

Although the **`R`** class is where resource IDs are specified, you should never need to look there to discover a resource ID. A resource ID is always composed of:

- The *resource type*: Each resource is grouped into a "type," such as **`string`**, **`drawable`**, and **`layout`**. For more about the different types, see Resource Types.
- The *resource name*, which is either: the filename, excluding the extension; or the value in the XML **`android:name`** attribute, if the resource is a simple

value (such as a string).

There are two ways you can access a resource:

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

  ```
  R.string.hello
  ```

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

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

  ```
  @string/hello
  ```

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

Accessing resources in code

You can use a resource in code by passing the resource ID as a method parameter. For example, you can set an `ImageView` to use the `res/drawable/myimage.png` resource

using `setImageResource()`:

Error
More

```
val imageView = findViewById(R.id.myimageview) as
imageView.setImageResource(R.drawable.myimage)
```

You can also retrieve individual resources using methods in `Resources`, which you can get an instance of with `getResources()`.

Syntax

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

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

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

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

Use cases

There are many methods that accept a resource ID parameter and you can retrieve resources using methods in `Resources`. You can get an instance of `Resources` with `Context.getResources()`.

Here are some examples of accessing resources in code:

Error
More

```
// Load a background for the current screen from
window.setBackgroundDrawableResource(R.drawable.r

// Set the Activity title by getting a string fro
//  this method requires a CharSequence rather th
window.setTitle(resources.getText(R.string.main_t

// Load a custom layout for the current screen
setContentView(R.layout.main_screen)

// Set a slide in animation by getting an Animat
flipper.setInAnimation(AnimationUtils.loadAnimat
        R.anim.hyperspace_in))

// Set the text on a TextView object using a reso
val msgTextView = findViewById(R.id.msg) as TextV
msgTextView.setText(R.string.hello_message)
```

Caution: You should never modify the `R.java` file by hand—it is generated by the `aapt` tool when your project is

compiled. Any changes are overridden next time you compile.

Accessing resources from XML

You can define values for some XML attributes and elements using a reference to an existing resource. You will often do this when creating layout files, to supply strings and images for your widgets.

For example, if you add a **`Button`** to your layout, you should use a string resource for the button text:

```
<Button
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:text="@string/submit" />
```

Syntax

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

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

- **`<package_name>`** is the name of the package in which the resource is located (not required when referencing resources from the same package)
- **`<resource_type>`** is the **`R`** subclass for the resource

type

- **`<resource_name>`** is either the resource filename without the extension or the **`android:name`** attribute value in the XML element (for simple values).

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

Use cases

In some cases you must use a resource for a value in XML (for example, to apply a drawable image to a widget), but you can also use a resource in XML any place that accepts a simple value. For example, if you have the following resource file that includes a color resource and a string resource:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
   <color name="opaque_red">#f00</color>
   <string name="hello">Hello!</string>
</resources>
```

You can use these resources in the following layout file to set the text color and text string:

```
<?xml version="1.0" encoding="utf-8"?>
<EditText xmlns:android="http://schemas.android.
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
```

```
    android:textColor="@color/opaque_red"
    android:text="@string/hello" />
```

In this case you don't need to specify the package name in the resource reference because the resources are from your own package. To reference a system resource, you would need to include the package name. For example:

```
<?xml version="1.0" encoding="utf-8"?>
<EditText xmlns:android="http://schemas.android.c
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:textColor="@android:color/secondary_t
    android:text="@string/hello" />
```

Note: You should use string resources at all times, so that your application can be localized for other languages. For information about creating alternative resources (such as localized strings), see Providing alternative resources. For a complete guide to localizing your application for other languages, see Localization.

You can even use resources in XML to create aliases. For example, you can create a drawable resource that is an alias for another drawable resource:

```
<?xml version="1.0" encoding="utf-8"?>
<bitmap xmlns:android="http://schemas.android.cor
    android:src="@drawable/other_drawable" />
```

This sounds redundant, but can be very useful when using alternative resource. Read more about Creating alias resources.

Referencing style attributes

A style attribute resource allows you to reference the value of an attribute in the currently-applied theme. Referencing a style attribute allows you to customize the look of UI elements by styling them to match standard variations supplied by the current theme, instead of supplying a hard-coded value. Referencing a style attribute essentially says, "use the style that is defined by this attribute, in the current theme."

To reference a style attribute, the name syntax is almost identical to the normal resource format, but instead of the at-symbol (`@`), use a question-mark (`?`), and the resource type portion is optional. For instance:

```
?[<package_name>:][<resource_type>/]<resource_nar
```

For example, here's how you can reference an attribute to set the text color to match the "secondary" text color of the system theme:

```
<EditText id="text"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:textColor="?android:textColorSecondar
```

```
    android:text="@string/hello_world" />
```

Here, the **`android:textColor`** attribute specifies the name of a style attribute in the current theme. Android now uses the value applied to the **`android:textColorSecondary`** style attribute as the value for **`android:textColor`** in this widget. Because the system resource tool knows that an attribute resource is expected in this context, you do not need to explicitly state the type (which would be **`?android:attr/textColorSecondary`**)—you can exclude the **`attr`** type.

Accessing original files

While uncommon, you might need access your original files and directories. If you do, then saving your files in **`res/`** won't work for you, because the only way to read a resource from **`res/`** is with the resource ID. Instead, you can save your resources in the **`assets/`** directory.

Files saved in the **`assets/`** directory are *not* given a resource ID, so you can't reference them through the **`R`** class or from XML resources. Instead, you can query files in the **`assets/`** directory like a normal file system and read raw data using **`AssetManager`**.

However, if all you require is the ability to read raw data (such as a video or audio file), then save the file in the **`res/raw/`** directory and read a stream of bytes using

`openRawResource()`.

Accessing platform resources

Android contains a number of standard resources, such as styles, themes, and layouts. To access these resource, qualify your resource reference with the `android` package name. For example, Android provides a layout resource you can use for list items in a `ListAdapter`:

Error
More

```
listAdapter = ArrayAdapter(this, android.R.layou
```

In this example, `simple_list_item_1` is a layout resource defined by the platform for items in a `ListView`. You can use this instead of creating your own layout for list items.

Providing the best device compatibility with resources

In order for your app to support multiple device configurations, it's very important that you always provide default resources for each type of resource that your app uses.

For example, if your app supports several languages, always include a `values/` directory (in which your strings are saved) *without* a language and region qualifier. If you

instead put all your string files in directories that have a language and region qualifier, then your app will crash when run on a device set to a language that your strings don't support. But, as long as you provide default **`values/`** resources, then your app will run properly (even if the user doesn't understand that language—it's better than crashing).

Likewise, if you provide different layout resources based on the screen orientation, you should pick one orientation as your default. For example, instead of providing layout resources in **`layout-land/`** for landscape and **`layout-port/`** for portrait, leave one as the default, such as **`layout/`** for landscape and **`layout-port/`** for portrait.

Providing default resources is important not only because your app might run on a configuration you hadn't anticipated, but also because new versions of Android sometimes add configuration qualifiers that older versions don't support. If you use a new resource qualifier, but maintain code compatibility with older versions of Android, then when an older version of Android runs your app, it will crash if you don't provide default resources, because it cannot use the resources named with the new qualifier. For example, if your **`minSdkVersion`** is set to 4, and you qualify all of your drawable resources using night mode (**`night`** or **`notnight`**, which were added in API Level 8), then an API level 4 device cannot access your drawable resources and will crash. In this case, you

probably want **`notnight`** to be your default resources, so you should exclude that qualifier so your drawable resources are in either **`drawable/`** or **`drawable-night/`**.

So, in order to provide the best device compatibility, always provide default resources for the resources your app needs to perform properly. Then create alternative resources for specific device configurations using the configuration qualifiers.

There is one exception to this rule: If your app's **`minSdkVersion`** is 4 or greater, you *don't* need default drawable resources when you provide alternative drawable resources with the screen density qualifier. Even without default drawable resources, Android can find the best match among the alternative screen densities and scale the bitmaps as necessary. However, for the best experience on all types of devices, you should provide alternative drawables for all three types of density.

How Android finds the best-matching resource

When you request a resource for which you provide alternatives, Android selects which alternative resource to use at runtime, depending on the current device configuration. To demonstrate how Android selects an alternative resource, assume the following drawable directories each contain different versions of the same images:

```
drawable/
drawable-en/
drawable-fr-rCA/
drawable-en-port/
drawable-en-notouch-12key/
drawable-port-ldpi/
drawable-port-notouch-12key/
```

And assume the following is the device configuration:

Locale = **`en-GB`**
Screen orientation = **`port`**
Screen pixel density = **`hdpi`**
Touchscreen type = **`notouch`**
Primary text input method = **`12key`**

By comparing the device configuration to the available alternative resources, Android selects drawables from **`drawable-en-port`**.

The system arrives at its decision for which resources to use with the following logic:

Figure 2. Flowchart of how Android finds the best-matching resource.

1. Eliminate resource files that contradict the device configuration.

   The **`drawable-fr-rCA/`** directory is eliminated, because it contradicts the **`en-GB`** locale.

```
drawable/
drawable-en/
~~drawable-fr-rCA/~~
drawable-en-port/
drawable-en-notouch-12key/
drawable-port-ldpi/
drawable-port-notouch-12key/
```

Exception: Screen pixel density is the one qualifier that is not eliminated due to a contradiction. Even though the screen density of the device is hdpi, **`drawable-port-ldpi/`** isn't eliminated because every screen density is considered to be a match at this point. More information is available in the Supporting Multiple Screens document.

2. Pick the (next) highest-precedence qualifier in the list (table 2). (Start with MCC, then move down.)
3. Do any of the resource directories include this qualifier?
    - If No, return to step 2 and look at the next qualifier. (In the example, the answer is "no" until the language qualifier is reached.)
    - If Yes, continue to step 4.
4. Eliminate resource directories that don't include this qualifier. In the example, the system eliminates all the directories that don't include a language qualifier:

```
~~drawable/~~
drawable-en/
drawable-en-port/
```

drawable-en-notouch-12key/
~~drawable-port-ldpi/~~
~~drawable-port-notouch-12key/~~

Exception: If the qualifier in question is screen pixel density, Android selects the option that most closely matches the device screen density. In general, Android prefers scaling down a larger original image to scaling up a smaller original image. See Supporting Multiple Screens.

5. Go back and repeat steps 2, 3, and 4 until only one directory remains. In the example, screen orientation is the next qualifier for which there are any matches. So, resources that don't specify a screen orientation are eliminated:

   ~~drawable-en/~~
   drawable-en-port/
   ~~drawable-en-notouch-12key/~~

   The remaining directory is **`drawable-en-port`**.

Though this procedure is executed for each resource requested, the system further optimizes some aspects. One such optimization is that once the device configuration is known, it might eliminate alternative resources that can never match. For example, if the configuration language is English ("en"), then any resource directory that has a language qualifier set to

something other than English is never included in the pool of resources checked (though a resource directory *without* the language qualifier is still included).

When selecting resources based on the screen size qualifiers, the system uses resources designed for a screen smaller than the current screen if there are no resources that better match (for example, a large-size screen uses normal-size screen resources if necessary). However, if the only available resources are *larger* than the current screen, the system **doesn't** use them and your app will crash if no other resources match the device configuration (for example, if all layout resources are tagged with the `xlarge` qualifier, but the device is a normal-size screen).

Note: The *precedence* of the qualifier (in table 2) is more important than the number of qualifiers that exactly match the device. For example, in step 4 above, the last choice on the list includes three qualifiers that exactly match the device (orientation, touchscreen type, and input method), while `drawable-en` has only one parameter that matches (language). However, language has a higher precedence than these other qualifiers, so `drawable-port-notouch-12key` is out.