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 (https://developer.android.com/tools/projects/index.html#mipmap).

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

Directory	Resource Type
`animator/`	XML files that define property animations (https://developer.android.com/guide/topics/graphics/prop-animation.html).
`anim/`	XML files that define tween animations (https://developer.android.com/guide/topics/graphics/view-animation.html#tween-animatio (Property animations can also be saved in this directory, but the `animator/` directory is pref property animations to distinguish between the two types.)
`color/`	XML files that define a state list of colors. See Color State List Resource (https://developer.android.com/guide/topics/resources/color-list-resource.html)
`drawable/`	Bitmap files (`.png`, `.9.png`, `.jpg`, `.gif`) or XML files that are compiled into the following dr resource subtypes: • Bitmap files • Nine-Patches (re-sizable bitmaps) • State lists • Shapes • Animation drawables • Other drawables See Drawable Resources (https://developer.android.com/guide/topics/resources/drawable-resource.html).
`mipmap/`	Drawable files for different launcher icon densities. For more information on managing launc with `mipmap/` folders, see Managing Projects Overview (https://developer.android.com/tools/projects/index.html#mipmap).
`layout/`	XML files that define a user interface layout. See Layout Resource (https://developer.android.com/guide/topics/resources/layout-resource.html).
`menu/`	XML files that define app menus, such as an Options Menu, Context Menu, or Sub Menu. See Resource (https://developer.android.com/guide/topics/resources/menu-resource.html).
`raw/`	Arbitrary files to save in their raw form. To open these resources with a raw `InputStream` (https://developer.android.com/reference/java/io/InputStream.html), call `Resources.openRawResource()` (https://developer.android.com/reference/android/content/res/Resources.html#openRawRe with the resource ID, which is `R.raw.`*`filename`*.

	However, if you need access to original file names and file hierarchy, you might consider savi resources in the **`assets/`** directory (instead of **`res/raw/`**). Files in **`assets/`** aren't given a r so you can read them only using [**`AssetManager`**](https://developer.android.com/reference/android/content/res/AssetManager.html) (https://developer.android.com/reference/android/content/res/AssetManager.html).
`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 t filename, files in the **`values/`** directory describe multiple resources. For a file in this director of the **`<resources>`** element defines a single resource. For example, a **`<string>`** element c **`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 and place different resource types in one file. However, for clarity, you might want to place un resource types in different files. For example, here are some filename conventions for resour create in this directory: • arrays.xml for resource arrays (typed arrays (https://developer.android.com/guide/topics/resources/more-resources.html#TypedArra • colors.xml for color values (https://developer.android.com/guide/topics/resources/more-resources.html#Color) • dimens.xml for dimension values (https://developer.android.com/guide/topics/resources/more-resources.html#Dimensior • strings.xml for string values (https://developer.android.com/guide/topics/resources/string-resource.html). • styles.xml for styles (https://developer.android.com/guide/topics/resources/style-resourc See String Resources (https://developer.android.com/guide/topics/resources/string-resourc Style Resource (https://developer.android.com/guide/topics/resources/style-resource.html), Resource Types (https://developer.android.com/guide/topics/resources/more-resources.htm
`xml/`	Arbitrary XML files that can be read at runtime by calling **`Resources.getXML()`** (https://developer.android.com/reference/android/content/res/Resources.html#getXml(int) XML configuration files must be saved here, such as a searchable configuration (https://developer.android.com/guide/topics/search/searchable-config.html).
`font/`	Font files with extensions such as **`.ttf`**, **`.otf`**, or **`.ttc`**, or XML files that include a **`<font-f`** element. For more information about fonts as resources, go to Fonts in XML (https://developer.android.com/preview/features/fonts-in-xml.html).

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 (https://developer.android.com/guide/topics/resources/available-resources.html) 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.

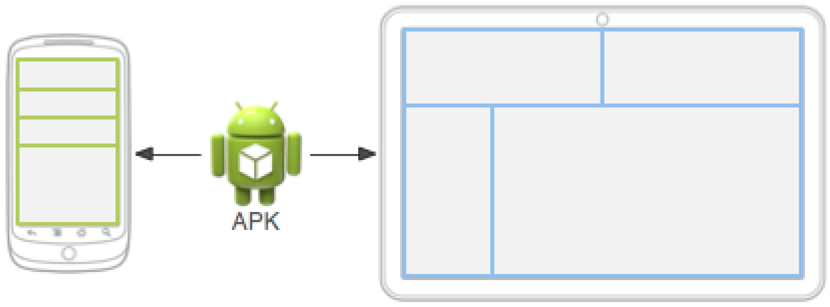

Figure 1. Two different devices, each using different layout resources.

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

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

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 (MCC), optionally followed by mobile network
		device. For example, **`mcc310`** is U.S. on any carrier, **`mcc310-mnc004`** is
		France on Orange.
		
		If the device uses a radio connection (GSM phone), the MCC and MNC v
		
		You can also use the MCC alone (for example, to include country-specif
		need to specify based on the language only, then use the *language and r*
		If you decide to use the MCC and MNC qualifier, you should do so with c
		
		Also see the configuration fields <u>**`mcc`**</u>
		(https://developer.android.com/reference/android/content/res/Configu
		(https://developer.android.com/reference/android/content/res/Configu
		current mobile country code and mobile network code, respectively.

Language and region	Examples: **`en`**	The language is defined by a two-letter <u>ISO 639-1</u> (http://www.loc.gov/s language code, optionally followed by a two letter <u>ISO 3166-1-alpha-2</u>

	`fr` `en-rUS` `fr-rFR` `fr-rCA` `b+en` `b+en+US` `b+es+419`	(https://www.iso.org/obp/ui/#iso:pub:PUB500001:en) region code (pr The codes are *not* case-sensitive; the **`r`** prefix is used to distinguish the region alone. Android 7.0 (API level 24) introduced support for BCP 47 language tags which you can use to qualify language- and region-specific resources. A sequence of one or more subtags, each of which refines or narrows the overall tag. For more information about language tags, see Tags for Ide (https://tools.ietf.org/html/rfc5646). To use a BCP 47 language tag, concatenate **b+** and a two-letter ISO 639 (http://www.loc.gov/standards/iso639-2/php/code_list.php) language subtags separated by **+**. The language tag can change during the life of your app if the users cha settings. See Handling Runtime Changes (https://developer.android.com/guide/topics/resources/runtime-chang can affect your app during runtime. See Localization (https://developer.android.com/guide/topics/resource to localizing your app for other languages. Also see the **`getLocales()`** (https://developer.android.com/reference/android/content/res/Configu which provides the defined list of locales. This list includes the primary
Layout Direction	`ldrtl` `ldltr`	The layout direction of your app. **`ldrtl`** means "layout-direction-right-tc left-to-right" and is the default implicit value. This can apply to any resource such as layouts, drawables, or values. For example, if you want to provide some specific layout for the Arabic l any other "right-to-left" language (like Persian or Hebrew) then you woul

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

★ **Note:** To enable right-to-left layout features for your app, you must set s (https://developer.android.com/guide/topics/manifest/application-eler set **`targetSdkVersion`** (https://developer.android.com/guide/topics/ to 17 or higher.

		Added in API level 17.

smallestWidth	**`sw<N>dp`**
	
	Examples:
	`sw320dp`
	`sw600dp`
	`sw720dp`
	etc.
		The fundamental size of a screen, as indicated by the shortest dimensic Specifically, the device's smallestWidth is the shortest of the screen's av think of it as the "smallest possible width" for the screen). You can use t the screen's current orientation, your app's has at least **`<N>`** dps of widtł
		
		For example, if your layout requires that its smallest dimension of scree then you can use this qualifier to create the layout resources, **`res/lay`** resources only when the smallest dimension of available screen is at le 600dp side is the user-perceived height or width. The smallest width is a device; **the device's smallest width doesn't change when the screen's o**
		
		Using smallest width to determine the general screen size is useful bec designing a layout. A UI will often scroll vertically, but have fairly hard co needs horizontally. The available width is also the key factor in determir handsets or multi-pane layout for tablets. Thus, you likely care most abo be on each device.
		
		The smallest width of a device takes into account screen decorations a has some persistent UI elements on the screen that account for space a system declares the smallest width to be smaller than the actual screer not available for your UI.
		
		Some values you might use here for common screen sizes:
		
		• 320, for devices with screen configurations such as:
		    • 240x320 ldpi (QVGA handset)
		    • 320x480 mdpi (handset)
		    • 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).
		
		When your app provides multiple resource directories with different valu system uses the one closest to (without exceeding) the device's smalle
		
		Added in API level 13.
		
		Also see the [`android:requiresSmallestWidthDp`](https://developer.android.com/guide/topics/manifest/supports-scree) (https://developer.android.com/guide/topics/manifest/supports-scree attribute, which declares the minimum smallestWidth with which your a **`smallestScreenWidthDp`** (https://developer.android.com/reference/android/content/res/Configu configuration field, which holds the device's smallestWidth value.

		For more information about designing for different screens and using th Screens (https://developer.android.com/guide/practices/screens_supp
Available width	**w<N>dp** Examples: **w720dp** **w1024dp** etc.	Specifies a minimum available screen width, in **dp** units at which the res **<N>** value. This configuration value changes when the orientation chang match the current actual width. This is often useful to determine whether to use a multi-pane layout, be won't want the same multi-pane layout for portrait orientation as you do specify the minimum width required for the layout, instead of using both qualifiers together. When your app provides multiple resource directories with different valu uses the one closest to (without exceeding) the device's current screen screen decorations, so if the device has some persistent UI elements or uses a value for the width that is smaller than the real screen size, acco reducing the app's available space. *Added in API level 13.* Also see the **`screenWidthDp`** (https://developer.android.com/reference/android/content/res/Configu configuration field, which holds the current screen width. For more information about designing for different screens and using th Screens (https://developer.android.com/guide/practices/screens_supp
Available height	**h<N>dp** Examples: **h720dp** **h1024dp** etc.	Specifies a minimum available screen height, in "dp" units at which the r **<N>** value. This configuration value changes when the orientation chang match the current actual height. Using this to define the height required by your layout is useful in the sa required width, instead of using both the screen size and orientation qu this qualifier, considering that UIs often scroll vertically and are thus mo available, whereas the width is more rigid. When your app provides multiple resource directories with different valu uses the one closest to (without exceeding) the device's current screen account screen decorations, so if the device has some persistent UI ele display, it uses a value for the height that is smaller than the real screen and reducing the app's available space. Screen decorations that aren't fi be hidden when full screen) are *not* accounted for here, nor are window bar, so apps must be prepared to deal with a somewhat smaller space t *Added in API level 13.* Also see the **`screenHeightDp`** (https://developer.android.com/reference/android/content/res/Configu configuration field, which holds the current screen width.

		For more information about designing for different screens and using th Screens (https://developer.android.com/guide/practices/screens_supp

Screen size	`small`
	`normal`
	`large`
	`xlarge`
		• `small`: Screens that are of similar size to a low-density QVGA screen
		screen is approximately 320x426 dp units. Examples are QVGA low-
		• `normal`: Screens that are of similar size to a medium-density HVGA
		normal screen is approximately 320x470 dp units. Examples of such
		medium-density, WVGA high-density.
		• `large`: Screens that are of similar size to a medium-density VGA scr
		large screen is approximately 480x640 dp units. Examples are VGA a
		• `xlarge`: Screens that are considerably larger than the traditional me
		minimum layout size for an xlarge screen is approximately 720x960
		extra-large screens would be too large to carry in a pocket and would
		Added in API level 9.
		
		Note: Using a size qualifier does not imply that the resources are *only* fo
		provide alternative resources with qualifiers that better match the curren
		use whichever resources are the best match (#BestMatch).
		
		Caution: If all your resources use a size qualifier that is *larger* than the c
		them and your app will crash at runtime (for example, if all layout resour
		qualifier, but the device is a normal-size screen).
		
		Added in API level 4.
		
		See Supporting Multiple Screens (https://developer.android.com/guide/
		more information.
		
		Also see the `screenLayout`
		(https://developer.android.com/reference/android/content/res/Configu
		field, which indicates whether the screen is small, normal, or large.

Screen aspect	`long` `notlong`	• `long`: Long screens, such as WQVGA, WVGA, FWVGA • `notlong`: Not long screens, such as QVGA, HVGA, and VGA *Added in API level 4.* This is based purely on the aspect ratio of the screen (a "long" screen is orientation. Also see the `screenLayout` (https://developer.android.com/reference/android/content/res/Configu field, which indicates whether the screen is long.
Round screen	`round` `notround`	• `round`: Round screens, such as a round wearable device • `notround`: Rectangular screens, such as phones or tablets

		Added in API level 23. Also see the **`isScreenRound()`** (https://developer.android.com/reference/android/content/res/Configu configuration method, which indicates whether the screen is round.
Wide Color Gamut	**`widecg`** **`nowidecg`**	• {@code widecg}: Displays with a wide color gamut such as Display F • {@code nowidecg}: Displays with a narrow color gamut such as sRG *Added in API level 26.* Also see the **`isScreenWideColorGamut()`** (https://developer.android.com/reference/android/content/res/Configu configuration method, which indicates whether the screen has a wide co
High Dynamic Range (HDR)	**`highdr`** **`lowdr`**	• {@code highdr}: Displays with a high-dynamic range • {@code lowdr}: Displays with a low/standard dynamic range *Added in API level 26.* Also see the **`isScreenHdr()`** (https://developer.android.com/reference/android/content/res/Configu configuration method, which indicates whether the screen has a HDR ca
Screen orientation	**`port`** **`land`**	• **`port`**: Device is in portrait orientation (vertical) • **`land`**: Device is in landscape orientation (horizontal) This can change during the life of your app if the user rotates the screen (https://developer.android.com/guide/topics/resources/runtime-chang affects your app during runtime. Also see the **`orientation`** (https://developer.android.com/reference/android/content/res/Configu field, which indicates the current device orientation.
UI mode	**`car`** **`desk`** **`television`** **`appliance`** **`watch`** **`vrheadset`**	• **`car`**: Device is displaying in a car dock • **`desk`**: Device is displaying in a desk dock • **`television`**: Device is displaying on a television, providing a "ten fo screen that the user is far away from, primarily oriented around DPAI • **`appliance`**: Device is serving as an appliance, with no display • **`watch`**: Device has a display and is worn on the wrist • **`vrheadset`**: Device is displaying in a virtual reality headset *Added in API level 8, television added in API 13, watch added in API 20.* For information about how your app can respond when the device is ins Determining and Monitoring the Docking State and Type (https://developer.android.com/training/monitoring-device-state/docki

		This can change during the life of your app if the user places the device some of these modes using **`UiModeManager`** (https://developer.android.com/reference/android/app/UiModeManag (https://developer.android.com/guide/topics/resources/runtime-chang affects your app during runtime.
Night mode	`night` `notnight`	• **`night`**: Night time • **`notnight`**: Day time *Added in API level 8.* This can change during the life of your app if night mode is left in auto r changes based on the time of day. You can enable or disable this mode (https://developer.android.com/reference/android/app/UiModeManag (https://developer.android.com/guide/topics/resources/runtime-chang affects your app during runtime.
Screen pixel density (dpi)	`ldpi` `mdpi` `hdpi` `xhdpi` `xxhdpi` `xxxhdpi` `nodpi` `tvdpi` `anydpi` *nnn*`dpi`	• **`ldpi`**: Low-density screens; approximately 120dpi. • **`mdpi`**: Medium-density (on traditional HVGA) screens; approximately • **`hdpi`**: High-density screens; approximately 240dpi. • **`xhdpi`**: Extra-high-density screens; approximately 320dpi. *Added in A* • **`xxhdpi`**: Extra-extra-high-density screens; approximately 480dpi. *Add* • **`xxxhdpi`**: Extra-extra-extra-high-density uses (launcher icon only, se (https://developer.android.com/guide/practices/screens_support.ht *Screens*); approximately 640dpi. *Added in API Level 18* • **`nodpi`**: This can be used for bitmap resources that you don't want to • **`tvdpi`**: Screens somewhere between mdpi and hdpi; approximately density group. It is mostly intended for televisions and most apps sh resources is sufficient for most apps and the system scales them as • **`anydpi`**: This qualifier matches all screen densities and takes prece for vector drawables (https://developer.android.com/training/materi *Added in API Level 21* • ***nnn*`dpi`**: Used to represent non-standard densities, where ***nnn*** is a p shouldn't be used in most cases. Use standard density buckets, whic supporting the various device screen densities on the market. There is a 3:4:6:8:12:16 scaling ratio between the six primary densities (bitmap in ldpi is 12x12 in mdpi, 18x18 in hdpi, 24x24 in xhdpi and so on If you decide that your image resources don't look good enough on a tel want to try tvdpi resources, the scaling factor is 1.33*mdpi. For example screens should be 133px x 133px for tvdpi. ★ **Note:** Using a density qualifier doesn't imply that the resources are *only* provide alternative resources with qualifiers that better match the curre

		use whichever resources are the best match (#BestMatch). See Supporting Multiple Screens (https://developer.android.com/guide/ more information about how to handle different screen densities and hc fit the current density.
Touchscreen type	`notouch` `finger`	• `notouch`: Device doesn't have a touchscreen. • `finger`: Device has a touchscreen that is intended to be used throu finger. Also see the `touchscreen` (https://developer.android.com/reference/android/content/res/Configu field, which indicates the type of touchscreen on the device.

Keyboard availability	`keysexposed`
	`keyshidden`
	`keyssoft`
		• `keysexposed`: Device has a keyboard available. If the device has a likely), this may be used even when the hardware keyboard *isn't* expc hardware keyboard. If no software keyboard is provided or it's disabl hardware keyboard is exposed.
		• `keyshidden`: Device has a hardware keyboard available but it is hid software keyboard enabled.
		• `keyssoft`: Device has a software keyboard enabled, whether it's vis
		
		If you provide `keysexposed` resources, but not `keyssoft` resources, t resources regardless of whether a keyboard is visible, as long as the sys
		
		This can change during the life of your app if the user opens a hardware Changes (https://developer.android.com/guide/topics/resources/runtin how this affects your app during runtime.
		
		Also see the configuration fields `hardKeyboardHidden` (https://developer.android.com/reference/android/content/res/Configu `keyboardHidden` (https://developer.android.com/reference/android/content/res/Configu indicate the visibility of a hardware keyboard and the visibility of any kin respectively.

Primary text input method	`nokeys`
	`qwerty`
	`12key`
		• `nokeys`: Device has no hardware keys for text input.
		• `qwerty`: Device has a hardware qwerty keyboard, whether it's visible
		• `12key`: Device has a hardware 12-key keyboard, whether it's visible t
		
		Also see the `keyboard` (https://developer.android.com/reference/android/content/res/Configu field, which indicates the primary text input method available.

Navigation key availability	`navexposed` `navhidden`	• `navexposed`: Navigation keys are available to the user. • `navhidden`: Navigation keys aren't available (such as behind a close

		This can change during the life of your app if the user reveals the naviga Changes (https://developer.android.com/guide/topics/resources/runtin how this affects your app during runtime. Also see the **navigationHidden** (https://developer.android.com/reference/android/content/res/Configu configuration field, which indicates whether navigation keys are hidden.

Primary non-touch navigation method	`nonav`
	`dpad`
	`trackball`
	`wheel`
		• `nonav`: Device has no navigation facility other than using the touchs
		• `dpad`: Device has a directional-pad (d-pad) for navigation.
		• `trackball`: Device has a trackball for navigation.
		• `wheel`: Device has a directional wheel(s) for navigation (uncommon
		
		Also see the **navigation**
		(https://developer.android.com/reference/android/content/res/Configu
		field, which indicates the type of navigation method available.

Platform Version (API level)	Examples:
	`v3`
	`v4`
	`v7`
	etc.
		The API level supported by the device. For example, `v1` for API level 1 (c
		`v4` for API level 4 (devices with Android 1.6 or higher). See the Android /
		(https://developer.android.com/guide/topics/manifest/uses-sdk-eleme
		information about these values.

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 (#Compatibility).

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 (#table2). 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-rES-rFR/`. Instead you need two resource directories, such as `drawable-rES/` and `drawable-rFR/`, 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 (#AliasResources) 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 (#BestMatch) (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</drawable>
</resources>
```

If you save this file as **`drawables.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 (https://developer.android.com/guide/topics/resources/more-resources.html) 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 (https://developer.android.com/guide/topics/resources/available-resources.html).
- 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 (#ResourcesFromCode).

- **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 (#ResourcesFromXml).

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` (https://developer.android.com/reference/android/widget/ImageView.html) to use the `res/drawable/myimage.png` resource using `setImageResource()` (https://developer.android.com/reference/android/widget/ImageView.html#setImageResource(int)):

```
ImageView imageView = (ImageView) findViewById(R.id.myimageview);
imageView.setImageResource(R.drawable.myimage);
```

You can also retrieve individual resources using methods in `Resources` (https://developer.android.com/reference/android/content/res/Resources.html), which you can get an instance of with `getResources()` (https://developer.android.com/reference/android/content/Context.html#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 (https://developer.android.com/guide/topics/resources/available-resources.html) 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` (https://developer.android.com/reference/android/content/res/Resources.html). You can get an instance of `Resources` (https://developer.android.com/reference/android/content/res/Resources.html) with `Context.getResources()` (https://developer.android.com/reference/android/content/Context.html#getResources()).

Here are some examples of accessing resources in code:

```
// Load a background for the current screen from a drawable resource
getWindow() (https://developer.android.com/reference/android/app/Activity.html#getWindow()).setE

// Set the Activity title by getting a string from the Resources object, becau
//  this method requires a CharSequence rather than a resource ID
getWindow() (https://developer.android.com/reference/android/app/Activity.html#getWindow()).setT

// Load a custom layout for the current screen
setContentView (https://developer.android.com/reference/android/app/Activity.html#setContentView

// Set a slide in animation by getting an Animation from the Resources object
mFlipper.setInAnimation (https://developer.android.com/reference/android/widget/ViewAnimator.
        R.anim.hyperspace_in));

// Set the text on a TextView object using a resource ID
TextView msgTextView = (TextView) findViewById(R.id.msg);
msgTextView.setText (https://developer.android.com/reference/android/widget/TextView.html#setT
```

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` (https://developer.android.com/reference/android/widget/Button.html) to your layout, you should use a string resource (https://developer.android.com/guide/topics/resources/string-resource.html) 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 (https://developer.android.com/guide/topics/resources/available-resources.html) 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 (https://developer.android.com/guide/topics/resources/more-resources.html#Color) and a string resource (https://developer.android.com/guide/topics/resources/string-resource.html):

```
<?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.com/apk/res/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.com/apk/res/android"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:textColor="@android:color/secondary_text_dark"
    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 (#AlternativeResources). For a complete guide to localizing your application for other languages, see Localization (https://developer.android.com/guide/topics/resources/localization.html).

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.com/apk/res/android"
    android:src="@drawable/other_drawable" />
```

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

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_name>
```

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

```
<EditText id="text"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:textColor="?android:textColorSecondary"
    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` (https://developer.android.com/reference/android/content/res/AssetManager.html).

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()` (https://developer.android.com/reference/android/content/res/Resources.html#openRawResource(int)).

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` (https://developer.android.com/reference/android/widget/ListAdapter.html):

```
setListAdapter (https://developer.android.com/reference/android/app/ListActivity.html#setL
```

In this example, `simple_list_item_1` (https://developer.android.com/reference/android/R.layout.html#simple_list_item_1) is a layout resource defined by the platform for items in a `ListView` (https://developer.android.com/reference/android/widget/ListView.html). You can use this instead of creating your own layout for list items. For more information, see the List View (https://developer.android.com/guide/topics/ui/layout/listview.html) developer guide.

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 (#LocaleQualifier). 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` (https://developer.android.com/guide/topics/manifest/uses-sdk-element.html#min) is set to 4, and you qualify all of your drawable resources using night mode (#NightQualifier) (`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` (https://developer.android.com/guide/topics/manifest/uses-sdk-element.html#min) is 4 or greater, you *don't* need default drawable resources when you provide alternative drawable resources with the screen density (#DensityQualifier) 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:

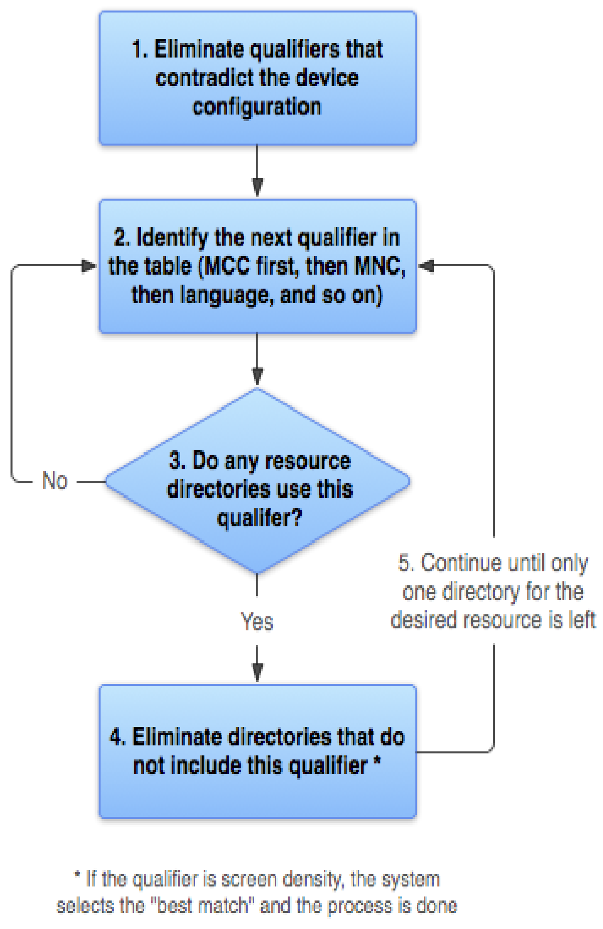
1. Eliminate qualifiers that contradict the device configuration
2. Identify the next qualifier in the table (MCC first, then MNC, then language, and so on)
3. Do any resource directories use this qualifer?
No
Yes
4. Eliminate directories that do not include this qualifier *
5. Continue until only one directory for the desired resource is left
* If the qualifier is screen density, the system selects the "best match" and the process is done

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 (https://developer.android.com/guide/practices/screens_support.html) document.

2. Pick the (next) highest-precedence qualifier in the list (table 2 (#table2)). (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 (https://developer.android.com/guide/practices/screens_support.html).

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 (#table2)) 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.