Providing Resources

You should always externalize application 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 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 application resources, you can access them using resource IDs that are generated in your project's `R` class. How to use resources in your application is discussed in Accessing Resources. This document shows you how to group your resources in your Android project and provide alternative resources for specific device configurations.

Quickview

- Different types of resources belong in different subdirectories of `res/`
- Alternative resources provide configuration-specific resource files
- Always include default resources so your app does not depend on specific device configurations

In this document

See also

Accessing Resources
Resource Types
Supporting Multiple Screens

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/
            icon.png
        layout/
            main.xml
            info.xml
        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, and a string resource file. The resource directory names are important and are described in table 1.

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.
`layout/`	XML files that define a user interface layout. See Layout Resource.
`menu/`	XML files that define application 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/` are not given a resource ID, so you can read them only using `AssetManager`.
`values/`	XML files that contain simple values, such as strings, integers, and colors. 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.

Caution: Never save resource files directly inside the `res/` directory—it will cause 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 application. 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 application 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 application.

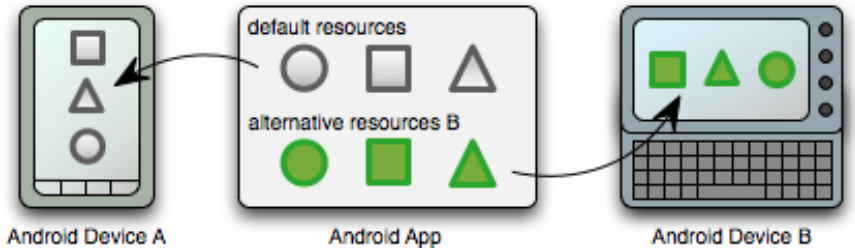

Figure 1. Two different devices, one using alternative resources.

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

1. Create a new directory in `res/` named in the form *`<resources_name>-<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.
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 alternative 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 one resource directory, they must be added to the directory name in the order they are listed in the table.

> **Note:** Some configuration qualifiers were added after Android 1.0, so not all versions of Android support all the qualifiers listed in table 2. New qualifiers indicate the version in which they were added. To avoid any issues, always include a set of default resources for resources that your application uses. For more information, see the section about Providing the Best Device Compatibility with Resources.

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 code (MNC) from the SIM card in the device. For example, `mcc310` is U.S. on any carrier, `mcc310-mnc004` is U.S. on Verizon, and `mcc208-mnc00` is France on Orange.
		
		If the device uses a radio connection (GSM phone), the MCC comes from the SIM, and the MNC comes from the network to which the device is connected.

		You can also use the MCC alone (for example, to include country-specific legal resources in your application). If you need to specify based on the language only, then use the *language and region* qualifier instead (discussed next). If you decide to use the MCC and MNC qualifier, you should do so with care and test that it works as expected. Also see the configuration fields `mcc`, and `mnc`, which indicate the current mobile country code and mobile network code, respectively.

Language and region	Examples:
	`en`
	`fr`
	`en-rUS`
	`fr-rFR`
	`fr-rCA`
	etc.
		The language is defined by a two-letter ISO 639-1 language code, optionally followed by a two letter ISO 3166-1-alpha-2 region code (preceded by lowercase "`r`").
		
		The codes are *not* case-sensitive; the `r` prefix is used to distinguish the region portion. You cannot specify a region alone.
		
		This can change during the life of your application if the user changes his or her language in the system settings. See Handling Runtime Changes for information about how this can affect your application during runtime.
		
		See Localization for a complete guide to localizing your application for other languages.
		
		Also see the `locale` configuration field, which indicates the current locale.

Screen size	`small`
	`normal`
	`large`
	`xlarge`
		`small`: Screens based on the space available on a low-density QVGA screen. Considering a portrait HVGA display, this has the same available width but less height—it is 3:4 vs. HVGA's 2:3 aspect ratio. The minimum layout size for this screen configuration is approximately 320x426 dp units. Examples are QVGA low density and VGA high density.
		`normal`: Screens based on the traditional medium-density HVGA screen. A screen is considered to be normal if it is at least this size (independent of density) and not larger. The minimum layout size for this screen configuration is approximately 320x470 dp units. Examples of such screens a WQVGA low density, HVGA medium density, WVGA high density.
		`large`: Screens based on the space available on a medium-density VGA screen. Such a screen has significantly more available space in both width and height than an HVGA display. The minimum layout size for this screen configuration is approximately 480x640 dp units. Examples are VGA and WVGA medium density screens.
		`xlarge`: Screens that are considerably larger than the traditional medium-density HVGA screen. The minimum layout size for this screen configuration is approximately 720x960 dp units. In most cases, devices with extra large screens would be too large to carry in a pocket and would most likely be tablet-style devices. *Added in API Level 9.*
		
		Added in API Level 4.
		
		See Supporting Multiple Screens for more information.
		
		Also see the `screenLayout` configuration 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 wider). This is not related to the screen orientation.

		Also see the `screenLayout` configuration field, which indicates whether the screen is long.
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 application if the user rotates the screen. See Handling Runtime Changes for information about how this affects your application during runtime. Also see the `orientation` configuration field, which indicates the current device orientation.
Dock mode	`car` `desk`	`car`: Device is in a car dock `desk`: Device is in a desk dock *Added in API Level 8.* This can change during the life of your application if the user places the device in a dock. You can enable or disable this mode using `UiModeManager`. See Handling Runtime Changes for information about how this affects your application 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 application if night mode is left in auto mode (default), in which case the mode changes based on the time of day. You can enable or disable this mode using `UiModeManager`. See Handling Runtime Changes for information about how this affects your application during runtime.

Screen pixel density (dpi)	`ldpi`
	`mdpi`
	`hdpi`
	`xhdpi`
	`nodpi`
		`ldpi`: Low-density screens; approximately 120dpi.
		`mdpi`: Medium-density (on traditional HVGA) screens; approximately 160dpi.
		`hdpi`: High-density screens; approximately 240dpi.
		`xhdpi`: Extra high-density screens; approximately 320dpi. *Added in API Level 8*
		`nodpi`: This can be used for bitmap resources that you do not want to be scaled to match the device density.
		
		Added in API Level 4.
		
		There is thus a 3:4:6:8 scaling ratio between the four densities, so a 9x9 bitmap in ldpi is 12x12 in mdpi, 18x18 in hdpi and 24x24 in xhdpi.
		
		When Android selects which resource files to use, it handles screen density differently than the other qualifiers. In step 1 of How Android finds the best matching directory (below), screen density is always considered to be a match. In step 4, if the qualifier being considered is screen density, Android selects the best final match at that point, without any need to move on to step 5.
		
		See Supporting Multiple Screens for more information about how to handle screen sizes and how Android might scale your bitmaps.

Touchscreen type	`notouch`
	`stylus`
	`finger`
		`notouch`: Device does not have a touchscreen.
		`stylus`: Device has a resistive touchscreen that's suited for use with a stylus.
		`finger`: Device has a touchscreen.

		Also see the `touchscreen` configuration 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 software keyboard enabled (which is likely), this may be used even when the hardware keyboard is *not* exposed to the user, even if the device has no hardware keyboard. If no software keyboard is provided or it's disabled, then this is only used when a hardware keyboard is exposed.
		`keyshidden`: Device has a hardware keyboard available but it is hidden *and* the device does *not* have a software keyboard enabled.
		`keyssoft`: Device has a software keyboard enabled, whether it's visible or not.
		
		If you provide `keysexposed` resources, but not `keyssoft` resources, the system uses the `keysexposed` resources regardless of whether a keyboard is visible, as long as the system has a software keyboard enabled.
		
		This can change during the life of your application if the user opens a hardware keyboard. See Handling Runtime Changes for information about how this affects your application during runtime.
		
		Also see the configuration fields `hardKeyboardHidden` and `keyboardHidden`, which indicate the visibility of a hardware keyboard and and the visibility of any kind of keyboard (including software), 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 to the user or not.
		`12key`: Device has a hardware 12-key keyboard, whether it's visible to the user or not.
		
		Also see the `keyboard` configuration 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 are not available (such as behind a closed lid). This can change during the life of your application if the user reveals the navigation keys. See Handling Runtime Changes for information about how this affects your application during runtime. Also see the `navigationHidden` 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 touchscreen.
		`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` configuration 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 (devices with Android 1.0 or higher) and `v4` for API Level 4 (devices with Android 1.6 or higher). See the Android API Levels document for more information about these values.
		
		Caution: Android 1.5 and 1.6 only match resources with this qualifier

		when it exactly matches the platform version. See the section below about Known Issues for more information.

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-rES-rFR/`. Instead you need two resource directories, such as `drawable-rES/` and `drawable-rFR/`, which contain the appropriate files. However, you are not 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 application 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 does not 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 do not want to provide as a default resource), you do not 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 do not offer this feature.

For example, imagine you have an application 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 `<bitmap>` element. For example:

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

If you save this file as `icon.xml` (in an alternative resource directory, such as `res/drawable-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="yellow">#f00</color>
    <color name="highlight">@color/red</color>
</resources>
```

Providing the Best Device Compatibility with Resources

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

For example, if your application 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 application will crash when run on a device set to a language that your strings do not support. But, as long as you provide default `values/` resources, then your application 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 application might run on a configuration you had not anticipated, but also because new versions of Android sometimes add configuration qualifiers that older versions do not 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 application, it will crash if you do not 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 application 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 application's `minSdkVersion` is 4 or greater, you *do not* 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. If your `minSdkVersion` is *less than* 4 (Android 1.5 or lower), be aware that the screen size, density, and aspect qualifiers are not supported on Android 1.5 or lower, so you might need to perform additional compatibility for these versions.

Providing screen resource compatibility for Android 1.5

Android 1.5 (and lower) does not support the following configuration qualifers:

Density
: `ldpi`, `mdpi`, `ldpi`, and `nodpi`

Screen size
: `small`, `normal`, and `large`

Screen aspect
: `long` and `notlong`

These configuration qualifiers were introduced in Android 1.6, so Android 1.5 (API Level 3) and lower does not support them. If you use these configuration qualifiers and do not provide corresponding default resources, then an Android 1.5 device might use any one of the resource directories named with the above screen configuration qualifiers, because it ignores these qualifiers and uses whichever otherwise-matching drawable resource it finds first.

For example, if your application supports Android 1.5 and includes drawable resources for each density type (`drawable-ldpi/`, `drawable-mdpi/`, and `drawable-ldpi/`), and does *not* include default drawable resources (`drawable/`), then an Android 1.5 will use drawables from any one of the alternative resource directories, which can result in a user interface that's less than ideal.

So, to provide compatibility with Android 1.5 (and lower) when using the screen configuration qualifiers:

1. Provide default resources that are for medium-density, normal, and notlong screens.

   Because all Android 1.5 devices have medium-density, normal, not-long screens, you can place these kinds of resources in the corresponding default resource directory. For example, put all medium density drawable resources in `drawable/` (instead of `drawable-mdpi/`), put `normal` size resources in the corresponding default resource directory, and `notlong` resources in the corresponding default resource directory.

2. Ensure that your SDK Tools version is r6 or greater.

   You need SDK Tools, Revision 6 (or greater), because it includes a new packaging tool that automatically applies an appropriate version qualifier to any resource directory named with a qualifier that does not exist in Android 1.0. For example, because the density qualifier was introduced in Android 1.6 (API Level 4), when the packaging tool encounters a resource directory using the density qualifier, it adds `v4` to the directory name to ensure that older versions do not use those resources (only API Level 4 and higher support that qualifier). Thus, by putting your medium-density resources in a directory *without* the `mdpi` qualifier, they are still accessible by Android 1.5, and any device that supports the density qualifer and has a medium-density screen also uses the default resources (which are mdpi) because they are the best match for the device (instead of using the `ldpi` or `hdpi` resources).

> **Note:** Later versions of Android, such as API Level 8, introduce other configuration qualifiers that older version do not support. To provide the best compatibility, you should always include a set of default resources for each type of resource that your application uses, as discussed above to provide the best device compatibility.

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`. It arrives at this decision using the following logic:

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/` is not 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 do not include this qualifier. In the example, the system eliminates all the directories that do not 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.

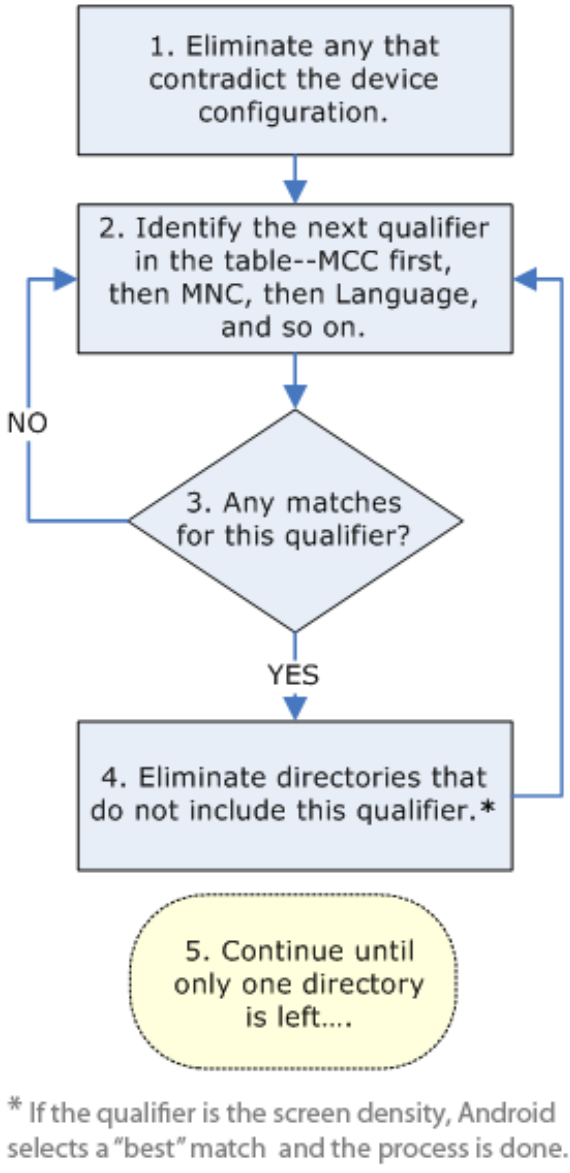

Figure 2. Flowchart of how Android finds the best-matching resource.

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 do not 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).

> **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.

To learn more about how to use resources in your application, continue to Accessing Resources.

Known Issues

Android 1.5 and 1.6: Version qualifier performs exact match, instead of best match

The correct behavior is for the system to match resources marked with a version qualifier equal to or less than the platform version on the device, but on Android 1.5 and 1.6, (API Level 3 and 4), there is a bug that causes the system to match resources marked with the version qualifier only when it exactly matches the version on the device.

The workaround: To provide version-specific resources, abide by this behavior. However, because this bug is fixed in versions of Android available after 1.6, if you need to differentiate resources between Android 1.5, 1.6, and later versions, then you only need to apply the version qualifier to the 1.6 resources and one to match all later versions. Thus, this is effectively a non-issue.

For example, if you want drawable resources that are different on each Android 1.5, 1.6, and 2.0.1 (and later), create three drawable directories: `drawable/` (for 1.5 and lower), `drawable-v4` (for 1.6), and `drawable-v6` (for 2.0.1 and later—version 2.0, v5, is no longer available).

← Back to Application Resources ↑ Go to top