Supporting Multiple Screens

Android runs on a variety of devices that offer different screen sizes and densities. For applications, the Android system provides a consistent development environment across devices and handles most of the work to adjust each application's user interface to the screen on which it is displayed. At the same time, the system provides APIs that allow you to control your application's UI for specific screen sizes and densities, in order to optimize your UI design for different screen configurations. For example, you might want a UI for tablets that's different from the UI for handsets.

Although the system performs scaling and resizing to make your application work on different screens, you should make the effort to optimize your application for different screen sizes and densities. In doing so, you maximize the user experience for all devices and your users believe that your application was actually designed for *their* devices—rather than simply stretched to fit the screen on their devices.

By following the practices described in this document, you can create an application that displays properly and provides an optimized user experience on all supported screen configurations, using a single `.apk` file.

> **Note:** The information in this document assumes that your application is designed for Android 1.6 (API Level 4) or higher. If your application supports Android 1.5 or lower, please first read Strategies for Android 1.5 (https://developer.android.com/guide/practices/screens-support-1.5.html).
>
> Also, be aware that **Android 3.2 has introduced new APIs** that allow you to more precisely control the layout resources your application uses for different screen sizes. These new features are especially important if you're developing an application that's optimized for tablets. For details, see the section about Declaring Tablet Layouts for Android 3.2 (#DeclaringTabletLayouts).

Overview of Screens Support

This section provides an overview of Android's support for multiple screens, including: an introduction to the terms and concepts used in this document and in the API, a summary of the screen configurations that the system supports, and an overview of the API and underlying screen-compatibility features.

Terms and concepts

Screen size

Actual physical size, measured as the screen's diagonal.

For simplicity, Android groups all actual screen sizes into four generalized sizes: small, normal, large, and extra-large.

Screen density

> The quantity of pixels within a physical area of the screen; usually referred to as dpi (dots per inch). For example, a "low" density screen has fewer pixels within a given physical area, compared to a "normal" or "high" density screen.
>
> For simplicity, Android groups all actual screen densities into six generalized densities: low, medium, high, extra-high, extra-extra-high, and extra-extra-extra-high.

Orientation

> The orientation of the screen from the user's point of view. This is either landscape or portrait, meaning that the screen's aspect ratio is either wide or tall, respectively. Be aware that not only do different devices operate in different orientations by default, but the orientation can change at runtime when the user rotates the device.

Resolution

> The total number of physical pixels on a screen. When adding support for multiple screens, applications do not work directly with resolution; applications should be concerned only with screen size and density, as specified by the generalized size and density groups.

Density-independent pixel (dp)

> A virtual pixel unit that you should use when defining UI layout, to express layout dimensions or position in a density-independent way.
>
> The density-independent pixel is equivalent to one physical pixel on a 160 dpi screen, which is the baseline density assumed by the system for a "medium" density screen. At runtime, the system transparently handles any scaling of the dp units, as necessary, based on the actual density of the screen in use. The conversion of dp units to screen pixels is simple: `px = dp * (dpi / 160)`. For example, on a 240 dpi screen, 1 dp equals 1.5 physical pixels. You should always use dp units when defining your application's UI, to ensure proper display of your UI on screens with different densities.

Range of screens supported

Starting with Android 1.6 (API Level 4), Android provides support for multiple screen sizes and densities, reflecting the many different screen configurations that a device may have. You can use features of the Android system to optimize your application's user interface for each screen configuration and ensure that your application not only renders properly, but provides the best user experience possible on each screen.

To simplify the way that you design your user interfaces for multiple screens, Android divides the range of actual screen sizes and densities into:

- A set of four generalized **sizes**: *small*, *normal*, *large*, and *xlarge*

> **Note:** Beginning with Android 3.2 (API level 13), these size groups are deprecated in favor of a new technique for managing screen sizes based on the available screen width. If you're developing for Android 3.2 and

> greater, see Declaring Tablet Layouts for Android 3.2 (#DeclaringTabletLayouts) for more information.

- A set of six generalized **densities**:
  - *ldpi* (low) ~120dpi
  - *mdpi* (medium) ~160dpi
  - *hdpi* (high) ~240dpi
  - *xhdpi* (extra-high) ~320dpi
  - *xxhdpi* (extra-extra-high) ~480dpi
  - *xxxhdpi* (extra-extra-extra-high) ~640dpi

The generalized sizes and densities are arranged around a baseline configuration that is a *normal* size and *mdpi* (medium) density. This baseline is based upon the screen configuration for the first Android-powered device, the T-Mobile G1, which has an HVGA screen (until Android 1.6, this was the only screen configuration that Android supported).

Each generalized size and density spans a range of actual screen sizes and densities. For example, two devices that both report a screen size of *normal* might have actual screen sizes and aspect ratios that are slightly different when measured by hand. Similarly, two devices that report a screen density of *hdpi* might have real pixel densities that are slightly different. Android makes these differences abstract to applications, so you can provide UI designed for the generalized sizes and densities and let the system handle any final adjustments as necessary. Figure 1 illustrates how different sizes and densities are roughly categorized into the different size and density groups.

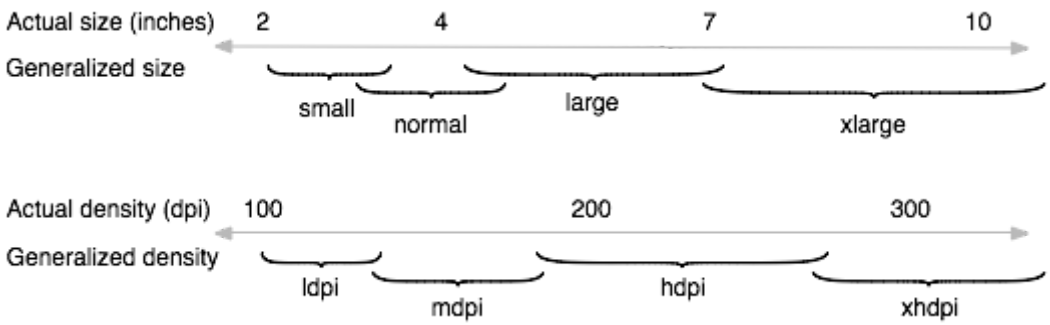

Figure 1. Illustration of how Android roughly maps actual sizes and densities to generalized sizes and densities (figures are not exact).

As you design your UI for different screen sizes, you'll discover that each design requires a minimum amount of space. So, each generalized screen size above has an associated minimum resolution that's defined by the system. These minimum sizes are in "dp" units—the same units you should use when defining your layouts—which allows the system to avoid worrying about changes in screen density.

- *xlarge* screens are at least 960dp x 720dp
- *large* screens are at least 640dp x 480dp
- *normal* screens are at least 470dp x 320dp
- *small* screens are at least 426dp x 320dp

> **Note:** These minimum screen sizes were not as well defined prior to Android 3.0, so you may encounter some devices that are mis-classified between normal and large. These are also based on the physical resolution of the screen, so may vary across devices—for example a 1024x720 tablet with a system bar actually has a bit less space available to the application due to it being used by the system bar.

To optimize your application's UI for the different screen sizes and densities, you can provide alternative resources (https://developer.android.com/guide/topics/resources/providing-resources.html#AlternativeResources) for any of the generalized sizes and densities. Typically, you should provide alternative layouts for some of the different screen sizes and alternative bitmap images for different screen densities. At runtime, the system uses the appropriate resources for your application, based on the generalized size or density of the current device screen.

You do not need to provide alternative resources for every combination of screen size and density. The system provides robust compatibility features that can handle most of the work of rendering your application on any device screen, provided that you've implemented your UI using techniques that allow it to gracefully resize (as described in the Best Practices (#screen-independence), below).

> **Note:** The characteristics that define a device's generalized screen size and density are independent from each other. For example, a WVGA high-density screen is considered a normal size screen because its physical size is about the same as the T-Mobile G1 (Android's first device and baseline screen configuration). On the other hand, a WVGA medium-density screen is considered a large size screen. Although it offers the same resolution (the same number of pixels), the WVGA medium-density screen has a lower screen density, meaning that each pixel is physically larger and, thus, the entire screen is larger than the baseline (normal size) screen.

Density independence

Your application achieves "density independence" when it preserves the physical size (from the user's point of view) of user interface elements when displayed on screens with different densities.

Maintaining density independence is important because, without it, a UI element (such as a button) appears physically larger on a low-density screen and smaller on a high-density screen. Such density-related size changes can cause problems in your application layout and usability. Figures 2 and 3 show the difference between an application when it does not provide density independence and when it does, respectively.

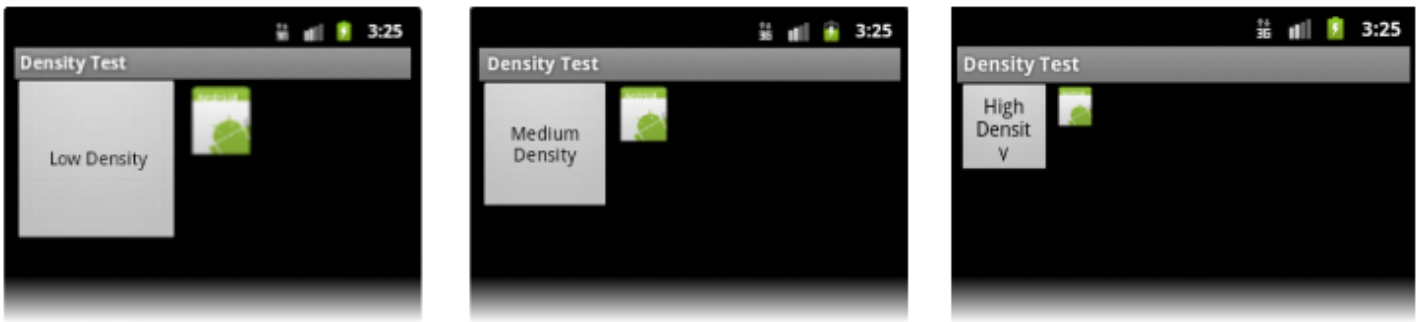

Figure 2. Example application without support for different densities, as shown on low, medium, and high-density screens.

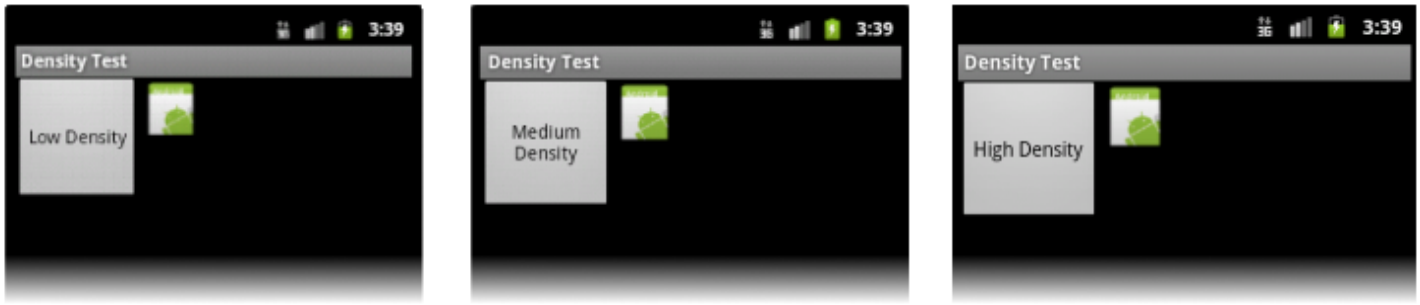

Figure 3. Example application with good support for different densities (it's density independent), as shown on low, medium, and high density screens.

The Android system helps your application achieve density independence in two ways:

- The system scales dp units as appropriate for the current screen density
- The system scales drawable resources to the appropriate size, based on the current screen density, if necessary

In figure 2, the text view and bitmap drawable have dimensions specified in pixels (`px` units), so the views are physically larger on a low-density screen and smaller on a high-density screen. This is because although the actual screen sizes may be the same, the high-density screen has more pixels per inch (the same amount of pixels fit in a smaller area). In figure 3, the layout dimensions are specified in density-independent pixels (`dp` units). Because the baseline for density-independent pixels is a medium-density screen, the device with a medium-density screen looks the same as it does in figure 2. For the low-density and high-density screens, however, the system scales the density-independent pixel values down and up, respectively, to fit the screen as appropriate.

In most cases, you can ensure density independence in your application simply by specifying all layout dimension values in density-independent pixels (`dp` units) or with `"wrap_content"`, as appropriate. The system then scales bitmap drawables as appropriate in order to display at the appropriate size, based on the appropriate scaling factor for the current screen's density.

However, bitmap scaling can result in blurry or pixelated bitmaps, which you might notice in the above screenshots. To avoid these artifacts, you should provide alternative bitmap resources for different densities. For example, you should provide higher-resolution bitmaps for high-density screens and the system will use those instead of resizing the bitmap designed for medium-density screens. The following section describes more about how to supply alternative resources for different screen configurations.

How to Support Multiple Screens

The foundation of Android's support for multiple screens is its ability to manage the rendering of an application's layout and bitmap drawables in an appropriate way for the current screen configuration. The system handles most of the work to render your application properly on each screen configuration by scaling layouts to fit the screen size/density and scaling bitmap drawables for the screen density, as appropriate. To more gracefully handle different screen configurations, however, you should also:

- **Explicitly declare in the manifest which screen sizes your application supports**

By declaring which screen sizes your application supports, you can ensure that only devices with the screens you support can download your application. Declaring support for different screen sizes can also affect how the system draws your application on larger screens—specifically, whether your application runs in screen compatibility mode (https://developer.android.com/guide/practices/screen-compat-mode.html).

To declare the screen sizes your application supports, you should include the `<supports-screens>` (https://developer.android.com/guide/topics/manifest/supports-screens-element.html) element in your manifest file.

- **Provide different layouts for different screen sizes**

  By default, Android resizes your application layout to fit the current device screen. In most cases, this works fine. In other cases, your UI might not look as good and might need adjustments for different screen sizes. For example, on a larger screen, you might want to adjust the position and size of some elements to take advantage of the additional screen space, or on a smaller screen, you might need to adjust sizes so that everything can fit on the screen.

  The configuration qualifiers you can use to provide size-specific resources are `small`, `normal`, `large`, and `xlarge`. For example, layouts for an extra-large screen should go in `layout-xlarge/`.

  Beginning with Android 3.2 (API level 13), the above size groups are deprecated and you should instead use the `sw<N>dp` configuration qualifier to define the smallest available width required by your layout resources. For example, if your multi-pane tablet layout requires at least 600dp of screen width, you should place it in `layout-sw600dp/`. Using the new techniques for declaring layout resources is discussed further in the section about Declaring Tablet Layouts for Android 3.2 (#DeclaringTabletLayouts).

- **Provide different bitmap drawables for different screen densities**

  By default, Android scales your bitmap drawables (`.png`, `.jpg`, and `.gif` files) and Nine-Patch drawables (`.9.png` files) so that they render at the appropriate physical size on each device. For example, if your application provides bitmap drawables only for the baseline, medium screen density (mdpi), then the system scales them up when on a high-density screen, and scales them down when on a low-density screen. This scaling can cause artifacts in the bitmaps. To ensure your bitmaps look their best, you should include alternative versions at different resolutions for different screen densities.

  The configuration qualifiers (#qualifiers) (described in detail below) that you can use for density-specific resources are `ldpi` (low), `mdpi` (medium), `hdpi` (high), `xhdpi` extra-high), `xxhdpi` (extra-extra-high), and `xxxhdpi` (extra-extra-extra-high). For example, bitmaps for high-density screens should go in `drawable-hdpi/`.

  > **Note:** The `mipmap-xxxhdpi` qualifier is necessary only to provide a launcher icon that can appear larger than usual on an xxhdpi device. You do not need to provide xxxhdpi assets for all your app's images.

  Some devices scale-up the launcher icon by as much as 25%. For example, if your highest density launcher icon image is already extra-extra-high-density, the scaling process will make it appear less crisp. So you should provide a higher density launcher icon in the `mipmap-xxxhdpi` directory, which the system uses instead of scaling up a smaller version of the icon.

  See Provide an xxx-high-density launcher icon (https://developer.android.com/design/style/iconography.html#xxxhdpi-launcher) for more information. You should not use the `xxxhdpi` qualifier for UI elements other than the launcher icon.

> **Note:** Place all your launcher icons in the `res/mipmap-[density]/` folders, rather than the `res/drawable-[density]/` folders. The Android system retains the resources in these density-specific folders, such as mipmap-xxxhdpi, regardless of the screen resolution of the device where your app is installed. This behavior allows launcher apps to pick the best resolution icon for your app to display on the home screen. For more information about using the mipmap folders, see Managing Projects Overview (https://developer.android.com/tools/projects/index.html#mipmap).

The size and density configuration qualifiers correspond to the generalized sizes and densities described in Range of screens supported (#range), above.

> **Note:** If you're not familiar with configuration qualifiers and how the system uses them to apply alternative resources, read Providing Alternative Resources (https://developer.android.com/guide/topics/resources/providing-resources.html#AlternativeResources) for more information.

At runtime, the system ensures the best possible display on the current screen with the following procedure for any given resource:

1. The system uses the appropriate alternative resource

   Based on the size and density of the current screen, the system uses any size- and density-specific resource provided in your application. For example, if the device has a high-density screen and the application requests a drawable resource, the system looks for a drawable resource directory that best matches the device configuration. Depending on the other alternative resources available, a resource directory with the `hdpi` qualifier (such as `drawable-hdpi/`) might be the best match, so the system uses the drawable resource from this directory.

2. If no matching resource is available, the system uses the default resource and scales it up or down as needed to match the current screen size and density

   The "default" resources are those that are not tagged with a configuration qualifier. For example, the resources in `drawable/` are the default drawable resources. The system assumes that default resources are designed for the baseline screen size and density, which is a normal screen size and a medium-density. As such, the system scales default density resources up for high-density screens and down for low-density screens, as appropriate.

   However, when the system is looking for a density-specific resource and does not find it in the density-specific directory, it won't always use the default resources. The system may instead use one of the other density-specific resources in order to provide better results when scaling. For example, when looking for a low-density resource and it is not available, the system prefers to scale-down the high-density version of the resource, because the system can easily scale a high-density resource down to low-density by a factor of 0.5, with fewer artifacts, compared to scaling a medium-density resource by a factor of 0.75.

For more information about how Android selects alternative resources by matching configuration qualifiers to the device configuration, read How Android Finds the Best-matching Resource (https://developer.android.com/guide/topics/resources/providing-resources.html#BestMatch).

Using configuration qualifiers

Android supports several configuration qualifiers that allow you to control how the system selects your alternative resources based on the characteristics of the current device screen. A configuration qualifier is a string that you can append to a resource directory in your Android project and specifies the configuration for which the resources inside are designed.

To use a configuration qualifier:

1. Create a new directory in your project's `res/` directory and name it using the format: `<resources_name>-<qualifier>`
   - `<resources_name>` is the standard resource name (such as `drawable` or `layout`).
   - `<qualifier>` is a configuration qualifier from table 1, below, specifying the screen configuration for which these resources are to be used (such as `hdpi` or `xlarge`).

   You can use more than one `<qualifier>` at a time—simply separate each qualifier with a dash.
2. Save the appropriate configuration-specific resources in this new directory. The resource files must be named exactly the same as the default resource files.

For example, `xlarge` is a configuration qualifier for extra-large screens. When you append this string to a resource directory name (such as `layout-xlarge`), it indicates to the system that these resources are to be used on devices that have an extra-large screen.

Table 1. Configuration qualifiers that allow you to provide special resources for different screen configurations.

Screen characteristic	Qualifier	Description
Size	`small`	Resources for *small* size screens.
	`normal`	Resources for *normal* size screens. (This is the baseline size.)
	`large`	Resources for *large* size screens.
	`xlarge`	Resources for *extra-large* size screens.
Density	`ldpi`	Resources for low-density (*ldpi*) screens (~120dpi).
	`mdpi`	Resources for medium-density (*mdpi*) screens (~160dpi). (This is the baseline density.)
	`hdpi`	Resources for high-density (*hdpi*) screens (~240dpi).
	`xhdpi`	Resources for extra-high-density (*xhdpi*) screens (~320dpi).
	`xxhdpi`	Resources for extra-extra-high-density (*xxhdpi*) screens (~480dpi).
	`xxxhdpi`	Resources for extra-extra-extra-high-density (*xxxhdpi*) uses (~640dpi). Use this for the launcher icon only, see note (#xxxhdpi-note) above.
	`nodpi`	Resources for all densities. These are density-independent resources. The system does not scale resources tagged with this qualifier, regardless of the current screen's density.

	`tvdpi`	Resources for screens somewhere between mdpi and hdpi; approximately 213dpi. This is not considered a "primary" density group. It is mostly intended for televisions and most apps shouldn't need it—providing mdpi and hdpi resources is sufficient for most apps and the system will scale them as appropriate. If you find it necessary to provide tvdpi resources, you should size them at a factor of 1.33*mdpi. For example, a 100px x 100px image for mdpi screens should be 133px x 133px for tvdpi.
Orientation	`land`	Resources for screens in the landscape orientation (wide aspect ratio).
	`port`	Resources for screens in the portrait orientation (tall aspect ratio).
Aspect ratio	`long`	Resources for screens that have a significantly taller or wider aspect ratio (when in portrait or landscape orientation, respectively) than the baseline screen configuration.
	`notlong`	Resources for use screens that have an aspect ratio that is similar to the baseline screen configuration.

> **Note:** If you're developing your application for Android 3.2 and higher, see the section about Declaring Tablet Layouts for Android 3.2 (#DeclaringTabletLayouts) for information about new configuration qualifiers that you should use when declaring layout resources for specific screen sizes (instead of using the size qualifiers in table 1).

For more information about how these qualifiers roughly correspond to real screen sizes and densities, see Range of Screens Supported (#range), earlier in this document.

For example, the following application resource directories provide different layout designs for different screen sizes and different drawables. Use the `mipmap/` folders for launcher icons.

```
res/layout/my_layout.xml               // layout for normal screen size ("default")
res/layout-large/my_layout.xml         // layout for large screen size
res/layout-xlarge/my_layout.xml        // layout for extra-large screen size
res/layout-xlarge-land/my_layout.xml   // layout for extra-large in landscape orientation

res/drawable-mdpi/graphic.png          // bitmap for medium-density
res/drawable-hdpi/graphic.png          // bitmap for high-density
res/drawable-xhdpi/graphic.png         // bitmap for extra-high-density
res/drawable-xxhdpi/graphic.png        // bitmap for extra-extra-high-density

res/mipmap-mdpi/my_icon.png         // launcher icon for medium-density
res/mipmap-hdpi/my_icon.png         // launcher icon for high-density
res/mipmap-xhdpi/my_icon.png        // launcher icon for extra-high-density
res/mipmap-xxhdpi/my_icon.png       // launcher icon for extra-extra-high-density
res/mipmap-xxxhdpi/my_icon.png      // launcher icon for extra-extra-extra-high-density
```

For more information about how to use alternative resources and a complete list of configuration qualifiers (not just for screen configurations), see Providing Alternative Resources (https://developer.android.com/guide/topics/resources/providing-resources.html#AlternativeResources).

Be aware that, when the Android system picks which resources to use at runtime, it uses certain logic to determine the "best matching" resources. That is, the qualifiers you use don't have to exactly match the current screen

configuration in all cases in order for the system to use them. Specifically, when selecting resources based on the size qualifiers, the system will use resources designed for a screen smaller than the current screen if there are no resources that better match (for example, a large-size screen will use normal-size screen resources if necessary). However, if the only available resources are *larger* than the current screen, the system will not use them and your application 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). For more information about how the system selects resources, read How Android Finds the Best-matching Resource (https://developer.android.com/guide/topics/resources/providing-resources.html#BestMatch).

> **Tip:** If you have some drawable resources that the system should never scale (perhaps because you perform some adjustments to the image yourself at runtime), you should place them in a directory with the `nodpi` configuration qualifier. Resources with this qualifier are considered density-agnostic and the system will not scale them.

Designing alternative layouts and drawables

The types of alternative resources you should create depends on your application's needs. Usually, you should use the size and orientation qualifiers to provide alternative layout resources and use the density qualifiers to provide alternative bitmap drawable resources.

The following sections summarize how you might want to use the size and density qualifiers to provide alternative layouts and drawables, respectively.

Alternative layouts

Generally, you'll know whether you need alternative layouts for different screen sizes once you test your application on different screen configurations. For example:

- When testing on a small screen, you might discover that your layout doesn't quite fit on the screen. For example, a row of buttons might not fit within the width of the screen on a small screen device. In this case you should provide an alternative layout for small screens that adjusts the size or position of the buttons.
- When testing on an extra-large screen, you might realize that your layout doesn't make efficient use of the big screen and is obviously stretched to fill it. In this case, you should provide an alternative layout for extra-large screens that provides a redesigned UI that is optimized for bigger screens such as tablets.

  Although your application should work fine without an alternative layout on big screens, it's quite important to users that your application looks as though it's designed specifically for their devices. If the UI is obviously stretched, users are more likely to be unsatisfied with the application experience.
- And, when testing in the landscape orientation compared to the portrait orientation, you might notice that UI elements placed at the bottom of the screen for the portrait orientation should instead be on the right side of the screen in landscape orientation.

To summarize, you should be sure that your application layout:

- Fits on small screens (so users can actually use your application)
- Is optimized for bigger screens to take advantage of the additional screen space

- Is optimized for both landscape and portrait orientations

If your UI uses bitmaps that need to fit the size of a view even after the system scales the layout (such as the background image for a button), you should use Nine-Patch (https://developer.android.com/guide/topics/graphics/2d-graphics.html#nine-patch) bitmap files. A Nine-Patch file is basically a PNG file in which you specify two-dimensional regions that are stretchable. When the system needs to scale the view in which the bitmap is used, the system stretches the Nine-Patch bitmap, but stretches only the specified regions. As such, you don't need to provide different drawables for different screen sizes, because the Nine-Patch bitmap can adjust to any size. You should, however, provide alternate versions of your Nine-Patch files for different screen densities.

Alternative drawables

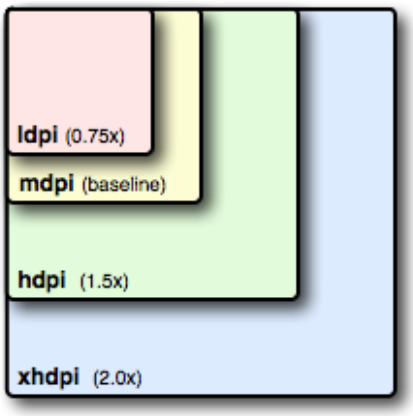

Figure 4. Relative sizes for bitmap drawables that support each density.

Almost every application should have alternative drawable resources for different screen densities, because almost every application has a launcher icon and that icon should look good on all screen densities. Likewise, if you include other bitmap drawables in your application (such as for menu icons or other graphics in your application), you should provide alternative versions or each one, for different densities.

> **Note:** You only need to provide density-specific drawables for bitmap files (`.png`, `.jpg`, or `.gif`) and Nine-Patch files (`.9.png`). If you use XML files to define shapes, colors, or other drawable resources (https://developer.android.com/guide/topics/resources/drawable-resource.html), you should put one copy in the default drawable directory (`drawable/`).

To create alternative bitmap drawables for different densities, you should follow the **3:4:6:8:12:16 scaling ratio** between the six generalized densities. For example, if you have a bitmap drawable that's 48x48 pixels for medium-density screens, all the different sizes should be:

- 36x36 (0.75x) for low-density
- 48x48 (1.0x baseline) for medium-density
- 72x72 (1.5x) for high-density
- 96x96 (2.0x) for extra-high-density
- 144x144 (3.0x) for extra-extra-high-density

- 192x192 (4.0x) for extra-extra-extra-high-density (launcher icon only; see note (#xxxhdpi-note) above)

For more information about designing icons, see the Icon Design Guidelines (https://developer.android.com/guide/practices/ui_guidelines/icon_design.html), which includes size information for various bitmap drawables, such as launcher icons, menu icons, status bar icons, tab icons, and more.

Declaring Tablet Layouts for Android 3.2

For the first generation of tablets running Android 3.0, the proper way to declare tablet layouts was to put them in a directory with the `xlarge` configuration qualifier (for example, `res/layout-xlarge/`). In order to accommodate other types of tablets and screen sizes—in particular, 7" tablets—Android 3.2 introduces a new way to specify resources for more discrete screen sizes. The new technique is based on the amount of space your layout needs (such as 600dp of width), rather than trying to make your layout fit the generalized size groups (such as *large* or *xlarge*).

The reason designing for 7" tablets is tricky when using the generalized size groups is that a 7" tablet is technically in the same group as a 5" handset (the *large* group). While these two devices are seemingly close to each other in size, the amount of space for an application's UI is significantly different, as is the style of user interaction. Thus, a 7" and 5" screen should not always use the same layout. To make it possible for you to provide different layouts for these two kinds of screens, Android now allows you to specify your layout resources based on the width and/or height that's actually available for your application's layout, specified in dp units.

For example, after you've designed the layout you want to use for tablet-style devices, you might determine that the layout stops working well when the screen is less than 600dp wide. This threshold thus becomes the minimum size that you require for your tablet layout. As such, you can now specify that these layout resources should be used only when there is at least 600dp of width available for your application's UI.

You should either pick a width and design to it as your minimum size, or test what is the smallest width your layout supports once it's complete.

> **Note:** Remember that all the figures used with these new size APIs are density-independent pixel (dp) values and your layout dimensions should also always be defined using dp units, because what you care about is the amount of screen space available after the system accounts for screen density (as opposed to using raw pixel resolution). For more information about density-independent pixels, read Terms and concepts (#terms), earlier in this document.

Using new size qualifiers

The different resource configurations that you can specify based on the space available for your layout are summarized in table 2. These new qualifiers offer you more control over the specific screen sizes your application supports, compared to the traditional screen size groups (small, normal, large, and xlarge).

> **Note:** The sizes that you specify using these qualifiers are **not the actual screen sizes**. Rather, the sizes are for the width or height in dp units that are **available to your activity's window**. The Android system might use some

> of the screen for system UI (such as the system bar at the bottom of the screen or the status bar at the top), so some of the screen might not be available for your layout. Thus, the sizes you declare should be specifically about the sizes needed by your activity—the system accounts for any space used by system UI when declaring how much space it provides for your layout. Also beware that the Action Bar (https://developer.android.com/guide/topics/ui/actionbar.html) is considered a part of your application's window space, although your layout does not declare it, so it reduces the space available for your layout and you must account for it in your design.

Table 2. New configuration qualifiers for screen size (introduced in Android 3.2).

Screen configuration	Qualifier values	Description

smallestWidth	`sw<N>dp`
	
	Examples:
	`sw600dp`
	`sw720dp`
		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 of the screen's available height and width (you may also think of it as the "smallest possible width" for the screen). You can use this qualifier to ensure that, regardless of the screen's current orientation, your application's 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 at least 600 dp at all times, then you can use this qualifier to create the layout resources, `res/layout-sw600dp/`. The system will use these resources only when the smallest dimension of available screen is at least 600dp, regardless of whether the 600dp side is the user-perceived height or width. The smallestWidth is a fixed screen size characteristic of the device; **the device's smallestWidth does not change when the screen's orientation changes**.
		
		The smallestWidth of a device takes into account screen decorations and system UI. For example, if the device has some persistent UI elements on the screen that account for space along the axis of the smallestWidth, the system declares the smallestWidth to be smaller than the actual screen size, because those are screen pixels not available for your UI.
		
		This is an alternative to the generalized screen size qualifiers (small, normal, large, xlarge) that allows you to define a discrete number for the effective size available for your UI. Using smallestWidth to determine the general screen size is useful because width is often the driving factor in designing a layout. A UI will often scroll vertically, but have fairly hard constraints on the minimum space it needs horizontally. The available width is also the key factor in determining whether to use a one-pane layout for handsets or multi-pane layout for tablets. Thus, you likely care most about what the smallest possible width will be on each device.

Available screen width	`w<N>dp`	Specifies a minimum available width in dp units at which the resources should be used—defined by the `<N>` value. The system's corresponding value for the

	Examples:
	`w720dp`
	`w1024dp`
		width changes when the screen's orientation switches between landscape and portrait to reflect the current actual width that's available for your UI.
		
		This is often useful to determine whether to use a multi-pane layout, because even on a tablet device, you often won't want the same multi-pane layout for portrait orientation as you do for landscape. Thus, you can use this to specify the minimum width required for the layout, instead of using both the screen size and orientation qualifiers together.

Available screen height	`h<N>dp`
	
	Examples:
	`h720dp`
	`h1024dp`
	etc.
		Specifies a minimum screen height in dp units at which the resources should be used—defined by the `<N>` value. The system's corresponding value for the height changes when the screen's orientation switches between landscape and portrait to reflect the current actual height that's available for your UI.
		
		Using this to define the height required by your layout is useful in the same way as `w<N>dp` is for defining the required width, instead of using both the screen size and orientation qualifiers. However, most apps won't need this qualifier, considering that UIs often scroll vertically and are thus more flexible with how much height is available, whereas the width is more rigid.

While using these qualifiers might seem more complicated than using screen size groups, it should actually be simpler once you determine the requirements for your UI. When you design your UI, the main thing you probably care about is the actual size at which your application switches between a handset-style UI and a tablet-style UI that uses multiple panes. The exact point of this switch will depend on your particular design—maybe you need a 720dp width for your tablet layout, maybe 600dp is enough, or 480dp, or some number between these. Using these qualifiers in table 2, you are in control of the precise size at which your layout changes.

For more discussion about these size configuration qualifiers, see the Providing Resources (https://developer.android.com/guide/topics/resources/providing-resources.html#SmallestScreenWidthQualifier) document.

Configuration examples

To help you target some of your designs for different types of devices, here are some numbers for typical screen widths:

- 320dp: a typical phone screen (240x320 ldpi, 320x480 mdpi, 480x800 hdpi, etc).
- 480dp: a tweener tablet like the Streak (480x800 mdpi).
- 600dp: a 7” tablet (600x1024 mdpi).
- 720dp: a 10” tablet (720x1280 mdpi, 800x1280 mdpi, etc).

Using the size qualifiers from table 2, your application can switch between your different layout resources for handsets and tablets using any number you want for width and/or height. For example, if 600dp is the smallest available width supported by your tablet layout, you can provide these two sets of layouts:

```
res/layout/main_activity.xml           # For handsets
res/layout-sw600dp/main_activity.xml   # For tablets
```

In this case, the smallest width of the available screen space must be 600dp in order for the tablet layout to be applied.

For other cases in which you want to further customize your UI to differentiate between sizes such as 7” and 10” tablets, you can define additional smallest width layouts:

```
res/layout/main_activity.xml           # For handsets (smaller than 600dp available width)
res/layout-sw600dp/main_activity.xml   # For 7” tablets (600dp wide and bigger)
res/layout-sw720dp/main_activity.xml   # For 10” tablets (720dp wide and bigger)
```

Notice that the previous two sets of example resources use the "smallest width" qualifier, `sw<N>dp`, which specifies the smallest of the screen's two sides, regardless of the device's current orientation. Thus, using `sw<N>dp` is a simple way to specify the overall screen size available for your layout by ignoring the screen's orientation.

However, in some cases, what might be important for your layout is exactly how much width or height is *currently* available. For example, if you have a two-pane layout with two fragments side by side, you might want to use it whenever the screen provides at least 600dp of width, whether the device is in landscape or portrait orientation. In this case, your resources might look like this:

```
res/layout/main_activity.xml         # For handsets (smaller than 600dp available width)
res/layout-w600dp/main_activity.xml  # Multi-pane (any screen with 600dp available width or
```

Notice that the second set is using the "available width" qualifier, `w<N>dp`. This way, one device may actually use both layouts, depending on the orientation of the screen (if the available width is at least 600dp in one orientation and less than 600dp in the other orientation).

If the available height is a concern for you, then you can do the same using the `h<N>dp` qualifier. Or, even combine the `w<N>dp` and `h<N>dp` qualifiers if you need to be really specific.

Declaring screen size support

Once you've implemented your layouts for different screen sizes, it's equally important that you declare in your manifest file which screens your application supports.

Along with the new configuration qualifiers for screen size, Android 3.2 introduces new attributes for the <supports-screens> (https://developer.android.com/guide/topics/manifest/supports-screens-element.html) manifest element:

`android:requiresSmallestWidthDp` (https://developer.android.com/guide/topics/manifest/supports-screens-element.html#requiresSmallest)

> Specifies the minimum smallestWidth required. The smallestWidth is the shortest dimension of the screen space (in `dp` units) that must be available to your application UI—that is, the shortest of the available screen's two dimensions. So, in order for a device to be considered compatible with your application, the device's smallestWidth must be equal to or greater than this value. (Usually, the value you supply for this is the "smallest width" that your layout supports, regardless of the screen's current orientation.)
>
> For example, if your application is only for tablet-style devices with a 600dp smallest available width:
>
> ```
> <manifest ... >
>     <supports-screens android:requiresSmallestWidthDp="600" />
> ```

```
    ...
</manifest>
```

However, if your application supports all screen sizes supported by Android (as small as 426dp x 320dp), then you don't need to declare this attribute, because the smallest width your application requires is the smallest possible on any device.

> **Caution:** The Android system does not pay attention to this attribute, so it does not affect how your application behaves at runtime. Instead, it is used to enable filtering for your application on services such as Google Play. However, **Google Play currently does not support this attribute for filtering** (on Android 3.2), so you should continue using the other size attributes if your application does not support small screens.

`android:compatibleWidthLimitDp` (https://developer.android.com/guide/topics/manifest/supports-screens-element.html#compatibleWidth)

This attribute allows you to enable screen compatibility mode (https://developer.android.com/guide/practices/screen-compat-mode.html) as a user-optional feature by specifying the maximum "smallest width" that your application supports. If the smallest side of a device's available screen is greater than your value here, users can still install your application, but are offered to run it in screen compatibility mode. By default, screen compatibility mode is disabled and your layout is resized to fit the screen as usual, but a button is available in the system bar that allows users to toggle screen compatibility mode on and off.

> **Note:** If your application's layout properly resizes for large screens, you do not need to use this attribute. We recommend that you avoid using this attribute and instead ensure your layout resizes for larger screens by following the recommendations in this document.

`android:largestWidthLimitDp` (https://developer.android.com/guide/topics/manifest/supports-screens-element.html#largestWidth)

This attribute allows you to force-enable screen compatibility mode (https://developer.android.com/guide/practices/screen-compat-mode.html) by specifying the maximum "smallest width" that your application supports. If the smallest side of a device's available screen is greater than your value here, the application runs in screen compatibility mode with no way for the user to disable it.

> **Note:** If your application's layout properly resizes for large screens, you do not need to use this attribute. We recommend that you avoid using this attribute and instead ensure your layout resizes for larger screens by following the recommendations in this document.

> **Caution:** When developing for Android 3.2 and higher, you should not use the older screen size attributes in combination with the attributes listed above. Using both the new attributes and the older size attributes might cause unexpected behavior.

For more information about each of these attributes, follow the respective links above.

Best Practices

The objective of supporting multiple screens is to create an application that can function properly and look good on any of the generalized screen configurations supported by Android. The previous sections of this document provide information about how Android adapts your application to screen configurations and how you can customize the look of your application on different screen configurations. This section provides some additional tips and an overview of techniques that help ensure that your application scales properly for different screen configurations.

Here is a quick checklist about how you can ensure that your application displays properly on different screens:

1. Use `wrap_content`, `match_parent`, or `dp` units when specifying dimensions in an XML layout file
2. Do not use hard coded pixel values in your application code
3. Do not use `AbsoluteLayout` (it's deprecated)
4. Supply alternative bitmap drawables for different screen densities

The following sections provide more details.

1. Use wrap_content, match_parent, or the dp unit for layout dimensions

When defining the `android:layout_width` (https://developer.android.com/reference/android/view/ViewGroup.LayoutParams.html#attr_android:layout_width) and `android:layout_height` (https://developer.android.com/reference/android/view/ViewGroup.LayoutParams.html#attr_android:layout_height) for views in an XML layout file, using `"wrap_content"`, `"match_parent"` or `dp` units guarantees that the view is given an appropriate size on the current device screen.

For instance, a view with a `layout_width="100dp"` measures 100 pixels wide on medium-density screen and the system scales it up to 150 pixels wide on high-density screen, so that the view occupies approximately the same physical space on the screen.

Similarly, you should prefer the `sp` (scale-independent pixel) to define text sizes. The `sp` scale factor depends on a user setting and the system scales the size the same as it does for `dp`.

2. Do not use hard-coded pixel values in your application code

For performance reasons and to keep the code simpler, the Android system uses pixels as the standard unit for expressing dimension or coordinate values. That means that the dimensions of a view are always expressed in the code using pixels, but always based on the current screen density. For instance, if `myView.getWidth()` returns 10, the view is 10 pixels wide on the current screen, but on a device with a higher density screen, the value returned

might be 15. If you use pixel values in your application code to work with bitmaps that are not pre-scaled for the current screen density, you might need to scale the pixel values that you use in your code to match the un-scaled bitmap source.

If your application manipulates bitmaps or deals with pixel values at runtime, see the section below about Additional Density Considerations (#DensityConsiderations).

3. Do not use AbsoluteLayout

Unlike the other layouts widgets, `AbsoluteLayout` (https://developer.android.com/reference/android/widget/AbsoluteLayout.html) enforces the use of fixed positions to lay out its child views, which can easily lead to user interfaces that do not work well on different displays. Because of this, `AbsoluteLayout` (https://developer.android.com/reference/android/widget/AbsoluteLayout.html) was deprecated in Android 1.5 (API Level 3).

You should instead use `RelativeLayout` (https://developer.android.com/reference/android/widget/RelativeLayout.html), which uses relative positioning to lay out its child views. For instance, you can specify that a button widget should appear "to the right of" a text widget.

4. Use size and density-specific resources

Although the system scales your layout and drawable resources based on the current screen configuration, you may want to make adjustments to the UI on different screen sizes and provide bitmap drawables that are optimized for different densities. This essentially reiterates the information from earlier in this document.

If you need to control exactly how your application will look on various screen configurations, adjust your layouts and bitmap drawables in configuration-specific resource directories. For example, consider an icon that you want to display on medium and high-density screens. Simply create your icon at two different sizes (for instance 100x100 for medium-density and 150x150 for high-density) and put the two variations in the appropriate directories, using the proper qualifiers:

```
res/drawable-mdpi/icon.png   //for medium-density screens
res/drawable-hdpi/icon.png   //for high-density screens
```

> **Note:** If a density qualifier is not defined in a directory name, the system assumes that the resources in that directory are designed for the baseline medium density and will scale for other densities as appropriate.

For more information about valid configuration qualifiers, see Using configuration qualifiers (#qualifiers), earlier in this document.

Additional Density Considerations

This section describes more about how Android performs scaling for bitmap drawables on different screen densities and how you can further control how bitmaps are drawn on different densities. The information in this

section shouldn't be important to most applications, unless you have encountered problems in your application when running on different screen densities or your application manipulates graphics.

To better understand how you can support multiple densities when manipulating graphics at runtime, you should understand that the system helps ensure the proper scale for bitmaps in the following ways:

1. *Pre-scaling of resources (such as bitmap drawables)*

   Based on the density of the current screen, the system uses any size- or density-specific resources from your application and displays them without scaling. If resources are not available in the correct density, the system loads the default resources and scales them up or down as needed to match the current screen's density. The system assumes that default resources (those from a directory without configuration qualifiers) are designed for the baseline screen density (mdpi), unless they are loaded from a density-specific resource directory. Pre-scaling is, thus, what the system does when resizing a bitmap to the appropriate size for the current screen density.

   If you request the dimensions of a pre-scaled resource, the system returns values representing the dimensions *after* scaling. For example, a bitmap designed at 50x50 pixels for an mdpi screen is scaled to 75x75 pixels on an hdpi screen (if there is no alternative resource for hdpi) and the system reports the size as such.

   There are some situations in which you might not want Android to pre-scale a resource. The easiest way to avoid pre-scaling is to put the resource in a resource directory with the `nodpi` configuration qualifier. For example:

   ```
   res/drawable-nodpi/icon.png
   ```

   When the system uses the `icon.png` bitmap from this folder, it does not scale it based on the current device density.

2. *Auto-scaling of pixel dimensions and coordinates*

   An application can disable pre-scaling by setting `android:anyDensity` (https://developer.android.com/guide/topics/manifest/supports-screens-element.html#any) to `"false"` in the manifest or programmatically for a `Bitmap` (https://developer.android.com/reference/android/graphics/Bitmap.html) by setting `inScaled` (https://developer.android.com/reference/android/graphics/BitmapFactory.Options.html#inScaled) to `"false"`. In this case, the system auto-scales any absolute pixel coordinates and pixel dimension values at draw time. It does this to ensure that pixel-defined screen elements are still displayed at approximately the same physical size as they would be at the baseline screen density (mdpi). The system handles this scaling transparently to the application and reports the scaled pixel dimensions to the application, rather than physical pixel dimensions.

   For instance, suppose a device has a WVGA high-density screen, which is 480x800 and about the same size as a traditional HVGA screen, but it's running an application that has disabled pre-scaling. In this case, the system will "lie" to the application when it queries for screen dimensions, and report 320x533 (the approximate mdpi translation for the screen density). Then, when the application does drawing operations, such as invalidating the rectangle from (10,10) to (100, 100), the system transforms the coordinates by scaling them the appropriate amount, and actually invalidate the region (15,15) to (150, 150). This discrepancy may cause unexpected behavior if your application directly manipulates the scaled bitmap, but this is considered a reasonable trade-off to keep the performance of applications as good as possible. If you encounter this situation, read the following section about Converting dp units to pixel units (#dips-pels).

   Usually, **you should not disable pre-scaling**. The best way to support multiple screens is to follow the basic techniques described above in How to Support Multiple Screens (#support).

If your application manipulates bitmaps or directly interacts with pixels on the screen in some other way, you might need to take additional steps to support different screen densities. For example, if you respond to touch gestures by counting the number of pixels that a finger crosses, you need to use the appropriate density-independent pixel values, instead of actual pixels.

Scaling Bitmap objects created at runtime

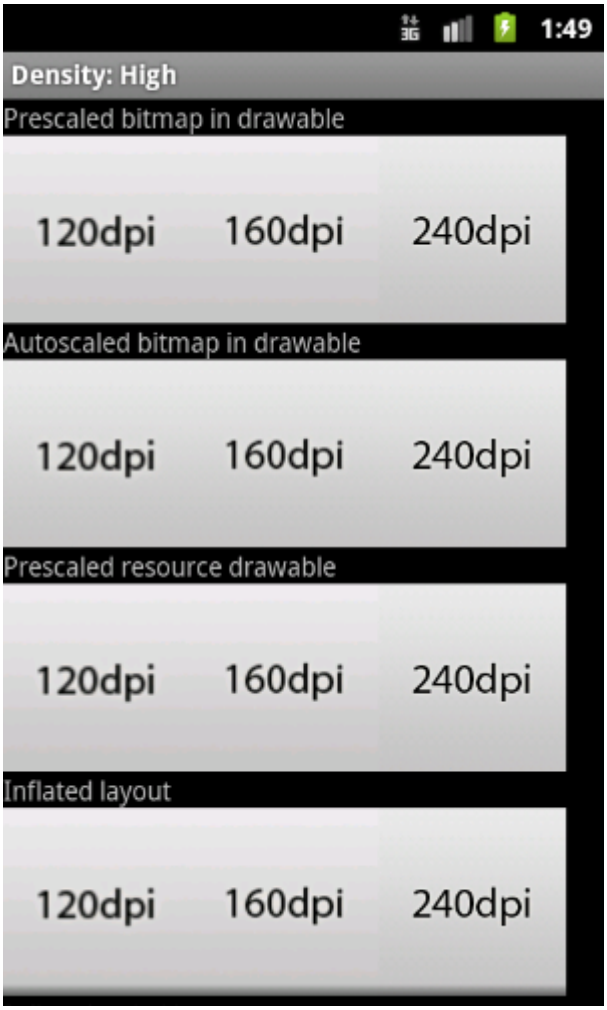

Figure 5. Comparison of pre-scaled and auto-scaled bitmaps.

If your application creates an in-memory bitmap (a `Bitmap` (https://developer.android.com/reference/android/graphics/Bitmap.html) object), the system assumes that the bitmap is designed for the baseline medium-density screen, by default, and auto-scales the bitmap at draw time. The system applies "auto-scaling" to a `Bitmap` (https://developer.android.com/reference/android/graphics/Bitmap.html) when the bitmap has unspecified density properties. If you don't properly account for the current device's screen density and specify the bitmap's density properties, the auto-scaling can result in scaling artifacts the same as when you don't provide alternative resources.

To control whether a `Bitmap` (https://developer.android.com/reference/android/graphics/Bitmap.html) created at runtime is scaled or not, you can specify the density of the bitmap with `setDensity()` (https://developer.android.com/reference/android/graphics/Bitmap.html#setDensity(int)), passing a density constant from `DisplayMetrics` (https://developer.android.com/reference/android/util/DisplayMetrics.html), such as

DENSITY_HIGH (https://developer.android.com/reference/android/util/DisplayMetrics.html#DENSITY_HIGH) or DENSITY_LOW (https://developer.android.com/reference/android/util/DisplayMetrics.html#DENSITY_LOW).

If you're creating a Bitmap (https://developer.android.com/reference/android/graphics/Bitmap.html) using BitmapFactory (https://developer.android.com/reference/android/graphics/BitmapFactory.html), such as from a file or a stream, you can use BitmapFactory.Options (https://developer.android.com/reference/android/graphics/BitmapFactory.Options.html) to define properties of the bitmap as it already exists, which determine if or how the system will scale it. For example, you can use the inDensity (https://developer.android.com/reference/android/graphics/BitmapFactory.Options.html#inDensity) field to define the density for which the bitmap is designed and the inScaled (https://developer.android.com/reference/android/graphics/BitmapFactory.Options.html#inScaled) field to specify whether the bitmap should scale to match the current device's screen density.

If you set the inScaled (https://developer.android.com/reference/android/graphics/BitmapFactory.Options.html#inScaled) field to false, then you disable any pre-scaling that the system may apply to the bitmap and the system will then auto-scale it at draw time. Using auto-scaling instead of pre-scaling can be more CPU expensive, but uses less memory.

Figure 5 demonstrates the results of the pre-scale and auto-scale mechanisms when loading low (120), medium (160) and high (240) density bitmaps on a high-density screen. The differences are subtle, because all of the bitmaps are being scaled to match the current screen density, however the scaled bitmaps have slightly different appearances depending on whether they are pre-scaled or auto-scaled at draw time.

> **Note:** In Android 3.0 and above, there should be no perceivable difference between pre-scaled and auto-scaled bitmaps, due to improvements in the graphics framework.

Converting dp units to pixel units

In some cases, you will need to express dimensions in dp and then convert them to pixels. Imagine an application in which a scroll or fling gesture is recognized after the user's finger has moved by at least 16 pixels. On a baseline screen, a user's must move by 16 pixels / 160 dpi, which equals 1/10th of an inch (or 2.5 mm) before the gesture is recognized. On a device with a high-density display (240dpi), the user's must move by 16 pixels / 240 dpi, which equals 1/15th of an inch (or 1.7 mm). The distance is much shorter and the application thus appears more sensitive to the user.

To fix this issue, the gesture threshold must be expressed in code in dp and then converted to actual pixels. For example:

```
// The gesture threshold expressed in dp
private static final float GESTURE_THRESHOLD_DP = 16.0f;

// Get the screen's density scale
final float scale = getResources() (https://developer.android.com/reference/android/content/ContextWrapp
// Convert the dps to pixels, based on density scale
mGestureThreshold = (int) (GESTURE_THRESHOLD_DP * scale + 0.5f);

// Use mGestureThreshold as a distance in pixels...
```

The `DisplayMetrics.density` (https://developer.android.com/reference/android/util/DisplayMetrics.html#density) field specifies the scale factor you must use to convert `dp` units to pixels, according to the current screen density. On a medium-density screen, `DisplayMetrics.density` (https://developer.android.com/reference/android/util/DisplayMetrics.html#density) equals 1.0; on a high-density screen it equals 1.5; on an extra-high-density screen, it equals 2.0; and on a low-density screen, it equals 0.75. This figure is the factor by which you should multiply the `dp` units on order to get the actual pixel count for the current screen. (Then add `0.5f` to round the figure up to the nearest whole number, when converting to an integer.) For more information, refer to the `DisplayMetrics` (https://developer.android.com/reference/android/util/DisplayMetrics.html) class.

However, instead of defining an arbitrary threshold for this kind of event, you should use pre-scaled configuration values that are available from `ViewConfiguration` (https://developer.android.com/reference/android/view/ViewConfiguration.html).

Using pre-scaled configuration values

You can use the `ViewConfiguration` (https://developer.android.com/reference/android/view/ViewConfiguration.html) class to access common distances, speeds, and times used by the Android system. For instance, the distance in pixels used by the framework as the scroll threshold can be obtained with `getScaledTouchSlop()` (https://developer.android.com/reference/android/view/ViewConfiguration.html#getScaledTouchSlop()):

```
private static final int GESTURE_THRESHOLD_DP = ViewConfiguration.get(myContext).getScaled
```

Methods in `ViewConfiguration` (https://developer.android.com/reference/android/view/ViewConfiguration.html) starting with the `getScaled` prefix are guaranteed to return a value in pixels that will display properly regardless of the current screen density.

How to Test Your Application on Multiple Screens

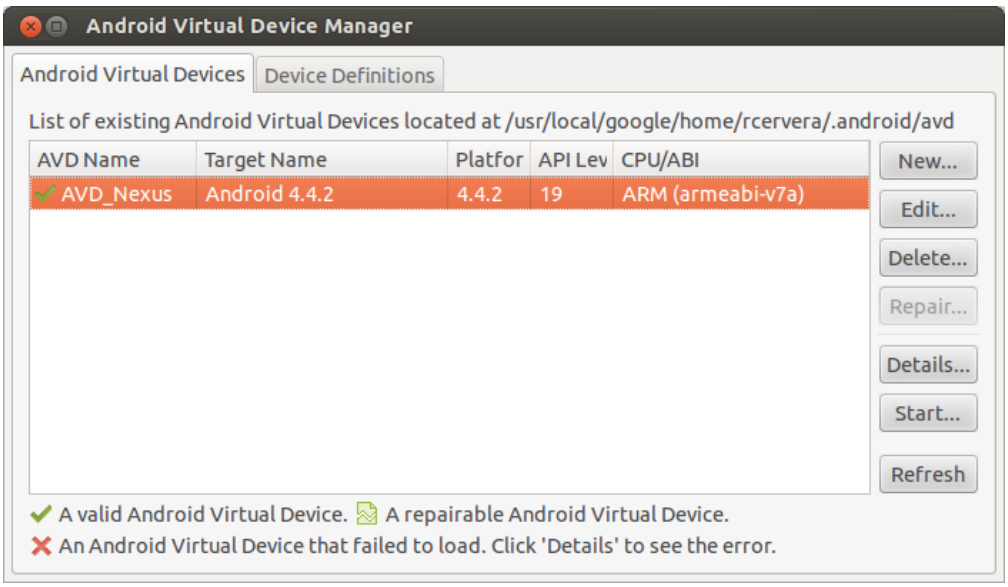

Figure 6. A set of AVDs for testing screens support.

Before publishing your application, you should thoroughly test it in all of the supported screen sizes and densities. The Android SDK includes emulator skins you can use, which replicate the sizes and densities of common screen configurations on which your application is likely to run. You can also modify the default size, density, and resolution of the emulator skins to replicate the characteristics of any specific screen. Using the emulator skins and additional custom configurations allows you to test any possible screen configuration, so you don't have to buy various devices just to test your application's screen support.

To set up an environment for testing your application's screen support, you should create a series of AVDs (Android Virtual Devices), using emulator skins and screen configurations that emulate the screen sizes and densities you want your application to support. To do so, you can use the AVD Manager to create the AVDs and launch them with a graphical interface.

To launch the Android SDK Manager, execute the `SDK Manager.exe` from your Android SDK directory (on Windows only) or execute `android` from the `<sdk>/tools/` directory (on all platforms). Figure 6 shows the AVD Manager with a selection of AVDs, for testing various screen configurations.

Table 3 shows the various emulator skins that are available in the Android SDK, which you can use to emulate some of the most common screen configurations.

For more information about creating and using AVDs to test your application, see Managing AVDs with AVD Manager (https://developer.android.com/tools/devices/managing-avds.html).

Table 3. Various screen configurations available from emulator skins in the Android SDK (indicated in bold) and other representative resolutions.

	Low density (120), *ldpi*	Medium density (160), *mdpi*	High density (240), *hdpi*	Extra-high-densi
Small screen	**QVGA (240x320)**		480x640	
Normal screen	**WQVGA400 (240x400)** **WQVGA432 (240x432)**	**HVGA (320x480)**	**WVGA800 (480x800)** **WVGA854 (480x854)** 600x1024	640x960
Large screen	**WVGA800** (480x800)** **WVGA854** (480x854)**	**WVGA800* (480x800)** **WVGA854* (480x854)** 600x1024		
Extra-Large screen	1024x600	**WXGA (1280x800)**† 1024x768 1280x768	1536x1152 1920x1152 1920x1200	2048x1536 2560x1536 2560x1600

* To emulate this configuration, specify a custom density of 160 when creating an AVD that uses a WVGA800 or WVGA854 skin.

** To emulate this configuration, specify a custom density of 120 when creating an AVD that uses a WVGA800 or WVGA854 skin.

† This skin is available with the Android 3.0 platform

To see the relative numbers of active devices that support any given screen configuration, see the Screen Sizes and Densities (https://developer.android.com/resources/dashboard/screens.html) dashboard.

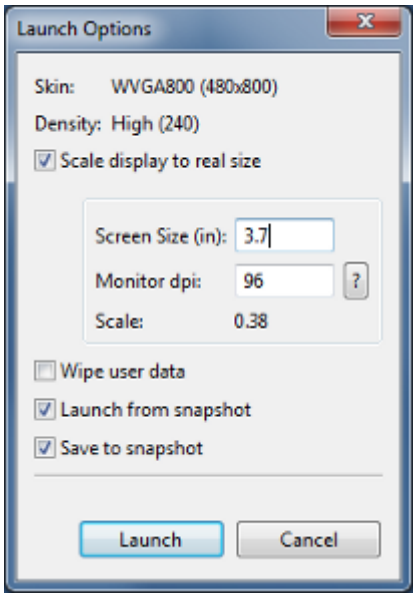

Figure 7. Size and density options you can set, when starting an AVD from the AVD Manager.

We also recommend that you test your application in an emulator that is set up to run at a physical size that closely matches an actual device. This makes it a lot easier to compare the results at various sizes and densities. To do so you need to know the approximate density, in dpi, of your computer monitor (for instance, a 30" Dell monitor has a density of about 96 dpi). When you launch an AVD from the AVD Manager, you can specify the screen size for the emulator and your monitor dpi in the Launch Options, as shown in figure 7.

If you would like to test your application on a screen that uses a resolution or density not supported by the built-in skins, you can create an AVD that uses a custom resolution or density. When creating the AVD from the AVD Manager, specify the Resolution, instead of selecting a Built-in Skin.

If you are launching your AVD from the command line, you can specify the scale for the emulator with the `-scale` option. For example:

```
emulator -avd <avd_name> -scale 96dpi
```

To refine the size of the emulator, you can instead pass the `-scale` option a number between 0.1 and 3 that represents the desired scaling factor.

For more information about creating AVDs from the command line, see Managing AVDs from the Command Line (https://developer.android.com/tools/devices/managing-avds-cmdline.html).