目录

[Android屏幕适配建议文档 2](#_Toc405385113)

[屏幕相关概念 2](#_Toc405385114)

[屏幕像素px 2](#_Toc405385115)

[屏幕大小size 2](#_Toc405385116)

[屏幕密度dpi 2](#_Toc405385117)

[屏幕方向 2](#_Toc405385118)

[分辨率 3](#_Toc405385119)

[独立密度像素DIP（Density-independent Pixels与密度无关的像素）： 3](#_Toc405385120)

[DPI与DIP的换算 3](#_Toc405385121)

[点9图片 4](#_Toc405385122)

[为什么会出现Android屏幕适配问题。 6](#_Toc405385123)

[Android使用版本的多样化 6](#_Toc405385124)

[Android设备屏幕尺寸和屏幕密度的多样化 7](#_Toc405385125)

[Android设备加载图片资源的流程和原理 9](#_Toc405385126)

[1、Android中放置图片资源的文件夹 9](#_Toc405385127)

[2、Android中view设置背景图片时查找图片资源的顺序 9](#_Toc405385128)

[3、设置view背景图片的过程 10](#_Toc405385129)

[屏幕适配的解决办法 13](#_Toc405385130)

[适配不同屏幕尺寸的解决办法。 13](#_Toc405385131)

[适配不同的设备密度解决办法 26](#_Toc405385132)

[适配特殊屏幕设备的解决办法 28](#_Toc405385133)

[Android 图标设计建议 32](#_Toc405385134)

# Android屏幕适配建议文档

## 屏幕相关概念

### 屏幕像素px

是屏幕的物理像素点，与密度相关，密度大了，单位面积上的px会比较多。

### 屏幕大小size

通过查阅设备信息可以知道设备的物理尺寸，也可以通过计算屏幕对角线的长度得到，单位为Inch。

### 屏幕密度dpi

屏幕在物理尺寸范围内的像素数量。也通常指DPI(dots per inch)。屏幕密度越小，所包含的像素也就越少。

DPI＝（√（横向分辨率^2+纵向分辨率^2））/屏幕尺寸  
在Android的设计规范中，把屏幕密度划分为6类

屏幕密度等级：

ldpi (low) ~120dpi（基本上已经淘汰了）

mdpi (medium) ~160dpi

hdpi (high) ~240dpi

xhdpi (extra-high) ~320dpi

xxhdpi (extra-extra-high) ~480dpi

xxxhdpi (extra-extra-extra-high) ~640dpi

### 屏幕方向

屏幕方向有横向和纵向，这两种情况下的屏幕长宽比是不同的。

### 分辨率

跟电脑分辨率的概念类似，表示屏幕横纵方向的像素数，例如480\*800。  
在为APP进行屏幕适配时，不能只考虑到像素，还有屏幕大小，屏幕密度等等。

分辨率等级：

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

### 独立密度像素DIP（Density-independent Pixels与密度无关的像素）：

一个基于density的抽象单位，这个和设备硬件有关，通常在开发中设置一些view的宽高推荐用这个，一般情况下，在不同分辨率，都不会有缩放的感觉。在运行时, Android根据使用中的屏幕的实际密度, 透明地处理任何所需dip单位的缩放。

### DPI与DIP的换算

dots per inch，就是每英寸多少像素，通过下面公式可以得到。  
DPI＝（√（横向分辨率^2+纵向分辨率^2））/屏幕尺寸

在Android的设计规范中，DPI分成了5个档次：MDPI，HDPI，XHDPI，XXHDPI，XXXHDPI，它们的比例是 2:3:4:6:8

[](http://isux.tencent.com/wp-content/uploads/2014/06/20140630193233262.png)

在MDPI的屏幕(即160DPI)，1dp 和1sp 基本等于 1px 。

* dp 和 px的换算公式：px ＝ dp\*(DPI/160)。
* sp 和 px的换算公式：px ＝ sp\*(DPI/160)。

### 点9图片

与传统的png 格式图片相比， 9.png 格式图片在图片四周有一圈一个像素点组成的边沿，该边沿用于对图片的可扩展区和内容显示区进行定义。

   这种格式的图片在android 环境下具有自适应调节大小的能力。

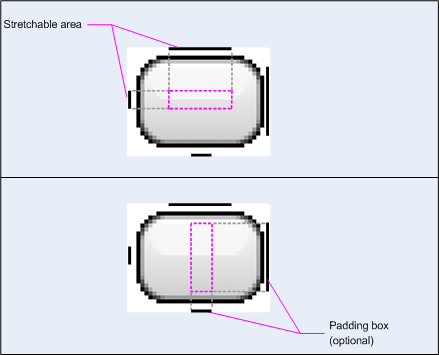
   （1）允许开发人员定义可扩展区域，当需要延伸图片以填充比图片本身更大区域时，可扩展区的内容被延展。

  （2）允许开发人员定义内容显示区，用于显示文字或其他内容

    如下图所示：

    左侧和上方的黑线交叉的部分即可扩展区域

    右侧和下方的黑线交叉的部分即内容显示区



点9图的处理过程和上面的普通png图片是一样的，会根据所放置的资源文件夹和屏幕的像素密度先进行缩放，在显示的时候点9图会再进行局部拉伸，所以如果将带圆角的点9图片放置在低像素密度资源文件夹下，当使用高像素密度设备显示时，图片会先进行放大在进行局部拉伸，这样会导致在放大过程中图片圆角和边缘被拉伸，显示时会变的模糊。

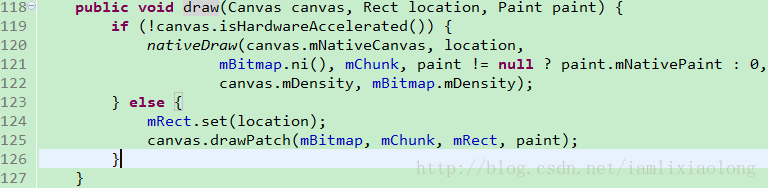
解决方案：

1、尽量将点9图片放置在高像素密度资源文件夹中，这样即使在低像素密度手机上显示时会先对图片进行缩小再进行局部拉伸，但是在低像素密度手机上运行应用时，所有使用点9图片的地方都会对图片进行一次计算缩放，影响性能；

2、针对不同像素密度手机做多套点9图片。

补充：点9图片在缩放过后，如何进行局部拉伸渲染到屏幕上的？

源码跟踪，在View的draw方法中根据Drawable对象将图片作为背景绘制到指定区域中，点9图的实际绘制过程在NinePatch的draw方法中，通过canvas对象调用了本地方法nativeDraw对图片进行了绘制。至于如何绘制局部暂时看不到JNI方法的源码。

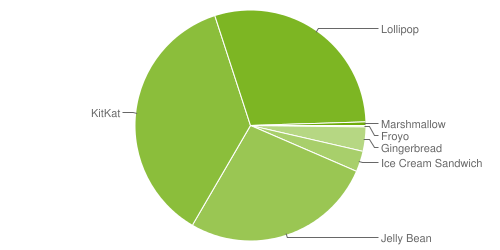


## 为什么会出现Android屏幕适配问题。

### Android使用版本的多样化

一下这张表格是反映现有Android版本的使用比重

|  |  |  |  |
| --- | --- | --- | --- |
| Version | Codename | API | Distribution |
| [2.2](http://developer.android.com/about/versions/android-2.2.html) | Froyo | 8 | 0.2% |
| [2.3.3 - 2.3.7](http://developer.android.com/about/versions/android-2.3.3.html) | Gingerbread | 10 | 3.4% |
| [4.0.3 - 4.0.4](http://developer.android.com/about/versions/android-4.0.html) | Ice Cream Sandwich | 15 | 2.9% |
| [4.1.x](http://developer.android.com/about/versions/android-4.1.html) | Jelly Bean | 16 | 10.0% |
| [4.2.x](http://developer.android.com/about/versions/android-4.2.html) | 17 | 13.0% |
| [4.3](http://developer.android.com/about/versions/android-4.3.html) | 18 | 3.9% |
| [4.4](http://developer.android.com/about/versions/android-4.4.html) | KitKat | 19 | 36.6% |
| [5.0](http://developer.android.com/about/versions/android-5.0.html) | Lollipop | 21 | 16.3% |
| [5.1](http://developer.android.com/about/versions/android-5.1.html) | 22 | 13.2% |
| [6.0](http://developer.android.com/about/versions/marshmallow/index.html) | Marshmallow | 23 | 0.5% |



数据截止到 on December 7, 2015.Any versions with less than 0.1% distribution are not shown.

以上数据统计截止到到2014年12月1日。（0.1%以下的样本没有列入以上表格和图表）

**由以上图表分析得现在使用Android4.0以上的用户比重占到了将近96%**

**但是这96%的占有率并不集中在某一版本之上，而是分布于从api15到22的不同版本上这就造成了我们写Android应用是需要去适配不同版本，是其原因之一。**

### Android设备屏幕尺寸和屏幕密度的多样化

以下图表提供有关具不同屏幕大小和密度占有比例。Android官方将Android设备的屏幕尺寸和屏幕密度人为的划分为不同的等级。

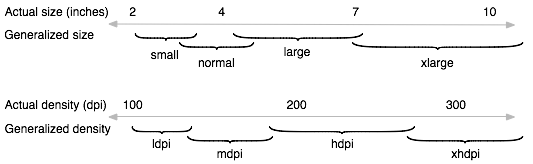
屏幕分辨率等级：

* 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

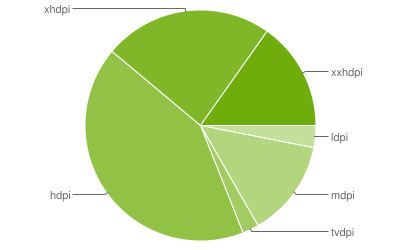
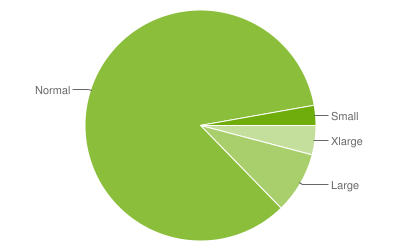
屏幕密度等级：

* ldpi (low) ~120dpi
* mdpi (medium) ~160dpi
* hdpi (high) ~240dpi
* xhdpi (extra-high) ~320dpi
* xxhdpi (extra-extra-high) ~480dpi
* xxxhdpi (extra-extra-extra-high) ~640dpi

大致分布情况统计如下表：



|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | ldpi | mdpi | tvdpi | hdpi | xhdpi | xxhdpi | **Total** |
| Small | 2.8% |  |  |  |  |  | 2.8% |
| Normal |  | 5.7% | 0.1% | 41.2% | 22.3% | 15.2% | 84.5% |
| Large | 0.3% | 4.8% | 2.2% | 0.6% | 0.7% |  | 8.6% |
| Xlarge |  | 3.1% |  | 0.3% | 0.7% |  | 4.1% |
| **Total** | 3.1% | 13.6% | 2.3% | 42.1% | 23.7% | 15.2% |  |



以上数据统计截止到*December 7, 2015.*

占有率0.1%以下设备没有纳入其中

**由以上数据分析得屏幕分辨率在470dp x 320dp以上的设备占到了将近95%，屏幕密度在240dpi以上的设备占到了将近80%，但是他们同样分布不均，从而导致了我们的Android应用需要去适配不同的屏幕分辨率和不同的屏幕密度的设备**

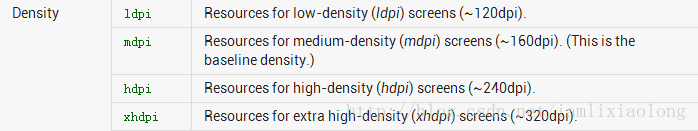
**屏幕尺寸单位是英寸而不是像素，也就说一个英寸在任何分辨率下显示的大小都是一样的，但是像素在密度不同的手机里面显示的实际的大小是不一样的**

## Android设备加载图片资源的流程和原理

### 1、Android中放置图片资源的文件夹

Android中一般有drawable-ldpi、drawable-mdpi、drawable-hdpi、drawable-xhdpi、drawable-xxhdpi等放置图片资源的文件夹，这几个文件夹分别对应的像素密度为：

|  |  |
| --- | --- |
| 文件夹 | 对应的像素密度 |
| drawable-ldpi | 120dpi |
| drawable-mdpi | 160dpi |
| drawable-hdpi | 240dpi |
| drawable-xhdpi | 320dpi |



另外自己可以创建一个默认的drawable文件夹，默认对应的像素密度为160dpi。

### 2、Android中view设置背景图片时查找图片资源的顺序

Android中view在setBackground加载图片时，通常会先去设备对应的像素密度的文件夹中去找对应的图片，如果没有找到就会去紧挨着的像素密度较高的文件夹中去找，然后再往上找，如果最高像素密度的文件夹中还没有找到该图片才会向像素密度较低的文件夹中去找。这是加载对应图片的一个查找过程。

比如一个设备的像素密度为240dpi，应用程序有drawable、drawable-ldpi、drawable-mdp、drawable-hdpi、drawable-xhdpi、drawable-xxhdpi六个文件夹，则在一个view设置背景图片时，查找图片的顺序为drawable-hdpi   ===== >  drawable-xhdpi  ====>  drawable-xxhdpi ====>   drawable-mdpi  ====>   drawable  ====>  drawable-ldpi。

这个顺序可以通过一个小的demo自己验证，是android查找图片资源的规则，不是通过几句话就能说明的。

### 3、设置view背景图片的过程

源码跟踪：

View view = new View(this);

view.setBackgroundResource(R.drawable.about\_logo);

使用这里是设置一个view的背景图片，setBackgroundResource方法的源码为：

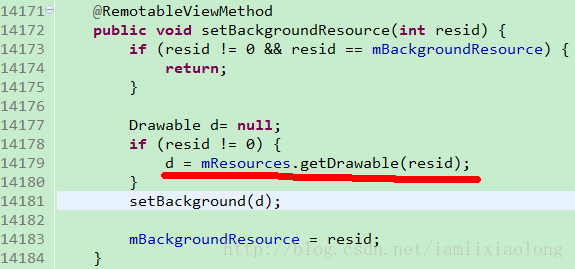


图-2

这个方法里面第14179行代码，红线标注部分，是通过Resources对象根据图片资源resid去获取图片对应的drawable对象，getDrawable方法源码如下：

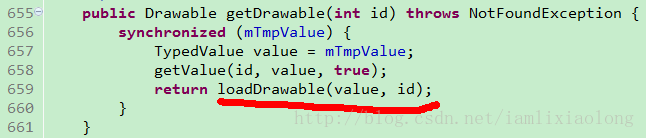


图-3

这个方法里面通过loadDrawable方法返回一个Drawable对象，loadDrawable方法里面传入了一个TypedValue对象，而TypedValue对象是通过getValue方法获得的，这里可以通过代码查看一下TypedValue对象中存放了哪些对应资源图片的信息。

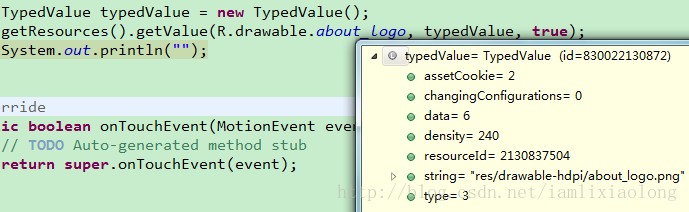


图-4

通过demo代码中，对代码进行debug，发现，根据图片资源resid获取的TypeValued对象中保存的信息主要有density=240和string=“res/drawable-hdpi/about-logo.png”，density是指找到的图片资源所在drawable-hdpi文件夹对应的像素密度，string是图片资源的路径。

实际上loadDrawable方法就是根据这个图片资源的路径去获取到相应的Drawable对象的。此时我是将图片放置到drawable-hdpi文件夹中的，那么如果我将图片移动到drawable-mdpi文件夹中，TypeValued值会一样吗？通过测试，发现如下结果：

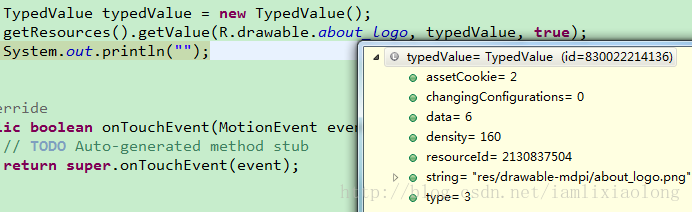


图-5

通过图-4和图-5得出的typedValue信息可知，其获得的相应变量值是不一样的，此时的density=160、string=”res/drawable-mdpi/about-logo.png”，density是drawable-mdpi对应的像素密度，那么同样的图片放置在不同的资源文件夹中，得到的Drawable对象一样吗，通过证明，它们是不一样的。

about\_logo.png原始图片大小为138\*64像素，在同样的480\*800像素240dpi的模拟器上运行，其得到的Drawable对象信息如下：

图片放置在drawable-mdpi文件夹下：

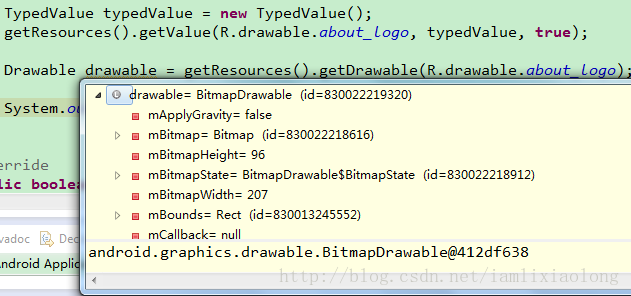


图-6

图片放置在drawable-hdpi文件夹下：

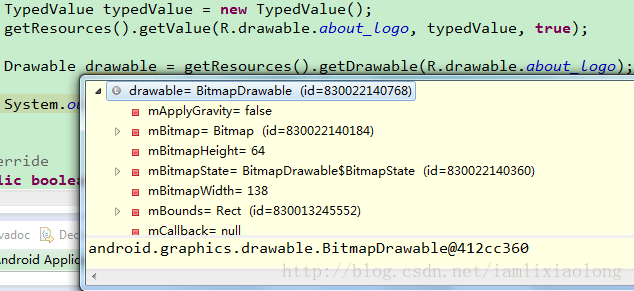


图-7

通过代码测试得出如下数据：

|  |  |  |  |
| --- | --- | --- | --- |
| 图片放置的文件夹 | 对应像素密度 | 设备像素密度 | 得到的图片对应的Bitmap的宽高值 |
| drawable-mdpi | 160dpi | 240dpi | 207\*96 |
| drawable-hdpi | 240dpi | 240dpi | 138\*64 |

由此看出，放置在不同文件夹下面的相同的图片，在相同像素密度下所取得的图片Bitmap大小是不同的，如上表格中，drawable-mdpi下的图片实际上是进行了缩放的。

程序得到的图片宽度 = 实际图片宽度 \* 设备像素密度 / 图片资源文件夹对应的像素密度

程序得到的图片高度 = 实际图片高度 \* 设备像素密度 / 图片资源文件夹对应的像素密度

由此可以看出如果图片放置在低密度文件夹中，而要在高像素密度设备上显示时，其会先进行放大，然后再显示，这样就会导致高像素密度设备上显示模糊。

注：图片Bitmap放大的过程可以在源码中找到，源码在BitmapFactory.decodeStream方法中。

## 屏幕适配的解决办法

### 适配不同屏幕尺寸的解决办法。

#### THIS LESSON TEACHES YOU TO

This lesson shows you how to support different screen sizes by:

* 确保您的布局可以充分调整大小以适应屏幕
* 根据屏幕配置提供合适的UI布局
* 确保正确的布局被施加到正确的屏幕
* 提供正确的缩放位图

#### Use "wrap\_content" and "match\_parent"

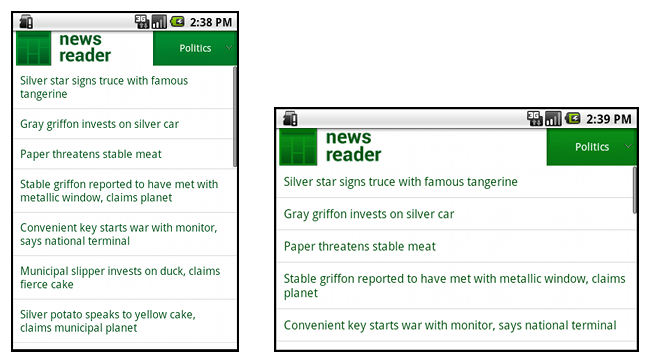
To ensure that your layout is flexible and adapts to different screen sizes, you should use "wrap\_content" and "match\_parent"for the width and height of some view components. If you use"wrap\_content", the width or height of the view is set to the minimum size necessary to fit the content within that view, while "match\_parent" (also known as "fill\_parent" before API level 8) makes the component expand to match the size of its parent view.

By using the "wrap\_content" and "match\_parent" size values instead of hard-coded sizes, your views either use only the space required for that view or expand to fill the available space, respectively. For example:

<LinearLayoutxmlns:android="http://schemas.android.com/apk/res/android"  
    android:orientation="vertical"  
    android:layout\_width="match\_parent"  
    android:layout\_height="match\_parent">  
    <LinearLayoutandroid:layout\_width="match\_parent"  
                  android:id="@+id/linearLayout1"    
                  android:gravity="center"  
                  android:layout\_height="50dp">  
        <ImageViewandroid:id="@+id/imageView1"  
                   android:layout\_height="wrap\_content"  
                   android:layout\_width="wrap\_content"  
                   android:src="@drawable/logo"  
                   android:paddingRight="30dp"  
                   android:layout\_gravity="left"  
                   android:layout\_weight="0"/>  
        <Viewandroid:layout\_height="wrap\_content"  
              android:id="@+id/view1"  
              android:layout\_width="wrap\_content"  
              android:layout\_weight="1"/>  
        <Buttonandroid:id="@+id/categorybutton"  
                android:background="@drawable/button\_bg"  
                android:layout\_height="match\_parent"  
                android:layout\_weight="0"  
                android:layout\_width="120dp"  
                style="@style/CategoryButtonStyle"/>  
    </LinearLayout>  
  
    <fragmentandroid:id="@+id/headlines"  
              android:layout\_height="fill\_parent"  
              android:name="com.example.android.newsreader.HeadlinesFragment"  
              android:layout\_width="match\_parent"/>  
</LinearLayout>

Notice how the sample uses "wrap\_content" and "match\_parent" for component sizes rather than specific dimensions. This allows the layout to adapt correctly to different screen sizes and orientations.

For example, this is what this layout looks like in portrait and landscape mode. Notice that the sizes of the components adapt automatically to the width and height:



**Figure 1.** The News Reader sample app in portrait (left) and landscape (right).

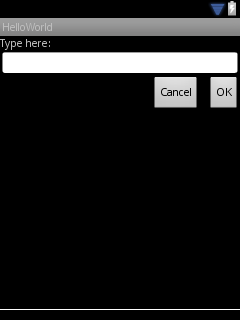
#### Use RelativeLayout

You can construct fairly complex layouts using nested instances of [LinearLayout](http://developer.android.com/reference/android/widget/LinearLayout.html) and combinations of"wrap\_content" and "match\_parent" sizes. However, [LinearLayout](http://developer.android.com/reference/android/widget/LinearLayout.html) does not allow you to precisely control the spacial relationships of child views; views in a [LinearLayout](http://developer.android.com/reference/android/widget/LinearLayout.html) simply line up side-by-side. If you need child views to be oriented in variations other than a straight line, a better solution is often to use a [RelativeLayout](http://developer.android.com/reference/android/widget/RelativeLayout.html), which allows you to specify your layout in terms of the spacial relationships between components. For instance, you can align one child view on the left side and another view on the right side of the screen.

For example:

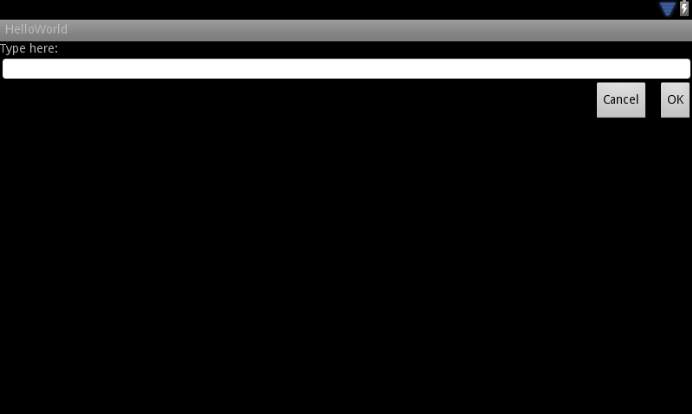
<?xml version="1.0" encoding="utf-8"?>  
<RelativeLayoutxmlns:android="http://schemas.android.com/apk/res/android"  
    android:layout\_width="match\_parent"  
    android:layout\_height="match\_parent">  
    <TextView  
        android:id="@+id/label"  
        android:layout\_width="match\_parent"  
        android:layout\_height="wrap\_content"  
        android:text="Type here:"/>  
    <EditText  
        android:id="@+id/entry"  
        android:layout\_width="match\_parent"  
        android:layout\_height="wrap\_content"  
        android:layout\_below="@id/label"/>  
    <Button  
        android:id="@+id/ok"  
        android:layout\_width="wrap\_content"  
        android:layout\_height="wrap\_content"  
        android:layout\_below="@id/entry"  
        android:layout\_alignParentRight="true"  
        android:layout\_marginLeft="10dp"  
        android:text="OK"/>  
    <Button  
        android:layout\_width="wrap\_content"  
        android:layout\_height="wrap\_content"  
        android:layout\_toLeftOf="@id/ok"  
        android:layout\_alignTop="@id/ok"  
        android:text="Cancel"/>  
</RelativeLayout>

Figure 2 shows how this layout appears on a QVGA screen.



**Figure 2.** Screenshot on a QVGA screen (small screen).

Figure 3 shows how it appears on a larger screen.



**Figure 3.** Screenshot on a WSVGA screen (large screen).

Notice that although the size of the components changed, their spatial relationships are preserved as specified by the [RelativeLayout.LayoutParams](http://developer.android.com/reference/android/widget/RelativeLayout.LayoutParams.html).

#### Use Size Qualifiers

There's only so much mileage you can get from a flexible layout or relative layout like the one in the previous sections. While those layouts adapt to different screens by stretching the space within and around components, they may not provide the best user experience for each screen size. Therefore, your application should not only implement flexible layouts, but should also provide several alternative layouts to target different screen configurations. You do so by using [configuration qualifiers](http://developer.android.com/guide/practices/screens_support.html#qualifiers), which allows the runtime to automatically select the appropriate resource based on the current device’s configuration (such as a different layout design for different screen sizes).

For example, many applications implement the "two pane" pattern for large screens (the app might show a list of items on one pane and the content on another pane). Tablets and TVs are large enough for both panes to fit simultaneously on screen, but phone screens have to show them separately. So, to implement these layouts, you could have the following files:

* res/layout/main.xml, single-pane (default) layout:

<LinearLayoutxmlns:android="http://schemas.android.com/apk/res/android"  
    android:orientation="vertical"  
    android:layout\_width="match\_parent"  
    android:layout\_height="match\_parent">  
  
    <fragmentandroid:id="@+id/headlines"  
              android:layout\_height="fill\_parent"  
              android:name="com.example.android.newsreader.HeadlinesFragment"  
              android:layout\_width="match\_parent"/>  
</LinearLayout>

* res/layout-large/main.xml, two-pane layout:

<LinearLayoutxmlns:android="http://schemas.android.com/apk/res/android"  
    android:layout\_width="fill\_parent"  
    android:layout\_height="fill\_parent"  
    android:orientation="horizontal">  
    <fragmentandroid:id="@+id/headlines"  
              android:layout\_height="fill\_parent"  
              android:name="com.example.android.newsreader.HeadlinesFragment"  
              android:layout\_width="400dp"  
              android:layout\_marginRight="10dp"/>  
    <fragmentandroid:id="@+id/article"  
              android:layout\_height="fill\_parent"  
              android:name="com.example.android.newsreader.ArticleFragment"  
              android:layout\_width="fill\_parent"/>  
</LinearLayout>

Notice the large qualifier in the directory name of the second layout. This layout will be selected on devices with screens classified as large (for example, 7" tablets and above). The other layout (without qualifiers) will be selected for smaller devices.

#### Use the Smallest-width Qualifier

One of the difficulties developers had in pre-3.2 Android devices was the "large" screen size bin, which encompasses the Dell Streak, the original Galaxy Tab, and 7" tablets in general. However, many applications may want to show different layouts for different devices in this category (such as for 5" and 7" devices), even though they are all considered to be "large" screens. That's why Android introduced the "Smallest-width" qualifier (amongst others) in Android 3.2.

The Smallest-width qualifier allows you to target screens that have a certain minimum width given in dp. For example, the typical 7" tablet has a minimum width of 600 dp, so if you want your UI to have two panes on those screens (but a single list on smaller screens), you can use the same two layouts from the previous section for single and two-pane layouts, but instead of the large size qualifier, use sw600dp to indicate the two-pane layout is for screens on which the smallest-width is 600 dp:

* res/layout/main.xml, single-pane (default) layout:

<LinearLayoutxmlns:android="http://schemas.android.com/apk/res/android"  
    android:orientation="vertical"  
    android:layout\_width="match\_parent"  
    android:layout\_height="match\_parent">  
  
    <fragmentandroid:id="@+id/headlines"  
              android:layout\_height="fill\_parent"  
              android:name="com.example.android.newsreader.HeadlinesFragment"  
              android:layout\_width="match\_parent"/>  
</LinearLayout>

* res/layout-sw600dp/main.xml, two-pane layout:

<LinearLayoutxmlns:android="http://schemas.android.com/apk/res/android"  
    android:layout\_width="fill\_parent"  
    android:layout\_height="fill\_parent"  
    android:orientation="horizontal">  
    <fragmentandroid:id="@+id/headlines"  
              android:layout\_height="fill\_parent"  
              android:name="com.example.android.newsreader.HeadlinesFragment"  
              android:layout\_width="400dp"  
              android:layout\_marginRight="10dp"/>  
    <fragmentandroid:id="@+id/article"  
              android:layout\_height="fill\_parent"  
              android:name="com.example.android.newsreader.ArticleFragment"  
              android:layout\_width="fill\_parent"/>  
</LinearLayout>

This means that devices whose smallest width is greater than or equal to 600dp will select the layout-sw600dp/main.xml (two-pane) layout, while smaller screens will select the layout/main.xml (single-pane) layout.

However, this won't work well on pre-3.2 devices, because they don't recognize sw600dp as a size qualifier, so you still have to use the large qualifier as well. So, you should have a file named res/layout-large/main.xmlwhich is identical to res/layout-sw600dp/main.xml. In the next section you'll see a technique that allows you to avoid duplicating the layout files this way.

#### Use Layout Aliases

The smallest-width qualifier is available only on Android 3.2 and above. Therefore, you should also still use the abstract size bins (small, normal, large and xlarge) to be compatible with earlier versions. For example, if you want to design your UI so that it shows a single-pane UI on phones but a multi-pane UI on 7" tablets, TVs and other large devices, you'd have to supply these files:

* res/layout/main.xml: single-pane layout
* res/layout-large: multi-pane layout
* res/layout-sw600dp: multi-pane layout

The last two files are identical, because one of them will be matched by Android 3.2 devices, and the other one is for the benefit of tablets and TVs with earlier versions of Android.

To avoid this duplication of the same file for tablets and TVs (and the maintenance headache resulting from it), you can use alias files. For example, you can define the following layouts:

* res/layout/main.xml, single-pane layout
* res/layout/main\_twopanes.xml, two-pane layout

And add these two files:

* res/values-large/layout.xml:

<resources>  
    <itemname="main"type="layout">@layout/main\_twopanes</item>  
</resources>

* res/values-sw600dp/layout.xml:

<resources>  
    <itemname="main"type="layout">@layout/main\_twopanes</item>  
</resources>

These latter two files have identical content, but they don’t actually define the layout. They merely set up mainto be an alias to main\_twopanes. Since these files have large and sw600dp selectors, they are applied to tablets and TVs regardless of Android version (pre-3.2 tablets and TVs match large, and post-3.2 will match sw600dp).

#### Use Orientation Qualifiers

Some layouts work well in both landscape and portrait orientations, but most of them can benefit from adjustments. In the News Reader sample app, here is how the layout behaves in each screen size and orientation:

* **small screen, portrait:** single pane, with logo
* **small screen, landscape:** single pane, with logo
* **7" tablet, portrait:** single pane, with action bar
* **7" tablet, landscape:** dual pane, wide, with action bar
* **10" tablet, portrait:** dual pane, narrow, with action bar
* **10" tablet, landscape:** dual pane, wide, with action bar
* **TV, landscape:** dual pane, wide, with action bar

So each of these layouts is defined in an XML file in the res/layout/ directory. To then assign each layout to the various screen configurations, the app uses layout aliases to match them to each configuration:

res/layout/onepane.xml:

<LinearLayoutxmlns:android="http://schemas.android.com/apk/res/android"  
    android:orientation="vertical"  
    android:layout\_width="match\_parent"  
    android:layout\_height="match\_parent">  
  
    <fragmentandroid:id="@+id/headlines"  
              android:layout\_height="fill\_parent"  
              android:name="com.example.android.newsreader.HeadlinesFragment"  
              android:layout\_width="match\_parent"/>  
</LinearLayout>

res/layout/onepane\_with\_bar.xml:

<LinearLayoutxmlns:android="http://schemas.android.com/apk/res/android"  
    android:orientation="vertical"  
    android:layout\_width="match\_parent"  
    android:layout\_height="match\_parent">  
    <LinearLayoutandroid:layout\_width="match\_parent"  
                  android:id="@+id/linearLayout1"    
                  android:gravity="center"  
                  android:layout\_height="50dp">  
        <ImageViewandroid:id="@+id/imageView1"  
                   android:layout\_height="wrap\_content"  
                   android:layout\_width="wrap\_content"  
                   android:src="@drawable/logo"  
                   android:paddingRight="30dp"  
                   android:layout\_gravity="left"  
                   android:layout\_weight="0"/>  
        <Viewandroid:layout\_height="wrap\_content"  
              android:id="@+id/view1"  
              android:layout\_width="wrap\_content"  
              android:layout\_weight="1"/>  
        <Buttonandroid:id="@+id/categorybutton"  
                android:background="@drawable/button\_bg"  
                android:layout\_height="match\_parent"  
                android:layout\_weight="0"  
                android:layout\_width="120dp"  
                style="@style/CategoryButtonStyle"/>  
    </LinearLayout>  
  
    <fragmentandroid:id="@+id/headlines"  
              android:layout\_height="fill\_parent"  
              android:name="com.example.android.newsreader.HeadlinesFragment"  
              android:layout\_width="match\_parent"/>  
</LinearLayout>

res/layout/twopanes.xml:

<LinearLayoutxmlns:android="http://schemas.android.com/apk/res/android"  
    android:layout\_width="fill\_parent"  
    android:layout\_height="fill\_parent"  
    android:orientation="horizontal">  
    <fragmentandroid:id="@+id/headlines"  
              android:layout\_height="fill\_parent"  
              android:name="com.example.android.newsreader.HeadlinesFragment"  
              android:layout\_width="400dp"  
              android:layout\_marginRight="10dp"/>  
    <fragmentandroid:id="@+id/article"  
              android:layout\_height="fill\_parent"  
              android:name="com.example.android.newsreader.ArticleFragment"  
              android:layout\_width="fill\_parent"/>  
</LinearLayout>

res/layout/twopanes\_narrow.xml:

<LinearLayoutxmlns:android="http://schemas.android.com/apk/res/android"  
    android:layout\_width="fill\_parent"  
    android:layout\_height="fill\_parent"  
    android:orientation="horizontal">  
    <fragmentandroid:id="@+id/headlines"  
              android:layout\_height="fill\_parent"  
              android:name="com.example.android.newsreader.HeadlinesFragment"  
              android:layout\_width="200dp"  
              android:layout\_marginRight="10dp"/>  
    <fragmentandroid:id="@+id/article"  
              android:layout\_height="fill\_parent"  
              android:name="com.example.android.newsreader.ArticleFragment"  
              android:layout\_width="fill\_parent"/>  
</LinearLayout>

Now that all possible layouts are defined, it's just a matter of mapping the correct layout to each configuration using the configuration qualifiers. You can now do it using the layout alias technique:

res/values/layouts.xml:

<resources>  
    <itemname="main\_layout"type="layout">@layout/onepane\_with\_bar</item>  
    <boolname="has\_two\_panes">false</bool>  
</resources>

res/values-sw600dp-land/layouts.xml:

<resources>  
    <itemname="main\_layout"type="layout">@layout/twopanes</item>  
    <boolname="has\_two\_panes">true</bool>  
</resources>

res/values-sw600dp-port/layouts.xml:

<resources>  
    <itemname="main\_layout"type="layout">@layout/onepane</item>  
    <boolname="has\_two\_panes">false</bool>  
</resources>

res/values-large-land/layouts.xml:

<resources>  
    <itemname="main\_layout"type="layout">@layout/twopanes</item>  
    <boolname="has\_two\_panes">true</bool>  
</resources>

res/values-large-port/layouts.xml:

<resources>  
    <itemname="main\_layout"type="layout">@layout/twopanes\_narrow</item>  
    <boolname="has\_two\_panes">true</bool>  
</resources>

#### Use Nine-patch Bitmaps

Supporting different screen sizes usually means that your image resources must also be capable of adapting to different sizes. For example, a button background must fit whichever button shape it is applied to.

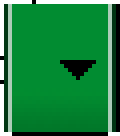
If you use simple images on components that can change size, you will quickly notice that the results are somewhat less than impressive, since the runtime will stretch or shrink your images uniformly. The solution is using nine-patch bitmaps, which are specially formatted PNG files that indicate which areas can and cannot be stretched.

Therefore, when designing bitmaps that will be used on components with variable size, always use nine-patches. To convert a bitmap into a nine-patch, you can start with a regular image (figure 4, shown with in 4x zoom for clarity).



**Figure 4.** button.png

And then run it through the draw9patch utility of the SDK (which is located in the tools/ directory), in which you can mark the areas that should be stretched by drawing pixels along the left and top borders. You can also mark the area that should hold the content by drawing pixels along the right and bottom borders, resulting in figure 5.

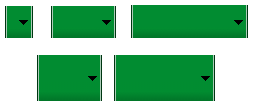


**Figure 5.** button.9.png

Notice the black pixels along the borders. The ones on the top and left borders indicate the places where the image can be stretched, and the ones on the right and bottom borders indicate where the content should be placed.

Also, notice the .9.png extension. You must use this extension, since this is how the framework detects that this is a nine-patch image, as opposed to a regular PNG image.

When you apply this background to a component (by setting android:background="@drawable/button"), the framework stretches the image correctly to accommodate the size of the button, as shown in various sizes in figure 6.



**Figure 6.** A button using the button.9.png nine-patch in various sizes.

### 适配不同的设备密度解决办法

#### THIS LESSON TEACHES YOU TO

This lesson shows you how to support different screen densities by providing different resources and using resolution-independent units of measurements.

#### Use Density-independent Pixels

One common pitfall you must avoid when designing your layouts is using absolute pixels to define distances or sizes. Defining layout dimensions with pixels is a problem because different screens have different pixel densities, so the same number of pixels may correspond to different physical sizes on different devices. Therefore, when specifying dimensions, always use either dp or sp units. A dp is a density-independent pixel that corresponds to the physical size of a pixel at 160 dpi. An sp is the same base unit, but is scaled by the user's preferred text size (it’s a scale-independent pixel), so you should use this measurement unit when defining text size (but never for layout sizes).

For example, when you specify spacing between two views, usedp rather than px:

<Buttonandroid:layout\_width="wrap\_content"  
    android:layout\_height="wrap\_content"  
    android:text="@string/clickme"  
    android:layout\_marginTop="20dp"/>

When specifying text size, always use sp:

<TextViewandroid:layout\_width="match\_parent"  
    android:layout\_height="wrap\_content"  
    android:textSize="20sp"/>

#### Provide Alternative Bitmaps

Since Android runs in devices with a wide variety of screen densities, you should always provide your bitmap resources tailored to each of the generalized density buckets: low, medium, high and extra-high density. This will help you achieve good graphical quality and performance on all screen densities.

To generate these images, you should start with your raw resource in vector format and generate the images for each density using the following size scale:

* xhdpi: 2.0
* hdpi: 1.5
* mdpi: 1.0 (baseline)
* ldpi: 0.75

This means that if you generate a 200x200 image for xhdpi devices, you should generate the same resource in 150x150 for hdpi, 100x100 for mdpi and finally a 75x75 image for ldpi devices.

Then, place the generated image files in the appropriate subdirectory under res/ and the system will pick the correct one automatically based on the screen density of the device your application is running on:

MyProject/

res/

drawable-xhdpi/

awesomeimage.png

drawable-hdpi/

awesomeimage.png

drawable-mdpi/

awesomeimage.png

drawable-ldpi/

awesomeimage.png

Then, any time you reference @drawable/awesomeimage, the system selects the appropriate bitmap based on the screen's dpi.

### 适配特殊屏幕设备的解决办法

#### Declaring an App is Only for Handsets

Because the system generally scales applications to fit larger screens well, you shouldn't need to filter your application from larger screens. As long as you follow the [Best Practices for Screen Independence](http://developer.android.com/guide/practices/screens_support.html#screen-independence), your application should work well on larger screens such as tablets. However, you might discover that your application can't scale up well or perhaps you've decided to publish two versions of your application for different screen configurations. In such a case, you can use the [<compatible-screens>](http://developer.android.com/guide/topics/manifest/compatible-screens-element.html) element to manage the distribution of your application based on combinations of screen size and density. External services such as Google Play use this information to apply filtering to your application, so that only devices that have a screen configuration with which you declare compatibility can download your application.

The [<compatible-screens>](http://developer.android.com/guide/topics/manifest/compatible-screens-element.html) element must contain one or more <screen> elements. Each <screen> element specifies a screen configuration with which your application is compatible, using both the android:screenSizeand android:screenDensity attributes. Each <screen> element **must include both attributes** to specify an individual screen configuration—if either attribute is missing, then the element is invalid (external services such as Google Play will ignore it).

For example, if your application is compatible with only small and normal size screens, regardless of screen density, you must specify eight different <screen> elements, because each screen size has four density configurations. You must declare each one of these; any combination of size and density that you do notspecify is considered a screen configuration with which your application is not compatible. Here's what the manifest entry looks like if your application is compatible with only small and normal screen sizes:

<manifest ... >  
    <compatible-screens>  
        <!-- all small size screens -->  
        <screenandroid:screenSize="small"android:screenDensity="ldpi"/>  
        <screenandroid:screenSize="small"android:screenDensity="mdpi"/>  
        <screenandroid:screenSize="small"android:screenDensity="hdpi"/>  
        <screenandroid:screenSize="small"android:screenDensity="xhdpi"/>  
        <!-- all normal size screens -->  
        <screenandroid:screenSize="normal"android:screenDensity="ldpi"/>  
        <screenandroid:screenSize="normal"android:screenDensity="mdpi"/>  
        <screenandroid:screenSize="normal"android:screenDensity="hdpi"/>  
        <screenandroid:screenSize="normal"android:screenDensity="xhdpi"/>  
    </compatible-screens>  
    ...  
    <application ... >  
        ...  
    <application>  
</manifest>

**Note:** Although you can also use the [<compatible-screens>](http://developer.android.com/guide/topics/manifest/compatible-screens-element.html) element for the reverse scenario (when your application is not compatible with smaller screens), it's easier if you instead use the [<supports-screens>](http://developer.android.com/guide/topics/manifest/supports-screens-element.html) as discussed in the next section, because it doesn't require you to specify each screen density your application supports.

#### Declaring an App is Only for Tablets

If you don't want your app to be used on handsets (perhaps your app truly makes sense only on a large screen) or you need time to optimize it for smaller screens, you can prevent small-screen devices from downloading your app by using the [<supports-screens>](http://developer.android.com/guide/topics/manifest/supports-screens-element.html) manifest element.

For example, if you want your application to be available only to tablet devices, you can declare the element in your manifest like this:

<manifest ... >  
    <supports-screensandroid:smallScreens="false"  
                      android:normalScreens="false"  
                      android:largeScreens="true"  
                      android:xlargeScreens="true"  
                      android:requiresSmallestWidthDp="600"/>  
    ...  
    <application ... >  
        ...  
    </application>  
</manifest>

This describes your app's screen-size support in two different ways:

* It declares that the app does not support the screen sizes "small" and "normal", which are traditionally not tablets.
* It declares that the app requires a screen size with a minimum usable area that is at least 600dp wide.

The first technique is for devices that are running Android 3.1 or older, because those devices declare their size based on generalized screen sizes. The [requiresSmallestWidthDp](http://developer.android.com/guide/topics/manifest/supports-screens-element.html#requiresSmallest) attribute is for devices running Android 3.2 and newer, which includes the capability for apps to specify size requirements based on a minimum number of density-independent pixels available. In this example, the app declares a minimum width requirement of 600dp, which generally implies a 7"-or-greater screen.

Your size choice might be different, of course, based on how well your design works on different screen sizes; for example, if your design works well only on screens that are 9" or larger, you might require a minimum width of 720dp.

The catch is that you must compile your application against Android 3.2 or higher in order to use therequiresSmallestWidthDp attribute. Older versions don't understand this attribute and will raise a compile-time error. The safest thing to do is develop your app against the platform that matches the API level you've set for[minSdkVersion](http://developer.android.com/guide/topics/manifest/uses-sdk-element.html#min). When you're making final preparations to build your release candidate, change the build target to Android 3.2 and add the requiresSmallestWidthDp attribute. Android versions older than 3.2 simply ignore that XML attribute, so there's no risk of a runtime failure.

For more information about why the "smallest width" screen size is important for supporting different screen sizes, read [New Tools for Managing Screen Sizes](http://android-developers.blogspot.com/2011/07/new-tools-for-managing-screen-sizes.html).

**Caution:** If you use the [<supports-screens>](http://developer.android.com/guide/topics/manifest/supports-screens-element.html) element for the reverse scenario (when your application is not compatible with larger screens) and set the larger screen size attributes to "false", then external services such as Google Play **do not** apply filtering. Your application will still be available to larger screens, but when it runs, it will not resize to fit the screen. Instead, the system will emulate a handset screen size (about 320dp x 480dp; see [Screen Compatibility Mode](http://developer.android.com/guide/practices/screen-compat-mode.html) for more information). If you want to prevent your application from being downloaded on larger screens, use [<compatible-screens>](http://developer.android.com/guide/topics/manifest/compatible-screens-element.html), as discussed in the previous section about[Declaring an App is Only for Handsets](http://developer.android.com/guide/practices/screens-distribution.html#FilteringHandsetApps).

Remember, you should strive to make your application available to as many devices as possible by applying all necessary techniques for [supporting multiple screens](http://developer.android.com/guide/practices/screens_support.html). You should use [<compatible-screens>](http://developer.android.com/guide/topics/manifest/compatible-screens-element.html) or [<supports-screens>](http://developer.android.com/guide/topics/manifest/supports-screens-element.html) only when you cannot provide compatibility on all screen configurations or you have decided to provide different versions of your application for different sets of screen configurations.

#### Publishing Multiple APKs for Different Screens

Although we recommend that you publish one APK for your application, Google Play allows you to publish multiple APKs for the same application when each APK supports a different set of screen configurations (as declared in the manifest file). For example, if you want to publish both a handset version and a tablet version of your application, but you're unable to make the same APK work for both screen sizes, you can actually publish two APKs for the same application listing. Depending on each device's screen configuration, Google Play will deliver it the APK that you've declared to support that device's screen.

Beware, however, that publishing multiple APKs for the same application is considered an advanced feature and **most applications should publish only one APK that can support a wide range of device configurations**. Supporting multiple screen sizes, especially, is within reason using a single APK, as long as you follow the guide to [Supporting Multiple Screens](http://developer.android.com/guide/practices/screens_support.html).

If you need more information about how to publish multiple APKs on Google Play, read [Multiple APK Support](http://developer.android.com/google/play/publishing/multiple-apks.html).

### Android 图标设计建议

#### What size are Android Icons?

It is a simple question but even Android, who provide better documentation than most, seems to make you want to work a little for the answer. Fear not, I have done the work for you!

#### Android App Launcher Icon Sizes

|  |  |  |
| --- | --- | --- |
| Android icons require five separate sizes for different screen pixel densities. Icons for lower resolution are created automatically from the baseline. | | |
| mdpi (Baseline): | 160 dpi | 1× |
| hdpi: | 240 dpi | 1.5× |
| xhdpi: | 320 dpi | 2× |
| xxhdpi: | 490 dpi | 3× |
| xxxhdpi: | 640 dpi | 4× |
|  | | |
|  |  |  |

What sizes do I need for Android App launcher icons?

**Quick answer**: 48 px, 72 px, 96 px, 144 px, 192 px& 512 px (for Google Play Store).



##### Launcher icons

48 × 48 (mdpi)  
72 × 72 (hdpi)  
96 × 96 (xhdpi)  
144 × 144 (xxhdpi)  
192 × 192 (xxxhdpi)  
512 × 512 (Google Play store).png

Three-dimensional, front view, with a slight perspective as if viewed from above, so that users perceive some depth.

##### Action bar, Dialog & Tab icons

24 × 24 area in 32 × 32 (mdpi)  
36 × 36 area in 48 × 48 (hdpi)  
48 × 48 area in 64 × 64 (xhdpi)  
72 × 72 area in 96 × 96 (xxhdpi)  
96 × 96 area in 128 × 128 (xxxhdpi).png

These icons are used in the action bar menu. The first number is the size of the icon area, and the second is file size.

##### Small Contextual Icons

16 × 16 (mdpi)  
24 × 24 (hdpi)  
32 × 32 (xhdpi)  
48 × 48 (xxhdpi)  
64 × 64 (xxxhdpi)  
.png

Small icons are used to surface actions and/or provide status for specific items. For example, in the Gmail app, each message has a star icon that marks the message as important.

##### Notification icons

22 × 22 area in 24 × 24 (mdpi)  
33 × 33 area in 36 × 36 (hdpi)  
44 × 44 area in 48 × 48 (xhdpi)  
66 × 66 area in 72 × 72 (xxhdpi)  
88 × 88 area in 96 × 96 (xxxhdpi)  
.pngThese are used to represent application notifications in the status bar. They should be flat (no gradients), white and face-on perspective

#### What about the Google Play Icon?

Google play requires a single additional version of the Icon in 512 x 512 pixels.



Android Play Store Icon Size: 512 x 512 pixels

#### Understanding Android Icon Sizes

Android approach their Icon Size documentation slightly differently to other platforms, concentrating on different pixel densities rather than the simple facts we want. However, it is worth taking a moment to understand why they use this approach.

As technology has progressed, manufacturers have brought out hardware with ever increasing resolutions and pixel density. An increase in pixel density or dots per inch (dpi), basically means more image information can be displayed with.

* MDPI – Medium Pixel Density – also known as the ‘Baseline’
* HDPI – High Pixel Density
* XHDPI – Extra High Pixel Density
* XXHDPI – Extra Extra High Pixel Density

**Note:** Android also supports low-density (LDPI) screens, but you normally don’t need to create custom assets at this size because Android effectively down-scales your HDPI assets by 1/2 to match the expected size.

#### Android User interface Icon Sizes

All [Android icons](http://www.creativefreedom.co.uk/icon-design/app-icons/android-icons/) follow the same scaling rules, so when creating your icon files you must make 5 versions to cover all the platforms available. The table below shows the pixel densities and icon sizes for the different types of user interface icons used by Android:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | MDPI (Baseline) | HDPI | XHDPI | XXHDPI | XXXHDPI |
| Scale | 1 x | 1.5 x | 2 x | 3 x | 4 x |
| DPI | ~ 160 dpi | ~ 240 dpi | ~ 320 dpi | ~ 480 dpi | ~ 640 dpi |
| App Launcher Icons | 48 px | 72 px | 96 px | 144 px | 192 px |
| Action bar Icons | 32 px (24px inset) | 48 px | 64 px | 96 px | 128 px |
| Small / Contextual Icons | 16 px (12px inset) | 24 px | 32 px | 48 px | 64 px |
| Notification Icons | 24 px (22px inset) | 36 px | 48 px | 72 px | 96 px |

#### Looking for gorgeous Custom Android Icons?

Professional custom designed App Icons, stand out in the Play store and get your App noticed. We have a team of 8 amazing Icon designers ready to transform your App.

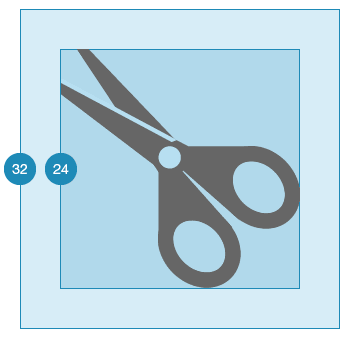
Check out our Android App Icon Design service:

[Android Icon Design](http://www.creativefreedom.co.uk/icon-design/app-icons/android-icons/)

[](http://www.creativefreedom.co.uk/icon-design-projects/app-icon-design/)

Whilst the [user interface icons](http://www.creativefreedom.co.uk/icon-designers-blog/the-different-facets-of-software-icons/) follow the same scaling rules as the app launcher icon, they also have some additional display rules of their own:

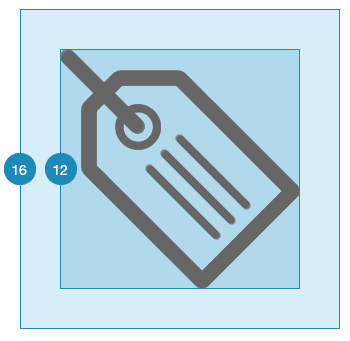
#### Android Action bar Icon Sizes



Action Bar Icons require a file size of 32×32 pixels at the baseline size (MDPI). However, the image itself must be no more than 24×24 pixels centred within the Icon file.

The Icon Design must be single colour #666666 with a transparency set to 60%

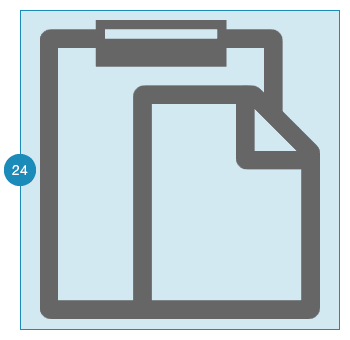
#### Android Small / Contextual Icon Sizes



Action Bar Icons require a file size of 16×16 pixels at the baseline size (MDPI). However, the image itself must be no more than 12×12 pixels centred within the Icon file.

The Icon Design must be single colour #666666 with a transparency set to 60%

#### Android Notification Icon Sizes



Action Bar Icons require a file size of 24×24 pixels at the baseline size (MDPI). However, the image itself must be no more than 22×22 pixels centred within the Icon file.

The Icon Design must be single colour white, #ffffff.

Android’s own icon guide can be found here:

* [Android Iconography](http://www.creativefreedom.co.uk/wp-content/uploads/2013/11/iconography.html)

I hope this Android Icon size guide has been helpful. Please let me know if there is something else you would like me to add or elaborate on. Thanks!

Adam :)

[ANDROID ICON GUIDE](http://www.creativefreedom.co.uk/icon-designers-blog/tag/android-icon-guide/) | [ANDROID ICON SIZES](http://www.creativefreedom.co.uk/icon-designers-blog/tag/android-icon-sizes/) | [ICON SIZE GUIDE](http://www.creativefreedom.co.uk/icon-designers-blog/tag/icon-size-guide/)