SMASHING MAGAZINE

Designing For Android

Imprint

Version 1: September 2012

ISBN: 978-3-943075-44-1

Cover Design: Ricardo Gimenes

PR & Press: Stephan Poppe

eBook Strategy: Talita Telma Stöckle

Technical Editing: Talita Telma Stöckle, Andrew Rogerson

Idea & Concept: Smashing Media GmbH

ABOUT SMASHING MAGAZINE

Smashing Magazine is an online magazine dedicated to Web designers and developers worldwide. Its rigorous quality control and thorough editorial work has gathered a devoted community exceeding half a million subscribers, followers and fans. Each and every published article is carefully prepared, edited, reviewed and curated according to the high quality standards set in Smashing Magazine's own publishing policy. Smashing Magazine publishes articles on a daily basis with topics ranging from business, visual design, typography, front-end as well as back-end development, all the way to usability and user experience design. The magazine is — and always has been — a professional and independent online publication neither controlled nor influenced by any third parties, delivering content in the best interest of its readers. These guidelines are continually revised and updated to assure that the quality of the published content is never compromised.

ABOUT SMASHING MEDIA GMBH

Smashing Media GmbH is one of the world's leading online publishing companies in the field of Web design. Founded in 2009 by Sven Lennartz and Vitaly Friedman, the company's headquarters is situated in southern Germany, in the sunny city of Freiburg im Breisgau. Smashing Media's lead publication, Smashing Magazine, has gained worldwide attention since its emergence back in 2006, and is supported by the vast, global Smashing community and readership. Smashing Magazine had proven to be a trustworthy online source containing high quality articles on progressive design and coding techniques as well as recent developments in the Web design industry.

About this eBook

If you intend to design for Mobile, you might have to pay attention to the democratic scenario that gives shape to this niche. Many developers are aware of this: by using the Android operating system to create apps, they acquire a huge market share. This eBook will guide you through the Android universe and help you to create interface designs for Android mobile gadgets.

Table of Contents

Getting To Know The Android Platform: Building, Testing & Distributing Apps

Juhani Lehtimaki

When iOS started to gain momentum, soon after the first iPhone launched, many businesses started to pay attention to apps. The number of apps for iOS grew exponentially, and every company, big and small, rushed to create their own app to support their business.

For some time, iOS was the only platform you really had to care about. The audience was there. For a few years now, there has been another player in the market. Android's marketshare growth has been phenomenal, and it simply cannot be ignored anymore. There are over 200 million Android users in the world—almost double the number of iOS users. For businesses, reaching the Android crowds is potentially a very lucrative investment.

Android as a platform can appear intimidating to new players. Blogs and media are littered with articles about Android fragmentation and malware. The Android platform can feel complex, although it is very flexible. However, before getting started with an Android project, understanding the platform and ecosystem is imperative. Trying to apply the methods and tools that work on other platforms could lead to disaster.

In this article, we'll explain parts of the application-building process and ecosystem for Android that could cause problems if misunderstood. We'll talk about an approach to building a scalable app that looks and feels right at home on Android, and we'll cover how to test it and your options for distributing it. The following topics would each need a full article to be explained fully, but this article should provide a good overview. After

reading this article, you should have a good understanding of what kinds of decisions and challenges you will face when creating an Android app.

Make The App Scalable

Android devices come in many forms and sizes. The last official count is that 600 Android devices are available, and that number is growing every day. Building an app that runs on all of them is more difficult than building for just one or two screen sizes and one set of hardware. Fortunately, Android was built from the ground up with this in mind. The framework provides tools to help developers tackle the problem. But as with all tools, they only work if used correctly.

Large preview.

An iOS app is designed and built by placing pixels at the proper coordinates until the UI looks just right. Not so on Android! Android designers must think about the scalability of each component and the relationships between components. The philosophy is much closer to Web app design than to iOS app design.

A CONTINUUM INSTEAD OF A SEPARATE TABLET UI

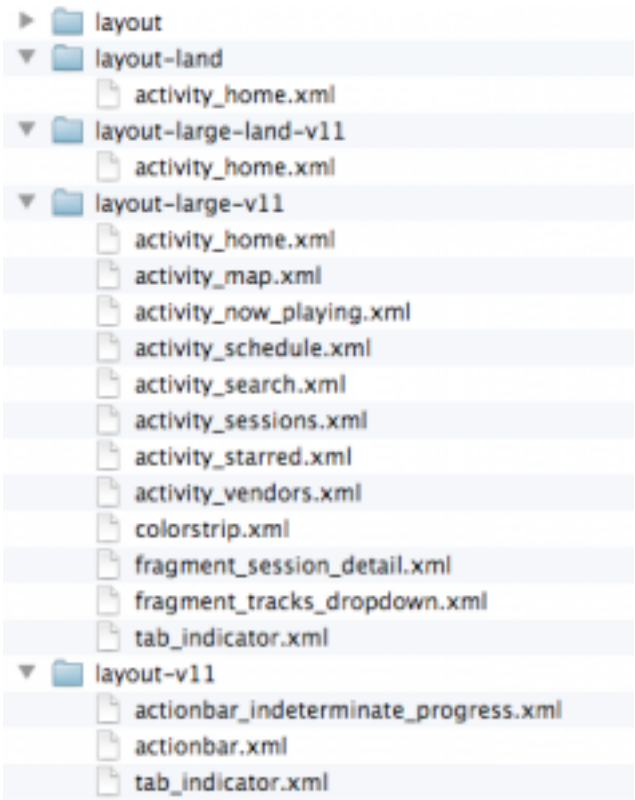
About half a year ago, Google rushed out the Android version named Honeycomb (3.0). Honeycomb was aimed at tablets and was never meant for anything else. The source code of Honeycomb was never released, and it never officially appeared on any phones. At the time, Apple had already established a practice by which developers provided two separate versions of their app, one for iPhone and one for iPad. Because of Apple's model and the separate Android version for tablets, everyone seemed to assume that two separate versions of an app are needed on Android, too. Soon, the Internet was full of blog posts complaining that Android didn't have enough tablet apps and that there was no way to search for them on the Google Play store.

Now, as Android Ice Cream Sandwich (4.0) is unifying all Android devices to run the same version of the OS, it all makes sense. Android is a continuum, and drawing a clear line between tablets and phones is impossible. In fact, checking whether an app is running on a tablet or phone is technically impossible. Checking the screen size (and many other features) at runtime, however, is possible.

This is where Android design starts to remind us of Web design. New technologies have enabled us to build websites that adapt automatically to the user's browser size by scaling and moving components around as needed. This approach is called responsive Web design. The very same principles can be used on Android. On Android, however, we are not bound by the limits of the browser. Responsive design can be taken even further.

RESPONSIVE ANDROID DESIGN

Android developers can define multiple layouts for every screen of their app, and the OS will pick the best-fitting one at runtime. The OS knows which one fits best by using definitions that developers add to their layout (and other) folders in the app's project resource tree.



An example of the structure of layout folders, which distinguish between screen sizes and Android versions.

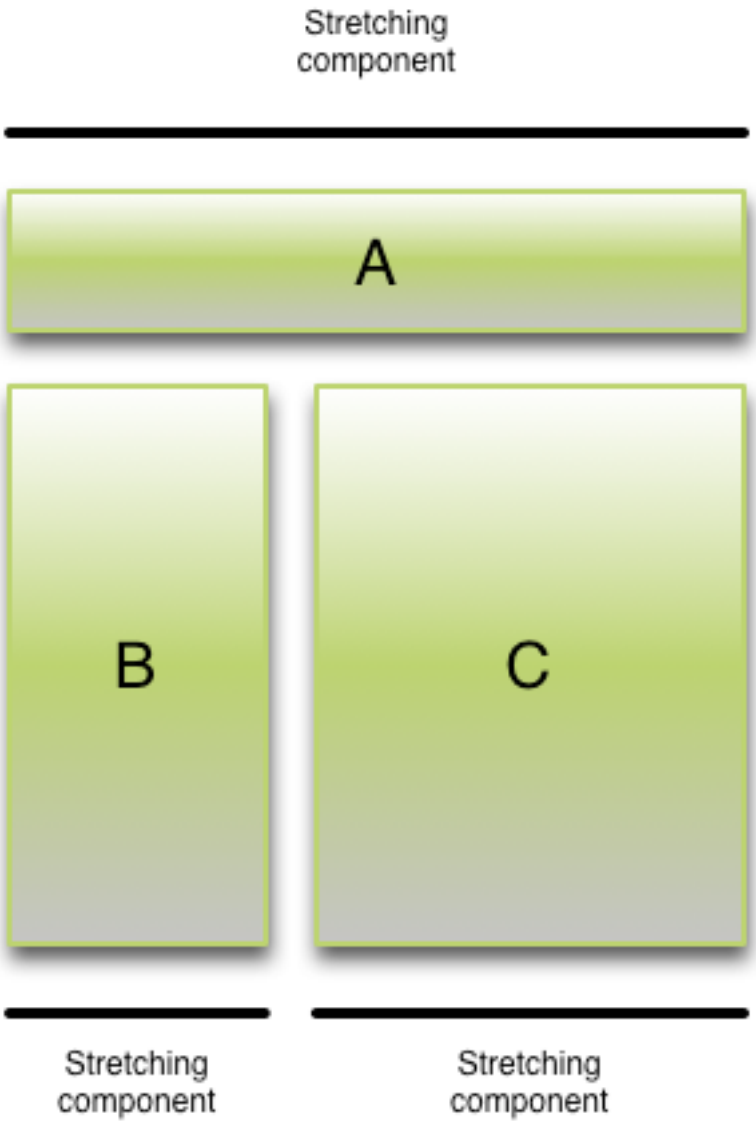
Starting from Android version 3.2 — and, therefore, also on Ice Cream Sandwich — a more fine-grained approach was introduced. Developers may now define layouts based on the screen's pixel density, independent of size, instead of using only the few categories that were available before.

```
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)
```

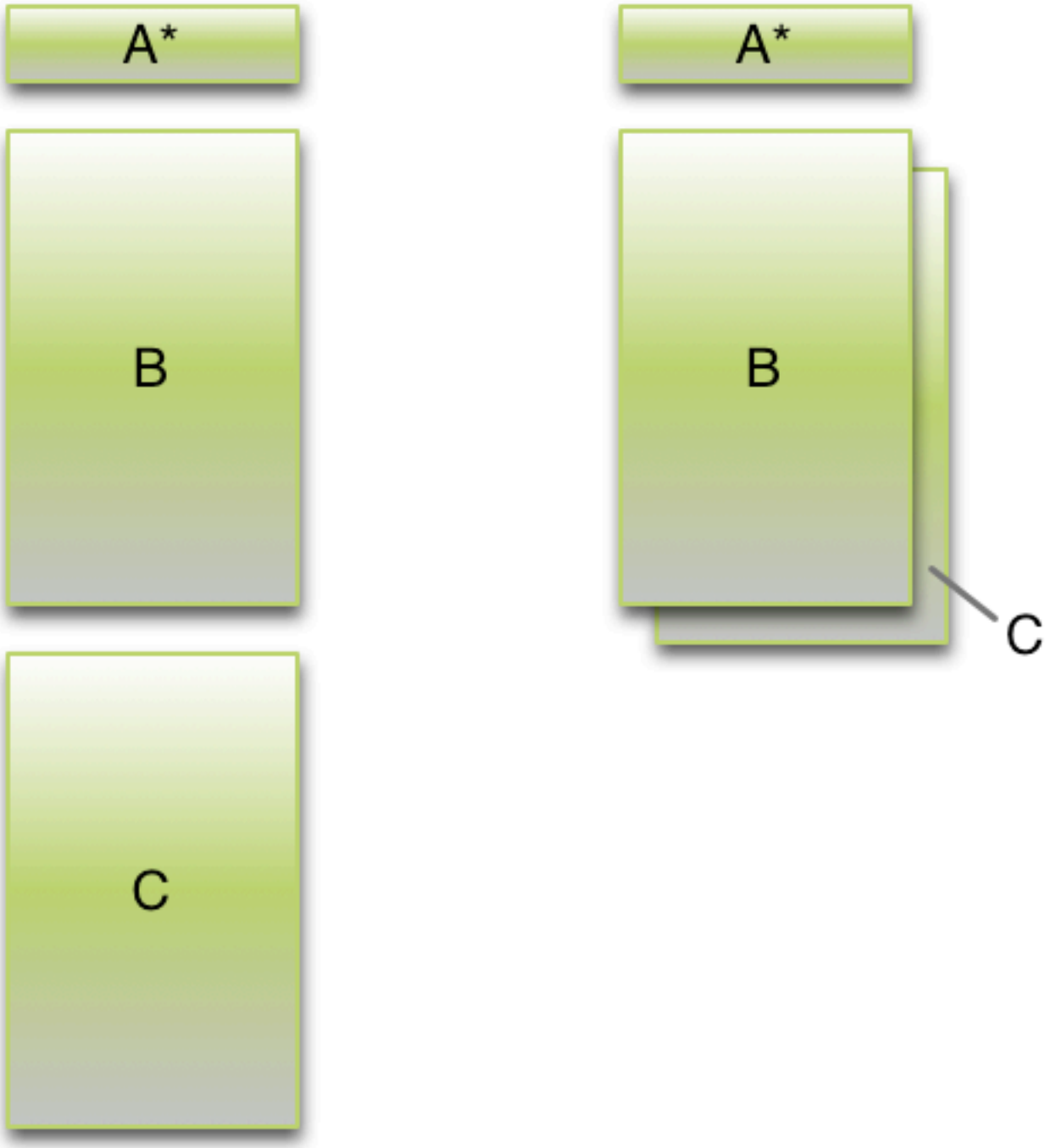
An example of the new layout specifications based on screen size. It is very similar to CSS' media queries. Android's documentation has more details.

USING FRAGMENTS TO IMPLEMENT RESPONSIVE DESIGN

Fragments are the building blocks of Android UIs. They can be programmed either to be standalone screens or to be displayed with other fragments; but the most powerful ones are both, depending on the device that the app is running on. This enables us not only to rearrange the fragments but to move them deeper into the activity stack. Dan McKenzie has written about issues related to designing for big Android screens.



Each component is itself stretchable and scales to screens with similar sizes.



When a screen's size is drastically different, the components need to be rearranged. They can be rearranged on the same level or moved deeper into the activity stack.

Make The App Look And Feel Android-Like

Consistency with other apps on the same platform is more important for an app's look and feel than consistency with the same developer's apps on other platforms. Having the look and feel of apps from a different platform will make the app feel foreign and make users unhappy.

(Remember to read Google's Android Design guidelines.)

TABS

In Android apps, tabs should always be on top. This convention was established and is driven by Google's design of its apps and by guidelines from advocates of Android development. Putting the tabs on top makes scaling an app to larger screen sizes easier. Putting tabs at the bottom of a tablet-sized UI wouldn't make sense.

Large preview.

Navigating between top-positioned tabs on a phone with a large screen can be difficult, especially when the person is using only one hand. The solution is to enable the user to swipe between tabs. This interaction model is not new, but in its latest release, Google has made it commonplace in Android apps. All bundled apps now support this interaction on tabbed UIs, and users will expect it to work in your app's tabbed screens, too.

ANDROID UI PATTERNS CAN PUT USERS AT EASE

Some UI patterns have become popular on Android — so much so that they are starting to define the look of Android apps. The action bar, one of the most popular patterns, is now part of Android's core libraries and can be used in any app running on Android 3.0 and up.

Good third-party libraries are available to bring the action bar to apps that run on older versions of Android. ActionBarSherlock is very stable and supports multiple versions and even automatically uses the native action bar when it detects a supported version of Android.

Another popular UI pattern is the dashboard. Many apps with a lot of functionality use the dashboard as their landing screen to give users a clear overview of and easy access to the app's most important functionality.

Large preview.

Google Play (left) and Evernote (right) both put an action bar at the top of their screens to provide quick access to contextually relevant actions. Evernote's landing screen clearly tells the user what they can do with the app, while providing easy access to those actions every time the app launches.

See Dan McKenzie's article "Designing for Android" for more on the look and feel of Android apps.

Integrate The App With Other Apps

The Android platform provides a powerful mechanism for apps to extend each other's functionality. This mechanism is called "intents." Apps can register to receive and launch intents. When an app registers to receive intents, it must tell the system what kind of intents it can handle. Your app could, for example, tell the system that it can show pictures or open Web page URLs. Now, whenever another app launches an intent to view an image or a Web page, the user has the option to choose your app to complete the action.

Large preview.

SOCIAL NETWORK INTEGRATION

On other mobile platforms, if an app wants to share something to Twitter, Facebook or another social network, it implements the sharing mechanisms internally in the app. Sharing requires a separate operation for each social network. On Android, this can be achieved more easily using intents. An app may launch an intent telling the system that it wants to share an image or text. Depending on which apps the user has installed, the user will be provided with a list of apps that can handle the operation. If the user chooses a Twitter or Facebook client, the client will open to its sharing screen with the text or image prefilled.

Large preview.

There are many benefits to integrating with social networks using intents rather than implementing sharing directly from your app:

1. Close to zero effort is required to build the functionality.
2. Users don't have to log into a separate application. The social network's app takes care of logging in.
3. You don't have to limit the social networks that users may use to share from your app. All apps installed on the user's device are available to be used.
4. If a social network's sharing protocol changes, you don't have to worry about it. That service's app will be updated to reflect the changes.
5. Users might be using an unofficial app for a social network. Using intents, they may continue using their app of choice with the interface they are familiar with.
6. The intents mechanism offers only options that the user actually uses (i.e. the apps that they have installed). No need to offer Facebook sharing to someone who doesn't have a Facebook account.

THINK OF OTHER OPPORTUNITIES

Extending the functionality of other apps via the intents system will benefit your app, too. Perhaps your app wouldn't get used every day and would get buried under apps that are used more often. But if your app extends the functionality of other apps and keeps popping up as an option every time the user wants to perform an action that your app can handle, then it will be thought of more by users.

Intents have limitless possibilities. You can build your own intents hierarchy to extend certain functionality to other apps, in effect providing an API that is easy to use and maintain. You are essentially recommending to users other apps that complement yours and, in turn, extending your app's features without having to write or maintain any code. The intents system is one of the most powerful features of the Android platform.

Quality Control

With the massive number of devices, testing an Android app is much more difficult than testing an iOS app. This is where the fragmentation causes the most problems. Testing on one or two devices is not enough; rather, you have to test on a variety of screen sizes, densities and Android versions.

In addition to what you would normally test on any other platform, you should the following:

- Test your app thoroughly on the lowest Android version that it runs on. Accidentally using an API that isn't actually available at runtime on some devices is easy.
- Test that the search button works on all relevant screens.

- Make sure that the D-pad and trackball navigation work on all screens.
- Test all supported screen densities, or at least extra-high, high and medium. Low-density devices can be difficult to find.
- Test on at least one tablet device. But try to test on as many screen sizes as possible.

TESTING IN THE CLOUD

New services are popping up to ease the pain of testing on multiple devices. Services such as Testdroid enable developers to test their apps on multiple real devices through a Web interface. Simply upload your app's package and automated testing script, and the service executes your scripts on dozens of devices. Results can be viewed in a Web browser. Examining screenshots from different devices is even possible, to ensure pixel-perfect UIs.

Testdroid is a cloud service for testing Android apps on multiple devices. Large preview.

Distribute The App

Once your app is tested and ready, you need to get it to users. You'll have to choose how to do it. Very few Android devices are restricted to one app store. The overwhelming majority of Android devices ship with Google Play, which is the most important route to reaching users on the platform.

GOOGLE PLAY

The Google Play store doesn't have a formal process for approving apps. Any application package uploaded to Play will appear in the store's listings to users. App guidelines do exist, but they are enforced only if there are complaints, and even then pretty randomly. This means that your app will be swamped by hundreds of other apps of varying quality.

So, how to rise above the masses and get the attention of users?

The first 30 days are important! Your app will appear in the listing for new paid or free apps during that time. Ranking relatively high in this listing during this time is much easier than ranking high in the overall top lists. Make sure that your app's website links to Google Play from the start, and use all social networks to tell people about your app's launch.

Getting recognized as a trusted brand is difficult. Google Play contains many apps that use registered trademarks without permission. Users have come to learn that a logo is no indication that an app was actually produced by that logo's company. To increase trust, make sure the "Visit Developer's Website" link points to the official website, and if possible link back to your app from there.

Top new apps on Google Play. Large preview.

Making an app work on all devices is sometimes impossible. Some devices lack the required hardware or simply run an old version of Android for which the required APIs don't exist. You can list all of the requirements in the app's manifest file, telling Google Play which devices the app is meant for and, thus, hiding it from listings that are being viewed on incompatible devices. But sometimes even that isn't enough. In these cases, Google Play allows developers to prevent certain devices from downloading their app. While this option should be used only as a last resort, it is still better than allowing users to download something that you know does not work on their device.

ALTERNATIVE APP STORES

Google Play is not the only place to distribute your app. Amazon's Appstore has lately gained attention due to the launch of Amazon's Android-based Kindle Fire tablet. Amazon's approach is fairly similar to Apple's in that it has a formal review process. The Appstore is also accessible to non-Amazon devices, but currently only in the US.

Multiplatform app store GetJar also distributes Android apps. GetJar has a lot of users and is a well-known and trusted source, especially among people with not-so-smart phones.

Barnes & Noble's app store is a US-only eBook-based app store. Unlike Amazon's, it is accessible only to B&N's Android hardware.

MULTIPLE APP STORES, JUST ONE, OR NONE?

Many people's first instinct is to try to get their app into all stores. This decision should not be made lightly, though. Distributing through multiple stores might make the app reach more potential users. However, being spread across multiple app stores could prevent the app from ranking as high as it could in the listings for downloads and ratings. Having a thousand installations across three app stores might sound better than having two thousand installations in one store, but maybe those two thousand would push the app into a more visible spot in the store and help it rocket to tens of thousands of installations later.

An app doesn't have to be in a store at all in order to be installed on devices. Android apps can be installed directly from websites or by transferring them from computer to phone. While you wouldn't reach the same audience and wouldn't benefit from the update mechanisms in app stores, there is definitely a place for direct distribution. Using forums and websites, developers can distribute their apps to alpha and beta communities without having to risk their reputation or low ratings in an app store. Distributing a major update or an unstable build to a limited number of dedicated testers and fans might be worth the extra effort.

Conclusion

Building a scalable and functional Android app is not impossible, but it requires careful planning and an understanding of the target platform. A blind approach or simply borrowing a design from another platform would likely end in failure. Achieving a successful end requires that you use Android's tools correctly and follow the right design approach. Writing an Android app takes effort, but if done right, the app could reach a massive numbers of users.

Designing For Android

Dan McKenzie

For designers, Android is the elephant in the room when it comes to app design. As much as designers would like to think it's an iOS world in which all anyones cares about are iPhones, iPads and the App Store, nobody can ignore that Android currently has the majority of smartphone market share and that it is being used on everything from tablets to e-readers. In short, the Google Android platform is quickly becoming ubiquitous, and brands are starting to notice.

But let's face it. Android's multiple devices and form factors make it feel like designing for it is an uphill battle. And its cryptic documentation is hardly a starting point for designing and producing great apps. Surf the Web for resources on Android design and you'll find little there to guide you.

If all this feels discouraging (and if it's the reason you're not designing apps for Android), you're not alone. Fortunately, Android is beginning to address the issues with multiple devices and screen sizes, and device makers are slowly arriving at standards that will eventually reduce complexity.

This article will help designers become familiar with what they need to know to get started with Android and to deliver the right assets to the development team. The topics we'll cover are:

- Demystifying Android screen densities,
- Learning the fundamentals of Android design via design patterns,
- Design assets your developer needs,
- How to get screenshots,
- What Android 3 is about, and what's on the horizon.

Android Smartphones And Display Sizes

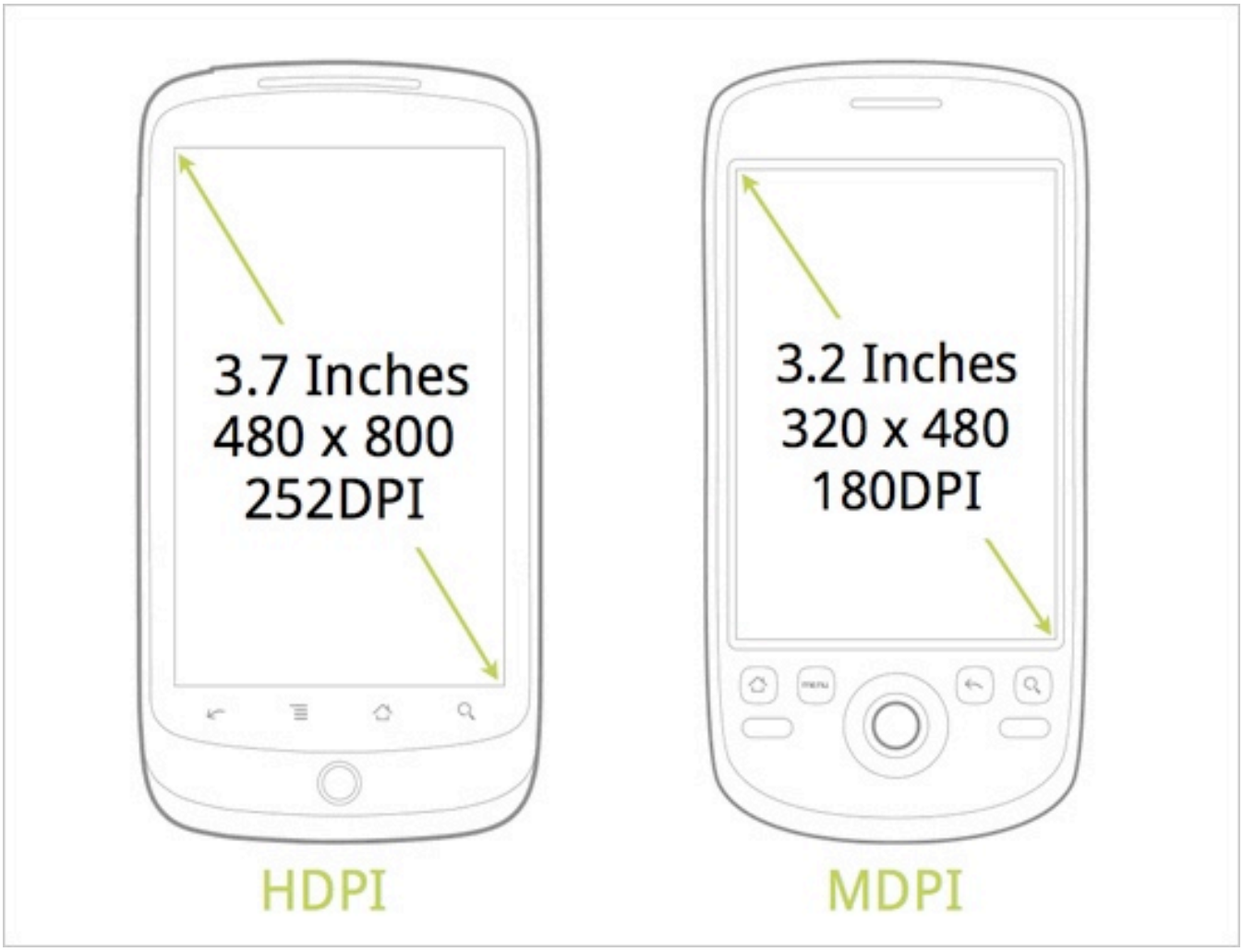
When starting any digital design project, understanding the hardware first is a good idea. For iOS apps, that would be the iPhone and iPod Touch. Android, meanwhile, spans dozens of devices and makers. Where to begin?

The old baseline for screens supported for Android smartphone devices was the T-Mobile G1, the first commercially available Android-powered device which has an HVGA screen measuring 320 x 480 pixels.

HVGA stands for “half-size video graphics array” (or half-size VGA) and is the standard display size for today’s smartphones. The iPhone 3GS, 3G and 2G use the same configuration.

T-Mobile G1, the first commercially available Android device and the baseline for Android screen specifications.

To keep things simple, Android breaks down physical screen sizes (measured as the screen's diagonal length from the top-left corner to bottom-right corner) into four general sizes: small, normal, large and xlarge.



Two common Android screen sizes. (Image from Google I/O 2010)

320 × 480 is considered a “normal” screen size by Android. As for “xlarge,” think tablets. However, the most popular Android smartphones today have WVGA (i.e. wide VGA) 800+ × 480-pixel HD displays. So, what’s “normal” is quickly changing. For now, we’ll say that most Android smartphones have large screens.

	Low density (120), *ldpi*	Medium density (160), *mdpi*	High density (240), *hdpi*	Extra high density (320), *xhdpi*
Small screen	**QVGA (240x320)**		480x640	
Normal screen	**WQVGA400 (240x 400)** **WQVGA (240x432)**	**HVGA (320x480)**	**WVGA (480x800)** **WVGA854 (480x854)** 600x1024	640x960
Large screen	**WVGA800 (480x800)** **WVGA854 (400x854)**	**WVGA800 (480x800)** **WVGA854 (480x854)** 600x1024		
Extra large screen	1024x600	**WXGA (1280x800)** 1024x768 1280x768	1536x1152 1920x1152 1920x1200	2048x1536 2560x1536 2560x1600

Diagram of various screen configurations available from emulator skins in the Android SDK. (Image: Android Developers website)

The variety of display sizes can be challenging for designers who are trying to create one-size-fits-all layouts. I've found the best approach is to design one set of layouts for 320 x 533 physical pixels and then introduce custom layouts for the other screen sizes.

While this creates more work for both the designer and developer, the larger physical screen size on bigger devices such as the Motorola Droid and HTC Evo might require changes to the baseline layouts that make better use of the extra real estate.

WHAT YOU NEED TO KNOW ABOUT SCREEN DENSITIES

Screen sizes are only half the picture! Developers don't refer to a screen's resolution, but rather its density. Here's how Android defines the terms in its Developers Guide:

- **Resolution**
  The total number of physical pixels on a screen.
- **Screen density**
  The quantity of pixels within a physical area of the screen, usually referred to as DPI (dots per inch).

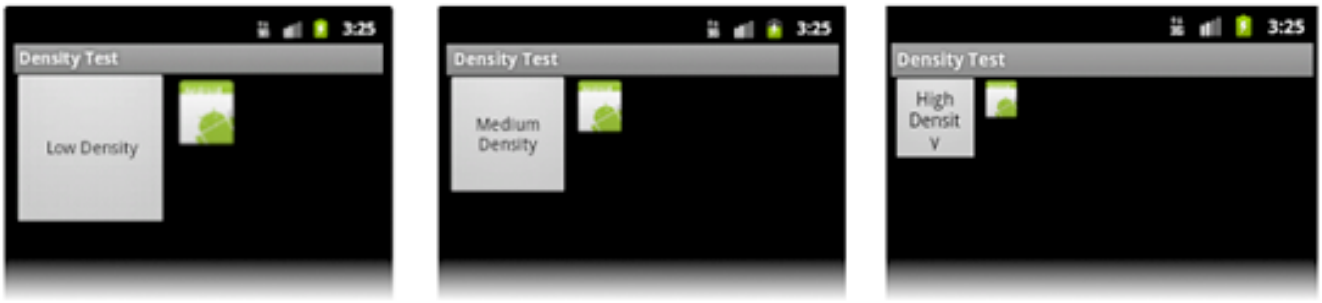
- **Density-independent pixel (DP)**
  This is a virtual pixel unit that you would use when defining a layout's UI in order to express the layout's dimensions or position in a density-independent way. The density-independent pixel is equivalent to one physical pixel on a 160 DPI screen, which is the baseline density assumed by the system of 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: pixels = DP * (DPI / 160). For example, on a 240 DPI screen, 1 DP equals 1.5 physical pixels. Always use DP units when defining your application's UI to ensure that the UI displays properly on screens with different densities.

It's a bit confusing, but this is what you need to know: Like screen sizes, Android divides screen densities into four basic densities: ldpi (low), mdpi (medium), hdpi (high), and xhdpi (extra high). This is important because you'll need to deliver all graphical assets (bitmaps) in sets of different densities. At the very least, you'll need to deliver mdpi and hdpi sets for any smartphone apps.

What this means is all bitmap graphics need to be scaled up or down from your baseline (320 x 533) screen layouts (note: there is also a way for parsing SVG files that provides a way to scale vector art on different screens sizes and densities without loss of image quality).

The bitmap requirement is similar to preparing graphics for print vs. the Web. If you have any experience with print production, you'll know that a 72 PPI image will look very pixelated and blurry when scaled up and printed. Instead, you would need to redo the image as a vector image or use a high-resolution photo and then set the file's resolution at around 300 PPI in order to print it without any loss of image quality. Screen density for Android works similar, except that we're not changing the file's resolution, only the image's size (i.e. standard 72 PPI is fine).

Let's say you took a bitmap icon measuring 100 × 100 pixels from one of the screens of your baseline designs (remember the "baseline" is a layout set at 320 × 480). Placing this same 100 × 100 icon on a device with an lDPI screen would make the icon appear big and blurry. Likewise, placing it on a device with an hDPI screen would make it appear too small (due to the device having more dots per inch than the mDPI screen).



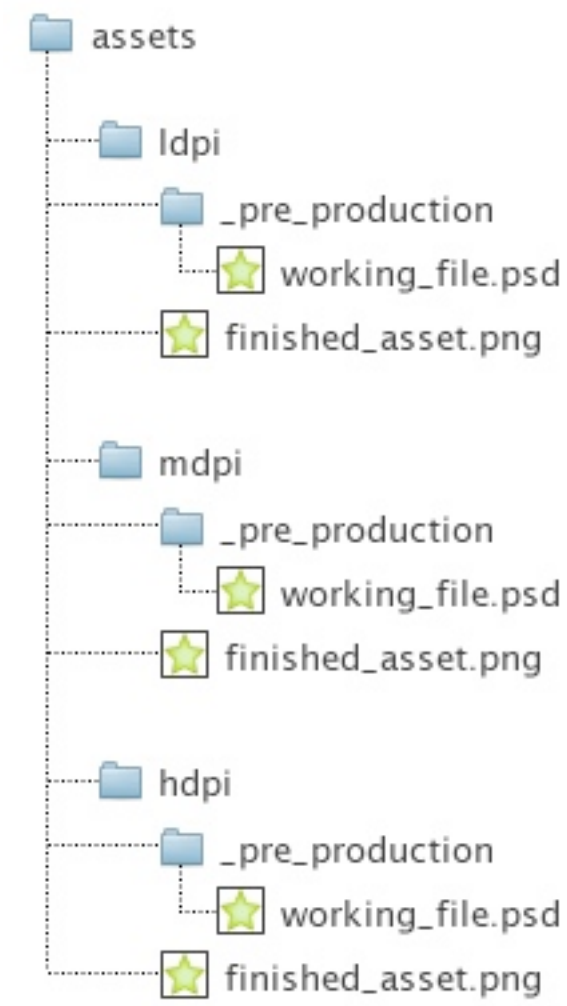
An application without density support. (Image: Android Developers website)

To adjust for the different device screen densities, we need to follow a 3:4:6:8 scaling ratio between the four density sizes. (For the iPhone, it's easy: it's just a 2:1 ratio between the iPhone 4 and 3GS.) Using our ratios and some simple math, we can create four different versions of our bitmap to hand off to our developer for production:

- 75 × 75 for low-density screens (i.e. ×0.75);
- 100 × 100 for medium-density screens (our baseline);
- 150 × 150 for high-density screens (×1.5);
- 200 × 200 for extra high-density screens (×2.0). (We're concerned with only lDPI, mDPI and hDPI for Android smartphone apps.)

The final graphic assets would appear like this using the four different screen densities.

After you've produced all of your graphics, you could organize your graphics library as follows:



The suggested organization and labeling of asset folders and files. In preparing our star graphic, all file prefixes could be preceded by the name ic_star, without changing the names of the respective densities.

You might be confused about what PPI (pixels per inch) to set your deliverables at. Just leave them at the standard 72 PPI, and scale the images accordingly.

Using Android Design Patterns

Clients often ask whether they can use their iPhone app design for Android. If you're looking for shortcuts, building an app for mobile Web browsers using something like Webkit and HTML5 is perhaps a better choice. But to produce a native Android app, the answer is no. Why? Because Android's UI conventions are different from iPhone's.

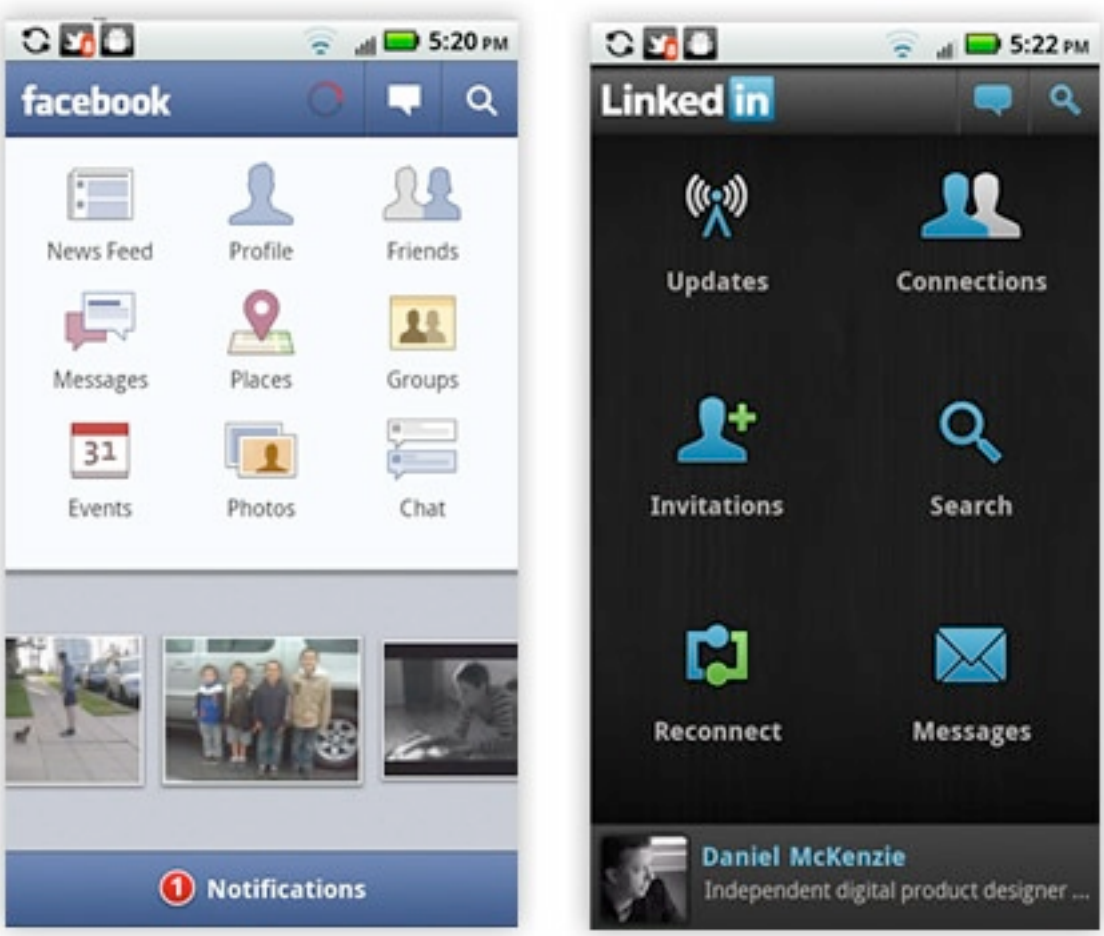
The big difference is the "Back" key, for navigating to previous pages. The Back key on Android devices is fixed and always available to the user, regardless of the app. It's either a physical part of the device or digitally fixed to the bottom of the screen, independent of any app, as in the recently released Android 3.0 for tablets (more on this later).

The hard “Back” key on a smartphone running Android 2.0.

The presence of a Back key outside of the app itself leaves space for other elements at the top of the screen, such as a logo, title or menu. While this navigational convention differs greatly from that of iOS, there are still other differentiators that Android calls “design patterns.” According to Android, a design pattern is a “general solution to a recurring problem.” Below are the main Android design patterns that were introduced with version 2.0.

DASHBOARD

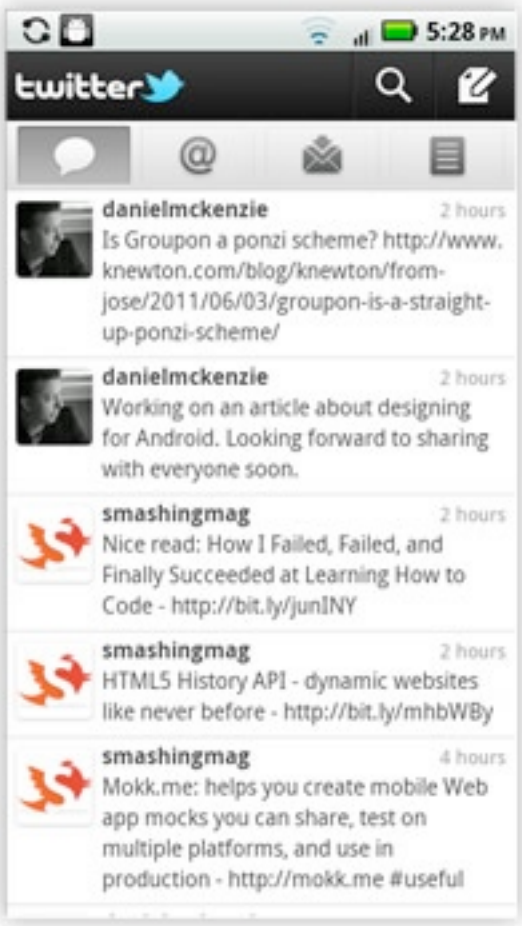
This pattern solves the problem of having to navigate to several layers within an app. It provides a launch pad solution for rich apps such as Facebook, LinkedIn and Evernote.



The dashboard design pattern, as used by Facebook and LinkedIn.

ACTION BAR

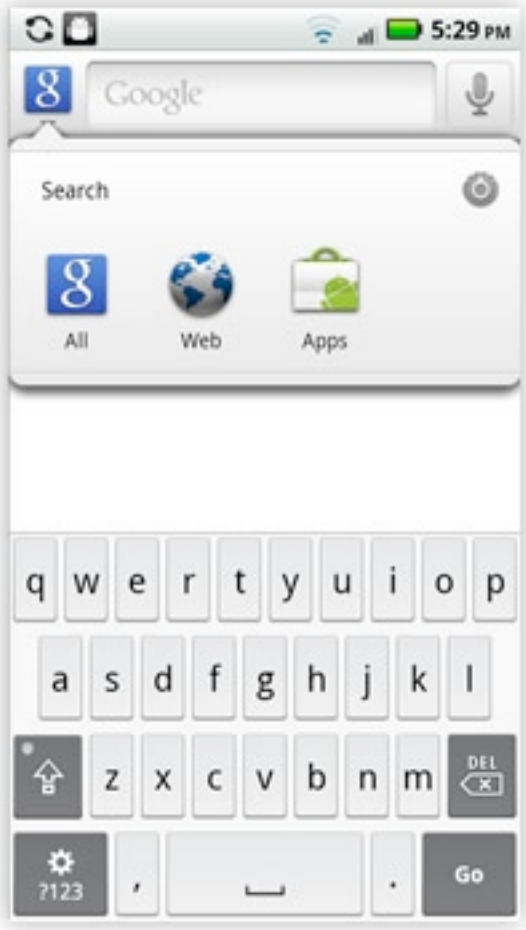
The action bar is one of Android's most important design patterns and differentiators. It works very similar to a conventional website's banner, with the logo or title typically on the left and the navigation items on the right. The action bar's design is flexible and allows for hovering menus and expanding search boxes. It's generally used as a global feature rather than a contextual one.



The action bar design pattern as used by Twitter.

SEARCH BAR

This gives the user a simple way to search by category, and it provides search suggestions.



The search bar design pattern as used in the Google Search app.

QUICK ACTIONS

This design pattern is similar to iOS' pop-up behavior that gives the user additional contextual actions. For example, tapping a photo in an app might trigger a quick action bar that allows the user to share the photo.

The quick action design pattern as used by Twitter.

COMPANION WIDGET

Widgets allow an app to display notifications on the user's launch screen. Unlike push notifications in iOS, which behave as temporary modal dialogs, companion widgets remain on the launch screen. (Tip: to select a widget for your Android device, simply tap and hold any empty space on one of the launch screens.)

Companion widgets by Engadget, New York Times and Pandora.

Using established design patterns is important for keeping the experience intuitive and familiar for your users. Users don't want an iPhone experience on their Android device any more than a Mac user wants a Microsoft experience in their Mac OS environment. Understanding design patterns is the first step to learning to speak Android and designing an optimal experience for its users. Your developers will also thank you!

Android Design Deliverables

OK, so you've designed your Android app and are ready to make it a reality. What do you need to hand off to the developer? Here's a quick list of deliverables:

1. Annotated wireframes of the user experience based on the baseline large screen size of 320 x 533 physical pixels. Include any additional screens for instances where a larger or smaller (320 x 480) screen size requires a modified layout or a landscape version is required.
2. Visual design mockups of key screens for WVGA large size (320 x 533) screens (based on a WVGA 800 x 480 hdpi physical pixel screen size) in addition to any custom layouts needed for other screen sizes.
3. Specifications for spacing, font sizes and colors, and an indication of any bitmaps.
4. A graphics library with lDPI, mDPI and hDPI versions of all bitmaps saved as transparent PNG files.
5. Density-specific app icons, including the app's launch icon, as transparent PNG files. Android already provides excellent tips for designers on this topic, along with some downloads, including graphic PSD templates and other goodies.

How To Take Screenshots

Your product manager has just asked for screenshots of the developer's build. The developer is busy and can't get them to you until tomorrow. What do you do?! As of this writing, Android has no built-in way to take screenshots (bummer, I know). The only way is to just deal with it, and that means pretending to be a developer for a while and downloading some really scary software. Let's get started!

The following software must be downloaded:

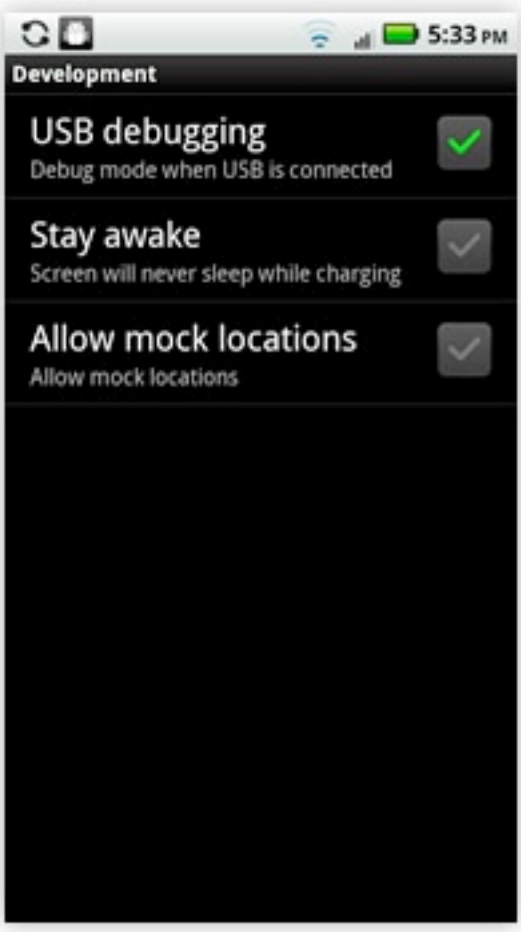
1. All USB drivers for your Android device,
2. Android software development kit (SDK),
3. Java SE SDK

Then, on your computer:

1. Extract the USB drivers to a folder on your desktop,
2. Extract the Android SDK to a folder on your desktop,
3. Install the Java SE SDK.

On your Android device:

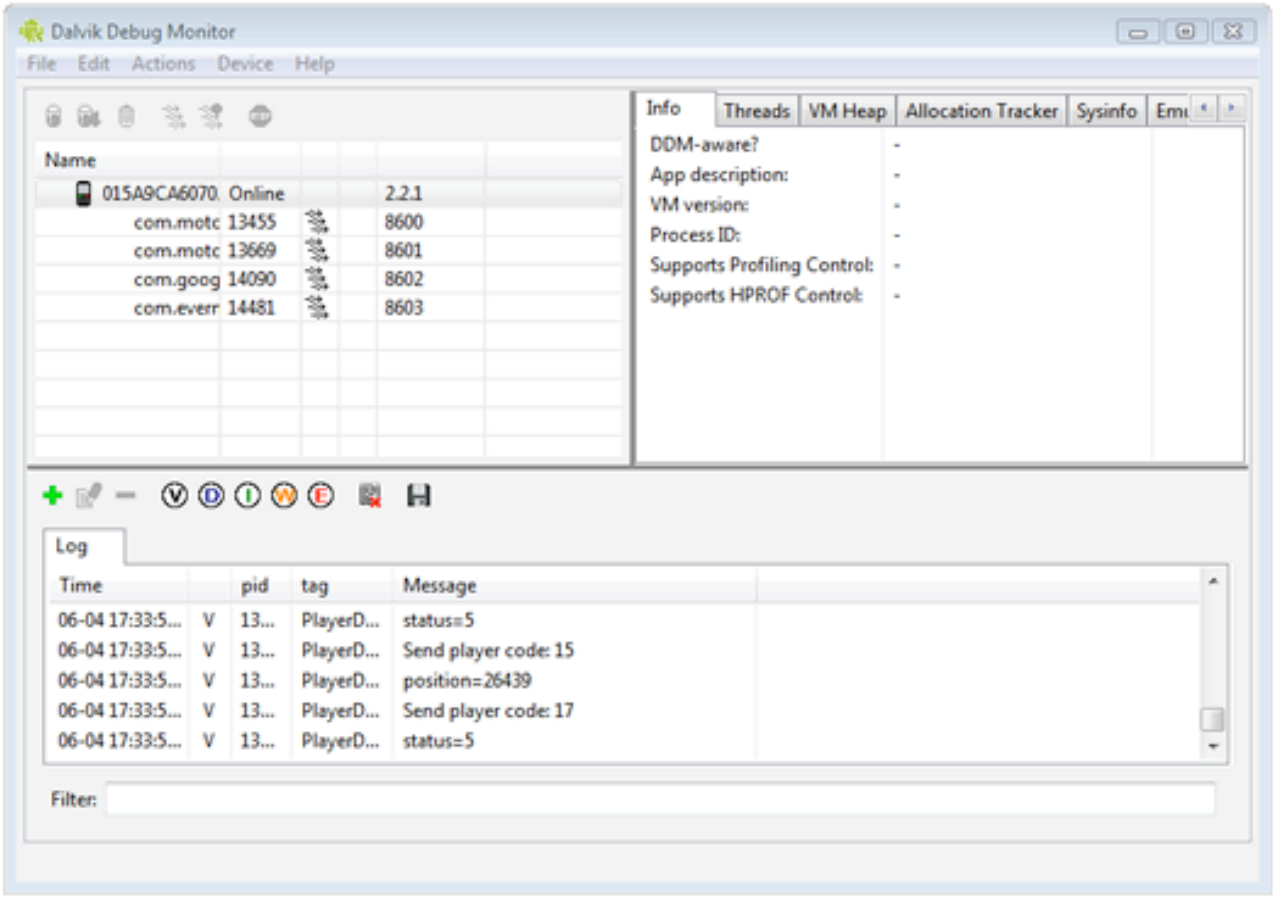
1. Open "Settings" (you'll find it in the apps menu),
2. Tap on "Applications,"
3. Tap on "Development,"
4. Check the box for "USB debugging."



Now, for the fun part:

1. Connect your Android device to your computer via USB. Windows users: allow Windows to install all drivers. One of the drivers may not be found and will require you to go to the Window's Device Manager under the Control Panel. There, you can locate the device (having a yellow warning icon next to it) and right-click on it.
2. Choose to "update/install" the driver for your device.
3. Go to your desktop. Open the Android SDK folder and select *SDK Setup.exe*.

4. Allow it to automatically refresh its list of the operating system SDKs that are available, and select to install all packages.
5. Once finished, exit the application.
6. Go back to the opened Android SDK folder on your desktop, and open the “Tools” folder.
7. Click on the file *ddms* to open the Dalvik Debug Monitor.
8. Select your device from the “Name” pane.
9. In the application’s top menu, open the “Device” menu, and choose “Screen capture…” A Device Screen Capture window will open, and you should see the launch screen of your Android device.



The Dalvik Debut Monitor.

To navigate:

1. Grab your Android device and navigate to any page. Go back to your computer and select “Refresh” in the Device Screen Capture window. The current screen from your Android device should appear.
2. If you’re on a Mac, you can just do the old Shift + Command + 4 trick to take a screenshot. In Windows, you can copy and paste it into one of the Windows media applications.

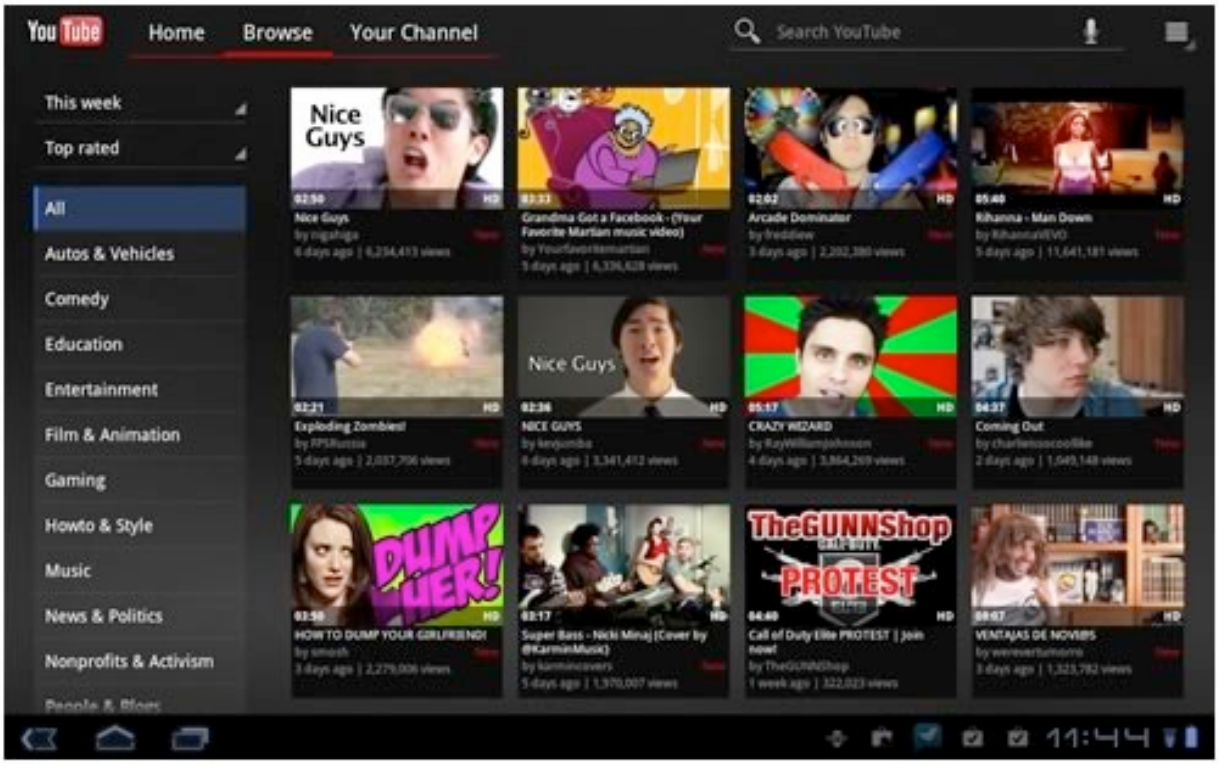
About Android Tablets

At CES 2011, companies rained down Android tablets, with an array of screen sizes. However, after a quick review of the most popular ones, we can conclude that the two important screen sizes to focus on in terms of physical pixels are 1280 × 800 and 800 × 480.

With the Android 3.0 Honeycomb release, Google provided device makers with an Android UI made for tablets. Gone is the hard “Back” button, replaced by an anchored software-generated navigation and system status bar at the bottom of the screen.

The anchored navigation and system bar in Android 3.0.

Android 3.0 got a visual refresh, while incorporating all of the design patterns introduced in Android 2.0. One of the major differences with 3.0 is the Action Bar which has been updated to include tabs, drop-down menus or breadcrumbs. The action bar can also change its appearance to show contextual actions when the user selects single or multiple elements on a screen.



The new action bar with tabs, introduced in Android 3.0.

Another new feature added to the Android framework with 3.0 is a mechanism called "fragments." A fragment is a self-contained component in a layout that can change size and position depending on the screen's orientation and size. This further addresses the problem of designing for multiple form factors by giving designers and developers a way to make their screen layout components elastic and stackable, depending on the screen limitations of the app. Screen components can be stretched, stacked, expanded and collapsed, and revealed and hidden.

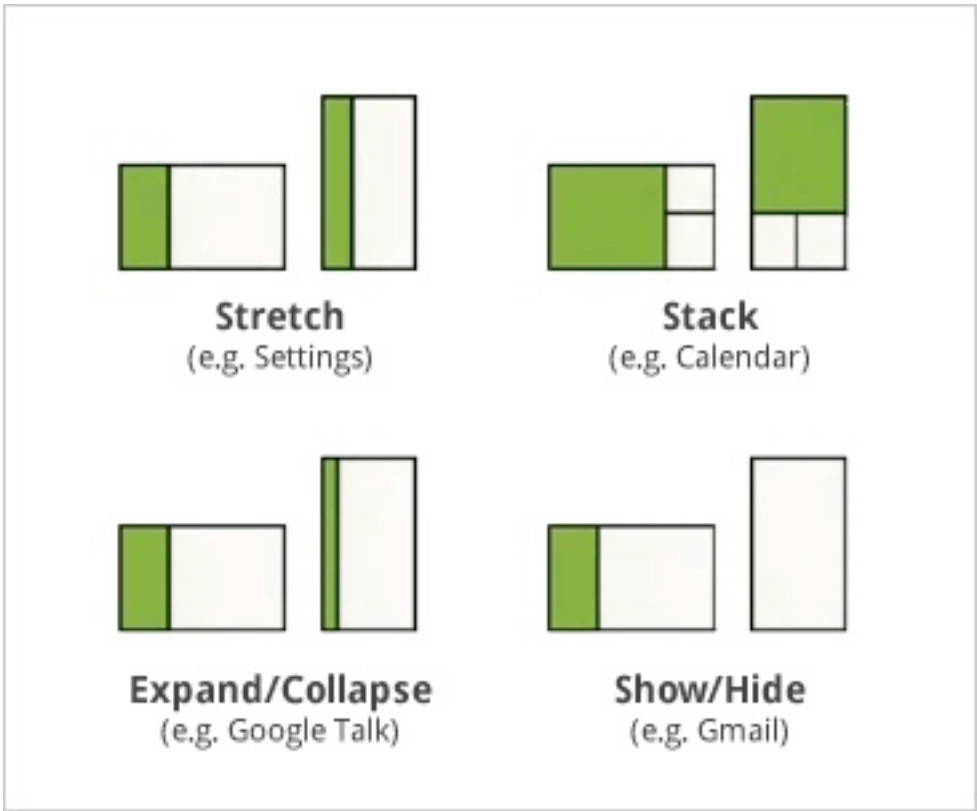


Diagram showing examples of how fragments can be used.

The next Android release, scrumptiously dubbed Ice Cream Sandwich, promises to bring this functionality to Android smartphones as well, giving designers and developers the option to build an app using a one-size-fits-all strategy. This could be a paradigm shift for designers and developers, who will need to learn to think of app design in terms of puzzle pieces that can be stretched, stacked, expanded or hidden to fit the form factor. In short, this will allow one Android OS to run anywhere (with infinite possibilities!).

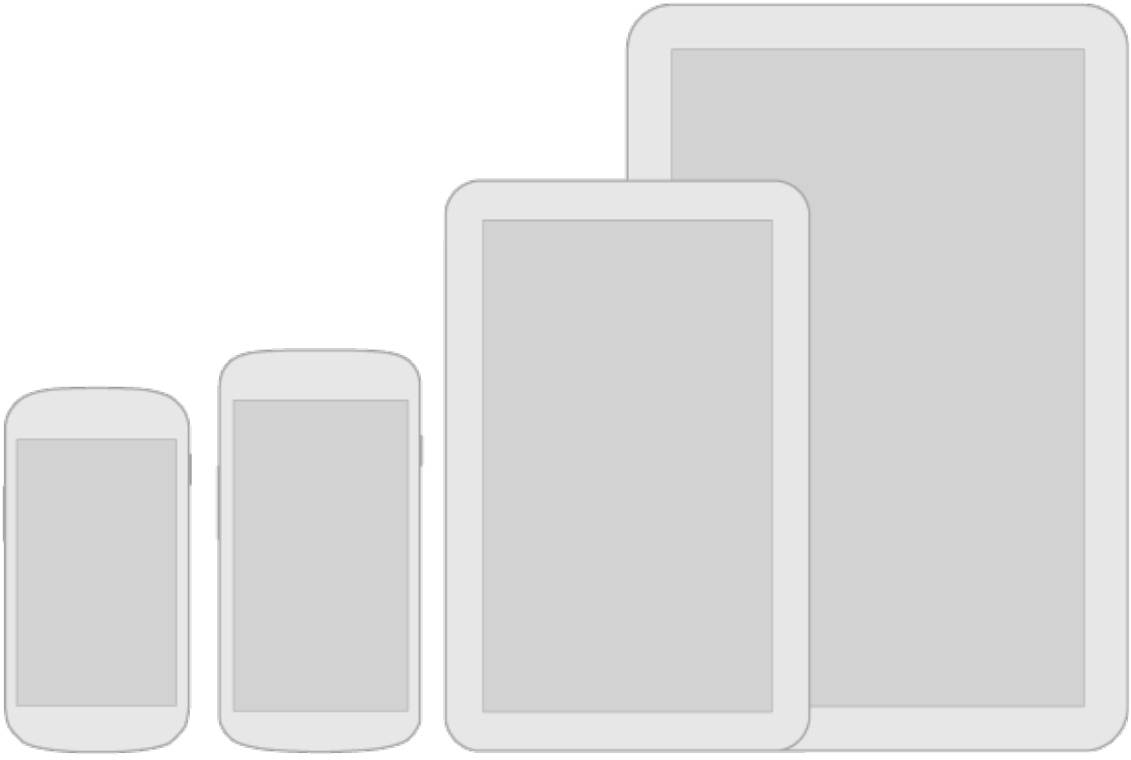
A WORD OF ADVICE

Do get your hands on an Android phone and tablet, and spend some time downloading apps and exploring their interfaces. In order to design for Android, you have to immerse yourself in the environment and know it intimately. This might sound obvious, but it's always surprising to hear when even the product manager doesn't have an Android device.

Designing For Android: Tips And Techniques

Jamie McDonald

Android is an attractive platform for developers, but not all designers share our enthusiasm. Making an app look and feel great across hundreds of devices with different combinations of screen size, pixel density and aspect ratio is no mean feat. Android's diversity provides plenty of challenges, but creating apps that run on an entire ecosystem of devices is rewarding too.



There are hundreds of Android devices with different screen sizes and resolutions. (Image credit: Android Design. Used under Creative Commons license.)

At Novoda, we build Android software for brands, start-ups and device manufacturers. We often work with visual designers who are new to Android. The new Android Design site is the first resource we recommend. You should definitely check it out. However, there is plenty more to pick up! The goal is to create apps that people love to use. Thoughtful UX and aesthetically pleasing visual designs help us get there.

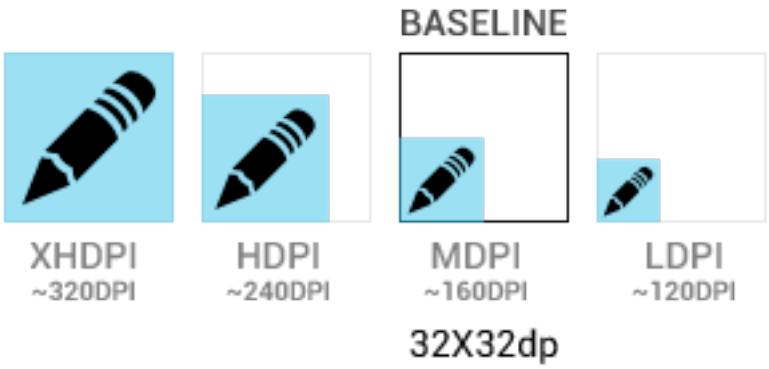
This article provides a set of practical tips and design considerations for creating Android apps. I've tried to include something useful whether you're crafting pixel-perfect graphic assets, finding an optimal user flow or getting your hands dirty developing XML layouts.

Pixels

Visual design is hugely important in the perceived quality of an app. It might even improve usability. Most developers have some exposure to UI patterns, but developers with visual design skills are *rare*. They really need you. Delivering high-fidelity mock-ups, drawable resources (i.e. graphic assets) and guidance to developers is the best way to deliver an aesthetically pleasing experience to the end user.

SCALE NICELY

Android is a platform of many screen densities. There is no set of resolutions to target, rather a density independent measurement scheme for graphics, widgets and layouts. This is covered in depth in a previous Smashing article and the official documentation, so I'll just add a mention of this neat web tool for calculating density pixels.



Optimize graphics for different screen densities. (Image credit: Android Design. Used under Creative Commons license.)

It's not always practical to hand optimize graphic assets for each density. The platform can scale resources down reasonably well. However, it's always worth testing designs on low-end devices and optimizing resources that scale badly.

BE STATE FRIENDLY

Touch states provide important confirmation of clicks and selections. When customizing widgets such as buttons, it's important to create drawables for all necessary states (such as default, focused, pressed and disabled). The focused state is essential user feedback on devices that support directional pad or trackball navigation.

Size is important too. Touch input is imprecise and fingers occlude the UI as they interact with the screen. Touch targets should normally be at least 45 density pixels in width and height.

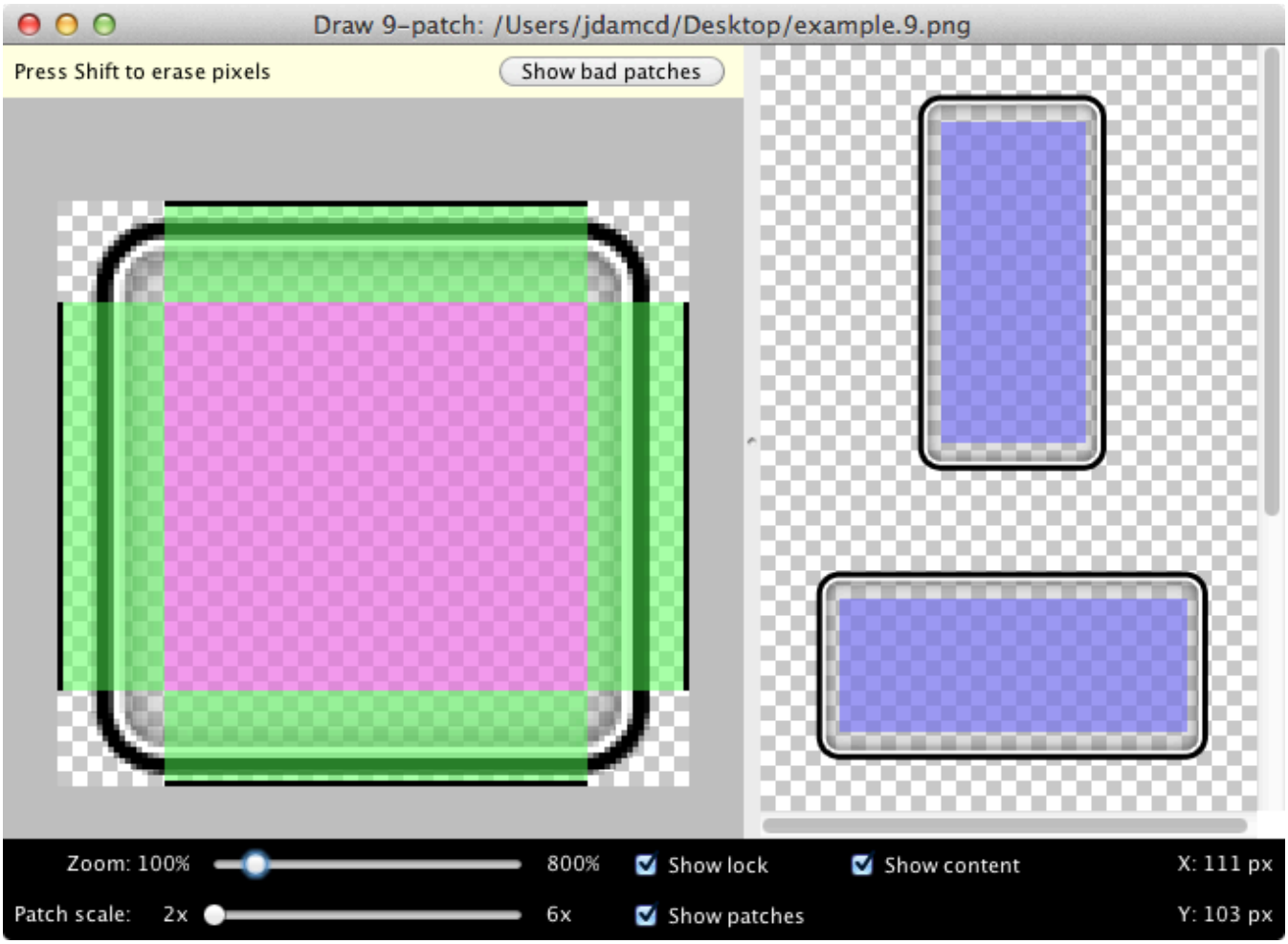
USE FONTS

Android has two fonts: Droid Sans and Roboto. Roboto was released in Ice Cream Sandwich (Android 4). It's been compared to Helvetica, but it's a little condensed, which is great for small screens. You're not limited to Roboto or Droid Sans, though. Any font can be packaged within an app in TTF format (with some memory overhead).

Roboto is Android's new font, introduced in Ice Cream Sandwich. (Image credit: Android Design. Used under Creative Commons license.)

USE 9-PATCH DRAWABLES

9-patch drawables allow PNGs to stretch and scale nicely in pre-defined ways. Markings along the top and left edges define the stretchable areas. The padded content area can optionally be defined with markings along the bottom and right edges. 9-patches are essential for creating and customizing UI widgets.

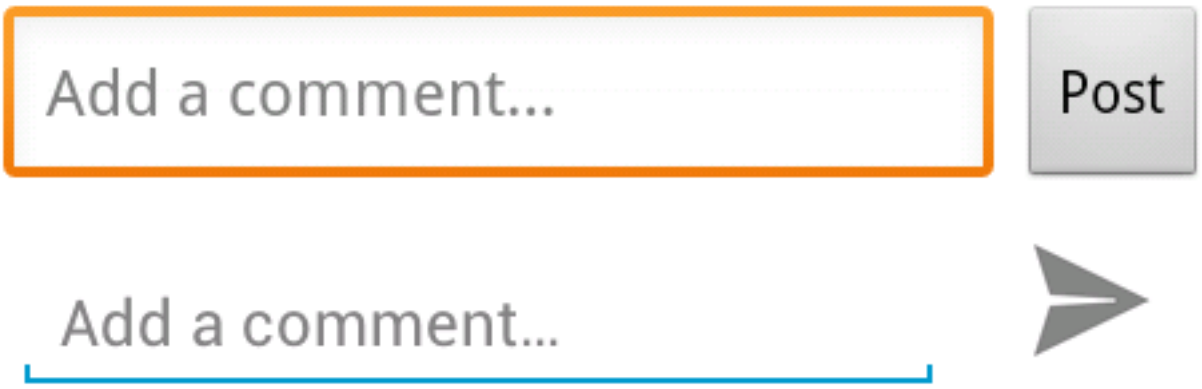


Create scalable widgets with Draw 9-patch.

It's possible to create 9-patches manually, but the Android SDK comes with an nice, simple tool called Draw 9-patch. This makes it quick and easy to convert a regular PNG in to a 9-patch. It highlights the stretchable area and displays previews of the resulting drawable with different widths and heights.

HANDLE DESIGN LEGACY

Honeycomb (Android 3) and Ice Cream Sandwich (Android 4) modernized Android's visual design with the Holo theme. However, some device manufacturers have a poor reputation for keeping platform versions up-to-date. Some of today's most popular devices will never be upgraded to Ice Cream Sandwich.



The Meetup app makes everybody feel at home with separate Gingerbread (Android 2.3) and Ice Cream Sandwich widgets.

So what can be done? There are two options. Deliver the current look, feel and experience to all devices or use a separate set of widgets styles and drawables for Gingerbread and below. Both approaches are valid. Would your users prefer *modern* or *comfortably familiar*?

SHOWCASE THE BRAND

Sometimes clients fear that sticking to recognized UI design patterns will make their apps less distinctive. I think the opposite is true. As patterns like the action bar become ubiquitous, they fade into the background. Users can spend less time wondering how to use an app and more time appreciating how elegantly your app solved their problem. That experience is much more valuable for the brand than a one-of-a-kind UI for the sake of differentiation.

The original Color app had an online FAQ for the UI controls. Make sure that navigation is intuitive.

Branding can be expressed through design of icons, drawables and widgets, as well as in the choice of colours and fonts. Subtle customization of the standard platform widgets can achieve a nice balance of brand values and platform consistency.

CREATE HIGH-FIDELITY MOCK-UPS

High fidelity mock-ups are the best way to communicate visual design to developer responsible for implementation. The Android Design website provides templates in PSD and other formats. It's important to try mock-ups out on real devices to confirm that they feel right, with UI components sensibly sized and placed. The Android Design Preview tool allows you to mirror mock-ups directly from your favourite design software to an attached Android device.

A practical approach for mock-ups is to work against the screen characteristics of the most popular devices. Ideally, create mock-ups for each alternative layout required by screen size or orientation.

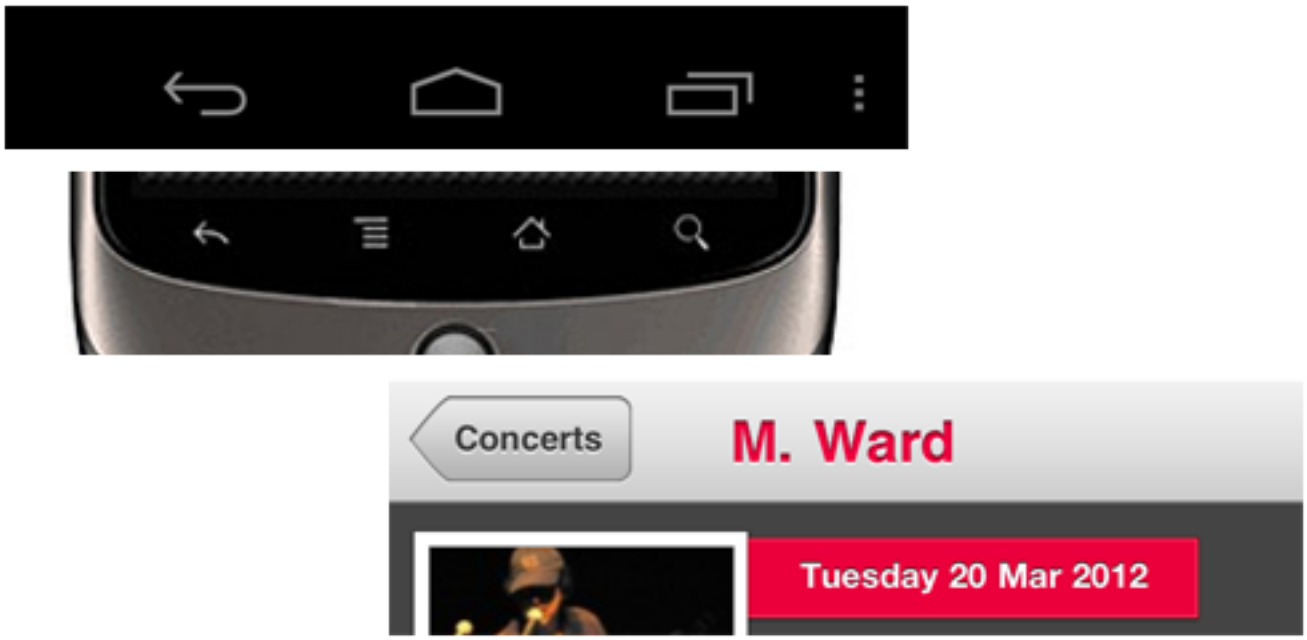
POLISH

Attention to detail is key. Become involved in the development process to ensure that your designs are realized. As a developer, I would always prefer to work with active designers than those who deliver mock-ups and resources before disappearing into thin air. Designs need to be iterated and refined as the app develops.

Animated transitions provide some visual polish that many Android apps lack. Developers might not include such things on their own initiative. Make them part of the design when they make sense. Aside from transitions, animations are a great way to keep users distracted or entertained when the app needs to make them wait.

User Experience

Android has patterns and conventions like any other platform. These help users to form expectations about how an unfamiliar app will behave. Porting an iOS experience directly to the Android platform almost always **results in a poor user experience**.



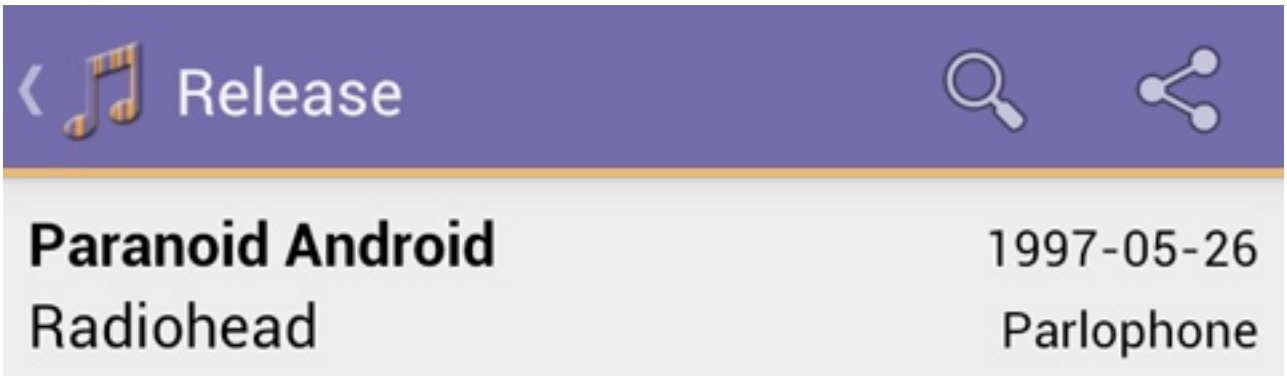
Back is a platform affordance in Android. In contrast, labeled back buttons within the app layout are the norm for iOS.

The back button is the best illustration of the interaction differences between Android and iOS. All Android devices have a hardware back button or on-screen navigation bar (including back button). This is universally available as a feature of the platform. Finding a back button within an Android app layout feels weird as an Android user. It makes me pause to think about which one to use and whether the behavior will differ.

DESIGN USER FLOWS

At the very simplest level, Android apps consist of a stack of screens. You can navigate in to the stack with buttons, action bar icons and list items. The platform back button allows you to reverse out of the stack.

The action bar mirrors a web convention, where the app icon to the left of the action bar usually takes you to the top level of the app. However, there is also the up affordance, intended to take advantage of structural rather than temporal memory. This is represented by a backwards facing chevron to the left of the app icon. This signals that pressing the icon will navigate one level up in the information hierarchy.



The up affordance allows the user to navigate up an information hierarchy instead of going to the top level of the app.

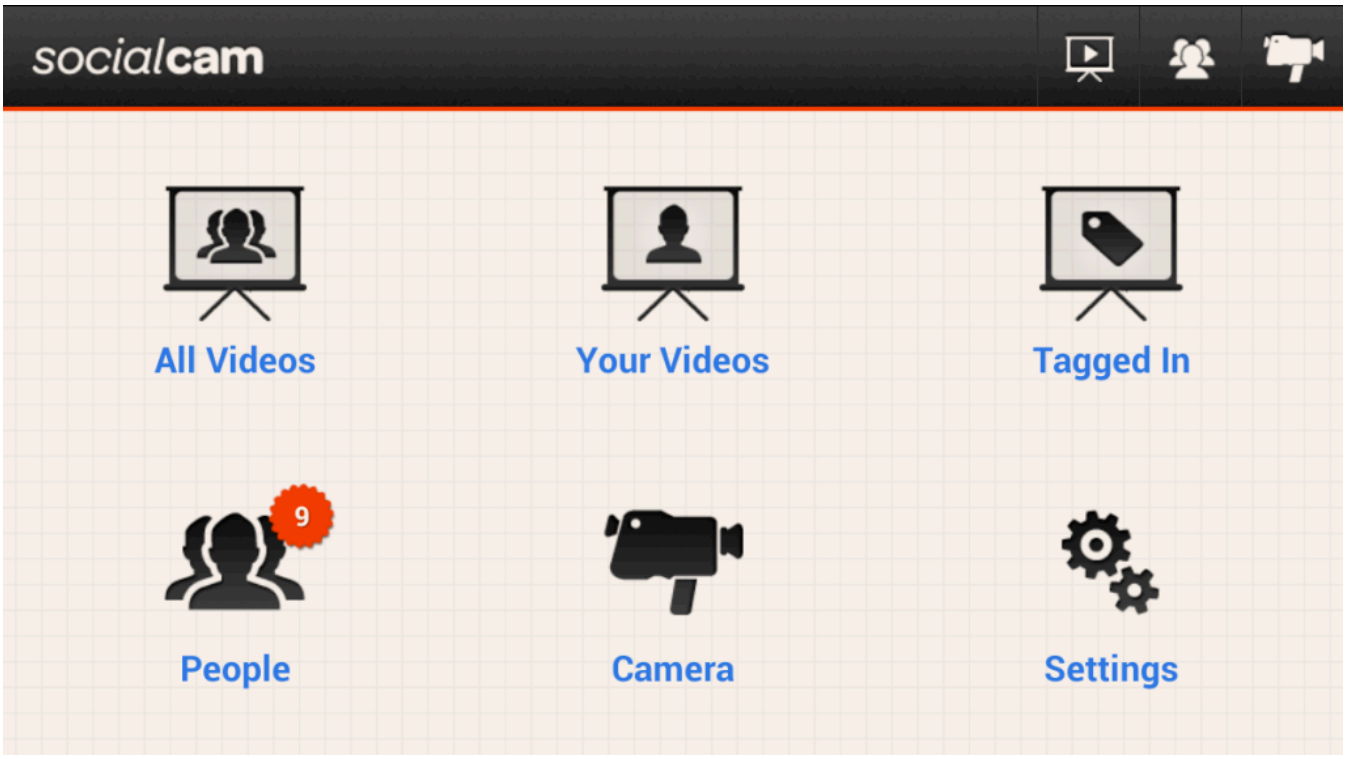
The purpose of the up affordance might be subtle at first. Android apps can have several entry points in addition to the launcher. **The Intent system allows apps to deep link each other** and home screen widgets or notifications might take you directly to specific content. The up affordance allows you to navigate up the information hierarchy regardless of where you came from.

Try user flows on potential users with wireframes or mock-ups and iterate. Prototypes on real devices are ideal because they allow you to test in realistic mobile environments. This might seem like a lot of effort, but remember, you only need to try things out with a few users.

BE PLATFORM CONSISTENT

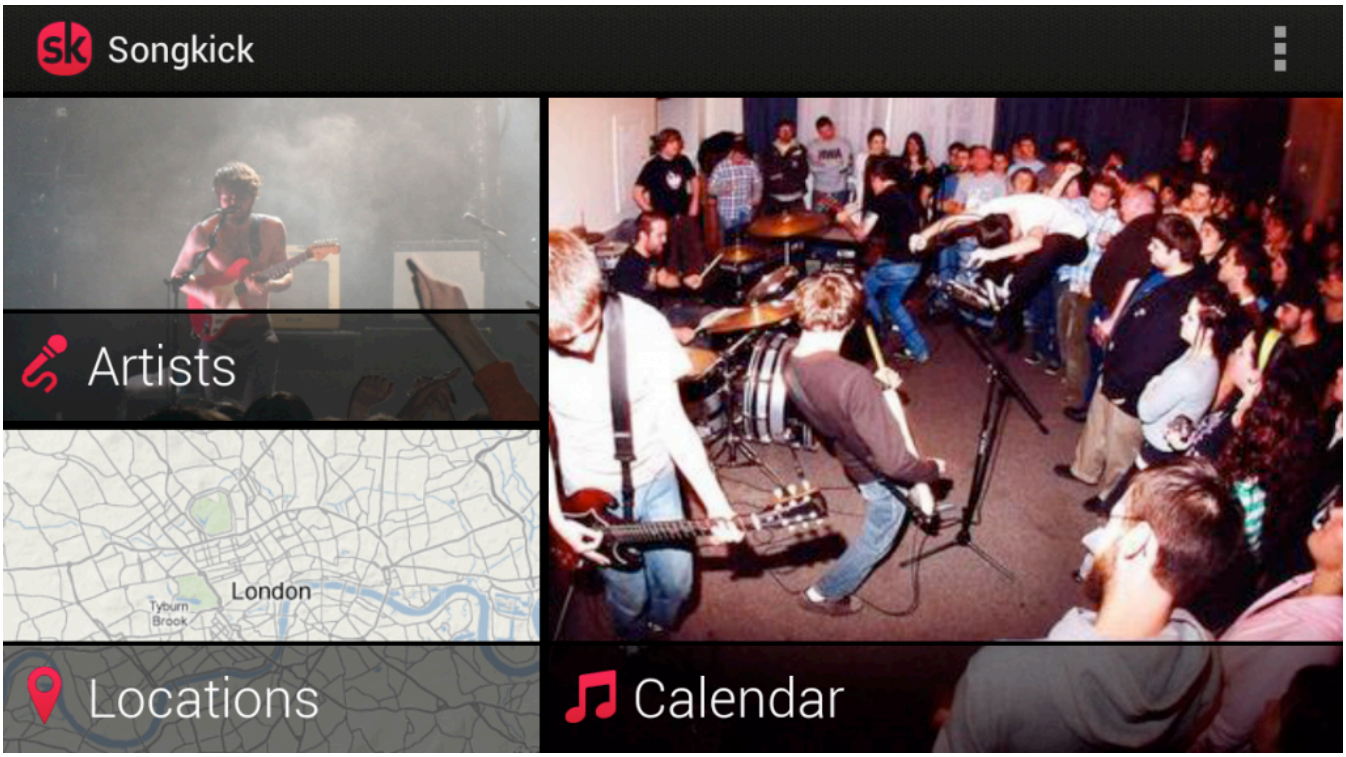
UI patterns are your friend. It's much better to think of these patterns as tools than constraints. Users would prefer not to have to learn to use your app, so patterns provide familiar hints about how to navigate and interact.

Action bar is the most widely adopted Android pattern. It tells you where you are and what you can do. It's a native feature of the platform since Honeycomb and the excellent Action Bar Sherlock library makes it available on older platform versions too.



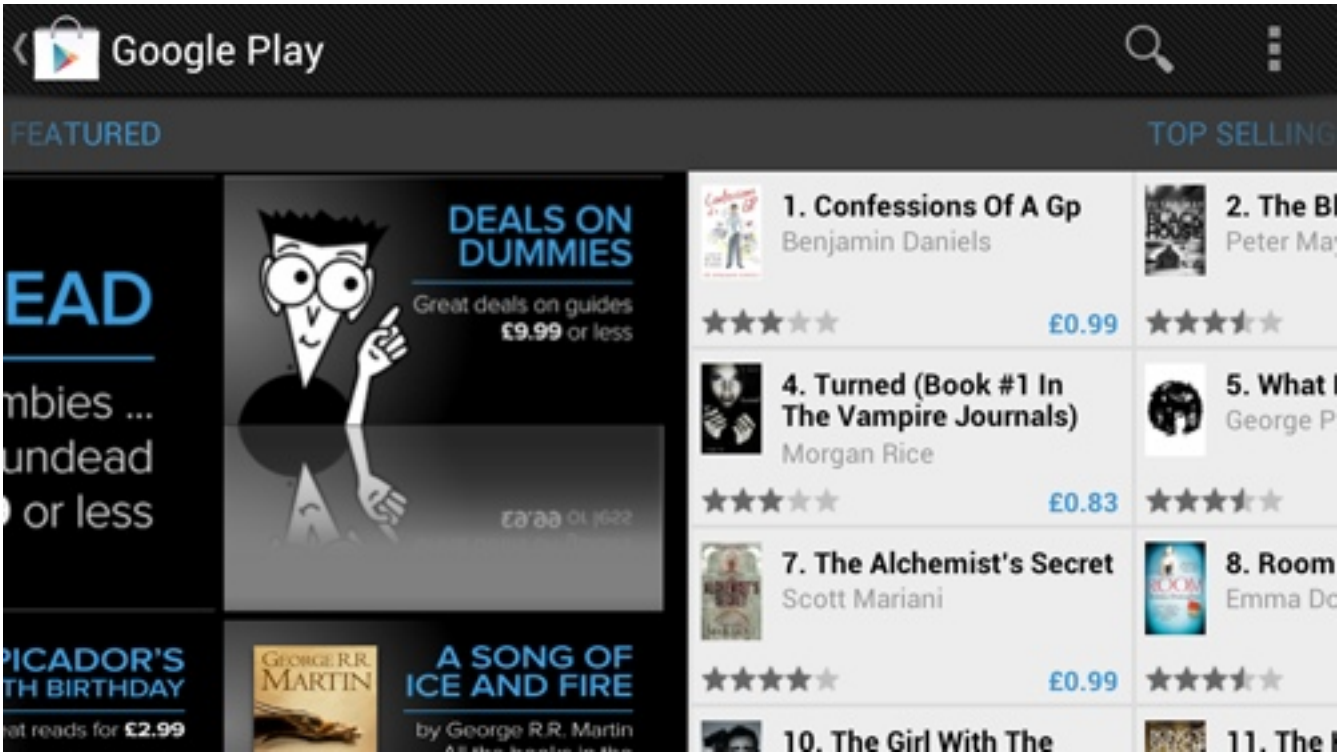
An example of the dashboard and action bar patterns.

The **dashboard** pattern is also quite widely used. These grids of icons are usually presented to the user when they launch an app. Dashboards provide top level navigation and describe the primary areas of the app.



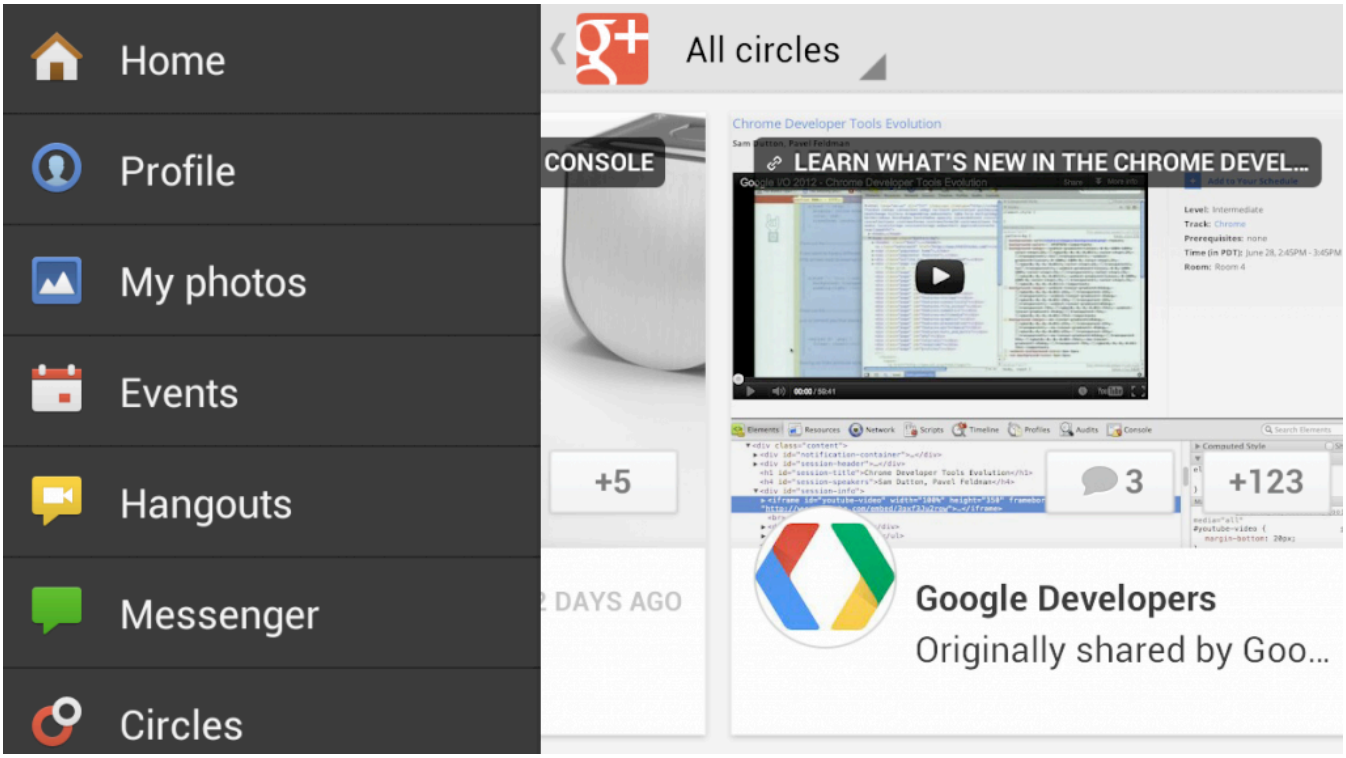
I worked on the Songkick app, where we used a dashboard draw out the content of the app with full-bleed images.

The **workspaces** pattern can be implemented with the ViewPager component. This allows users to swipe screens left and right between content. This can be used in conjunction with tabs to provide a more fluid browsing experience with tabbed data.



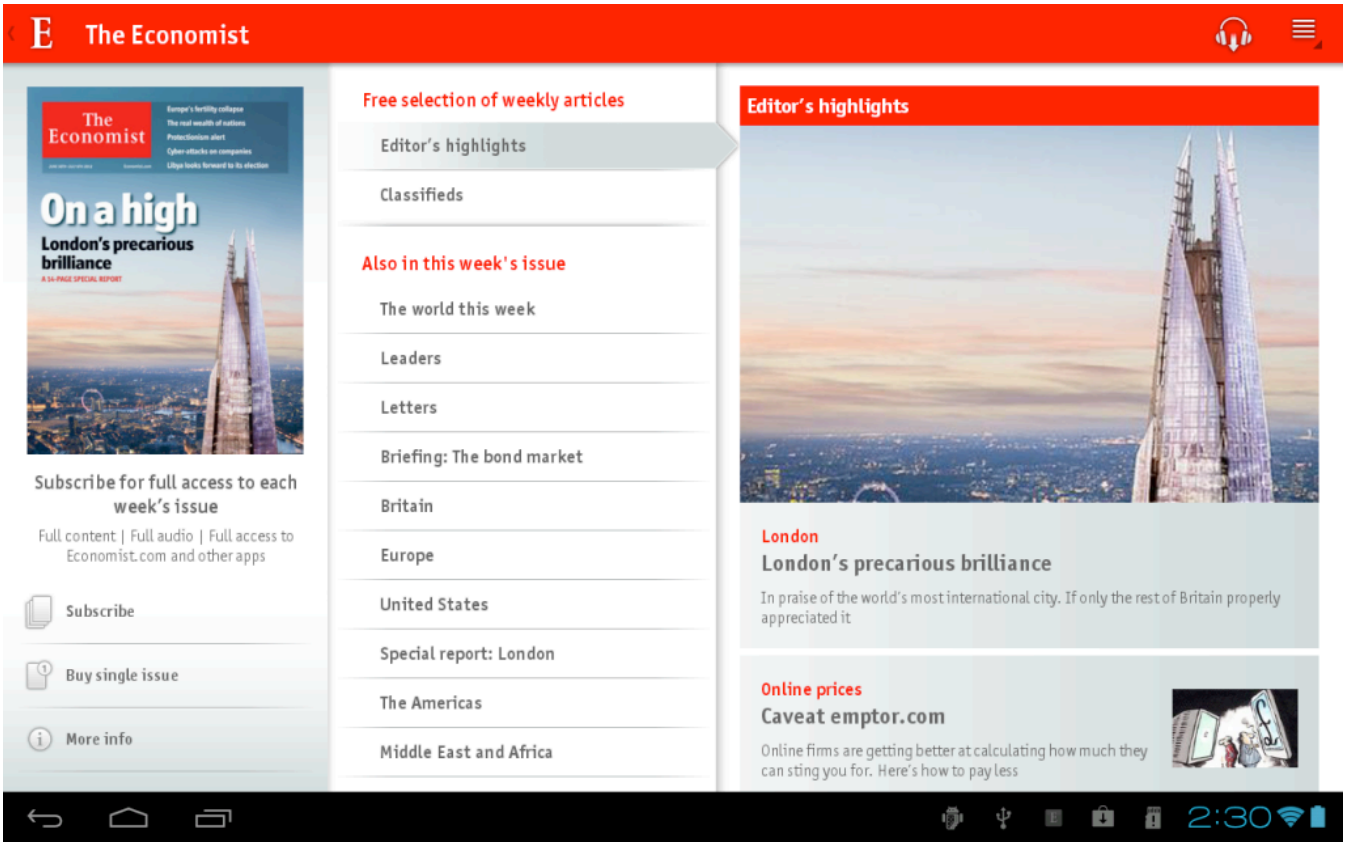
ViewPagers allow users to swipe left and right. Page indicators or tabs make this navigation discoverable.

The **ribbon menu** is an emerging navigation pattern. This allows us to launch the user directly into content and provide the top level navigation in a menu, which slides in from the left side of the screen when you press **up**.



The ribbon menu is an alternative to dashboard navigation.

Tablet optimized apps often take advantage of **multi-pane layouts**. A single tablet screen can display the content of several separate phone screens side by side. Optimising for tablets can involve creating several alternative layouts for different screen widths. Sections of UI can be designed once and laid out in different configurations for different screen sizes. Multi-pane layouts help to avoid overly wide list items and sparse layouts.



The Economist news app uses multi-pane tablet layouts so users can explore the hierarchy of content on a single screen.

These are familiar and proven UI patterns. They're the best tools for starting to sketch out your app layouts and navigation. However, they shouldn't discourage you from trying something new. Just ensure that the app behaves predictably.

DESIGN RESPONSIVELY

Android is a platform of many screen sizes. The devices that I can lay my hands on in our office compose a spectrum of screen sizes from 1.8 to 10.1 inches (as long as we ignore the Google TV). With variable screen area, **Android has something in common with responsive web design**. There is no getting away from the fact that design and implementation of a responsive experience across the full range of devices takes a lot of work. Supporting every screen is the ideal, but there are also sensible strategies for coping with the diversity of the platform.

Knowing a little about your target users and popular devices can help focus efforts and avoid premature optimisation. A good default strategy is to target popular, middle sized phones (3.2″ – 4.6″) and then optimize as necessary with alternate layouts and user flows for particularly small (<3″) devices and tablets.

It's always best to be orientation agnostic. Some devices have physical keyboards that require the device to be held in landscape. The on-screen keyboard is also easier to use in landscape. Text entry on touch screens is awkward an error prone, so let's at least give our users the benefit of the landscape keyboard.

UNDERSTAND MOBILE INTERACTIONS

People interact with mobile apps differently from websites or desktop software. Mobile apps rarely have the undivided attention of a user and most interactions use touch input, which is not as precise as we might like.

Mobile interactions can often be measured in seconds. We recently developed a location-based app that allows users to check-in at bars. We counted the clicks on user paths such as check-in, considering whether each step could be avoided or simplified. We specify everything that an app should do as user stories. The most frequent stories should be as quick and easy to accomplish as possible. It's particularly important in this scenario, because the user might be under the influence of alcohol…

OPTIMIZE FIRST USE

First launch experience is crucial. Apps are often installed in response to a real world problem. If the first run isn't satisfying then the user might never return. If the app requires sign up, offer preview functionality so that users get a feel for the experience. They probably need to be convinced that it's worth filling out that sign-up form. Also consider using analytics to measure points where users drop off in the launch and sign-up process.

Many apps launch with a tutorial. This is usually an admission that the app is too complicated, but if you're sure that you need one, keep it brief and visual. You might also want to use analytics to confirm that a tutorial serving a purpose. Are users that complete the tutorial more active? How many users just skip the tutorial?

BRING THE APP TO PLAY

User experience considerations shouldn't end in the app. It's worth putting a bit of thought in to the Google Play Store listing to ensure that it's immediately obvious what the app does and why the user would want it.

These Graphic Asset Guidelines will help you to create promotional material suitable for the various contexts and scales in which they appear. Some of these graphics are a pre-requisite for being featured too.

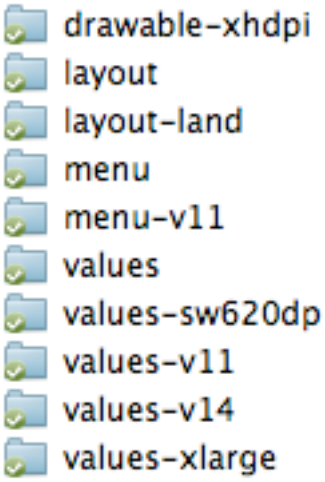
Layouts, Styles and Themes

Android has a visual layout editor and it's getting better all the time. However, I still find myself developing XML layouts by hand. This section gets down to implementation details, covering some best practices for crafting maintainable and performant layouts. Visual designers might want to skim this section, but some awareness of implementation details can't hurt.

The most general purpose layouts are *RelativeLayout* and *LinearLayout*. *RelativeLayout* should be favoured for efficiency, whilst *LinearLayout* is useful for distributing space between views using weights. GridLayout was new in Honeycomb. This is useful for creating complex layouts on large screens without nesting. Nesting layouts too deep is bad for performance and code readability alike!

LET THE FRAMEWORK DO THE WORK

The Android framework provides automated resource switching based on folder structure. This means that you can have separate graphic assets and layouts for different screen sizes and densities by arranging them in the correct folders. It goes much further than that. For example, you could switch color resources for different platform versions or even animation durations for different screen sizes.



The framework provides automatic resource switching.

Since Honeycomb, it's also possible to switch resources on available screen width in density pixels. This is a move away from the bucketed small, normal, large and extra-large screen size switching. It facilitates responsive design and allows multiple layout switching points (perhaps switching to a tablet-optimized layout at 600dp with another alternative at 800dp). It's typical to have multiple layout files with different configurations of the same components for different screen characteristics.

State list drawables make being state-friendly easy. These allow you to specify different drawables for different states of a UI component in an XML file. As mentioned earlier, representing states properly provides important user feedback.

```
<selector xmlns:android="http://schemas.android.com/apk/res/
android">

  <item
    android:state_focused="true"
    android:state_pressed="true"
    android:drawable="@drawable/my_button_pressed_focused" />

  <item
    android:state_focused="false"
    android:state_pressed="true"
    android:drawable="@drawable/my_button_pressed" />

  <item
    android:state_focused="true"
    android:drawable="@drawable/my_button_focused" />

  <item
    android:state_focused="false"
    android:state_pressed="false"
    android:drawable="@drawable/my_button_default" />

</selector>
```

EXTRACT VALUES

It's good practice to keep layout XML clean of explicit colours and dimensions. These can be defined separately and referenced in your layouts. Defining colours and dimensions separately promotes visual consistency and makes things easier to change later on. Extracting these values allows switching of dimensions on different screen sizes, orientations and platform versions. This is useful for tweaking padding for small screens or increasing text size for readability on large screens, which tend to be held further away from the face. Perhaps **res/values/dimens.xml** contains:

```
<dimen name="my_text_size">16sp</dimen>
```

whilst **res/values-sw600dp/dimens.xml** contains:

```
<dimen name="my_text_size">20sp</dimen>.
```

USE STYLES AND THEMES

A good technique to keep layout XML maintainable is to separate the styling concern from the positioning concern. Every **View** in a layout needs to have at least a width and height attribute. This results in a lot of boilerplate, which you can eliminate by inheriting from a set of base parent styles.

```
<style name="Match">
  <item name="android:layout_width">match_parent</item>
  <item name="android:layout_height">match_parent</item>
</style>

<style name="Wrap">
  <item name="android:layout_width">wrap_content</item>
  <item name="android:layout_height">wrap_content</item>
</style>

<style
```

```
  name="MatchHeight"
  parent="Match">
  <item name="android:layout_width">wrap_content</item>
</style>

<style
  name="MatchWidth"
  parent="Match">
  <item name="android:layout_height">wrap_content</item>
</style>
```

Recurring sets of attributes can be moved into styles. Widget styles that occur almost universally throughout the app can be moved into the theme. If a particular type of button always has the same text color and padding, it's much cleaner to specify the style than duplicate these attributes for each occurrence.

```
<style
  name="MyButtonStyle"
  parent="MatchWidth">
  <item name="android:padding">@dimen/my_button_padding</item>
  <item name="android:textColor">@color/my_button_text_color</
item>
</style>
```

We save four lines of attributes when we add the button to a layout. The layout file can be concerned with just the positioning and unique attributes of widgets.

```
<Button
  android:id="@+id/my_button"
  style="@style/MyButtonStyle"
  android:text="Hello, styled world!">
```

You can take this further by overriding default button style in a theme and applying it to an Activity or the entire app in the **AndroidManifest.xml**.

```
<style
  name="MyTheme"
  parent="@android:style/Theme.Holo">
  <item name="android:buttonStyle">@style/MyButtonStyle</item>
</style>

<style
  name="MyButtonStyle"
  parent="@android:style/Widget.Button">
  <item name="android:padding">@dimen/my_button_padding</item>
  <item name="android:textColor">@color/my_button_text_color</
item>
</style>
```

OPTIMIZE

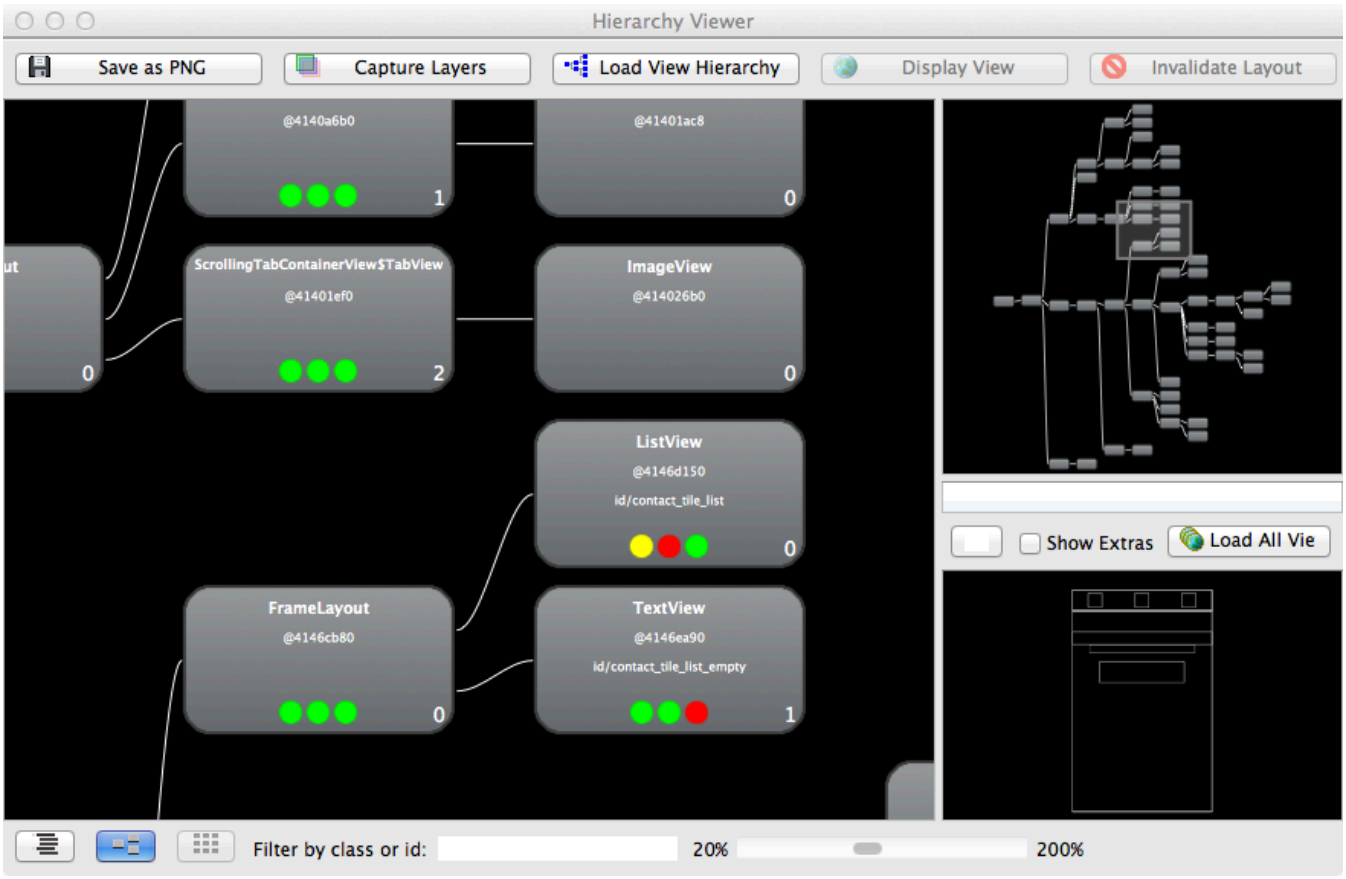
The **include** and **merge** XML tags allow you to drop reusable sections of UI into your layouts, minimizing duplicate XML when the same set of views occurs in multiple layout configurations.

```
<include
  layout="@layout/my_layout"
  style="@style/MatchWidth" />
```

A relatively new addition to the Android Developer Tools is Lint. This tool scans the resources in a project and creates warnings about potential performance optimizations and unused or duplicated resources. This is incredibly useful for eliminating clutter as an app changes over time and it's certainly worth checking lint for warnings regularly during your development process.

DEBUG

Sometimes layouts just don't turn out how you expected. It can be hard to spot bugs amongst all those angle brackets. This is where Hierarchy Viewer comes in. This tool allows you to **inspect the layout tree of an app running in the emulator**. You can inspect the detailed properties of each view.



Inspect your layout trees with Hierarchy Viewer. Those colored dots can tell you about your layout performance.

Hierarchy Viewer has a couple neat tricks for visual designers too. It allows you to inspect screens in zoomed **pixel perfect** mode and export the layers of a layout as a PSD.

Conclusion

So you've been introduced to the platform and the tools. What next? Ultimately, the best way to get a feel for Android is to *use it* every day. The most satisfying app designs have a few things in common: platform consistency, attention to detail and clean visual design. The first two, at least, can be picked up by using and analysing existing Android apps.

Android has come a long way in the last few years. The platform and the apps have become gradually more refined. Functionality is not enough at this point. There are almost half a million other apps out there, and users want polish.

Designing For Android Tablets

Dan McKenzie

More than ever, designers are being asked to create experiences for a variety of mobile devices. As tablet adoption increases and we move into the post-PC world, companies will compete for users' attention with the quality of their experience. Designing successful apps for Android tablets requires not only a great concept that will encourage downloads, usage and retention, but also an experience that Android users will find intuitive and native to the environment.

The following will help designers become familiar with Android tablet app design by understanding the differences between the iPad iOS user interface and Android 3.x "Honeycomb" UI conventions and elements. We will also look at Honeycomb design patterns and layout strategies, and then review some of the best Android tablet apps out there.

Note that while Android 2.x apps for smartphones can run on tablets, Android 3.0 Honeycomb was designed and launched specifically for tablets. Future updates promise to bring Honeycomb features to smartphone devices, as well as make it easier to design and build for multiple screen types.

For most of us, our first exposure to tablets was via the iPad. For this reason, it's reasonable to begin comparing the two user interfaces. By comparing, we can align what we've learned about tablets and begin to focus on the key differences between the two, so that we can meet the unique UI needs of Android users. Not only will this help us get up to speed, but it becomes especially important when designing an Android tablet app from an existing iPad one.

It's Just Like The iPad, Right?

While the Android tablet and iPad experience share many similarities (touch gestures, app launch icons, modals, etc.), designers should be aware of the many differences before making assumptions and drawing up screens.

SCREEN SIZE AND ORIENTATION

The biggest difference between the two platforms is the form factor. Layouts for the iPad are measured at 768 × 1024 physical pixels, and the iPad uses portrait mode as its default viewing orientation.

With Android tablets, it's a bit more complicated, due to the multiple device makers. In general, there are 7- and 10-inch Android tablets screen sizes (measured diagonally from the top-left corner to the bottom-right), with sizes in between. However, most tablets are around 10 inches.

What does this mean in pixels? A good baseline for your layouts is 1280 × 752 pixels (excluding the system bar), based on a 10-inch screen size and using landscape (not portrait) as the default orientation. Like on the iPad, content on Android tablets can be viewed in both landscape or portrait view, but landscape mode is usually preferred.

The portrait view on a typical 10-inch Android tablet (left) and on the iPad (right).

The landscape view on a typical 10-inch Android tablet (left) and on the iPad (right).

However, with Android, screen size is only the half of it. Android tablets also come in different “screen densities” — that is, the number of pixels within a given area of the screen. Without going into too much detail, designers have to prepare all production-ready bitmaps for three different screen densities, by scaling each bitmap to 1.5×and 2× its original size. So, a bitmap set to 100 × 100 pixels would also have copies at 150 × 150 and 200 × 200. By making three batches of graphics scaled at these sizes, you will be able to deliver your bitmaps to medium, high and extra-high density tablet screens without losing image quality.

For more information on screen densities and preparing graphics for Android devices, refer to my previous article on designing for Android.

SYSTEM BAR

While iOS makes minimal use of the system bar, Android Honeycomb expands its size to include notifications and soft navigation buttons. There is a “Back” button, a home button and a “Recent apps” button.

The Android Honeycomb system bar.

Android Honeycomb's system bar and buttons are always present and appear at the bottom of the screen regardless of which app is open. Think of it like a permanent UI fixture. The only exception is a "Lights out" mode, which dims the system bar to show immersive content, such as video and games.

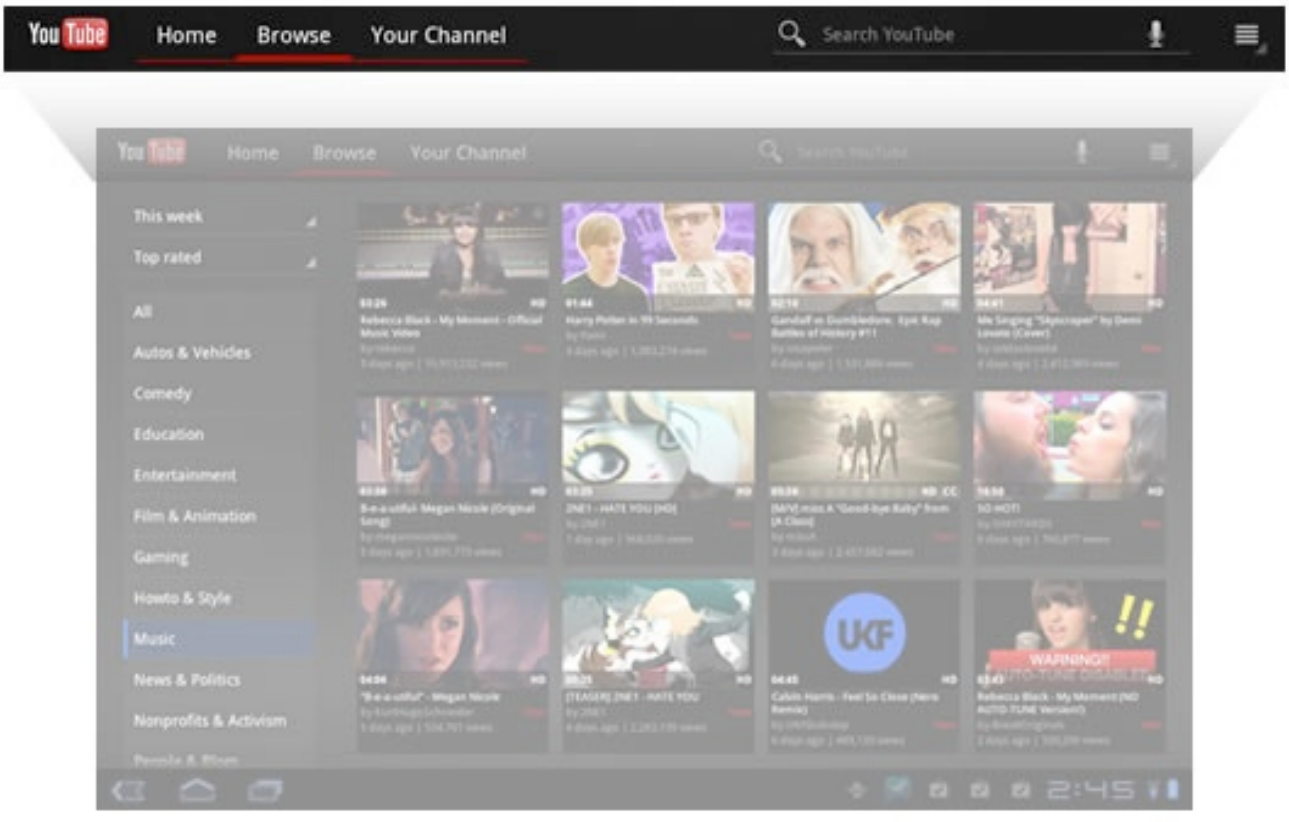
"BACK" BUTTON

While the bulkiness and permanence of the Honeycomb system bar might seem an obstacle to designers, it does free up the real estate that is typically taken by the "Back" button in iPad apps. The "Back" button in Honeycomb's system bar works globally across all apps.

The "Back" button in the system bar.

ACTION BAR

The bulk of the UI differences between platforms is found in the top action bar. Android suggests a specific arrangement of elements and a specific visual format for the action bar, including the placement of the icon or logo, the navigation (e.g. drop-down menu or tabs) and common actions. This is one of the most unifying design patterns across Android Honeycomb apps, and familiarizing yourself with it before attempting customizations or something iPad-like would be worthwhile. More on the pervasive action bar later.



The action bar.

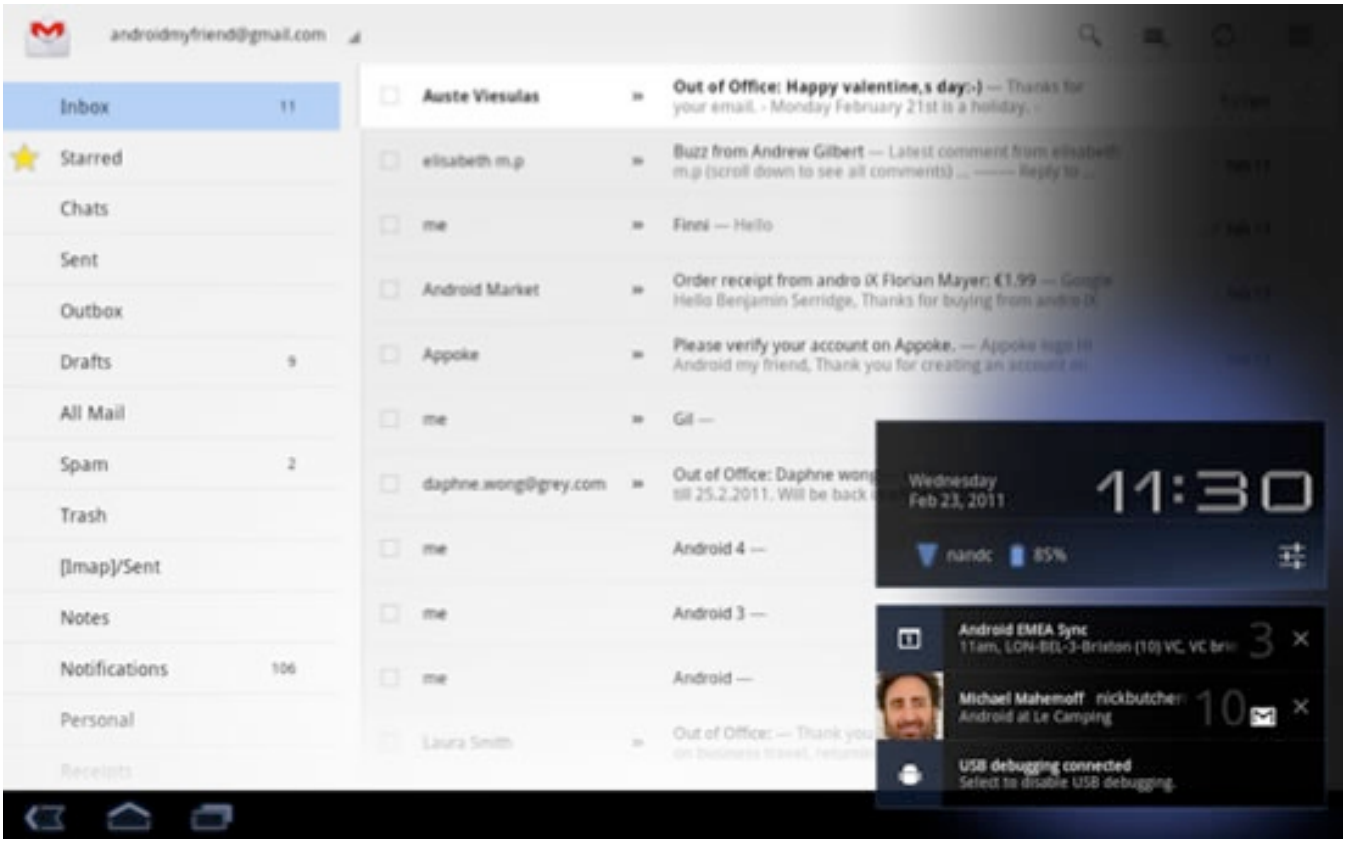
WIDGETS

New to iPad users will be Android's widgets. As the name implies, these are small notification and shortcuts tools that users can set to appear on their launch screen. Widgets can be designed to show stack views, grid views and list views, and with Android 3.1, they are now resizable.

Several widgets on a launch screen.

NOTIFICATIONS

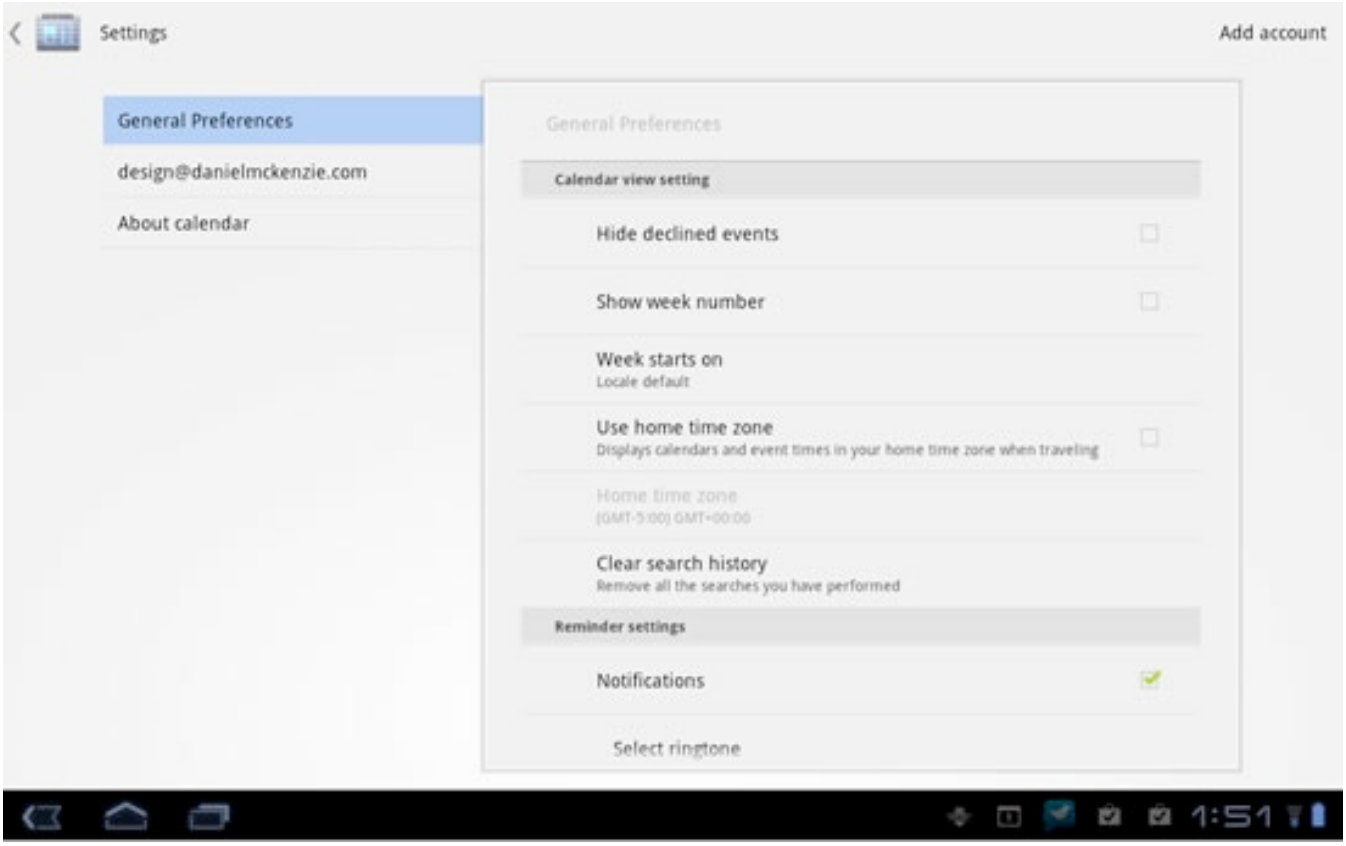
While iOS' notifications system pushes simple alerts to your launch screen, Honeycomb offers rich notifications that pop up ("toast" we used to call them) in the bottom-right area of the screen, much like Growl on Mac OS X. Custom layouts for notifications can be anything from icons to ticker text to actionable buttons.



A notification on Android.

SETTINGS

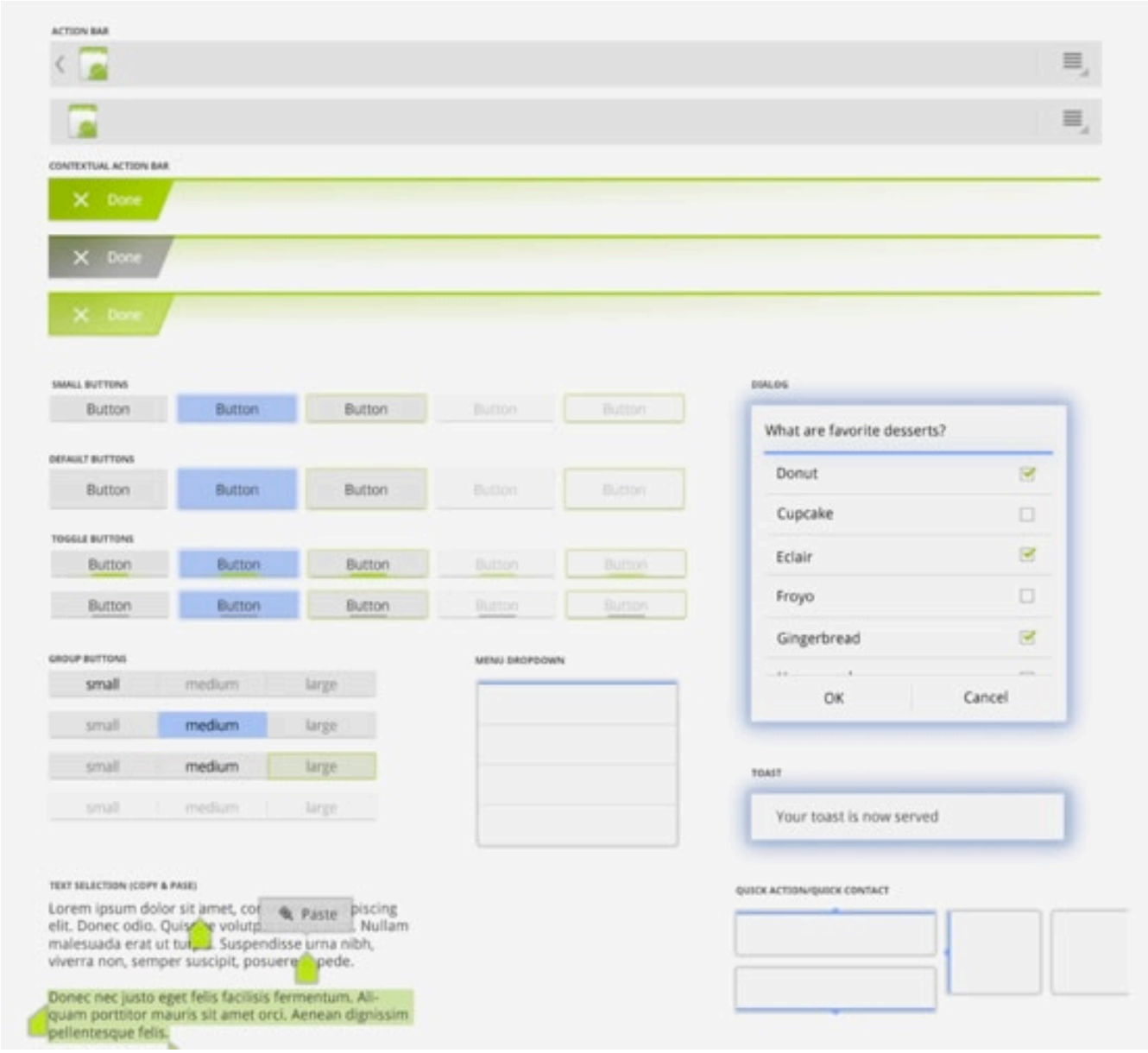
Access to settings in an iPad app are usually presented in a pop-over, triggered by an “i” button; and settings categories are broken up into tables for easy visual identification. Honeycomb has a different convention. It looks more like the iOS’ “General Settings” screen, where the user navigates categories on the left and views details on the right. This is the preferred (and more elegant) way to present multiple settings in Honeycomb.



The settings design pattern in the Calendar app.

UI ELEMENTS

As you can imagine, Android goes to great lengths to do everything opposite from its competitor (that's called differentiation!). Honeycomb has its own UI conventions, and it now has a new "holographic UI" visual language for such routine actions as picking a time and date, selecting an option, setting the volume, etc. Understanding this UI language is important to building screen flows and creating layouts.



A sampling of UI elements, taken from a slide in a Google I/O 2011 presentation.

FONTS

How many fonts does iPad 4.3 make available? The answer is fifty-seven.

How many does Android? Just three.

Yep, those three are Droid Sans, Droid Serif and Droid Sans Mono. But there is an upside. While only these three ship with the platform, developers are free to bundle any other fonts with their apps.

Anything the Same?

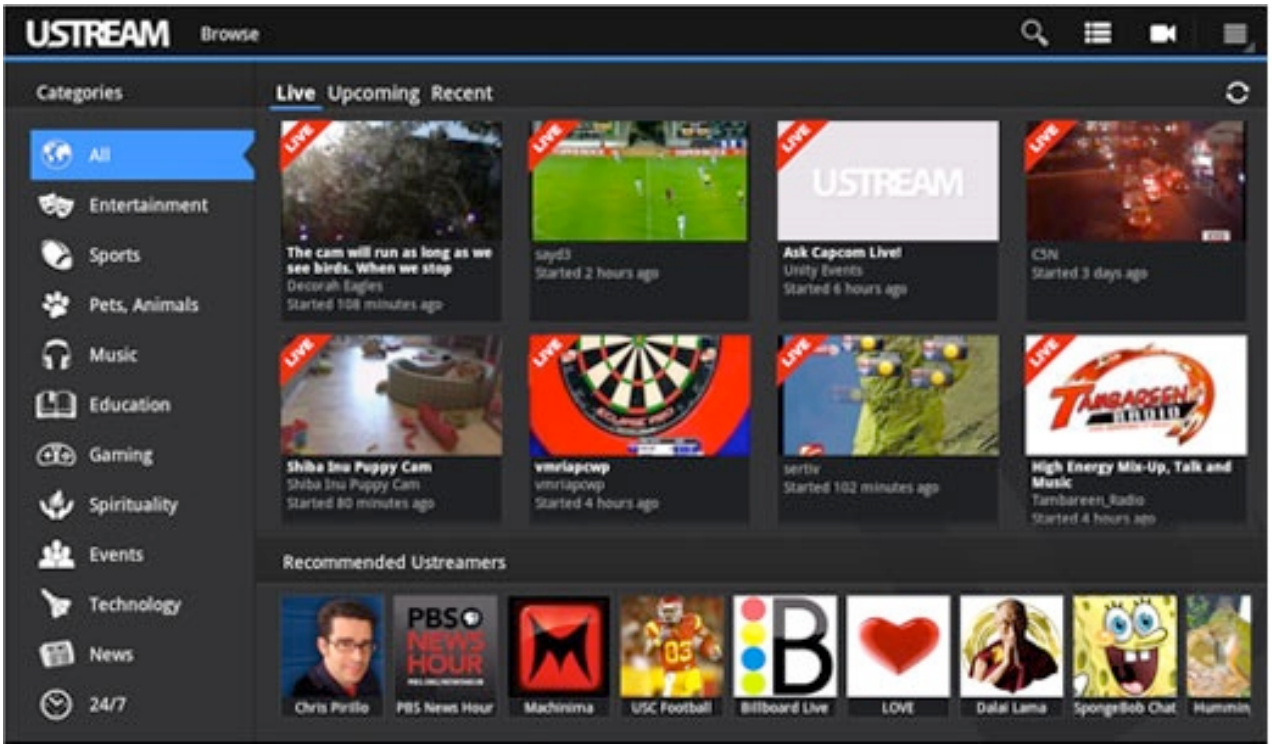
Luckily for designers who are already familiar with the iPad, the two platforms have some similarities.

TOUCH GESTURES

Tap, double-tap, flick, drag, pinch, rotate and scroll at will.

SPLIT VIEW AND MULTI-PANE UI

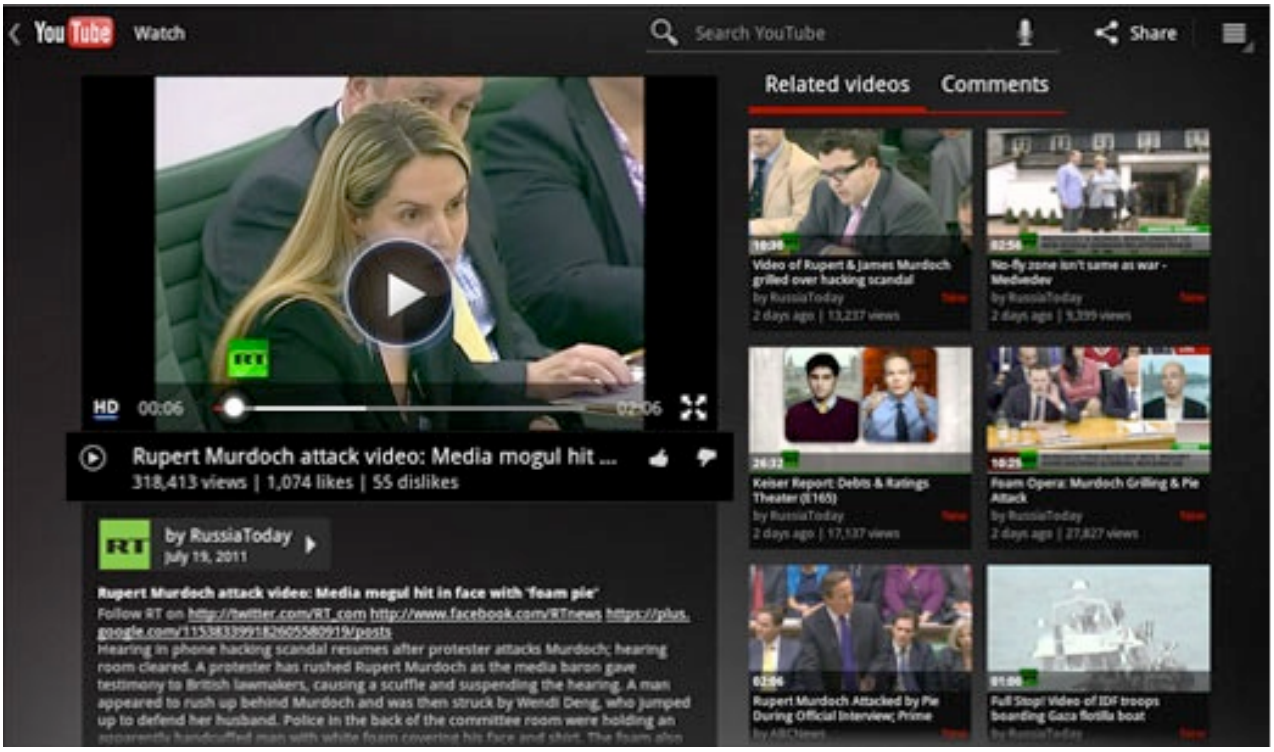
The split view is one of the most common layouts for tablets. It consists of two side-by-side panes. Of course, you can add panes for more complex layouts.



Ustream's split-view layout, with categories on the left and content on the right.

EMBEDDED MULTIMEDIA

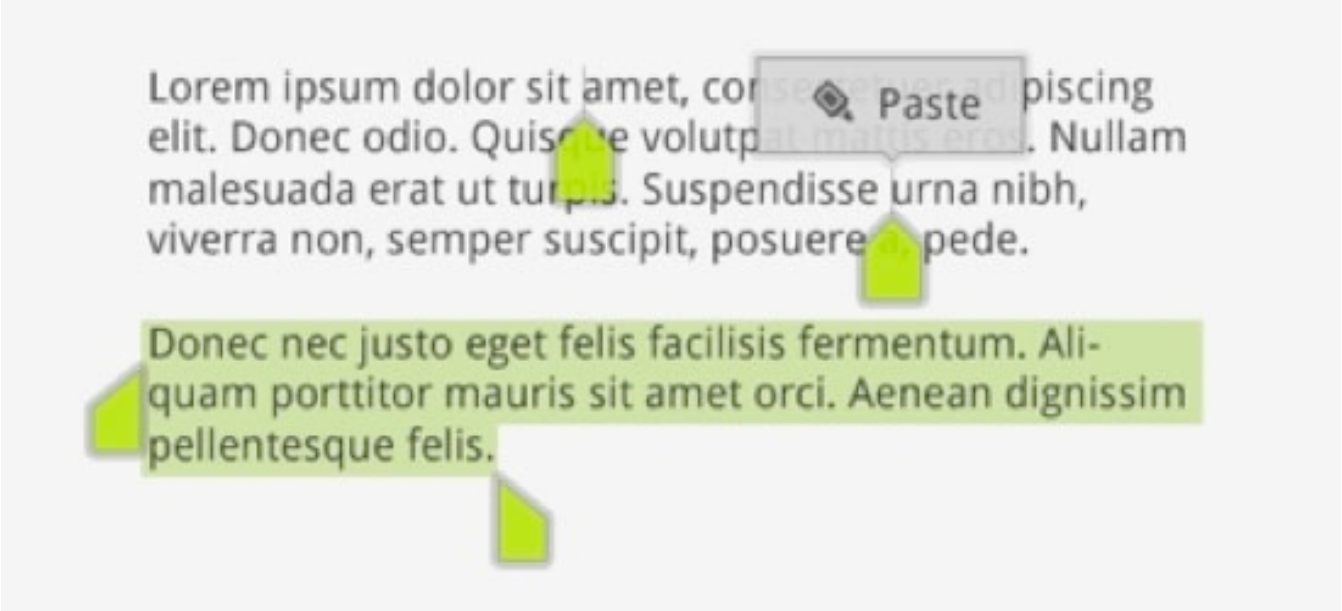
Both platforms allow embedded audio, video and maps.



The YouTube app, with an embedded video player.

CLIPBOARD

For copying and pasting data into and out of applications.



DRAG AND DROP

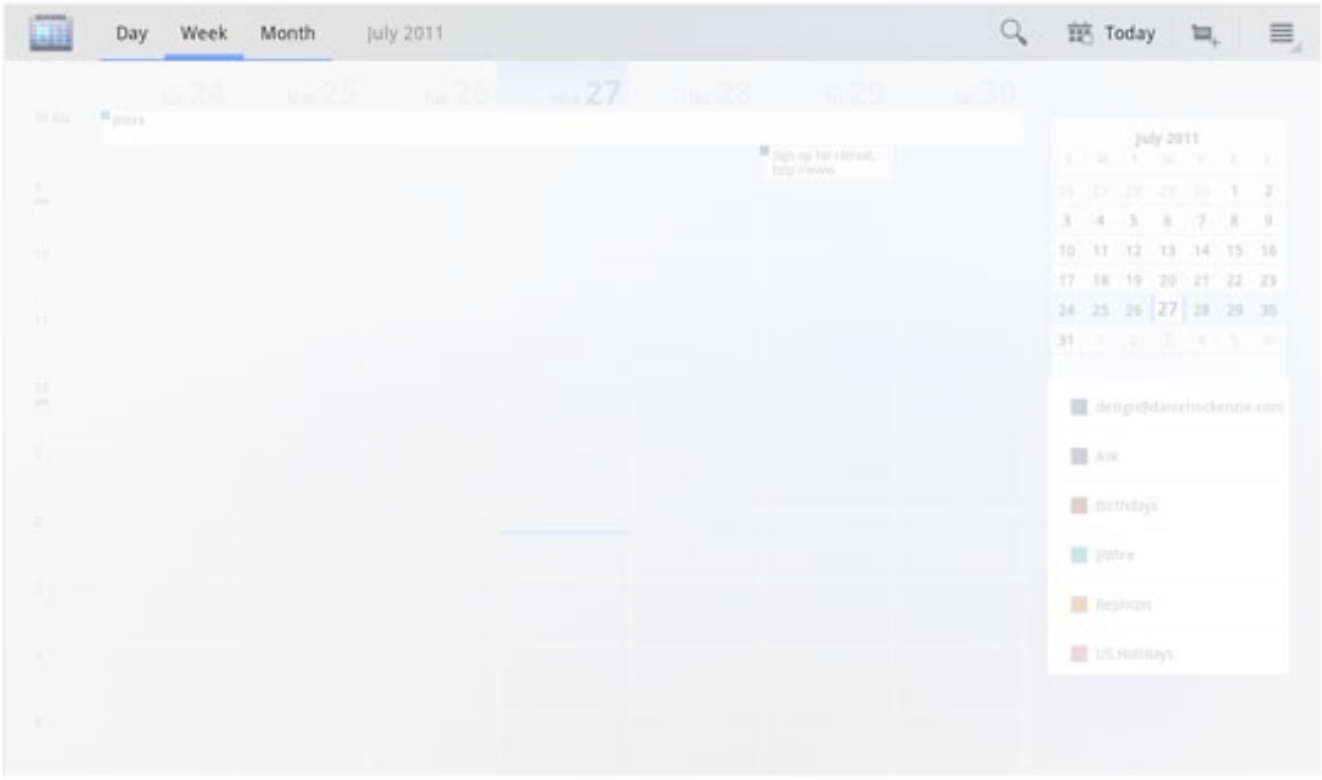
Both platforms have drag-and-drop capabilities.

The drag-and-drop feature in the Gmail app.

Design Patterns

Honeycomb continues many of the design patterns introduced in Android 2.0 and expands on them. In case you're not familiar with design patterns, they are, as defined in Android, a "general solution to a recurring problem." Design patterns are key UI conventions designed by Android's maintainers to help unify the user experience and to give designers and developers a template to work from. They are also customizable, so no need to fret!

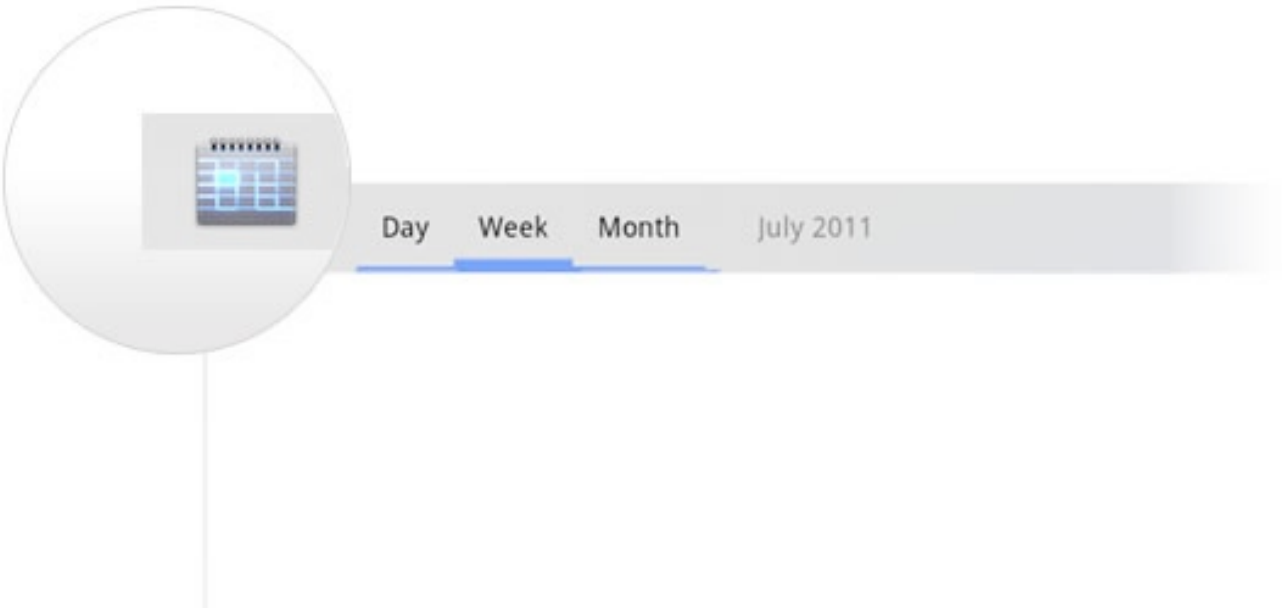
As mentioned, the action bar is the most prominent Android UI component and is the one we'll focus on here.



The action bar highlighted in the Calendar app.

ICON OR LOGO

The action bar starts with an icon or logo on the far left. It is actionable; by tapping on it, the user is directed to the app's home screen.

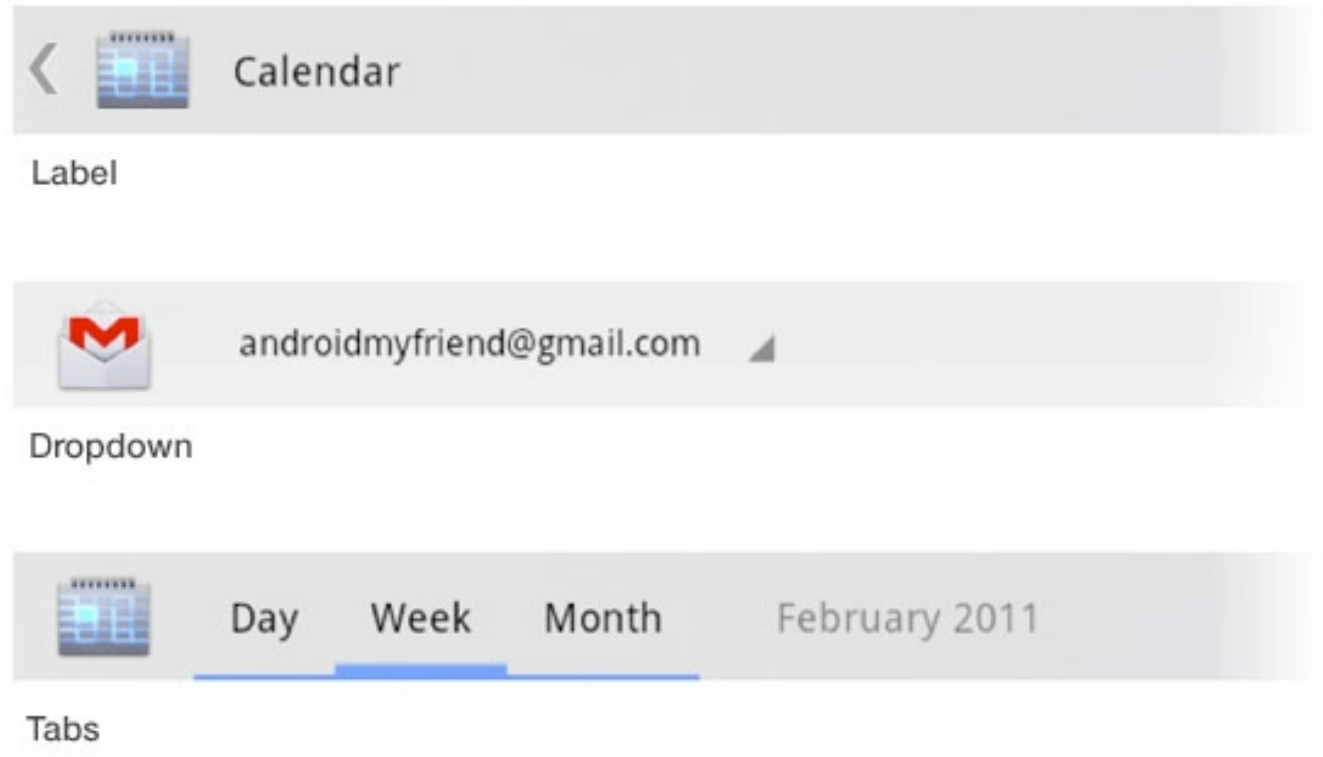


The Calendar app icon.

NAVIGATION

Next, we'll typically find some form of navigation, in the form of a drop-down or tab menu. Honeycomb uses a triangle graphic to indicate a drop-down menu and a series of underlines for tabs, which typically take up most of the action bar's real estate.

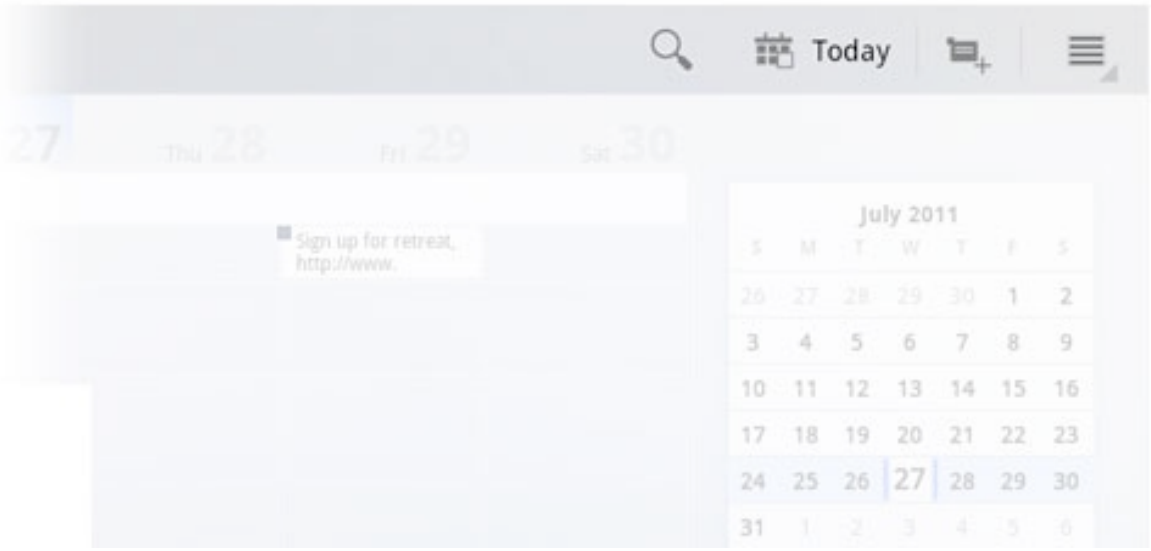
A left arrow button might also appear to the left of the icon or logo or the label, for navigating back or cancelling a primary action.



Three different kinds of action bar navigation.

COMMON ACTIONS

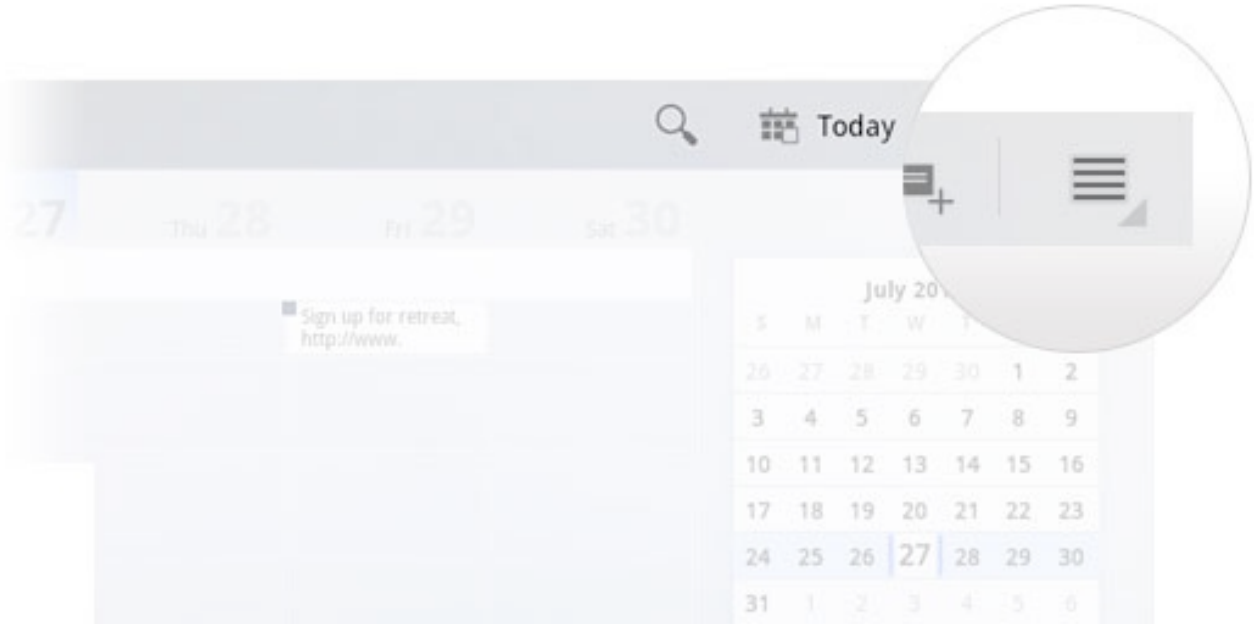
Common actions, as the name implies, gives user such things as search, share and an overflow menu. They are always positioned to the right of the action bar, away from any tabs.



Common actions in the Calendar app.

OVERFLOW MENU

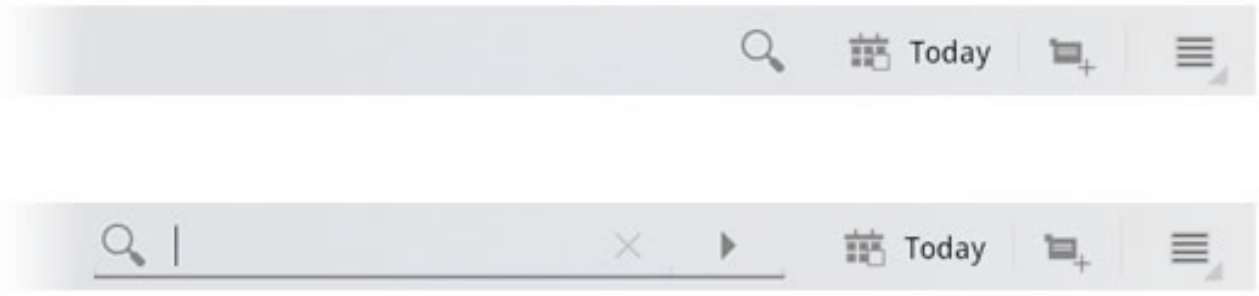
The overflow menu is part of the common actions group and is sometimes separated by a vertical rule. It is a place for miscellaneous menu items, such settings, help and feedback.



An overflow menu.

SEARCH

Search is also a part of the common actions group. Unique to search is its expand and collapse action. Tap on the search icon and a search box expands out, letting you enter a query. Tap the “x” to cancel, and it collapses to its single-button state. This is a space saver when many actions or tabs need to be shown.



The search function collapsed (top) and expanded (bottom). Tapping the magnifying glass opens the search box, while tapping the “x” closes it.

CONTEXTUAL ACTIONS

Contextual actions change the format of the action bar when an item is selected, revealing options unique to that item. For example, if a photo app is displaying thumbnails, the action bar might change once an image is selected, providing contextual actions for editing that particular image.

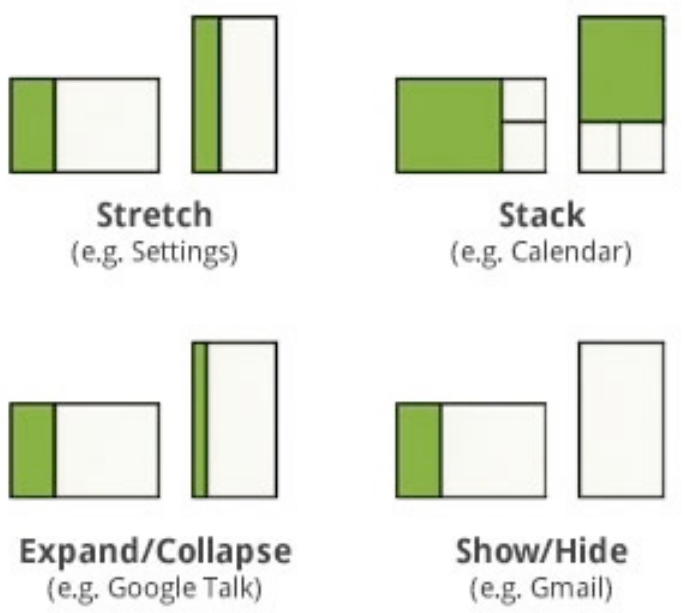
To exit the contextual action bar, users can tap either “Cancel” or “Done” at the far right of the bar.

The contextual action bar is triggered when tapping and holding an email in the Gmail app.

Tablet Layout Strategies

USING FRAGMENTS AND MULTI-PANE VIEWS

The building blocks of Honeycomb design are "Fragments." A Fragment is a self-contained layout component that can change size or layout position depending on the screen's orientation and size. This further addresses the problem of designing for multiple form factors by giving designers and developers a way to make their screen layout components elastic and stackable, depending on the screen restraints of the device that is running the app. Screen components can be stretched, stacked, expanded or collapsed and shown or hidden.

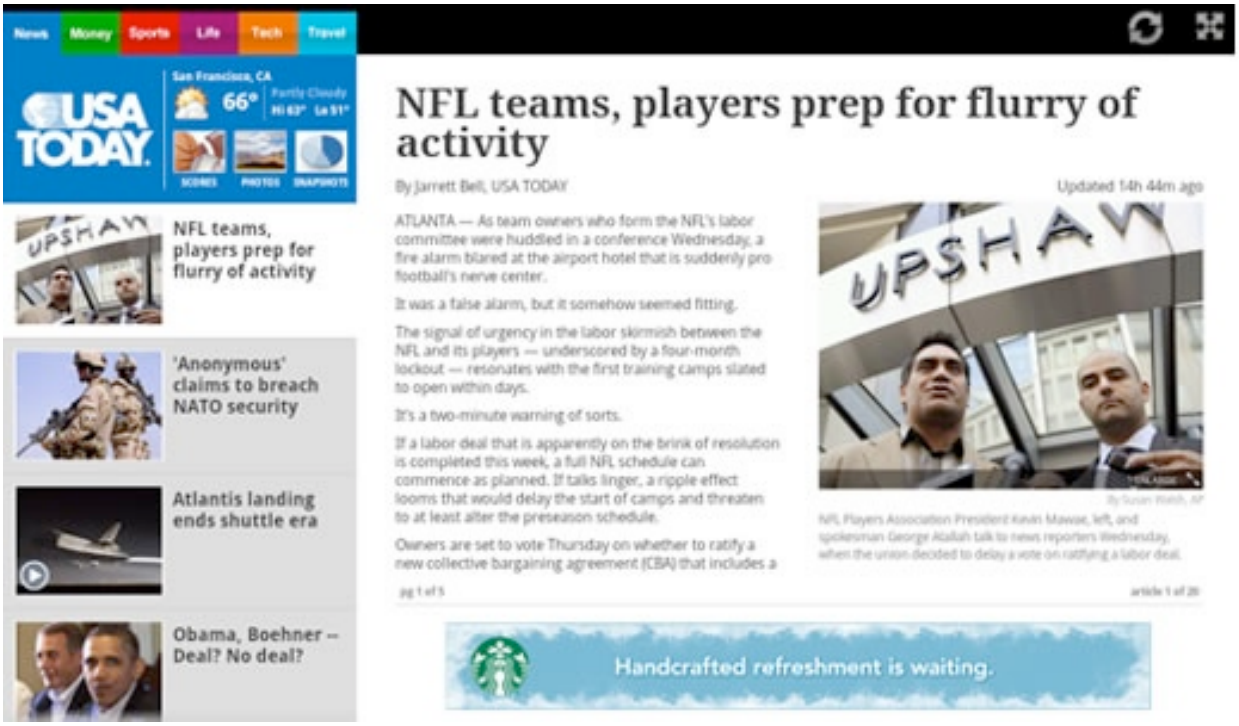


The Fragments framework gives designers and developers several options for maintaining their layouts across screen sizes and orientation modes.

What makes Fragments so special? With a compatibility library, developers can bring this functionality to Android smartphones going back to version 1.6, allowing them to build apps using a one-size-fits-all strategy. In short, it enables designers and developers to build one app for everything.

While Fragments may be a term used more by developers, designers should still have a basic understanding of how capsules of content can be stretched, stacked, expanded and hidden at will.

The most common arrangement of Fragments is the split view. This layout is common in news apps and email clients, where a list is presented in a narrow column and a detailed view in a wider one.



The split view used by USA Today

Another way to present a split view is to turn it on its side. In this case, the sideways list Fragment becomes a carousel, navigating horizontally instead of vertically.

ORIENTATION STRATEGIES

While Fragments are great for applying one design to multiple screen sizes, they are also useful for setting orientation strategies. Your screen design might look great in landscape view, but what will you do with three columns in a narrow portrait view? Again, you have the option to stretch, stack or hide content. Think of Fragments like a bunch of stretchy puzzle pieces that you can move around, shape and eliminate as needed.

A Word About Animation

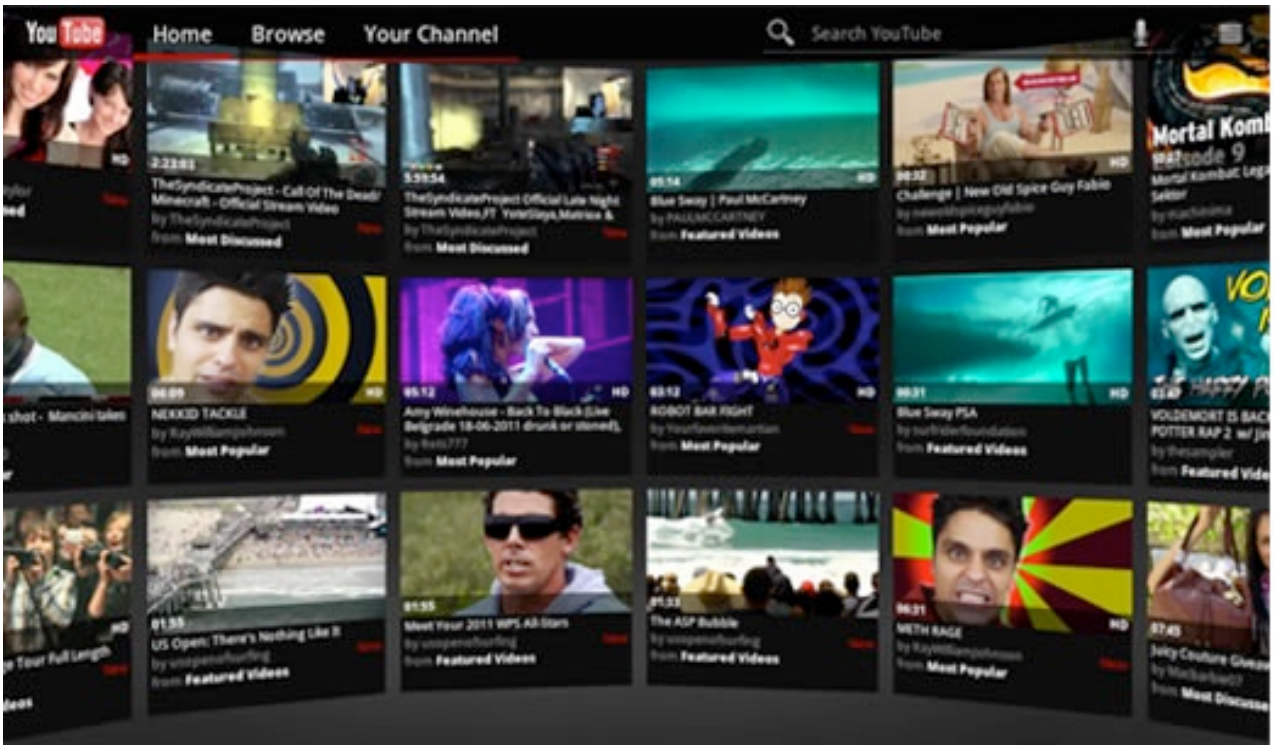
The Honeycomb framework allows designers and developers to use a variety of animation effects. According to the Android 3.0 Highlights page, "Animations can create fades or movement between states, loop an animated image or an existing animation, change colors, and much more." Honeycomb also boasts high-performance mechanisms for presenting 2-D and 3-D graphics. For a good overview of what Honeycomb is capable of, check out this video.

Learning from Example

Android tablet apps are still a relatively new space, and some brands are only beginning to test the waters. Below is a list of apps for inspiration. You can download any of them from the Android Market or Amazon.

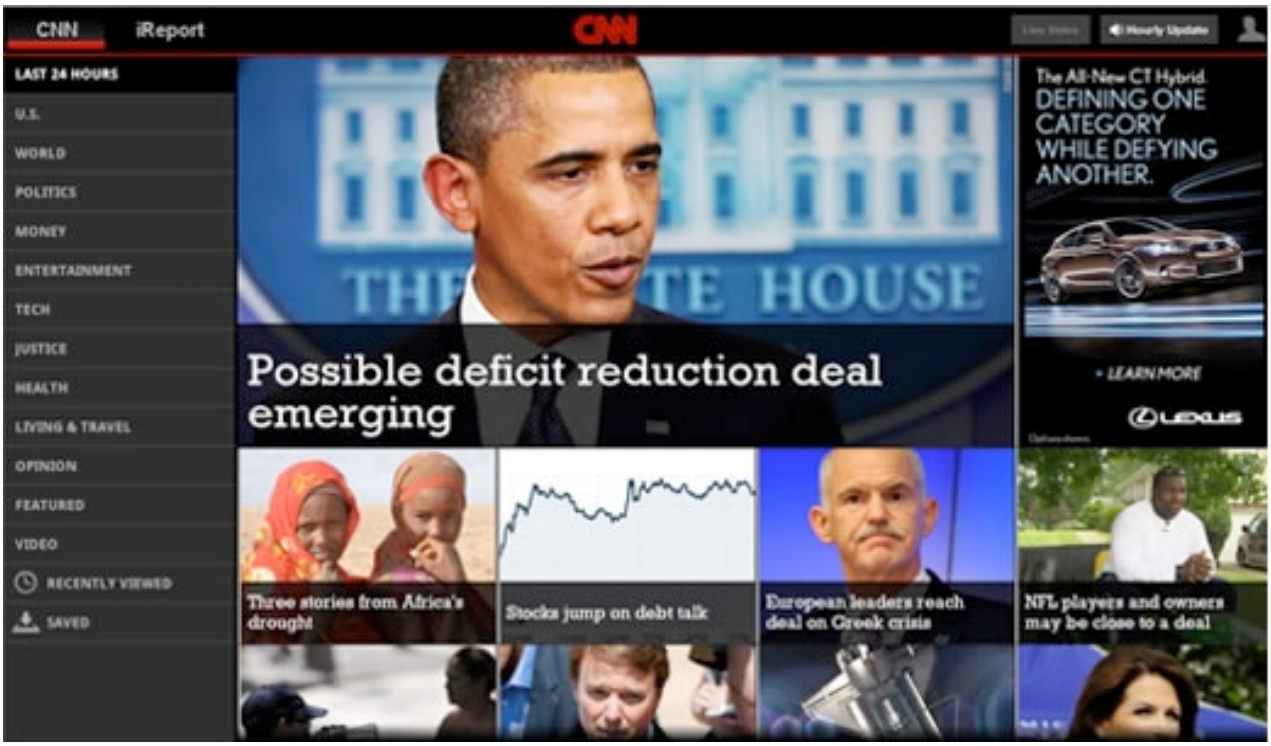
YouTube

Naturally, Google's YouTube app for Honeycomb is exemplary, showcasing all of the design patterns and UI elements discussed above. To get a good feel for Honeycomb, download this app first and take it for a spin.



CNN

The CNN app makes good use of touch gestures (including flicking to view more content), the split view and fonts! A custom font (Rockwell) is used for news headlines.



CNBC

Another good news app, with animation (the stock ticker tape) and rich graphics and gradients. CNBC has one of the most visually compelling apps.

Plume

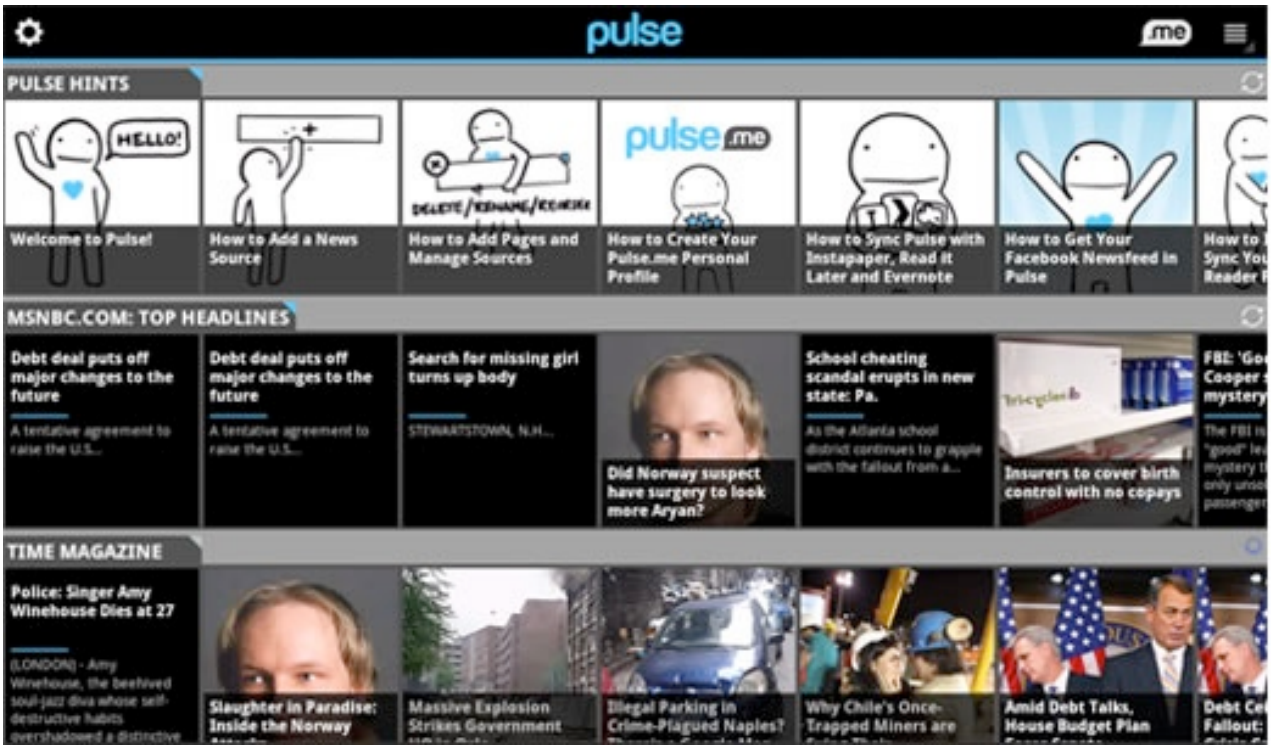
With its three-column layout, Plume is a good example of how layouts might need to be changed dramatically from landscape to portrait views.

FlightTrack

A data-heavy app done elegantly. Includes nice maps, subtle animation and standard Honeycomb UI elements.

Pulse

What else can you say: it's Pulse for Android tablets! But comparing the Android and iPad versions, which are identical in almost every way, is still fun anyway.

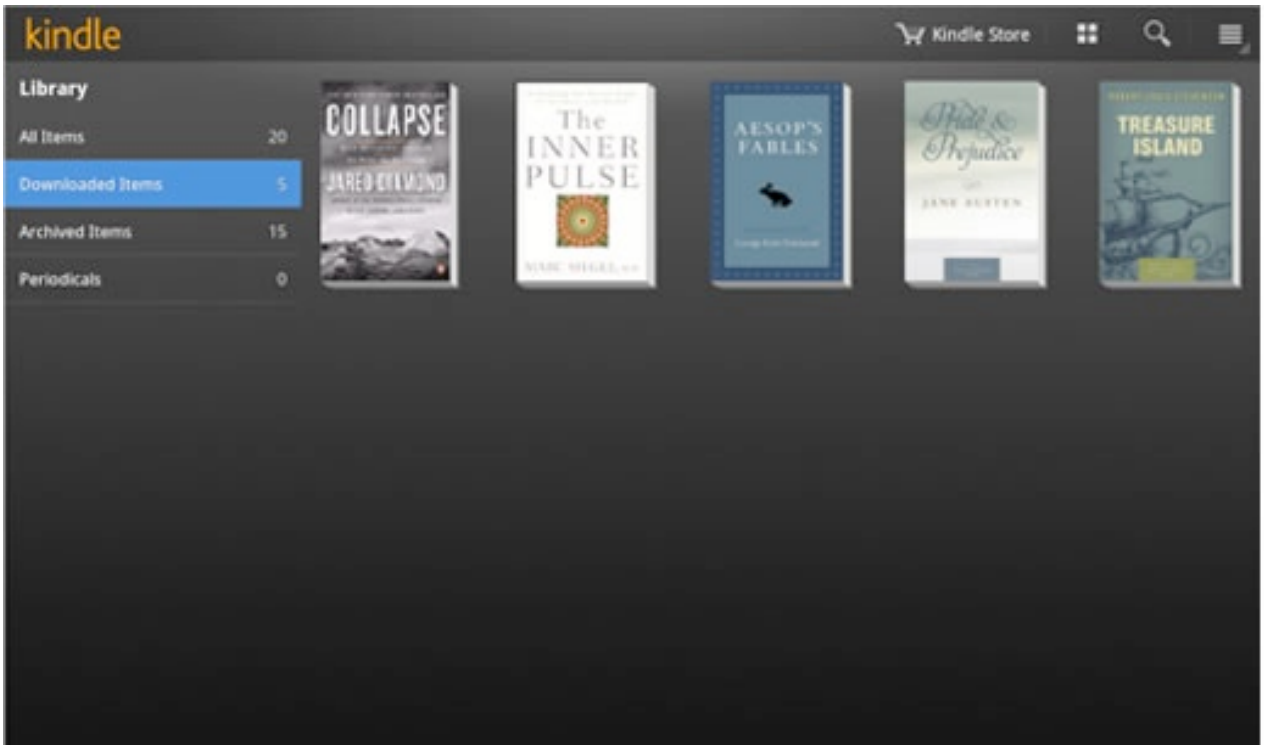


WeatherBug

This was one of the first Honeycomb apps in the Android Market. It makes good use of maps and of the holographic UI for showing pictures from weather cams.

Kindle

Kindle pretty much sticks to the book in using design patterns and Honeycomb UI elements. The outcome is elegant, while staying true to Android's best practices.



HONORABLE MENTIONS

- IMDb
- News360
- USA Today
- AccuWeather
- Ustream
- Google Earth
- Think Space

Getting The Best Out Of Eclipse For Android Development

Sue Smith

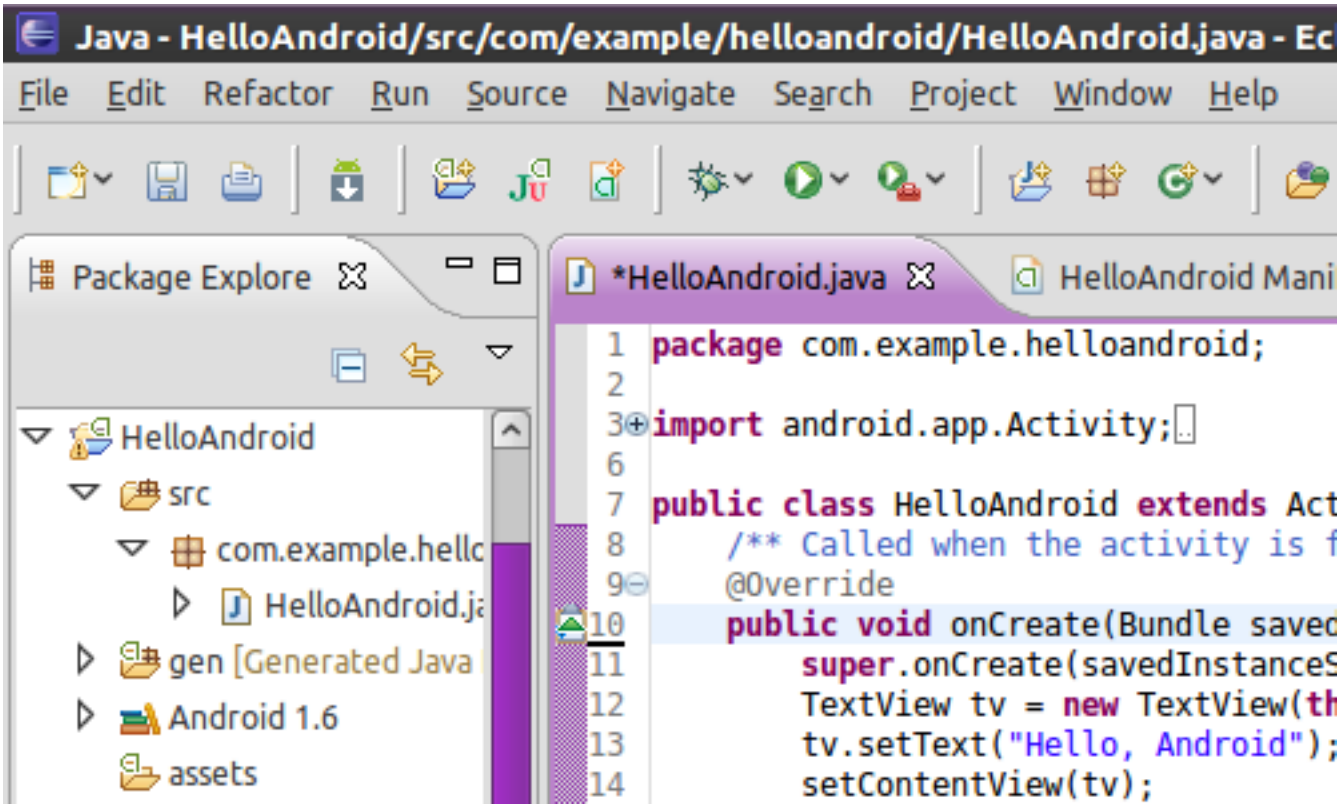
Getting into Android development can be quite a challenge, particularly if you're new to Java or Eclipse or both. Whatever your past experience, you might feel tempted to start working away without checking that you're making the best use of the IDE. In this article, we'll go over a few tips, tools and resources that can maximize Eclipse's usefulness and hopefully save you a few headaches. You might of course already be familiar with some (or all) of them and even be aware of others that we haven't covered. If so, please do feel free to mention them.

I've used Eclipse for Java development on and off for a few years, having recently started learning Android casually. I'm surprised at the lack of easily digestible material online about basic aspects of Android development, such as the topic of this article. I've found some useful information out there in disparate locations that are not particularly easy to come across. Most of the online content is still in the official Android Developer Guide, but it has to be said that it is pretty limited in practical material.

The aim here, then, is to provide a concise overview of Android development tools specifically in Eclipse. If you've already started developing for Android, you will almost certainly have come across some of them, but a few might be new to you, especially if you're learning Android casually or part time. If you're approaching Android as a Java developer and are already accustomed to Eclipse, you'll likely grasp this material well.

Get To Know Eclipse

Going over some features of Eclipse itself that would be useful for developing Android projects would be worthwhile. If you already know your way around Eclipse, you can probably skip these first few sections, because they're really aimed at people who are learning to use the IDE purely for Android development. Later sections include tips on using Eclipse specifically for Android, so you might find a bit or two in there that you haven't explored yet.



The Eclipse IDE, with the "Hello Android" project open.

Eclipse has a huge amount of flexibility for creating a working environment for your development projects. Many display options, tools and shortcuts in Eclipse enable you to interact with your projects in a way that will make sense to you directly. Having been an Eclipse user for a reasonable amount of time now, I still discover features in it that I had no idea existed and that could have saved me a lot of hassle in past projects.

Explore Perspectives

The Eclipse user interface provides a variety of ways to view the elements in any project. When you open Eclipse, depending on which “perspective” is open, you will typically see the screen divided into a number of distinct sections. In addition to the code editing area, you should see various aspects of the project represented in particular “views.”

A perspective in Eclipse is a group of views arranged to present a project in a particular way. If you open the “Window” menu, then select “Open Perspective,” you will usually see at least two options: “Debug” and “Java.” The Java perspective will likely form the basis of your Android development process; but for debugging, there is an Android-specific perspective called DDMS. Once you have set a perspective, you can still make alterations to the visible views, as well as adjust the layout by dragging the view areas around to suit yourself.

In general, when developing Android projects, some variation of the Java perspective is likely to be open, with the option of a few additional views, such as “LogCat,” where you can send and read the runtime output of your app. The Dalvik Debug Monitor Server (DDMS) perspective will likely be useful when the time comes to debug your Android applications. The tools in this perspective come as part of the Android software development kit (SDK), with Eclipse integration courtesy of the Android Developer Tools (ADT) plugin. The DDMS perspective provides a wide range of debugging tools, which we’ll cover in brief later on.

Make Use Of Views

By default, the Java perspective contains the main area to edit code and a handful of views with varying levels of usefulness for Android projects:

- **Package Explorer**
  This provides a hierarchical way to explore your projects, allowing you to browse and control the various elements in them through a directory structure.
- **Outline**
  Displays an interactive representation of the file currently open in the editor, arranged as logical code units, which can be useful for quickly jumping to a particular point in a big file.
- **Problems**
  Lists errors and warnings generated at the system level during the development process.
- **Javadoc**
  Useful if you’re creating your own documentation using Javadoc or using other language resources that have this documentation.

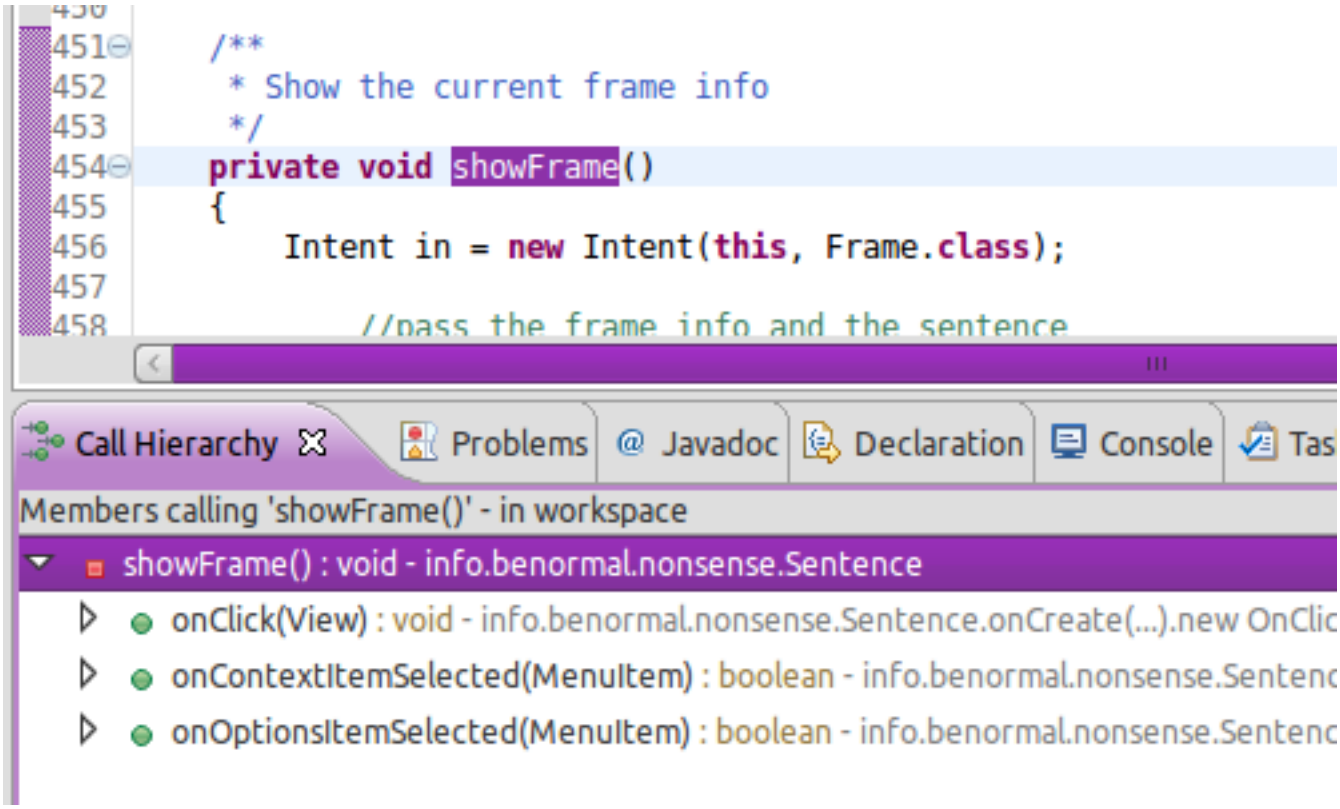
- **Declaration**
  Particularly useful if your Android projects have a lot of classes and methods. You can click on a variable or method in your code to see its outline here, and then click within the view to jump to the point where the item is declared.

You can open new views in Eclipse by selecting “Show View” in the Window menu and clicking from there. Many of the available views won’t likely be of any use to your Android project, but it is worth experimenting with them. If you open a view and decide that you don’t want to display it after all, simply close it. Adjust or move open views to different locations in Eclipse by clicking and dragging their edges or the tabbed areas at the top.

JAVA VIEWS RELEVANT TO ANDROID DEVELOPMENT

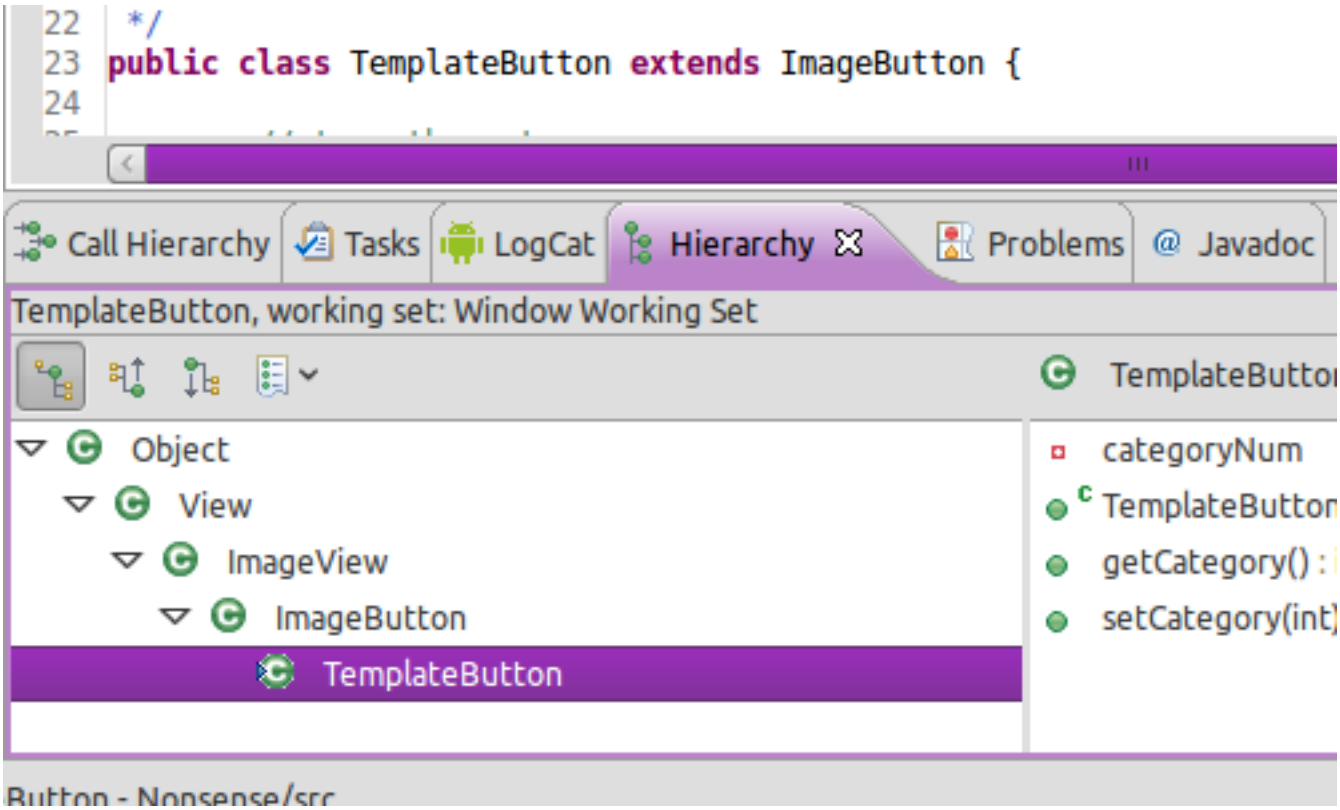
The ADT plugin for Eclipse includes a variety of resources for Android development within the IDE, some of which we’ll cover in the sections below. Eclipse has additional views for general Java programming, some of which naturally have a greater potential for usefulness in Android development but don’t appear by default in the Java perspective. In my experience, views that are useful for Android development include “Call Hierarchy,” “Type Hierarchy” and “Tasks.”

The Call Hierarchy view displays an interactive list of the calls for any particular method, class or variable, so you can keep track of where your code excerpts are being called from. This is particularly useful if you're altering existing code and need a sense of what will be affected.



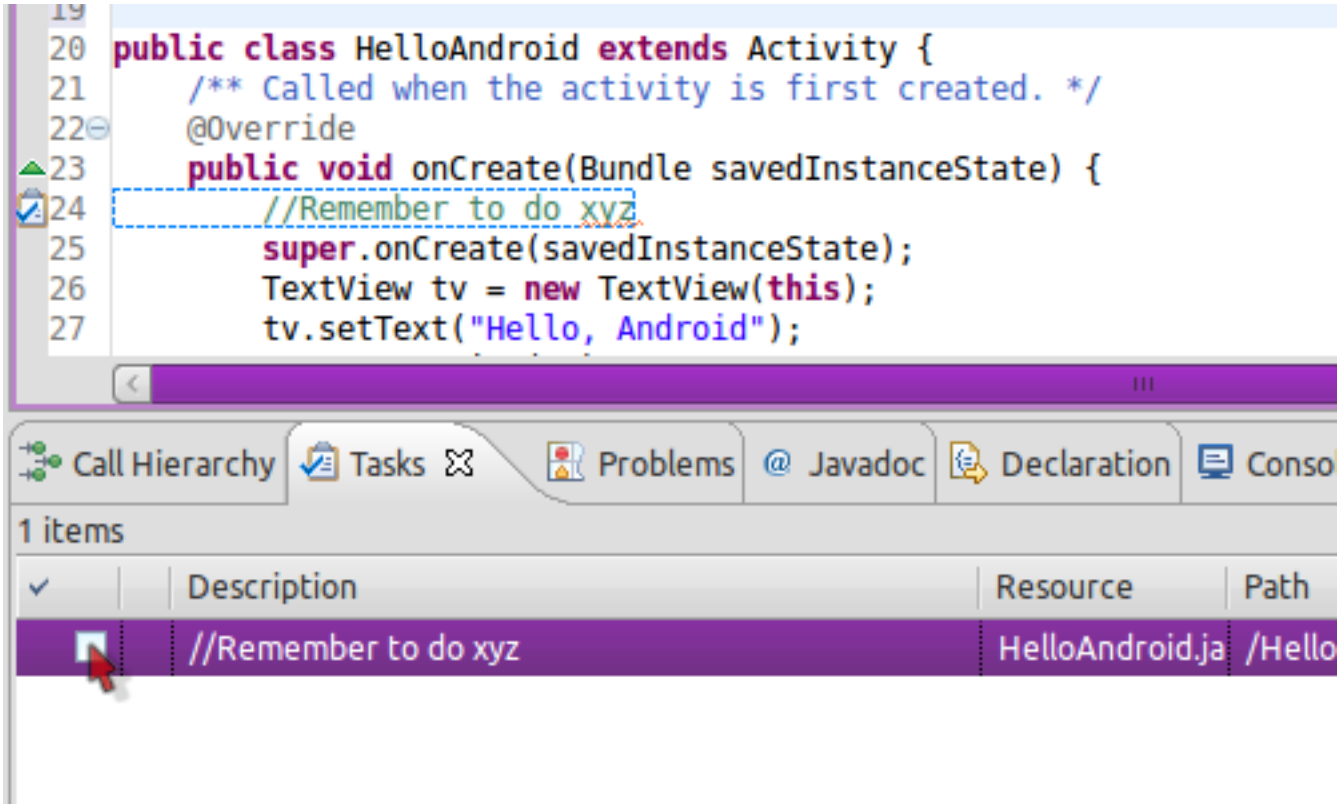
The Call Hierarchy view for a method in one of my projects.

The Type Hierarchy view is relevant to Java projects that involve inheritance, which basically means that it's relevant to all Android projects. Whenever you have a class that extends another class, this is inheritance and, thus, an instance when the Type Hierarchy view can be informative. If you're not yet familiar with inheritance in Java, taking the time to at least read up on the basics is well worth it, because the technique is key to Android programming.



The Type Hierarchy for a user interface element in an Android project.

The Tasks view is one I personally find helpful, but it really depends on how you choose to organize your projects. Think of the Tasks view as an interactive to-do list. For example, to add a task when you haven't quite finished a particular bit of implementation but need to address something else right away, write a code comment, then right-click in the border area just to the left of the editor section, choose "Add Task," and set your options from there. Once tasks are in a project, you can read them in the Tasks view, jumping to an individual task in your code by selecting it here. Tasks are also highlighted in the right-hand border of the editor.



The Tasks view, with a random comment in the "Hello Android" project.

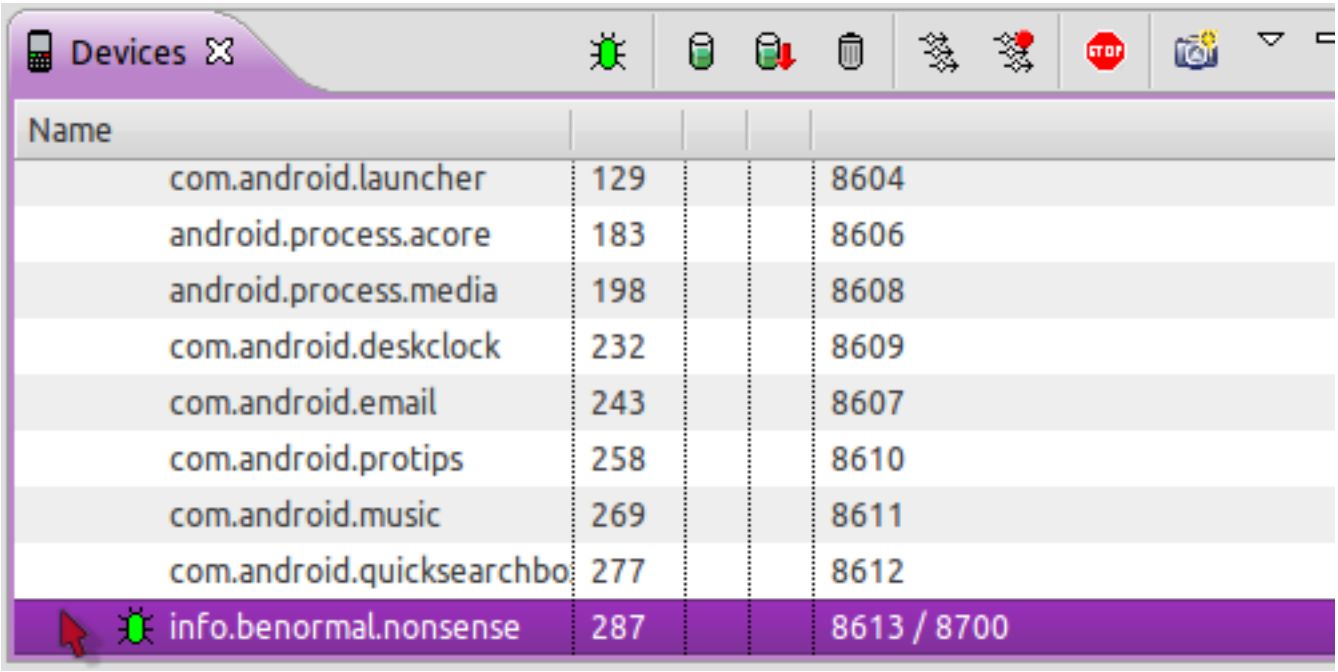
ANDROID DEBUGGING VIEWS

The DDMS perspective contains several views for Android development and debugging. When you first open the perspective, you might feel a little overwhelmed, particularly because the views are all presented side by side in a rather convoluted arrangement. Chances are that once you learn the purpose of one, you'll decide whether it's relevant to your project; for a basic project, many of them will not be.

If you feel that closing some views would make the interface an easier environment to work in, then go ahead and close them; you can always open them again later. Starting simple is sometimes best, and then adding elements over time as they become relevant. In the early stages, your Android project is not likely to be complex anyway.

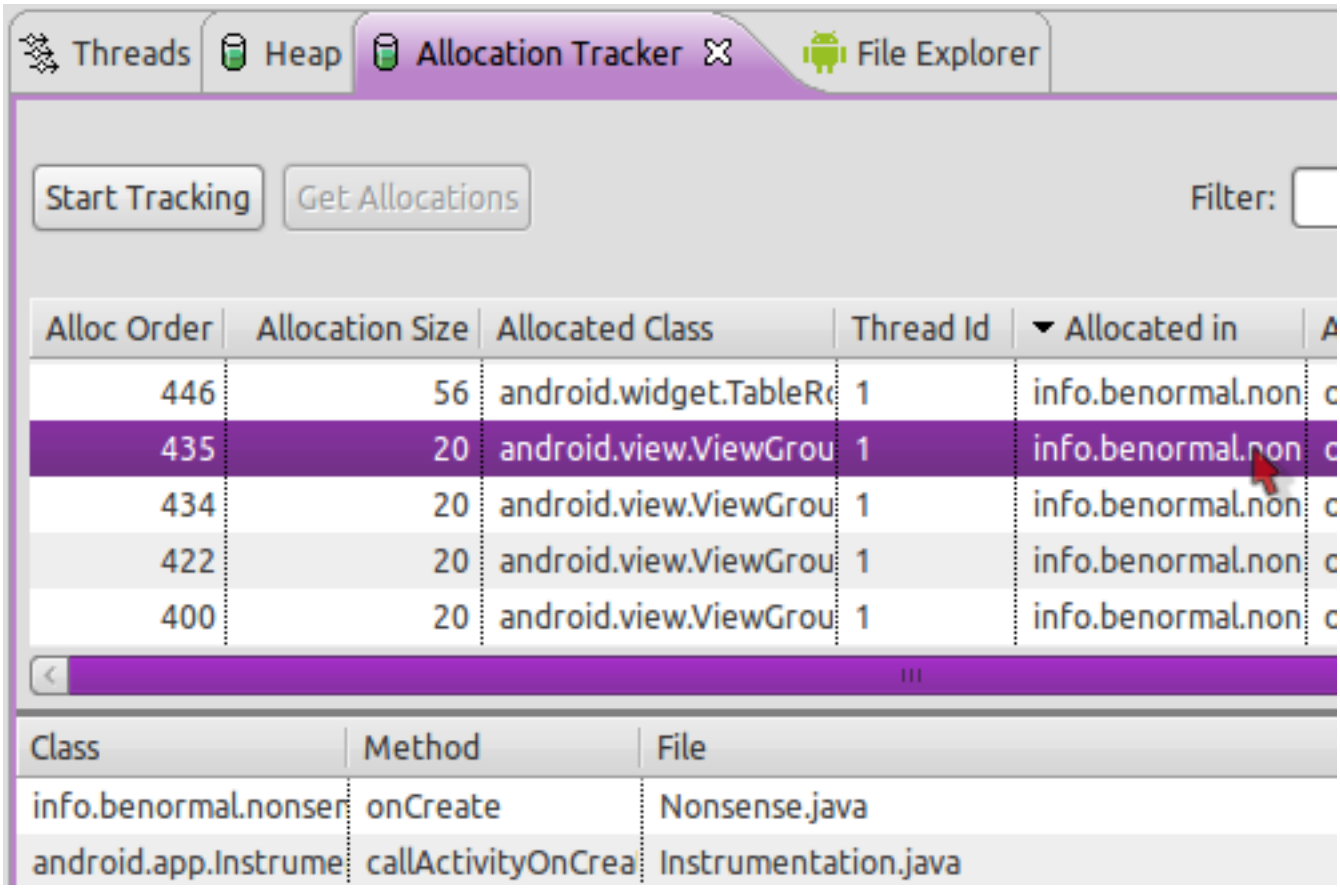
Let's go through each DDMS view in Eclipse, with a brief note on its purpose and use. The information is tailored to each view and so is pretty varied, but includes device, memory and process management.

The Devices view provides an overview of your running AVD emulators together with processes in operation, and it is the starting point for exploring your projects for the purpose of debugging. This view presents shortcuts to some debugging operations, so you can select your app's processes from here to see the other debugging operations in action. In most cases, you can view information in the other DDMS views by selecting a running process in the Devices view while your app is running in an emulator, then using the buttons at the top of the view to capture debugging information. When you do this, you'll see some of the other views being populated with data.



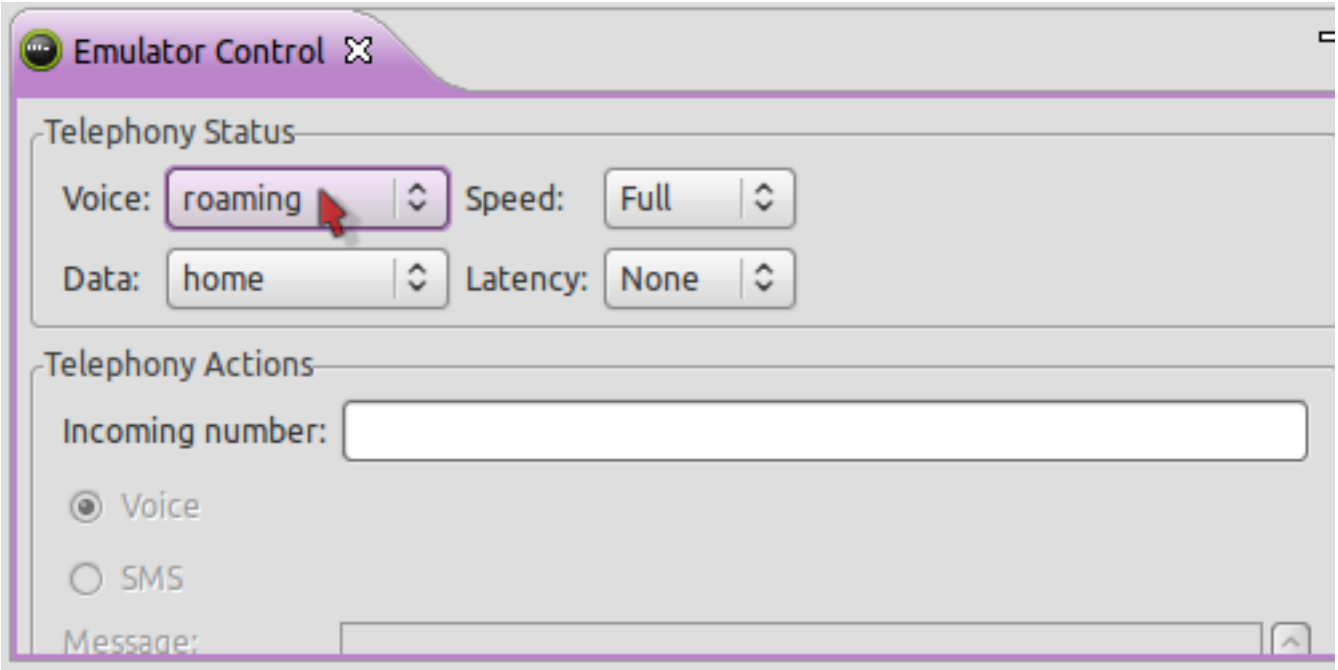
The Devices view, with a process selected for debugging.

The Allocation Tracker view is particularly useful for apps that have significant demands on performance. This view provides insight into how the Dalvik Garbage Collector is managing memory for your app. If you're not familiar with garbage collection in Java, you might want to read up on it, as well as familiarize yourself with basic principles of efficiency in Java programming, such as variable scope.



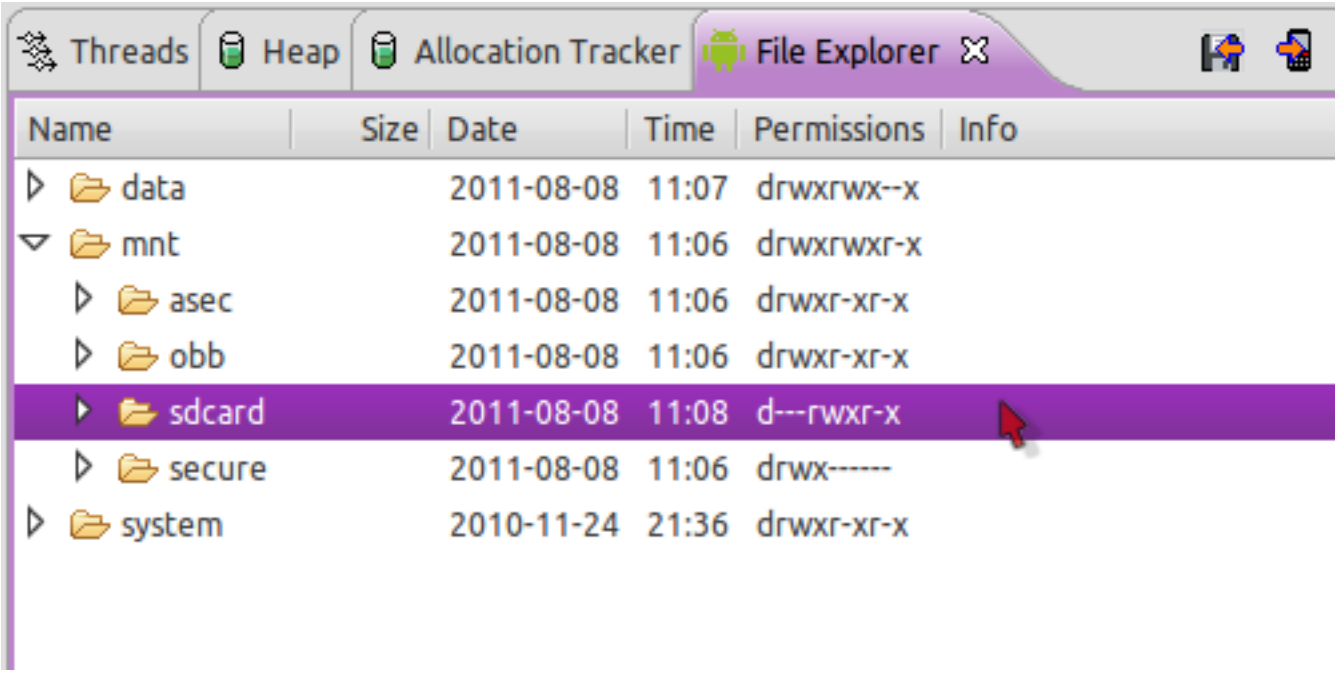
The Allocation Tracker view in the DDMS perspective.

The Emulator Control view enables you to emulate specific network contexts when testing your apps, allowing you to see how they function with varying constraints on speed, latency and network connectivity. A vital tool if your app depends on such external connections.



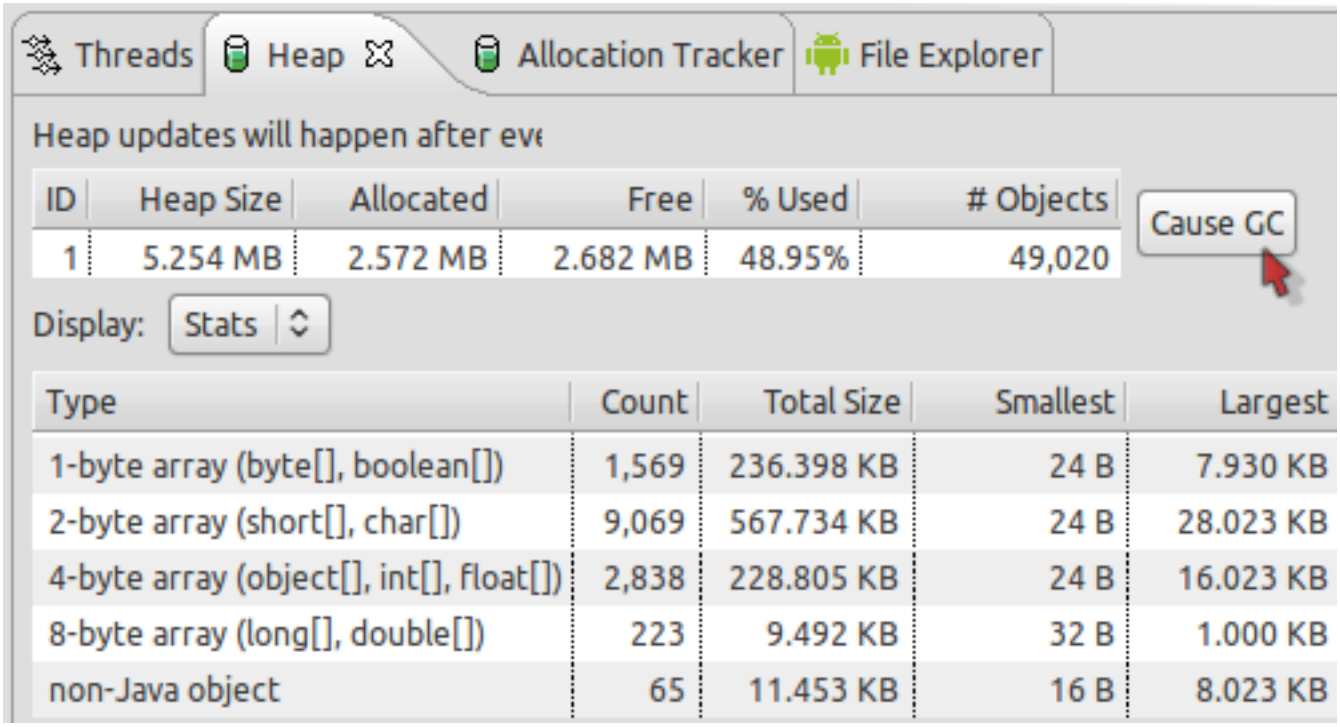
The Emulator Control view with telephony settings.

The File Explorer view enables you to access and control the device's file system (not the file system in your application's structure as with the Package Explorer view). Using this, you can copy files to and from the device's storage system; for example, in the case of an app storing data on the user's SD card.



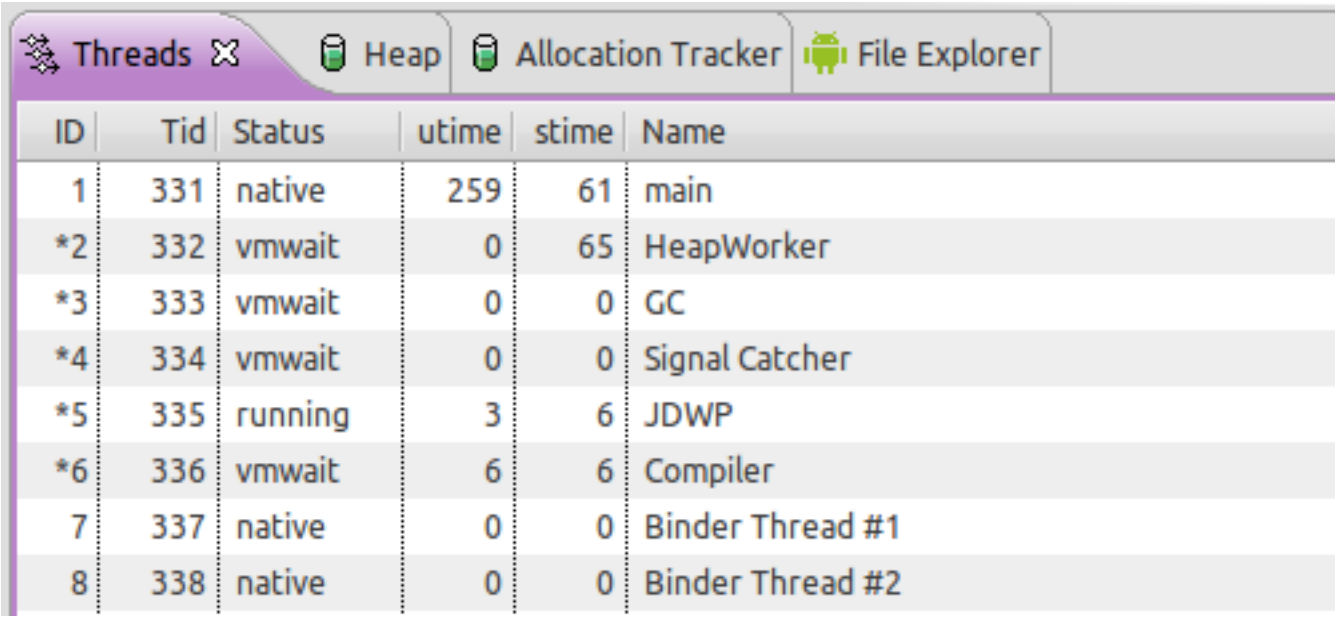
Looking at the SD card directory on a virtual device using the File Explorer view.

The Heap view is another tool for analyzing and improving memory management in your app. You can manually control the Garbage Collector from this view. If you are unclear on how the constructs of your Java code relate to heap memory, this is another area you might want to read up on.



The Heap view for a running app.

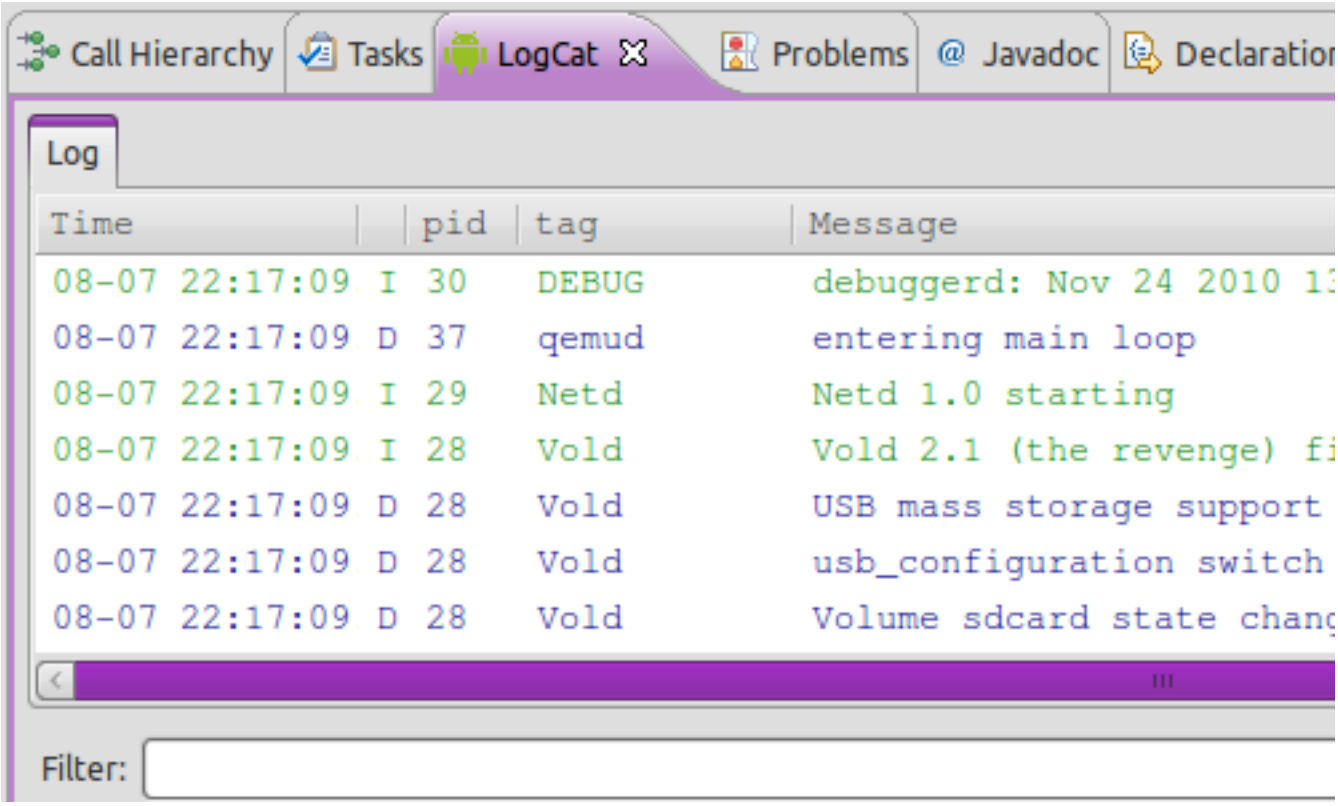
The Threads view enables you to access information about the threads running in your application. If you are unfamiliar with the concept of threads, then this view won't likely be of use to you. Typically, an Android application runs in a single process on a device, but if you implement a deeper level of control over your app's processing at the level of threads, then this view will be relevant.



ID	Tid	Status	utime	stime	Name
1	331	native	259	61	main
*2	332	vmwait	0	65	HeapWorker
*3	333	vmwait	0	0	GC
*4	334	vmwait	0	0	Signal Catcher
*5	335	running	3	6	JDWP
*6	336	vmwait	6	6	Compiler
7	337	native	0	0	Binder Thread #1
8	338	native	0	0	Binder Thread #2

The Threads view for a particular running process.

The LogCat view is useful for most Android apps, regardless of their complexity. A variety of messages are automatically outputted here when you run your app, but you can optionally use the view as your own custom log. You can create log messages in various categories, including “Debug,” “Info,” “Warning,” “Error” and “Verbose,” which is typically used for low-priority information such as development trace statements. In the LogCat view, you can then filter the messages in order to quickly scroll to those you’re interested in. The messages are also color-coded, making them a little easier to absorb.



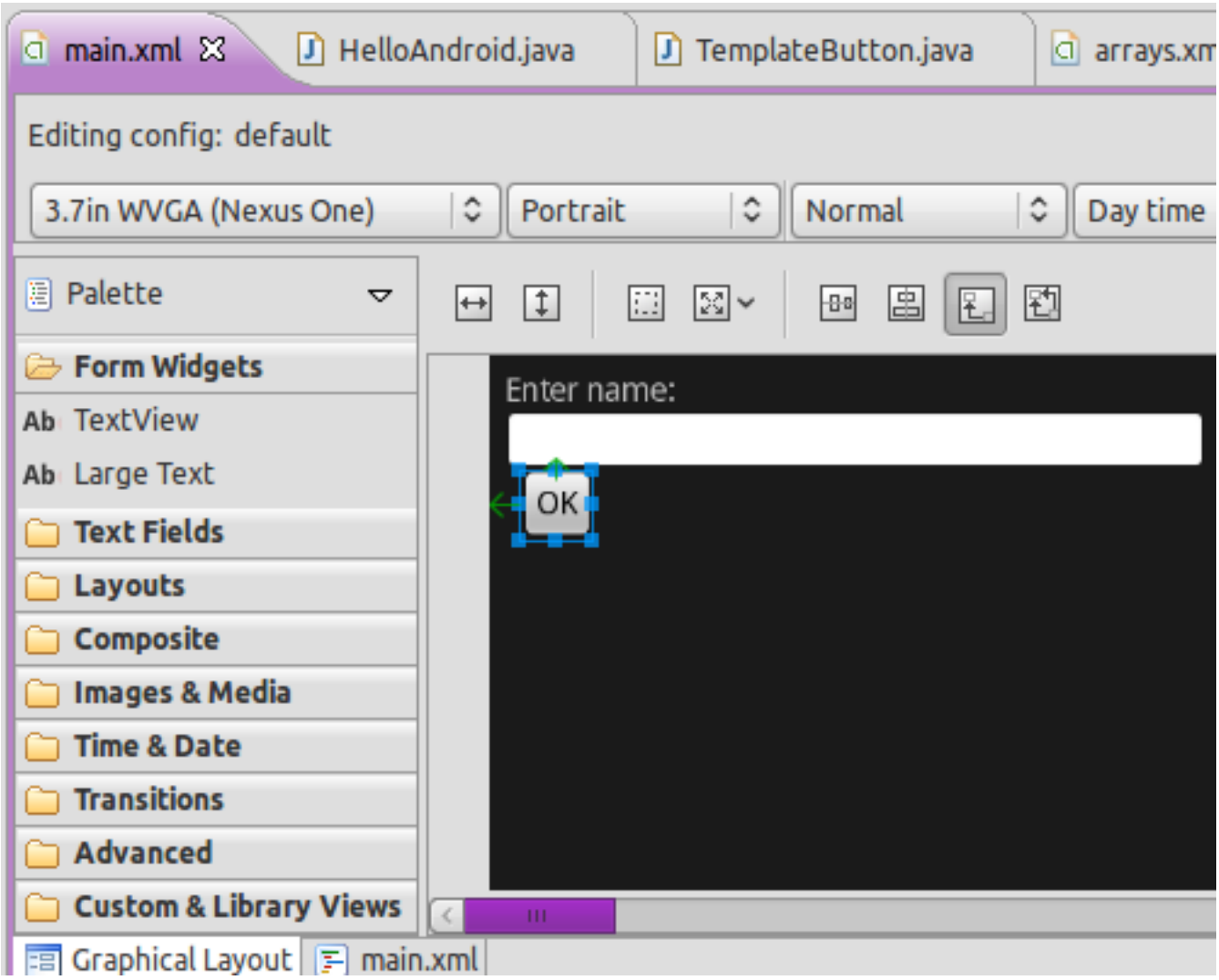
The LogCat view as a virtual Android device starts up.

A DDMS perspective view can optionally be added individually to the Java perspective if you find it useful for development as well as debugging. Eclipse is designed to enable you to interact with the project's elements any way you like, so don't be afraid to experiment with the interface.

Exploit The Design Tools

The design tools in the ADT have undergone major improvements in recent releases, so if you've been disappointed by them in the past and perhaps opted for other tools to design your Android apps, having another look at them now is well worth it.

In the Graphical Layout view, you can view your designs with various settings for device software, hardware as well as orientation. The current version generally gives a greatly improved level of control over widgets, views and rendering than most previous versions. You can manage user-interface elements here using the graphical interface, rather than having to implement every aspect of your design in XML code.

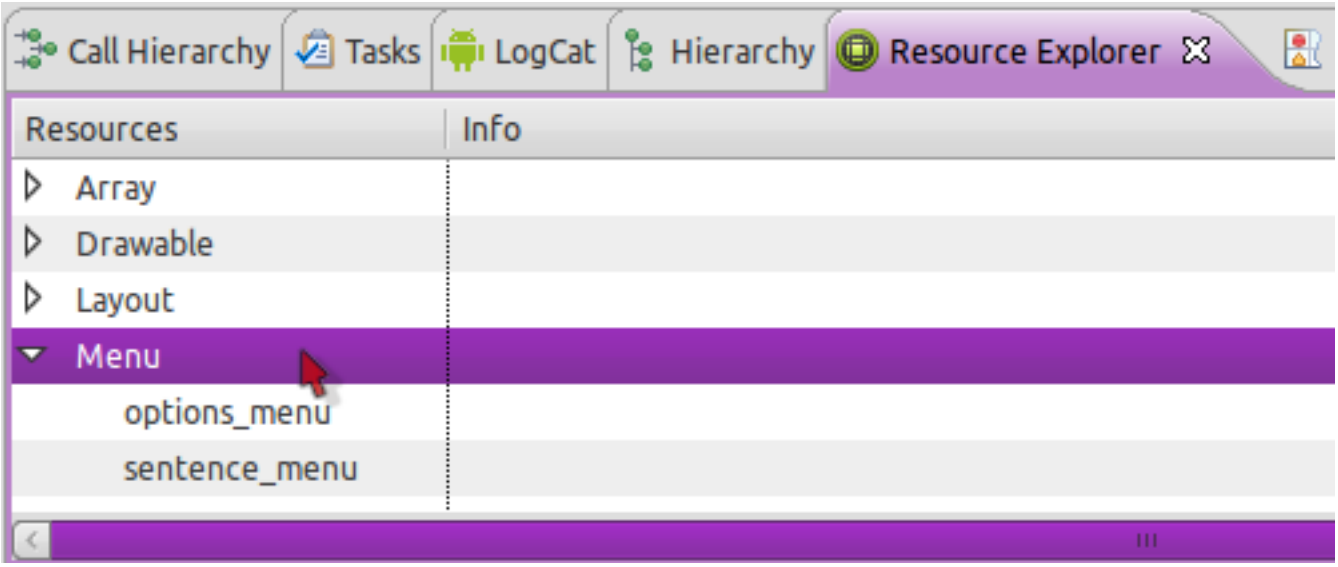


The graphical view of an app's XML layout.

DESIGN AND GRAPHICAL UTILITIES

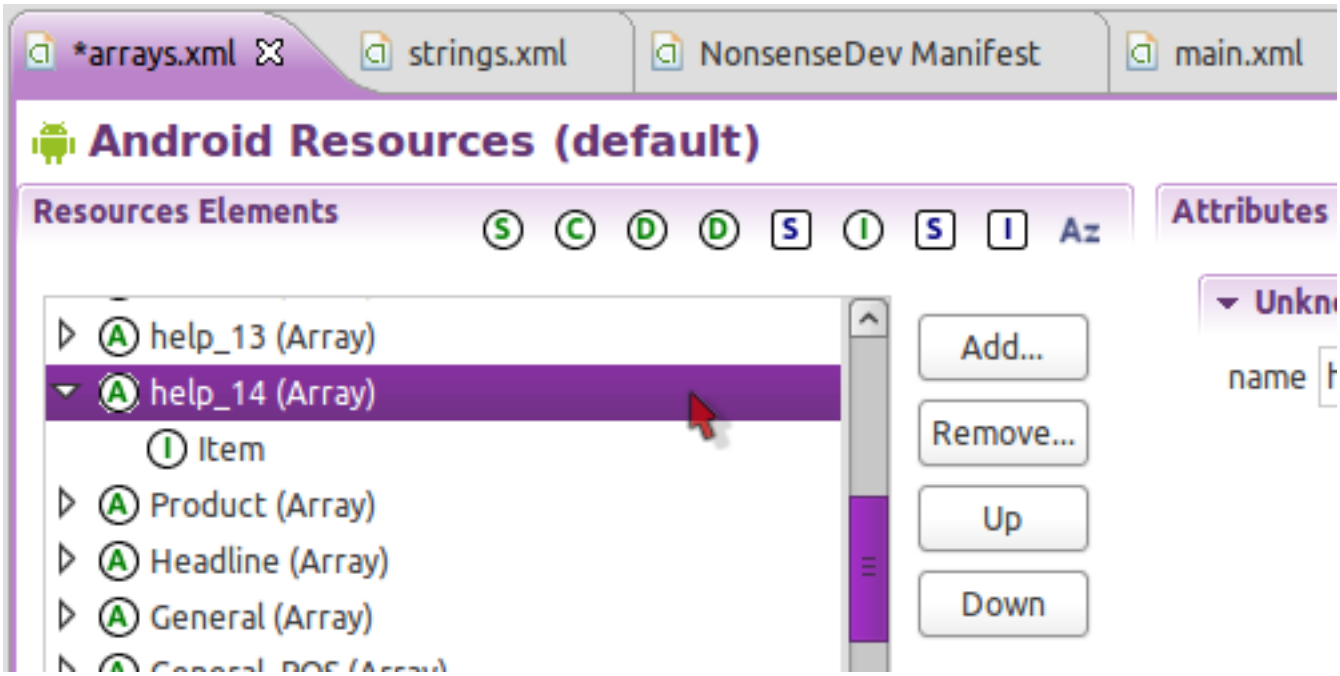
The ADT plugin update of June 2011 introduced a variety of visual refactoring utilities. While your XML layout files are open in the editor, you can browse the options in Eclipse's Refactoring menu for Android, choosing from various labor-saving tools such as "Extract Style," which provides a shorthand method of copying style attributes for reuse.

The Resource Explorer is a handy interactive way to explore your application's resources without the hassle of searching the file structure in the "res" folder via the Package Explorer. Although not exclusively for design purposes, the tool is useful for checking what design resources are in your application's structure. Depending on the app, your Resources directory might also contain menu, array, string and value items.



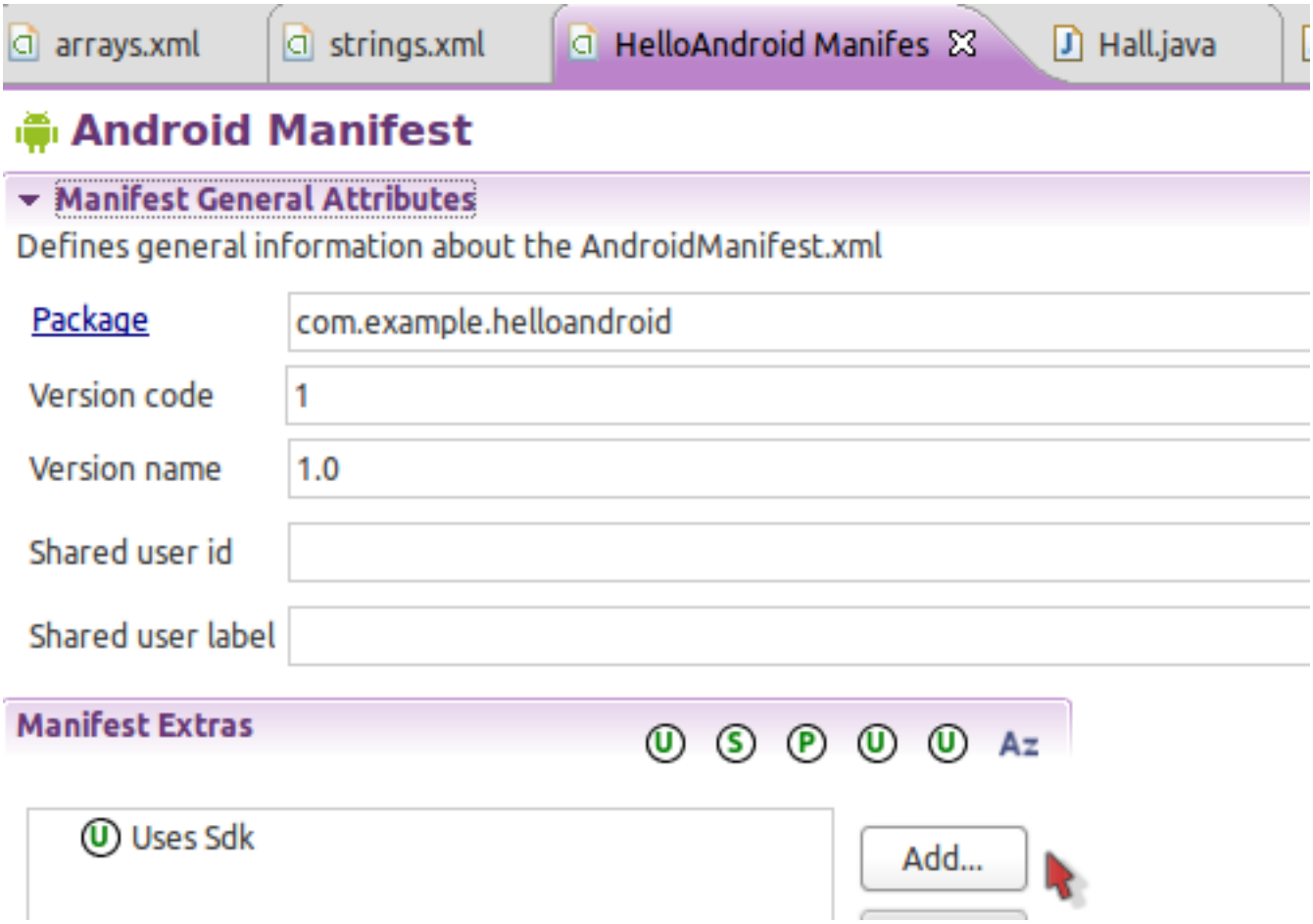
The Resources Explorer showing the Resources directory for an app.

On the subject of resources, the Resources interface for any XML files in your app enables you to manage such data graphically, if you prefer not to handle the XML code manually. This can be an effective way to manage both elements and attributes for your application's XML data.



A graphical view of the XML file for arrays in an app.

Various visual interaction tools for XML resources have gradually been added and improved in the ADT plugin. For example, the Manifest file can also be viewed graphically when you want to control the tags in it, including “Permissions,” “Application” and “Instrumentation.”



An application’s Manifest file, presented graphically.

Set Your Android Preferences

The “Eclipse Window” menu provides access to the Preferences settings for the environment as a whole, with a dedicated section for Android. You may have used this when you first installed the SDK but might not have been in there since then. It’s worth having a look through the options and experimenting with them to create the environment you want.

In the Android section, the “Build” settings include your saved edits to the project’s files, and they also point Eclipse to your keystore for signing the application. The LogCat section lets you set properties for appearance and behavior in the LogCat view. Other options include choosing properties for the DDMS, the code editors and the launch process for the emulator.

Use The Android Run Configuration Options

When you launch your Android project using the Run Configurations menu, you can choose from a range of additional options to suit the functionality of your app. With your Android project selected, choose the “Target” tab to see these options. The first section covers the more general launch parameters for the emulator, including control over speed and latency on the device. This feature is useful if your app depends on a network connection and you want to see how it functions with limited connectivity.

EMULATOR COMMAND-LINE OPTIONS

The “Target” tab also allows you to enter command-line options for the emulator that launches your app. Using these, you can tailor the appearance, properties and behavior of the emulator as it launches your app. A lot of options are here for various aspects of the device, including the following properties:

- **UI**
  Includes scaling the display window and setting the DPI.
- **Disk image**
  Includes the device's SD card and cache memory.
- **Media**
  Covers audio and radio properties.
- **Debugging**
  Enables and disables specific debug tags.
- **Network**
  In addition to network speed and latency, you can specify DNS and proxy servers.
- **System**
  Includes slowing down the device CPU and setting the time zone.

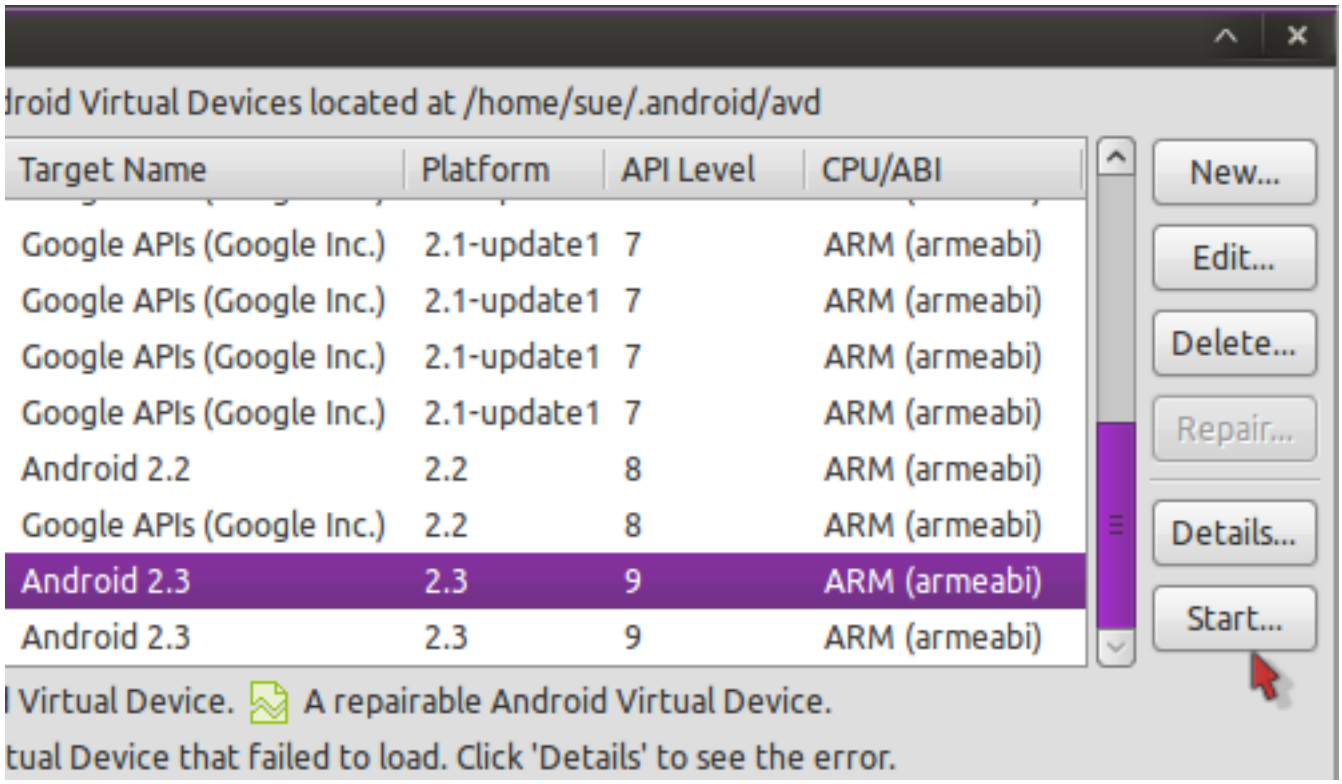
For a full list, see the Developer Guide.

Choose Useful AVD Configurations

When you create virtual devices in Eclipse using the Android SDK and AVD Manager, through the “Window” menu you can choose various settings, including API versions and hardware configurations, such as the size of the SD card and screen, plus optional hardware components. The sheer range of options can leave you wondering where to start, but the feature at least gives you *some* idea of how your apps will look and function on various devices.

Entering the details for a new AVD.

To create a virtual device, select “New,” enter a name and choose “Options.” You can edit your devices later if necessary. To start an AVD running, select it in the SDK and AVD Manager and click “Start.”



Choosing from a variety of AVDs to start running.

Some developers prefer configuration options that match the range of Android devices currently in use, which obviously vary widely, with multiple phone sizes and types, not to mention tablets. There is no dedicated resource for finding this data, but a number of websites list these configuration details periodically, so you can do a quick Web search when testing an app for release to find the most current information.

It must be said that there is a real limit in value to using actual device settings, because the emulator is merely a guide: it doesn't replicate the look and feel of different devices with any authenticity. These limitations are especially pertinent when you consider the third parties involved, such as HTC's Sense interface, which will substantially alter your application's UI. This is why testing on actual devices is also important, but that's another topic. The range of devices running Android will obviously continue to expand, so the feasibility of covering a significant chunk of them becomes more and more remote.

Rather than focusing on actual devices in use, another way to approach the AVD configuration settings while you're in Eclipse (i.e. when you're still developing rather than testing) is to simply try to cover a range of settings and to explore particular hardware and software configurations that could affect your app's functionality. Visually speaking, it goes without saying that a relative rather than fixed design will cope better with various screen sizes.

Get And Keep Yourself Acquainted With Developer Resources

As you're by now aware, if you weren't already, the Android Developer Tools plugin is what allows you to develop, run, test and debug Android applications in Eclipse, together with the software development kit. Like Eclipse itself, both the SDK and ADT plugins offer many tools, many of which you could very easily remain unaware of and only a few of which we have touched on in this article.

The Android development resources undergo regular updates. You might be one of those people who are always on the ball, but if you're anything like me, you'll rarely find the time to keep abreast of these developments, especially if (like me) Android development is not your full-time job.

The Android Developer Guide lists the various tools in the SDK, many of which you will not need to access directly, because the ADT plugin and Eclipse will handle interaction with them. But it's worth browsing the list fairly regularly, just to be aware of the tools available. Even if you're only getting into Android casually, these utilities can make an enormous difference to how productive and enjoyable your app's experience is. The Android Developers Blog is another good source to keep abreast of developments.

The ADT plugin itself comes with a wealth of features and is regularly updated. The Android SDK Guide also has a page on the plugin, outlining versions and features; again, well worth checking out regularly, because there may be significant updates. Checking for updates in Eclipse through the "Help" menu is another good habit.

Android apps are, of course, extremely varied, so the utility of any given development tool is necessarily relative. Keeping a eye on updates to resources needn't take up much time, because in many cases the updates will be irrelevant to you. But discovering the ones that *are* relevant could seriously improve your Android development efforts.

Following Best Practices

If you're interested in developing a best-practices approach to managing and documenting your apps, check out Apache Maven, which uses the project object model (POM). If you've already used Maven for Java development in Eclipse, the Android plugin and Integration tools will enable you to adopt the same development practices for your Android app. You can learn more about Maven on the Apache website, and you can learn about getting started with it for Android on the Sonatype website.

Get Started Developing For Android With Eclipse

Chris Blunt

There's a lot to get excited about in mobile application development today. With increasingly sophisticated hardware, tablet PCs and a variety of software platforms (Symbian OS, iOS, WebOS, Windows Phone 7…), the landscape for mobile developers is full of opportunities — and a little complex as well.

So much choice can be overwhelming when you just want to get started building mobile applications. Which platform should you choose? What programming language should you learn? What kit do you need for your planned project? In this tutorial, you'll learn how to start writing applications for Android, the open-source mobile operating system popularized by Google.

Why Develop for Android?

Android is an open-source platform based on the Linux kernel, and is installed on thousands of devices from a wide range of manufacturers. Android exposes your application to all sorts of hardware that you'll find in modern mobile devices — digital compasses, video cameras, GPS, orientation sensors, and more.

Android's free development tools make it possible for you to start writing software at little or no cost. When you're ready to show off your application to the world, you can publish it to Google's Android Market. Publishing to Android Market incurs a one-off registration fee (US $25 at the time of writing) and, unlike Apple's App Store which famously reviews each submission, makes your application available for customers to download and buy after a quick review process — unless the application is blatantly illegal.

Here are a few other advantages Android offers you as a developer:

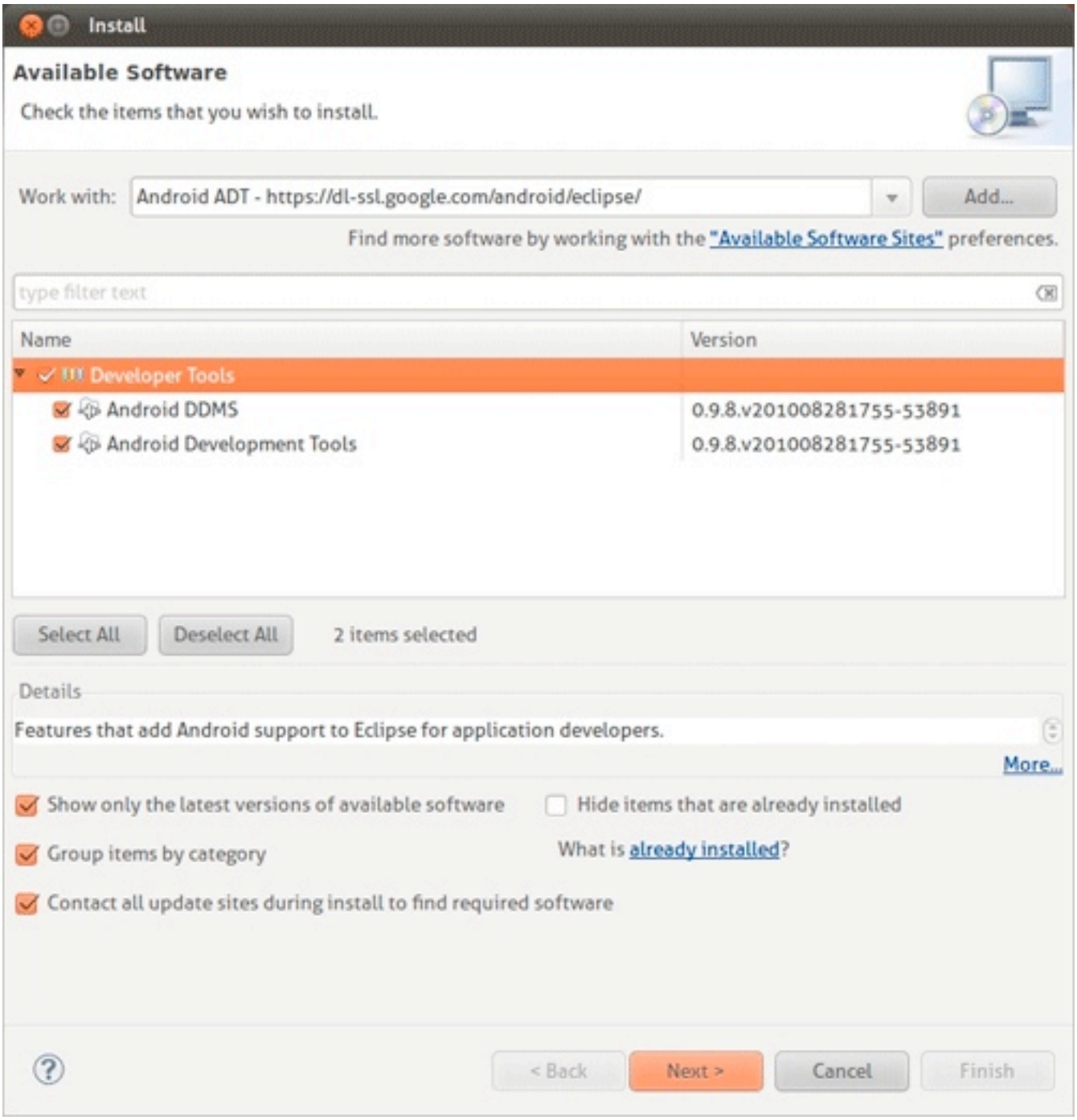
- The Android SDK is available for Windows, Mac and Linux, so you don't need to pay for new hardware to start writing applications.
- An SDK built on Java. If you're familiar with the Java programming language, you're already halfway there.
- By distributing your application on Android Market, it's available to hundreds of thousands of users instantly. You're not just limited to one store, because there are alternatives, too. For instance, you can release your application on your own blog. Amazon have recently been rumoured to be preparing their own Android app store also.
- As well as the technical SDK documentation, new resources are being published for Android developers as the platform gains popularity among both users and developers.

Enough with the talk — let's get started developing for Android!

Installing Eclipse and the Android SDK

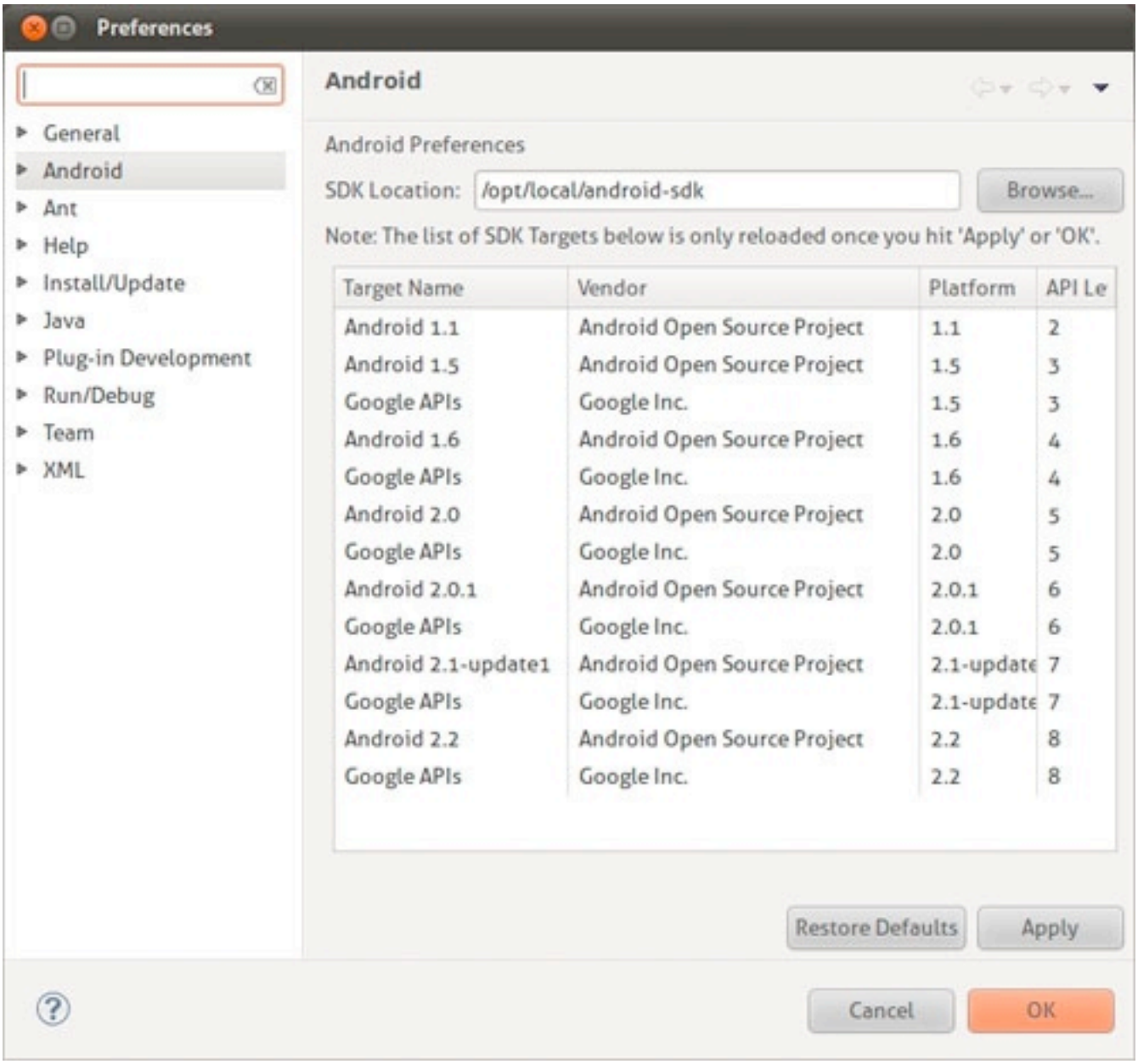
The recommended environment for **developing Android applications** is Eclipse with the Android Development Toolkit (ADT) plugin installed. I'll summarize the process here. If you need more detail, Google's own developer pages do a good job of explaining the installation and configuration process.

- Download the Android SDK for your platform (Windows, Mac OS X, or Linux).
- Extract the downloaded file to somewhere memorable on your hard drive (on Linux, I use **`/opt/local/`**).
- If you don't already have Eclipse installed, download and install the Eclipse IDE for Java Developers package. For programming, Google recommends using Eclipse 3.5 (Galileo).
- Run Eclipse and choose *Help->Install New Software*.
- Click *Add* in the Available Software window.
- Enter Android Development Tools in the *Name* field, and https://dl-ssl.google.com/android/eclipse/ in the *Location* field.
- Click *OK* and check *Developer Tools* in the list of available software. This will install the Android Development Tools and DDMS, Android's debugging tool.


Install
Available Software
Check the items that you wish to install.
Work with:
Android ADT - https://dl-ssl.google.com/android/eclipse/
Add...
Find more software by working with the "Available Software Sites" preferences.
type filter text
Name
Version
Developer Tools
Android DDMS
0.9.8.v201008281755-53891
Android Development Tools
0.9.8.v201008281755-53891
Select All
Deselect All
2 items selected
Details
Features that add Android support to Eclipse for application developers.
More...
Show only the latest versions of available software
Hide items that are already installed
Group items by category
What is already installed?
Contact all update sites during install to find required software
< Back
Next >
Cancel
Finish

Large image

- Click *Next* and *Finish* to install the plugin. You'll need to restart Eclipse once everything is installed.
- When Eclipse restarts, choose *Window->Preferences* and you should see *Android* listed in the categories.
- You now need to tell Eclipse where you've installed the Android SDK. Click *Android* and then *Browse* to select the location where you extracted the SDK files. For example, **`/opt/local/android-sdk`**.



Large view

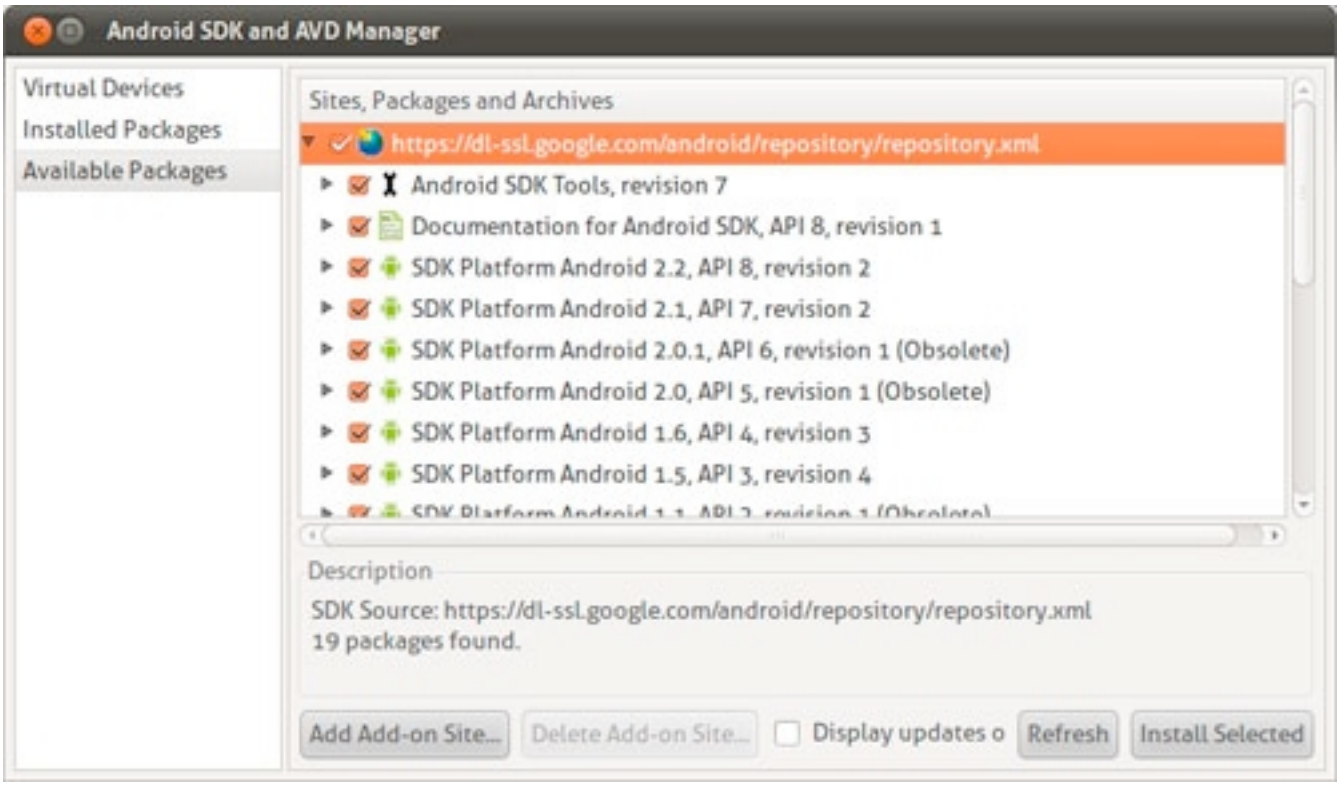
- Click *OK* to have Eclipse save the location of your SDK.

Targeting Android Platforms

Before you can start writing applications for Android, you need to download the SDK platforms for the Android devices for which you want to develop apps. Each platform has a different version of the Android SDK that may be installed on users' devices. For versions of Android 1.5 and above, there are two platforms available: *Android Open Source Project* and *Google*.

The *Android Open Source Project* platforms are open source, but do not include Google's proprietary extensions such as *Google Maps*. If you choose not to use the Google APIs, Google's mapping functionality won't be available to your application. Unless you have a specific reason not to, I'd recommended you to target one of the Google platforms, as this will allow you to take advantage of Google's proprietary extensions.

- Choose Window->Android SDK and AVD Manager.
- Click *Available Packages* in the left column and check the repository to show a list of the available Android platforms.
- You can choose which platforms to download from the list, or leave everything checked to download all the available platforms. When you're done, click *Install Selected* and follow the installation instructions.



Large image

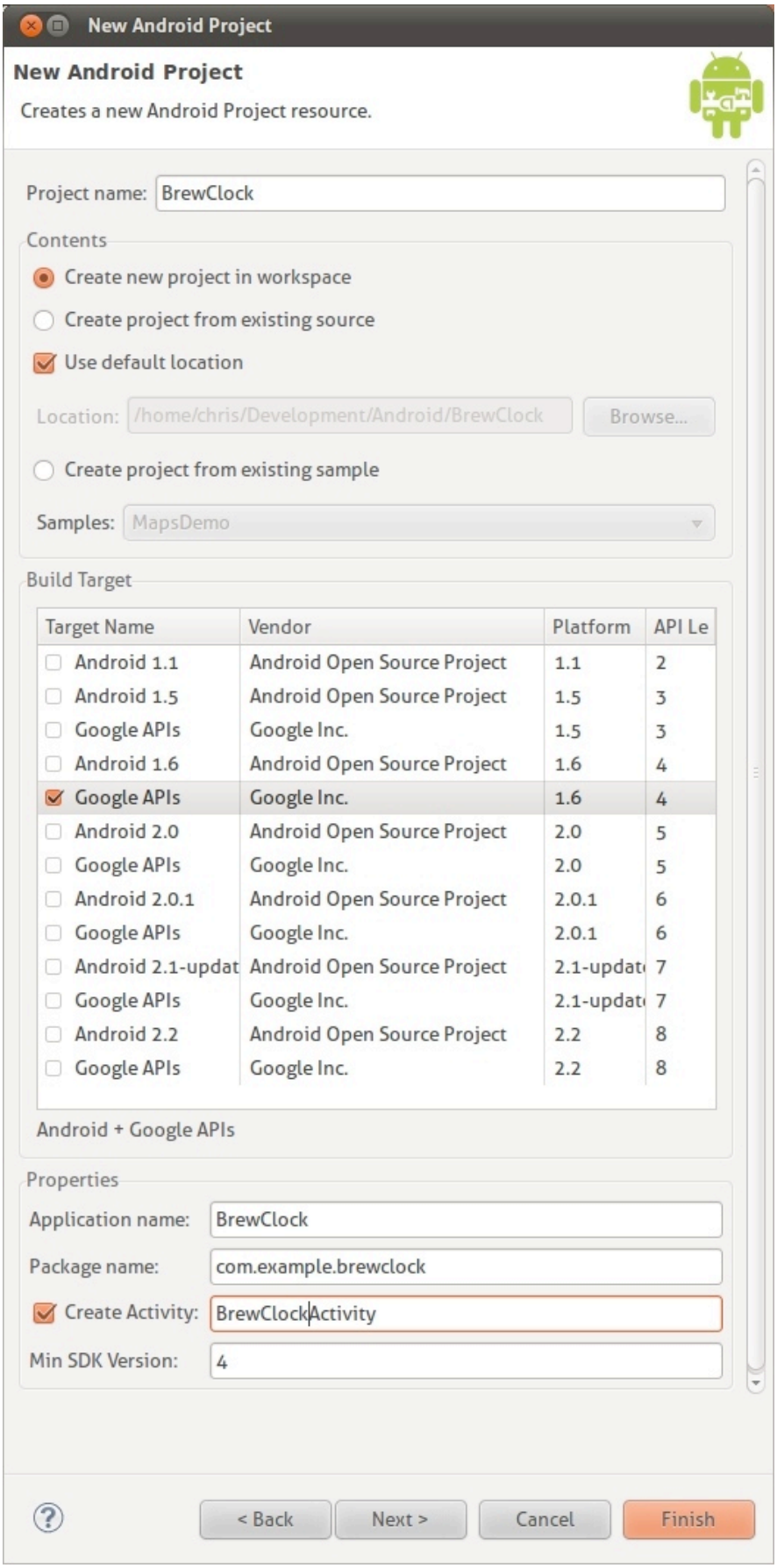
Once everything has been successfully downloaded, you're ready to start developing for Android.

Creating a New Android Project

Eclipse's New Project Wizard can create a new Android application for you, generating files and code that are ready to run right out of the box. It's a quick way to see something working, and a good starting point from which to develop your own applications:

- Choose File->New->Project...
- Choose Android Project
- In the *New Project* dialog, enter the following settings:

```
Project Name: BrewClock
Build Target: Google Inc. 1.6 (Api Level 4)
Application Name: BrewClock
Package Name: com.example.brewclock
Create Activity: BrewClockActivity
Min SDK Version: 4
```


New Android Project
New Android Project
Creates a new Android Project resource.
Project name: BrewClock
Contents
Create new project in workspace
Create project from existing source
Use default location
Location: /home/chris/Development/Android/BrewClock
Browse...
Create project from existing sample
Samples: MapsDemo
Build Target
Target Name | Vendor | Platform | API Le
Android 1.1 | Android Open Source Project | 1.1 | 2
Android 1.5 | Android Open Source Project | 1.5 | 3
Google APIs | Google Inc. | 1.5 | 3
Android 1.6 | Android Open Source Project | 1.6 | 4
Google APIs | Google Inc. | 1.6 | 4
Android 2.0 | Android Open Source Project | 2.0 | 5
Google APIs | Google Inc. | 2.0 | 5
Android 2.0.1 | Android Open Source Project | 2.0.1 | 6
Google APIs | Google Inc. | 2.0.1 | 6
Android 2.1-updat | Android Open Source Project | 2.1-updat | 7
Google APIs | Google Inc. | 2.1-updat | 7
Android 2.2 | Android Open Source Project | 2.2 | 8
Google APIs | Google Inc. | 2.2 | 8
Android + Google APIs
Properties
Application name: BrewClock
Package name: com.example.brewclock
Create Activity: BrewClockActivity
Min SDK Version: 4
< Back
Next >
Cancel
Finish

After clicking *Finish*, Eclipse will create a new Android project that's ready to run. Notice you told Eclipse to generate an Activity called **`BrewClockActivity?`** This is the code that Android actually uses to run your application. The generated code will display a simple 'Hello World' style message when the application runs.

PACKAGES

The package name is an identifier for your application. When the time comes and you are willing to publish on Android Market, it's exactly this identifier that will be used to track your application for updates, so it's important to **make sure it's unique**. Although we're using the **`com.example.brewclock`** namespace here, for a real application it's best to choose something like **`com.yourcompanyname.yourapplication`**.

SDK VERSIONS

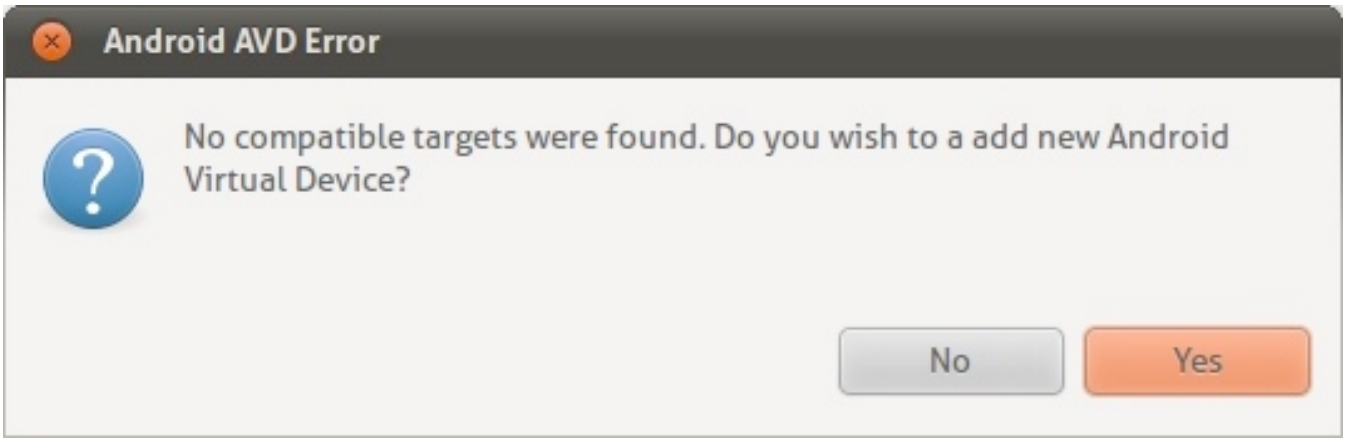
The **`Min SDK Version`** is the earliest version of Android on which your application will run. With each new release of Android, the SDK adds and changes methods. By choosing an SDK version, Android (and the Android Market) knows that your application will only run on devices with a version of Android later or equal than the specified version.

Running Your Application

Now let's try running the application in Eclipse. As this is the first run, Eclipse will ask what type of project you are working on:

- Choose *Run->Run* or press *Ctrl+F11*.
- Choose Android Application and click OK.

Eclipse will now try to run the application on an Android device. At the moment, though, you don't have any Android devices running, so the run will fail and you'll be asked to create a new ***Android Virtual Device*** (AVD).



ANDROID VIRTUAL DEVICES

An Android Virtual Device (AVD) is an emulator that simulates a real-world Android device, such as a mobile phone or Tablet PC. You can use AVDs to test how your application performs on a wide variety of Android devices, without having to buy every gadget on the market.

You can create as many AVDs as you like, each set up with different versions of the Android Platform. For each AVD you create, you can configure various hardware properties such as whether it has a physical keyboard, GPS support, the camera resolution, and so on.

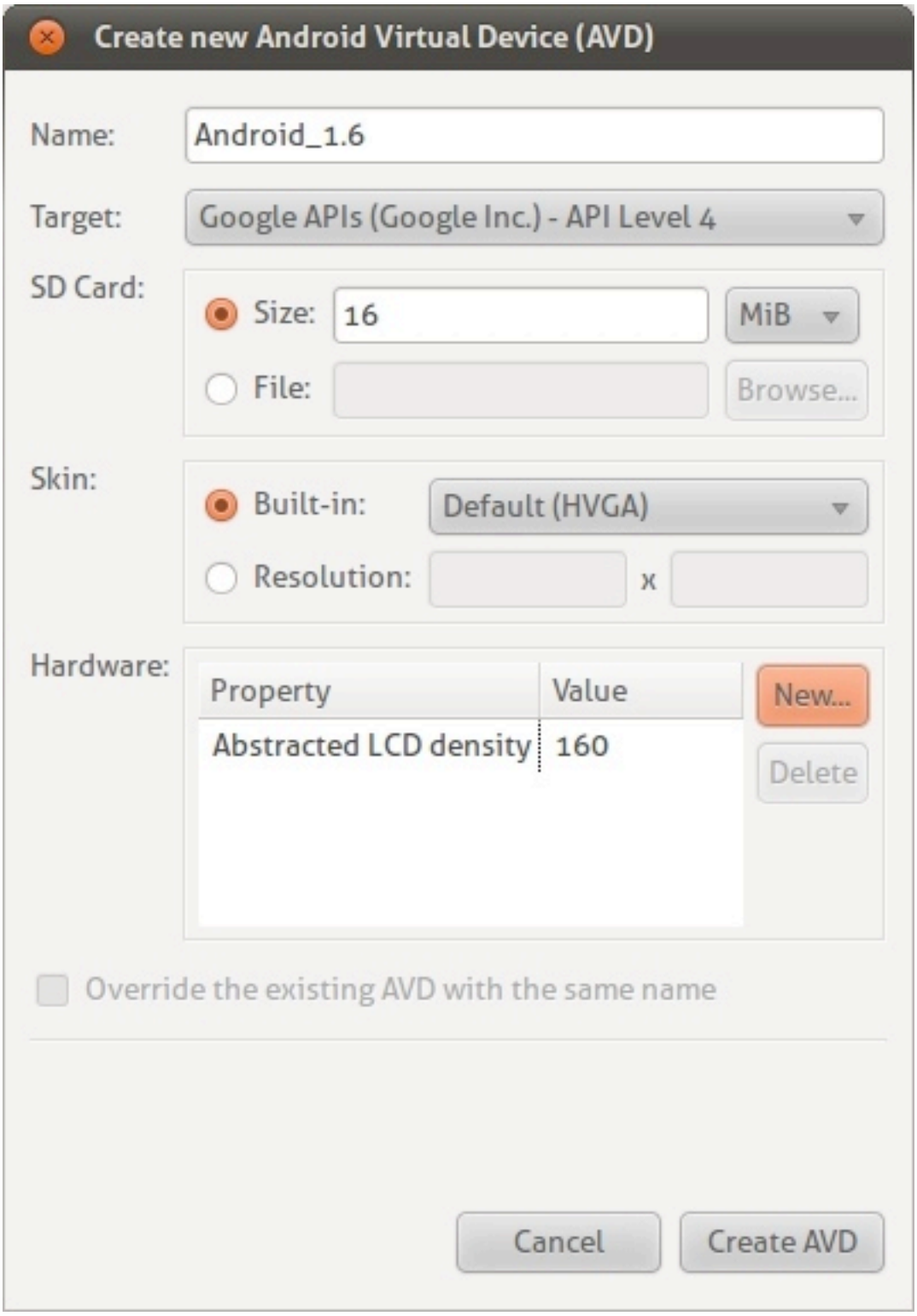
Before you can run your application, you need to create your first AVD running the target SDK platform (Google APIs 1.6).

Let's do that now:

- If you haven't tried to run your application yet, click *Run* now (or hit *Ctrl +F11*)
- When the target device warning pops up, click *Yes* to create a new AVD.
- Click New in the Android SDK and AVD Manager dialog that appears.
- Enter the following settings for the AVD:

```
Name: Android_1.6
Target: Google APIs (Google Inc.) - API Level 4
SD Card Size: 16 MiB
Skin Built In: Default (HVGA)
```

- Click *Create AVD* to have Android build your new AVD.
- Close the Android SDK and AVD Manager dialog.


Create new Android Virtual Device (AVD)
Name:
Android_1.6
Target:
Google APIs (Google Inc.) - API Level 4
SD Card:
Size:
16
MiB
File:
Browse...
Skin:
Built-in:
Default (HVGA)
Resolution:
x
Hardware:
Property
Value
Abstracted LCD density
160
New...
Delete
Override the existing AVD with the same name
Cancel
Create AVD

RUNNING THE CODE

Try running your application again (*Ctrl+F11*). Eclipse will now build your project and launch the new AVD. Remember, the AVD emulates a complete Android system, so you'll even need to sit through the slow boot process just like a real device. For this reason, once the AVD is up and running, it's best not to close it down until you've finished developing for the day.

When the emulator has booted, Eclipse automatically installs and runs your application:

Large image

Building Your First Android Application

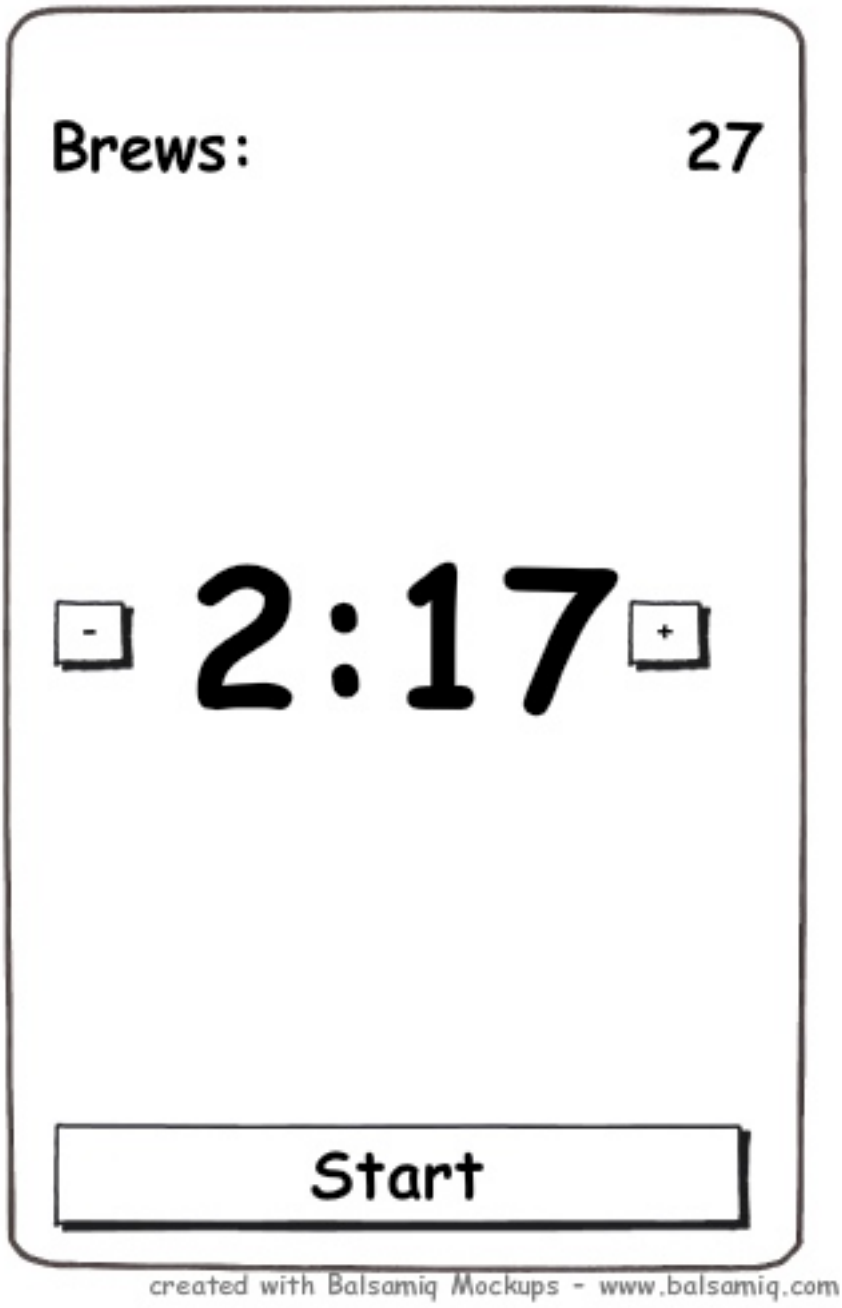
Testing generated code is all well and good, but you want to start building a real application. For this, we'll step through a simple design process and build an application that you can deploy to your Android device.

Most developers (myself included) like a **constant supply of good tea or coffee**. In the next section of this article you'll build a simple tea counter application to track how many cups of tea (*brews*) the user has drunk, and let them set a timer for brewing each cup.

You can download the complete code for this tutorial on GitHub.

DESIGNING THE USER INTERFACE

One of the first steps to building any Android application is to design and build the user interface. Here's a quick sketch of how the application's interface will look:


Brews:
27
-
2:17
+
Start
created with Balsamiq Mockups - www.balsamiq.com

Large image

The user will be able to set a brew time in minutes using the + and - buttons. When they click *Start*, a countdown will start for the specified number of minutes. Unless the user cancels the brew by tapping the button again, the brew count will be increased when the countdown timer reaches 0.

BUILDING THE INTERFACE

Android user interfaces, or *layouts*, which are described in XML documents, can be found in the **`res/layouts`** folder. The template code that Eclipse generated already has a simple layout declared in **`res/layouts/main.xml`** which you may have seen previously while the application was running on the emulator.

Eclipse has a graphical layout designer that lets you build the interface by 'dragging' and 'dropping' controls around the screen. However, I often find it easier to write the interface in XML and use the graphical layout to preview the results.

Let's do this now by changing **`main.xml`** to match the design sketch above:

- Open **`res/layouts/main.xml`** in Eclipse by double-clicking it in the *Package Explorer*.
- Click the **`main.xml`** tab along the bottom of the screen to switch to XML view.

Now change the content of **`main.xml`** to:

```
# /res/layouts/main.xml
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
  xmlns:android="http://schemas.android.com/apk/res/android"
  android:orientation="vertical"
  android:layout_width="fill_parent"
```

```
android:layout_height="fill_parent">
<LinearLayout
  android:orientation="horizontal"
  android:layout_width="fill_parent"
  android:layout_height="wrap_content"
  android:padding="10dip">
  <TextView
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:textSize="20dip"
    android:text="Brews: " />
  <TextView
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:text="None"
    android:gravity="right"
    android:textSize="20dip"
    android:id="@+id/brew_count_label" />
</LinearLayout>
<LinearLayout
  android:orientation="horizontal"
  android:layout_width="fill_parent"
  android:layout_height="wrap_content"
  android:layout_weight="1"
  android:gravity="center"
  android:padding="10dip">
  <Button
    android:id="@+id/brew_time_down"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="-"
    android:textSize="40dip" />
  <TextView
    android:id="@+id/brew_time"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="0:00"
```

```
        android:textSize="40dip"
        android:padding="10dip" />
      <Button
        android:id="@+id/brew_time_up"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="+"
        android:textSize="40dip" />
    </LinearLayout>
    <Button
      android:id="@+id/brew_start"
      android:layout_width="fill_parent"
      android:layout_height="wrap_content"
      android:layout_gravity="bottom"
      android:text="Start" />
  </LinearLayout>
```

As you can see, Android's XML layout files are verbose, but allow you to control virtually every aspect of elements on the screen.

One of the most important interface elements in Android are **`Layout`** containers, such as the **`LinearLayout`** used in this example. These elements are invisible to the user but act as layout containers for other elements such as **`Buttons and TextViews`**.

There are several types of layout views, each of which is used to build different types of layout. As well as the **`LinearLayout`** and **`AbsoluteLayout`**, the **`TableLayout`** allows the use of complex grid-based interfaces. You can find out more about Layouts in the Common Layout Objects section of the API documents.

LINKING YOUR LAYOUT WITH CODE

After saving your layout, try running your application in the emulator again by pressing *Ctrl+F11*, or clicking the *Run* icon in Eclipse. Now instead of the 'Hello World' message you saw earlier, you'll see Android now displays your application's new interface.

If you click any of the buttons, they'll highlight as expected, but don't do anything yet. Let's remedy that by writing some code behind the interface layout:

```
# /src/com/example/brewclock/BrewClockActivity.java
...
import android.widget.Button;
import android.widget.TextView;

public class BrewClockActivity extends Activity {
  /** Properties **/
  protected Button brewAddTime;
  protected Button brewDecreaseTime;
  protected Button startBrew;
  protected TextView brewCountLabel;
  protected TextView brewTimeLabel;

  ...
 }
```

Next, we'll change the call to **`onCreate`**. This is the method that gets called whenever Android starts your application. In the code that Eclipse generated, **`onCreate`** sets the activity's view to be **`R.layout.main`**. It's that line of code that tells Android to decode our layout XML document and display it to the user.

THE RESOURCE OBJECT

In Android, `R` is a special object that is automatically generated to allow access to your project's resources (layouts, strings, menus, icons…) from within the code. Each resource is given an **`id`**. In the layout file above, these are the **`@+id`** XML attributes. We'll use those attributes to connect the **`Buttons`** and **`TextViews`** in our layout to the code:

```
# /src/com/example/brewclock/BrewClockActivity.java
...
public class BrewClockActivity extends Activity {
  ...
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main);

    // Connect interface elements to properties
    brewAddTime = (Button) findViewById(R.id.brew_time_up);
    brewDecreaseTime = (Button)
findViewById(R.id.brew_time_down);
    startBrew = (Button) findViewById(R.id.brew_start);
    brewCountLabel = (TextView)
findViewById(R.id.brew_count_label);
    brewTimeLabel = (TextView) findViewById(R.id.brew_time);
  }
}
```

LISTENING FOR EVENTS

In order to detect when the user taps one of our buttons, we need to implement a listener. You may be familiar with listeners or *callbacks* from other event-driven platforms, such as Javascript/jQuery events or Rails' callbacks.

Android provides a similar mechanism by providing **`Listener`** interfaces, such as **`OnClickListener`**, that define methods to be triggered when an event occurs. Implementing the **`OnClickListener`** interface will notify your application when the user taps the screen, and on which button they tapped. You also need to tell each button about the **`ClickListener`** so that it knows which listener to notify:

```
# /src/com/example/brewclock/BrewClockActivity.java
...
// Be sure not to import
// `android.content.dialoginterface.OnClickListener`.
import android.view.View.OnClickListener;

public class BrewClockActivity extends Activity
  implements OnClickListener {
  ...
  public void onCreate(Bundle savedInstanceState) {
    ...
    // Setup ClickListeners
    brewAddTime.setOnClickListener(this);
    brewDecreaseTime.setOnClickListener(this);
    startBrew.setOnClickListener(this);
  }
  ...
  public void onClick(View v) {
    // TODO: Add code to handle button taps
  }
}
```

Next we'll add code that handles each of our button presses. We'll also add four new properties to the Activity that will let the user set and track the brewing time, how many brews have been made, and whether the timer is currently running.

```
# /src/com/example/brewclock/BrewClockActivity.java
...
public class BrewClockActivity extends Activity
  implements OnClickListener {
  ...
  protected int brewTime = 3;
  protected CountDownTimer brewCountDownTimer;
  protected int brewCount = 0;
  protected boolean isBrewing = false;
  ...
  public void onClick(View v) {
    if(v == brewAddTime)
      setBrewTime(brewTime + 1);
    else if(v == brewDecreaseTime)
      setBrewTime(brewTime -1);
    else if(v == startBrew) {
      if(isBrewing)
        stopBrew();
      else
        startBrew();
    }
  }
}
```

Notice we're using the **`CountDownTimer`** class provided by Android. This lets you easily create and start a simple countdown, and be notified at regular intervals whilst the countdown is running. You'll use this in the **`startBrew`** method below.

The following methods are all model logic that handles setting the brew time, starting and stopping the brew and maintaining a count of brews made. We'll also initialize the **`brewTime`** and **`brewCount`** properties in **`onCreate`**.

It would be good practice to move this code to a separate model class, but for simplicity we'll add the code to our **`BrewClockActivity`**:

```
# /src/com/example/brewclock/BrewClockActivity.java
...
public class BrewClockActivity extends Activity
  implements OnClickListener {
  ...
  public void onCreate(Bundle savedInstanceState) {
    ...
    // Set the initial brew values
    setBrewCount(0);
    setBrewTime(3);
  }

  /**
   * Set an absolute value for the number of minutes to brew.
   * Has no effect if a brew is currently running.
   * @param minutes The number of minutes to brew.
   */
  public void setBrewTime(int minutes) {
    if(isBrewing)
      return;

    brewTime = minutes;
```

```
    if(brewTime < 1)
      brewTime = 1;

    brewTimeLabel.setText(String.valueOf(brewTime) + "m");
  }

  /**
   * Set the number of brews that have been made, and update
   * the interface.
   * @param count The new number of brews
   */
  public void setBrewCount(int count) {
    brewCount = count;
    brewCountLabel.setText(String.valueOf(brewCount));
  }

  /**
   * Start the brew timer
   */
  public void startBrew() {
    // Create a new CountDownTimer to track the brew time
    brewCountDownTimer = new CountDownTimer(brewTime * 60 *
1000, 1000) {
      @Override
      public void onTick(long millisUntilFinished) {

brewTimeLabel.setText(String.valueOf(millisUntilFinished /
1000) + "s");
      }

      @Override
      public void onFinish() {
        isBrewing = false;
        setBrewCount(brewCount + 1);

        brewTimeLabel.setText("Brew Up!");
```

```
            startBrew.setText("Start");
          }
        };

        brewCountDownTimer.start();
        startBrew.setText("Stop");
        isBrewing = true;
      }

      /**
       * Stop the brew timer
       */
      public void stopBrew() {
        if(brewCountDownTimer != null)
          brewCountDownTimer.cancel();

        isBrewing = false;
        startBrew.setText("Start");
      }
      ...
  }
```

The only parts of this code specific to Android are setting the display labels using the **`setText`** method. In **`startBrew`**, we create and start a **`CountDownTimer`** to start counting down every second until a brew is finished. Notice that we define **`CountDownTimer's`** listeners (**`onTick`** and **`onFinish`**) inline. **`onTick`** will be called every 1000 milliseconds (1 second) the timer counts down, whilst **`onFinish`** is called when the timer reaches zero.

AVOIDING HARD-CODED TEXT IN YOUR CODE

To keep this tutorial code simple, I've intentionally written label strings directly in the code (e.g. "**Brew Up!**", "**Start**", "**Stop**"). Generally, this isn't good practice, as it makes finding and changing those strings harder in large projects.

Android provides a neat way to keep your text strings separate from code with the **R** object. **R** lets you define all your application's strings in an xml file (**res/values/strings.xml**) which you can then access in code by reference. For example:

```
# /res/values/strings.xml
<string name="brew_up_label">Brew Up!</string>
...

# /res/com/example/brewclock/BrewClockActivity.java
...
brewLabel.setText(R.string.brew_up_label);
...
```

Now if you wanted to change **Brew Up!** to something else, you would only need to change it once in the *strings.xml* file. Your application starts to span dozens of code files which keeps all your strings in one place and makes a lot of sense!

TRYING BREWCLOCK

With the code complete, it's time to try out the application. Hit *Run* or *Ctrl +F11* to start BrewClock in the emulator. All being well, you'll see the interface set up and ready to time your tea brewing! Try setting different brew times, and pressing *Start* to watch the countdown.

Large image

Summary

In this short introduction to Android, you've installed the Android SDK and Eclipse Android Development Tools (ADT) plugin. You've set up an emulator, or virtual device that can test your applications. You've also built a working Android application which has highlighted a number of key concepts that you'll use when developing your own Android applications.

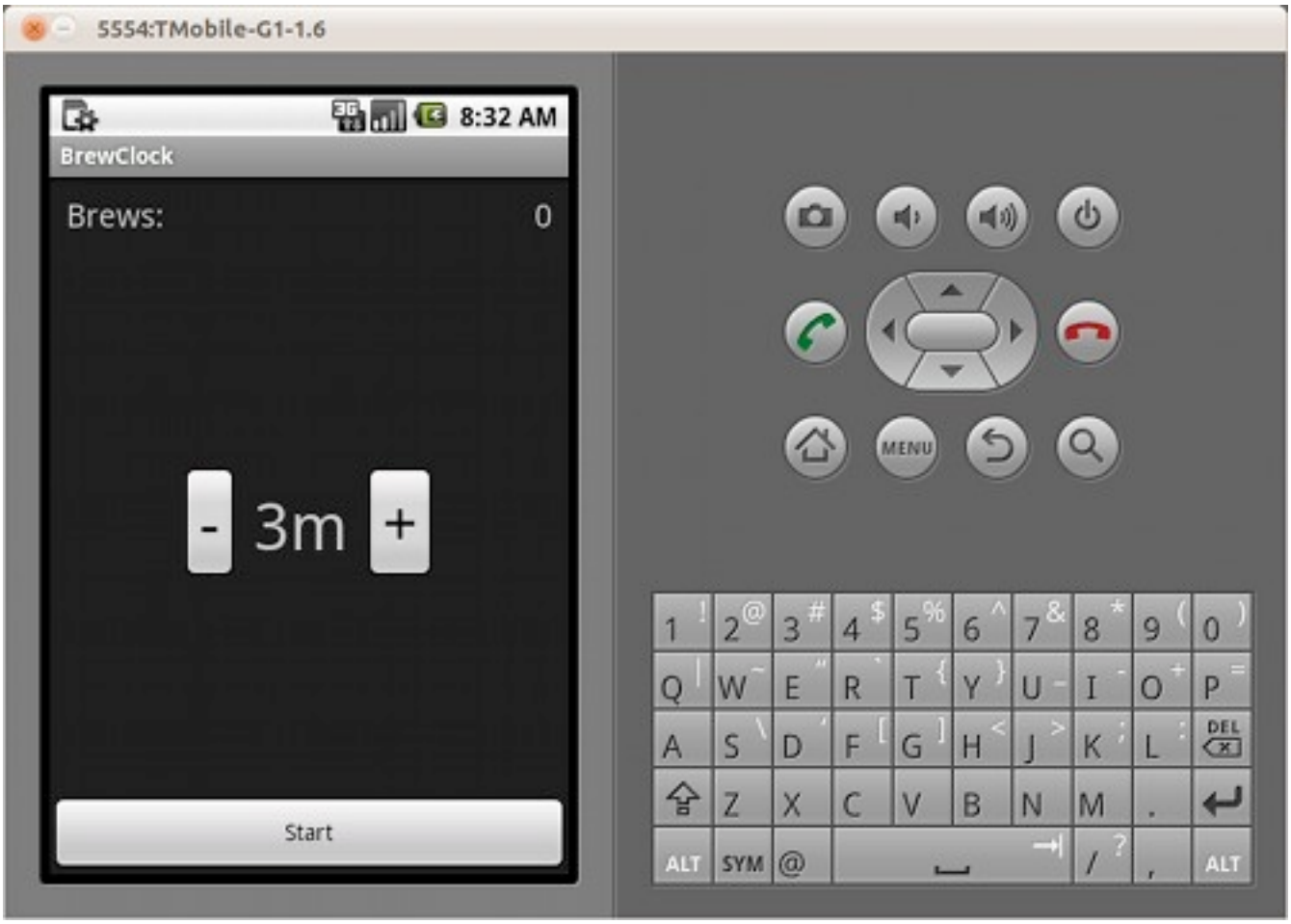
Hopefully, this has whet your appetite for building mobile applications, and experimenting in this exciting field. Android offers a great way to start writing applications for a range of current and upcoming mobile devices.

Get Started Developing For Android With Eclipse: Reloaded

Chris Blunt

In the first part of this tutorial series, we built a simple brew timer application using Android and Eclipse. In this second part, we'll continue developing the application by adding extra functionality. In doing this, you'll be introduced to some important and powerful features of the Android SDK, including Persistent data storage, Activities and Intent as well as Shared user preferences.

To follow this tutorial, you'll need the code from the previous article. If you want to get started right away, grab the code from GitHub and check out the *tutorial_part_1* tag using this:




```
$ git clone git://github.com/cblunt/BrewClock.git
$ cd BrewClock
$ git checkout tutorial_part_1
```

Once you've checked out the code on GitHub, you'll need to import the project into Eclipse:

1. Launch Eclipse and choose *File* → *Import...*
2. In the Import window, select "Existing Projects into Workspace" and click "Next."
3. On the next screen, click "Browse," and select the project folder that you cloned from GitHub.

4. Click “Finish” to import your project into Eclipse.

After importing the project into Eclipse, you might receive a warning message:

```
Android required .class compatibility set to 5.0.
Please fix project properties.
```

If this is the case, right-click on the newly imported “BrewClock” project in the “Project Explorer,” choose “Fix Project Properties,” and then restart Eclipse.

Getting Started With Data Storage

Currently, BrewClock lets users set a specific time for brewing their favorite cups of tea. This is great, but what if they regularly drink a variety of teas, each with their own different brewing times? At the moment, users have to remember brewing times for all their favorite teas! This doesn’t make for a great user experience. So, in this tutorial we’ll develop functionality to let users store brewing times for their favorite teas and then choose from that list of teas when they make a brew.

To do this, we’ll take advantage of Android’s rich data-storage API. Android offers several ways to store data, two of which we’ll cover in this article. The first, more powerful option, uses the SQLite database engine to store data for our application.

SQLite is a popular and lightweight SQL database engine that saves data in a single file. It is often used in desktop and embedded applications, where running a client-server SQL engine (such as MySQL or PostgreSQL) isn’t feasible.

Every application installed on an Android device can save and use any number of SQLite database files (subject to storage capacity), which the system will manage automatically. An application's databases are private and so cannot be accessed by any other applications. (Data can be shared through the **`ContentProvider`** class, but we won't cover content providers in this tutorial.) Database files persist when the application is upgraded and are deleted when the application is uninstalled.

We'll use a simple SQLite database in BrewClock to maintain a list of teas and their appropriate brewing times. Here's an overview of how our database schema will look:

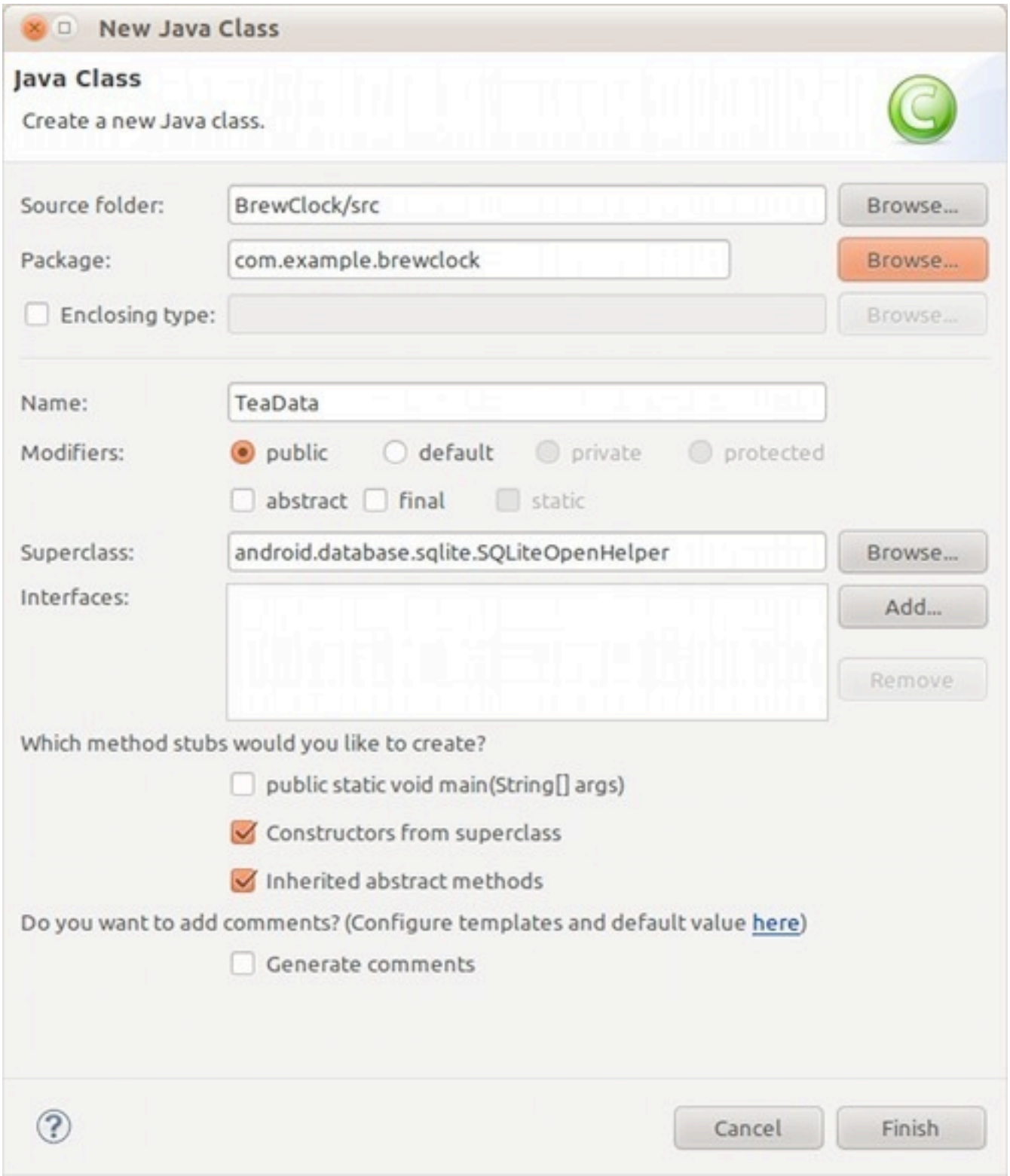
```
+-------------------------------------+
| Table: teas                         |
+-------------+-----------------------+
| Column      | Description           |
+-------------+-----------------------+
| _ID         | integer, autoincrement |
| name        | text, not null        |
| brew_time   | integer, not null     |
+-------------+-----------------------+
```

If you've worked with SQL before, this should look fairly familiar. The database table has three columns: a unique identifier (**`_ID`**), name and brewing time. We'll use the APIs provided by Android to create the database table in our code. The system will take care of creating the database file in the right location for our application.

ABSTRACTING THE DATABASE

To ensure the database code is easy to maintain, we'll abstract all the code for handling database creation, inserts and queries into a separate class, **`TeaData`**. This should be fairly familiar if you're used to the model-view-controller approach. All the database code is kept in a separate class from our **`BrewClockActivity`**. The Activity can then just instantiate a new **`TeaData`** instance (which will connect to the database) and do what it needs to do. Working in this way enables us to easily change the database in one place without having to change anything in any other parts of our application that deal with the database.

Create a new class called **`TeaData`** in the BrewClock project by going to File ➜ New ➜ Class. Ensure that **`TeaData`** extends the **`android.database.sqlite.SQLiteOpenHelper`** class and that you check the box for "Constructors from superclass."


New Java Class
Java Class
Create a new Java class.
Source folder:
BrewClock/src
Browse...
Package:
com.example.brewclock
Browse...
Enclosing type:
Browse...
Name:
TeaData
Modifiers:
public
default
private
protected
abstract
final
static
Superclass:
android.database.sqlite.SQLiteOpenHelper
Browse...
Interfaces:
Add...
Remove
Which method stubs would you like to create?
public static void main(String[] args)
Constructors from superclass
Inherited abstract methods
Do you want to add comments? (Configure templates and default value here)
Generate comments
Cancel
Finish

The `TeaData` class will automatically handle the creation and versioning of a SQLite database for your application. We'll also add methods to give other parts of our code an interface to the database.

Add two constants to `TeaData` to store the name and version of the database, the table's name and the names of columns in that table. We'll use the Android-provided constant `BaseColumns._ID` for the table's unique id column:

```
// src/com/example/brewclock/TeaData.java
import android.app.Activity;
import android.content.ContentValues;
import android.content.Context;
import android.database.Cursor;
import android.database.DatabaseUtils;
import android.provider.BaseColumns;

public class TeaData extends SQLiteOpenHelper {
  private static final String DATABASE_NAME = "teas.db";
  private static final int DATABASE_VERSION = 1;

  public static final String TABLE_NAME = "teas";

  public static final String _ID = BaseColumns._ID;
  public static final String NAME = "name";
  public static final String BREW_TIME = "brew_time";

  // …
}
```

Add a constructor to **TeaData** that calls its parent method, supplying our database name and version. Android will automatically handle opening the database (and creating it if it does not exist).

```
// src/com/example/brewclock/TeaData.java
public TeaData(Context context) {
  super(context, DATABASE_NAME, null, DATABASE_VERSION);
}
```

We'll need to override the **onCreate** method to execute a string of SQL commands that create the database table for our tea. Android will handle this method for us, calling **onCreate** when the database file is first created.

On subsequent launches, Android checks the version of the database against the **DATABASE_VERSION** number we supplied to the constructor. If the version has changed, Android will call the **onUpgrade** method, which is where you would write any code to modify the database structure. In this tutorial, we'll just ask Android to drop and recreate the database.

So, add the following code to **onCreate**:

```
// src/com/example/brewclock/TeaData.java
@Override
public void onCreate(SQLiteDatabase db) {
  // CREATE TABLE teas (id INTEGER PRIMARY KEY AUTOINCREMENT,
name TEXT NOT NULL, brew_time INTEGER);
  String sql =
    "CREATE TABLE " + TABLE_NAME + " ("
      + _ID + " INTEGER PRIMARY KEY AUTOINCREMENT, "
      + NAME + " TEXT NOT NULL, "
      + BREW_TIME + " INTEGER"
      + ");";

  db.execSQL(sql);
}
```

```
@Override
public void onUpgrade(SQLiteDatabase db, int oldVersion, int
newVersion) {
  db.execSQL("DROP TABLE IF EXISTS " + TABLE_NAME);
  onCreate(db);
}
```

Next, we'll add a new method to **`TeaData`** that lets us easily add new tea records to the database. We'll supply the method with a name and brewing time for the tea to be added. Rather than forcing us to write out the raw SQL to do this, Android supplies a set of classes for inserting records into the database. First, we create a set of **`ContentValues`**, pushing the relevant values into that set.

With an instance of **`ContentValues`**, we simply supply the column name and the value to insert. Android takes care of creating and running the appropriate SQL. Using Android's database classes ensures that the writes are safe, and if the data storage mechanism changes in a future Android release, our code will still work.

Add a new method, **`insert()`**, to the **`TeaData`** class:

```
// src/com/example/brewclock/TeaData.java
public void insert(String name, int brewTime) {
  SQLiteDatabase db = getWritableDatabase();

  ContentValues values = new ContentValues();
  values.put(NAME, name);
  values.put(BREW_TIME, brewTime);

  db.insertOrThrow(TABLE_NAME, null, values);
}
```

RETRIEVING DATA

With the ability to save data into the database, we'll also need a way to get it back out. Android provides the `Cursor` interface for doing just this. A `Cursor` represents the results of running a SQL query against the database, and it maintains a pointer to one row within that result set. This pointer can be moved forwards and backwards through the results, returning the values from each column. It can help to visualize this:

```
SQL Query: SELECT * from teas LIMIT 3;

+-----------------------------------+
|  _ID  |  name        |  brew_time  |
+-----------------------------------+
|    1  |  Earl Grey   |          3  |
|    2  |  Green       |          1  | <= Cursor
|    3  |  Assam       |          5  |
+-------+--------------+-------------+
```

In this example, the Cursor is pointing at the second row in the result set (Green tea). We could move the Cursor back a row to represent the first row (Earl Grey) by calling **`cursor.moveToPrevious()`**, or move forward to the Assam row with **`moveToNext()`**. To fetch the name of the tea that the Cursor is pointing out, we would call **`cursor.getString(1)`**, where **`1`** is the column index of the column we wish to retrieve (note that the index is zero-based, so column 0 is the first column, 1 the second column and so on).

Now that you know about Cursors, add a method that creates a `Cursor` object that returns all the teas in our database. Add an **`all`** method to **`TeaData`**:

```
// src/com/example/brewclock/TeaData.java
public Cursor all(Activity activity) {
  String[] from = { _ID, NAME, BREW_TIME };
  String order = NAME;
```

```
    SQLiteDatabase db = getReadableDatabase();
    Cursor cursor = db.query(TABLE_NAME, from, null, null, null,
  null, order);
    activity.startManagingCursor(cursor);

    return cursor;
  }
```

Let's go over this method in detail, because it looks a little strange at first. Again, rather than writing raw SQL to query the database, we make use of Android's database interface methods.

First, we need to tell Android which columns from our database we're interested in. To do this, we create an array of strings—each one of the column identifiers that we defined at the top of **`TeaData`**. We'll also set the column that we want to order the results by and store it in the **`order`** string.

Next, we create a read-only connection to the database using **`getReadableDatabase()`**, and with that connection, we tell Android to run a query using the **`query()`** method. The **`query()`** method takes a set of parameters that Android internally converts into a SQL query. Again, Android's abstraction layer ensures that our application code will likely continue to work, even if the underlying data storage changes in a future version of Android.

Because we just want to return every tea in the database, we don't apply any joins, filters or groups (i.e. **`WHERE`**, **`JOIN`**, and **`GROUP BY`** clauses in SQL) to the method. The **`from`** and **`order`** variables tell the query what columns to return on the database and the order in which they are retrieved. We use the **`SQLiteDatabase.query()`** method as an interface to the database.

Last, we ask the supplied Activity (in this case, our **`BrewClockActivity`**) to manage the Cursor. Usually, a Cursor must be manually refreshed to reload any new data, so if we added a new tea to our database, we would have to remember to refresh our Cursor. Instead, Android can take care of this for us, recreating the results whenever the Activity is suspended and resumed, by calling **`startManagingCursor()`**.

Finally, we'll add another utility method to return the number of records in the table. Once again, Android provides a handy utility to do this for us in the **`DatabaseUtils`** class:

Add the following method, **`count`**, to your **`TeaData`** class:

```
// src/com/example/brewclock/TeaData.java
  public long count() {
    SQLiteDatabase db = getReadableDatabase();
    return DatabaseUtils.queryNumEntries(db, TABLE_NAME);
  }
```

Save the **`TeaData`** class, and fix any missing imports using Eclipse (Source → Organize Imports). With our data class finished, it's time to change BrewClock's interface to make use of the database!

MODIFY BREWCLOCK'S INTERFACE TO ALLOW TEA SELECTION

The purpose of storing preset teas and brew times is to let the user quickly select their favorite tea from the presets. To facilitate this, we'll add a **`Spinner`** (analogous to a pop-up menu in desktop interfaces) to the main BrewClock interface, populated with the list of teas from **`TeaData`**.

As in the previous tutorial, use Eclipse's layout editor to add the Spinner to BrewClock's main interface layout XML file. Add the following code just below the **`LinearLayout`** for the brew count label (around line 24). Remember, you can switch to the "Code View" tab along the bottom of the window if Eclipse opens the visual layout editor.

```
<!-- /res/layout/main.xml -->

<!-- Tea Selection -->
<LinearLayout
  android:orientation="vertical"
  android:layout_width="fill_parent"
  android:layout_height="wrap_content">

  <Spinner
    android:id="@+id/tea_spinner"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content" />

</LinearLayout>
```

In the **`BrewClockActivity`** class, add a member variable to reference the Spinner, and connect it to the interface using **`findViewById`**:

```
// src/com/example/brewclock/BrewClockActivity.java
protected Spinner teaSpinner;
protected TeaData teaData;

// …

public void onCreate(Bundle savedInstanceState) {
  // …
  teaData = new TeaData(this);
  teaSpinner = (Spinner) findViewById(R.id.tea_spinner);
}
```

Try running your application to make sure the new interface works correctly. You should see a blank pop-up menu (or Spinner) just below the brew count. If you tap the spinner, Android handles displaying a pop-up menu so that you can choose an option for the spinner. At the moment, the menu is empty, so we'll remedy that by binding the Spinner to our tea database.


5554:TMobile-G1-1.6
9:08 AM
BrewClock
Brews:
0
-
3m
+
Start
MENU

DATA BINDING

When Android retrieves data from a database, it returns a **`Cursor`** object. The Cursor represents a set of results from the database and can be moved through the results to retrieve values. We can easily bind these results to a view (in this case, the Spinner) using a set of classes provided by Android called "Adapters." Adapters do all the hard work of fetching database results from a **`Cursor`** and displaying them in the interface.

Remember that our **`TeaData.all()`** method already returns a Cursor populated with the contents of our teas table. Using that Cursor, all we need to do is create a **`SimpleCursorAdapter`** to bind its data to our **`teaSpinner`**, and Android will take care of populating the spinner's options.

Connect the Cursor returned by **`teaData.all()`** to the Spinner by creating a **`SimpleCursorAdapter`**:

```
// com/example/brewclock/BrewClockActivity.java

public void onCreate(Bundle savedInstanceState) {
  // …
  Cursor cursor = teaData.all(this);

  SimpleCursorAdapter teaCursorAdapter = new
SimpleCursorAdapter(
    this,
    android.R.layout.simple_spinner_item,
    cursor,
    new String[] { TeaData.NAME },
    new int[] { android.R.id.text1 }
  );

  teaSpinner.setAdapter(teaCursorAdapter);
```

```
teaCursorAdapter.setDropDownViewResource(android.R.layout.simp
le_spinner_dropdown_item);
}
```

Notice that we've made use of Android's built-in **`android.R`** object. This provides some generic default resources for your application, such as simple views and layouts. In this case, we've used **`android.R.layout.simple_spinner_item`**, which is a simple text label layout.

If you run the application again, you'll see that the spinner is still empty! Even though we've connected the spinner to our database, there are no records in the database to display.

Let's give the user a choice of teas by adding some default records to the database in BrewClock's constructor. To avoid duplicate entries, we'll add only the default teas if the database is empty. We can make use of TeaData's **`count()`** method to check if this is the case.

Add code to create a default set of teas if the database is empty. Add this line just above the code to fetch the teas from **`teaData`**:

```
// com/example/brewclock/BrewClockActivity.java
public void onCreate(Bundle savedInstanceState) {
  // …

  // Add some default tea data! (Adjust to your preference :)
  if(teaData.count() == 0) {
    teaData.insert("Earl Grey", 3);
    teaData.insert("Assam", 3);
    teaData.insert("Jasmine Green", 1);
    teaData.insert("Darjeeling", 2);
  }
```

```
    // Code from the previous step:
    Cursor cursor = teaData.all(this);

    // …
  }
```

Now run the application again. You'll now see that your tea Spinner has the first tea selected. Tapping on the Spinner lets you select one of the teas from your database!

Congratulations! You've successfully connected your interface to a data source. This is one of the most important aspects of any software application. As you've seen, Android makes this task fairly easy, but it is extremely powerful. Using cursors and adapters, you can take virtually any data source (from a simple array of strings to a complex relational database query) and bind it to any type of view: a spinner, a list view or even an iTunes-like cover-flow gallery!

Although now would be a good time for a brew, our work isn't over yet. While you can choose different teas from the Spinner, making a selection doesn't do anything. We need to find out which tea the user has selected and update the brew time accordingly.

READ SELECTED TEA, AND UPDATE BREW TIME

To determine which tea the user has selected from our database, **`BrewClockActivity`** needs to listen for an event. Similar to the **`OnClickListener`** event that is triggered by button presses, we'll implement the **`OnItemSelectedListener`**. Events in this listener are triggered when the user makes a selection from a view, such as our Spinner.

Enable the **`onItemSelectedListener`** in **`BrewClockActivity`** by adding it to the class declaration. Remember to implement the interface methods **`onItemSelected()`** and **`onNothingSelected()`**:

```
// src/com/example/brewclock/BrewClockActivity.java
public class BrewClockActivity extends Activity implements
OnClickListener, OnItemSelectedListener {
  // …
  public void onItemSelected(AdapterView<?> spinner, View
view, int position, long id) {
    if(spinner == teaSpinner) {
      // Update the brew time with the selected tea's brewtime
```

```
        Cursor cursor = (Cursor) spinner.getSelectedItem();
        setBrewTime(cursor.getInt(2));
      }
    }

    public void onNothingSelected(AdapterView<?> adapterView) {
      // Do nothing
    }
  }
```

Here we check whether the spinner that triggered the **`onItemSelected`** event was BrewClock's **`teaSpinner`**. If so, we retrieve a Cursor object that represents the selected record. This is all handled for us by the **`SimpleCursorAdapter`** that connects **`teaData`** to the Spinner. Android knows which query populates the Spinner and which item the user has selected. It uses these to return the single row from the database, representing the user's selected tea.

Cursor's **`getInt()`** method takes the index of the column we want to retrieve. Remember that when we built our Cursor in **`teaData.all()`**, the columns we read were **`_ID`, `NAME`** and **`BREW_TIME`**. Assuming we chose Jasmine tea in **`teaSpinner`**, the Cursor returned by our selection would be pointing at that record in the database.

We then ask the Cursor to retrieve the value from column 2 (using **`getInt(2)`**), which in this query is our **`BREW_TIME`** column. This value is supplied to our existing **`setBrewTime()`** method, which updates the interface to show the selected tea's brewing time.

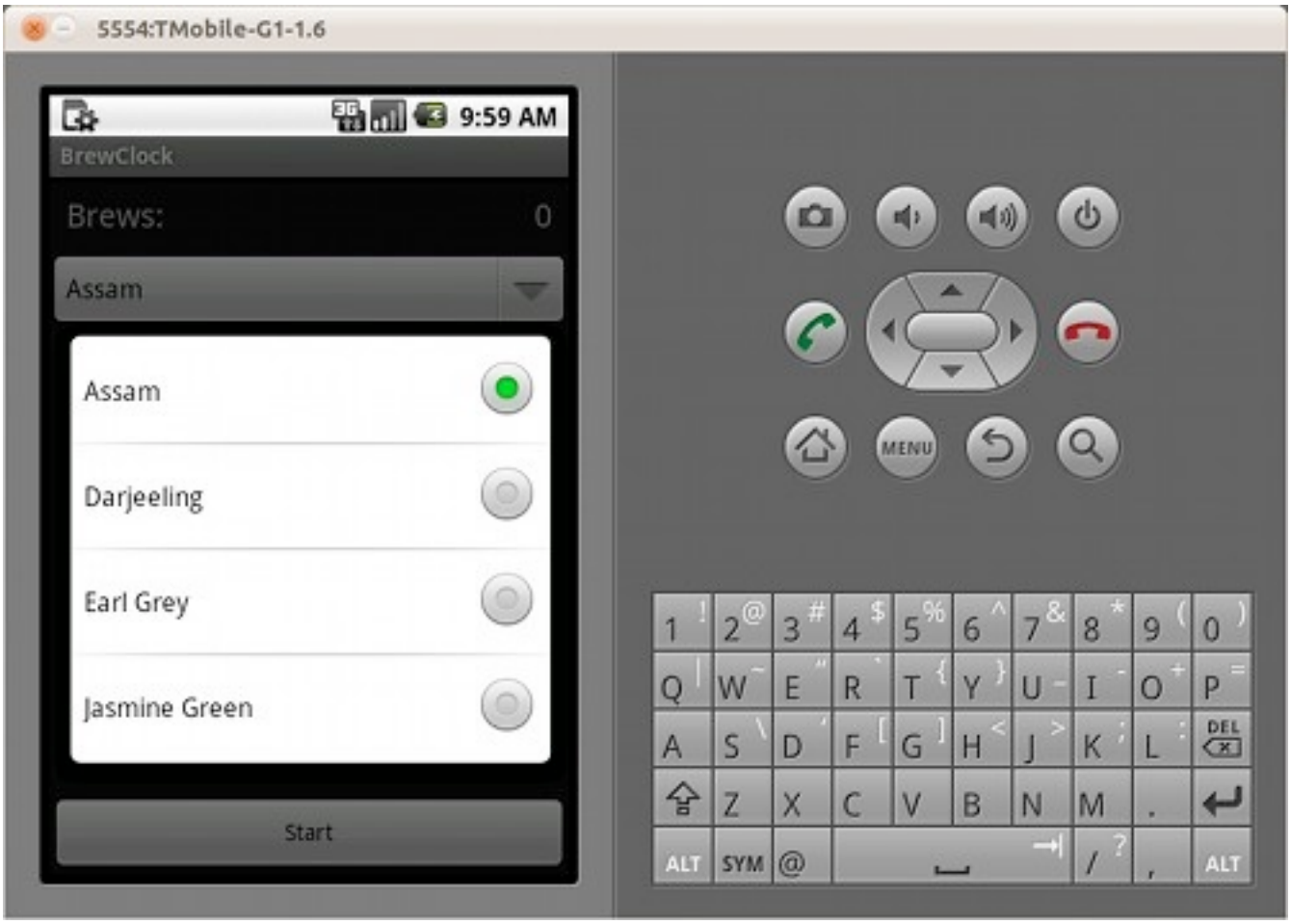
Finally, we need to tell the **`teaSpinner`** that **`BrewClockActivity`** is listening for **`OnItemSelected`** events. Add the following line to **`BrewClockActivity`**'s **`onCreate`** method:

```
// src/com/example/brewclock/BrewClockActivity.java
```

```
public void onCreate() {
  // …
  teaSpinner.setOnItemSelectedListener(this);
}
```

That should do it! Run your application again, and try selecting different teas from the Spinner. Each time you select a tea, its brew time will be shown on the countdown clock. The rest of our code already handles counting down from the current brew time, so we now have a fully working brew timer, with a list of preset teas.

You can, of course, go back into the code and add more preset teas to the database to suit your tastes. But what if we released BrewClock to the market? Every time someone wanted to add a new tea to the database, we'd need to manually update the database, and republish the application; everyone would need to update, and everybody would have the same list of teas. That sounds pretty inflexible, and a lot of work for us!



It would be much better if the user had some way to add their own teas and preferences to the database. We'll tackle that next…

Introducing Activities

Each screen in your application and its associated code is an **`Activity`**. Every time you go from one screen to another, Android creates a new Activity. In reality, although an application may comprise any number of screens/activities, Android treats them as separate entities. Activities work together to form a cohesive experience because Android lets you easily pass data between them.

In this final section, you'll add a new Activity (**`AddTeaActivity`**) to your application and register it with the Android system. You'll then pass data from the original **`BrewClockActivity`** to this new Activity.

First, though, we need to give the user a way to switch to the new Activity. We'll do this using an options menu.

OPTIONS MENUS

Options menus are the pop-up menus that appear when the user hits the "Menu" key on their device. Android handles the creation and display of options menus automatically; you just need to tell it what options to display and what to do when an option is chosen by the user.

However, rather than hard-coding our labels into the menu itself, we'll make use of Android string resources. String resources let you maintain all the human-readable strings and labels for your application in one file, calling them within your code. This means there's only one place in your code where you need to change strings in the future.

In the project explorer, navigate to “res/values” and you will see that a *strings.xml* file already exists. This was created by Eclipse when we first created the project, and it is used to store any strings of text that we want to use throughout the application.

Open *strings.xml* by double clicking on it, and switch to the XML view by clicking the *strings.xml* tab along the bottom of the window.

Add the following line within the **<resources>…</resources>** element:

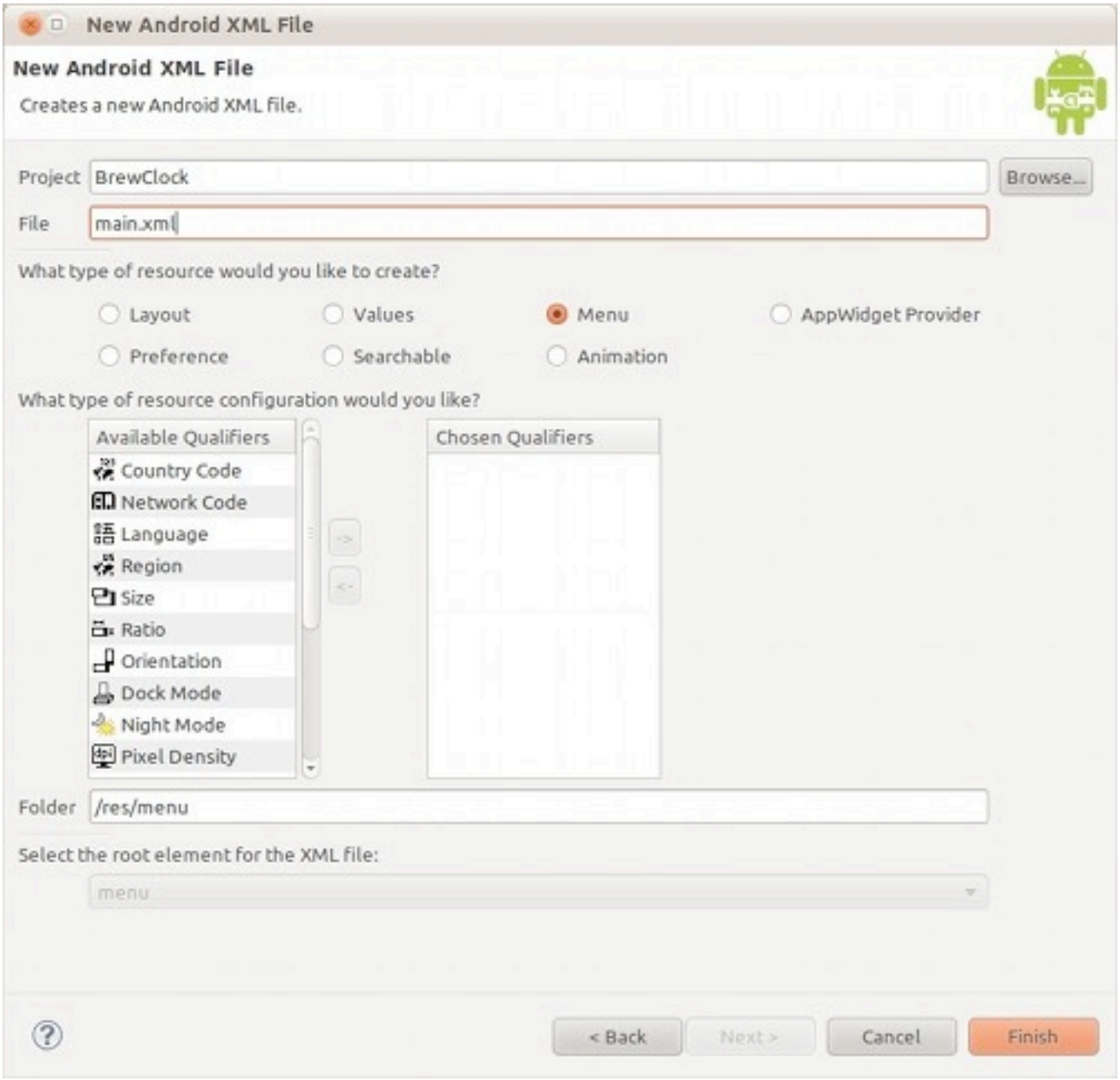
```
<!-- res/values/strings.xml -->
  <resources>
    <!-- … -->
    <string name="add_tea_label">Add Tea</string>
  </resources>
```

Here you’ve defined a string, **add_tea_label**, and its associated text. We can use **add_tea_label** to reference the string throughout the application’s code. If the label needs to change for some reason in the future, you’ll need to change it only once in this file.

Next, let’s create a new file to define our options menu. Just like strings and layouts, menus are defined in an XML file, so we’ll start by creating a new XML file in Eclipse:

Create a new Android XML file in Eclipse by choosing File ➜ New ➜ Other, and then select “Android XML File.”

Select a resource type of “Menu,” and save the file as *main.xml*. Eclipse will automatically create a folder, *res/menu*, where your menu XML files will be stored.



Open the *res/menus/main.xml* file, and switch to XML view by clicking the "main.xml" tab along the bottom of the window.

Add a new menu item, **`add_tea`**.

```
<?xml version="1.0" encoding="utf-8"?>
<menu xmlns:android="http://schemas.android.com/apk/res/
android">
```

```
  <item android:id="@+id/add_tea" android:title="@string/
add_tea_label" />
</menu>
```

Notice the **`android:title`** attribute is set to **`@string/add_tea_label`**. This tells Android to look up **`add_tea_label`** in our *strings.xml* file and return the associated label. In this case, our menu item will have a label "Add Tea."

Next, we'll tell our Activity to display the options menu when the user hits the "Menu" key on their device.

Back in *BrewClockActivity.java*, override the **`onCreateOptionsMenu`** method to tell Android to load our menu when the user presses the "Menu" button:

```
// src/com/example/brewclock/BrewClockActivity.java
@Override
public boolean onCreateOptionsMenu(Menu menu) {
  MenuInflater inflater = getMenuInflater();
  inflater.inflate(R.menu.main, menu);

  return true;
}
```

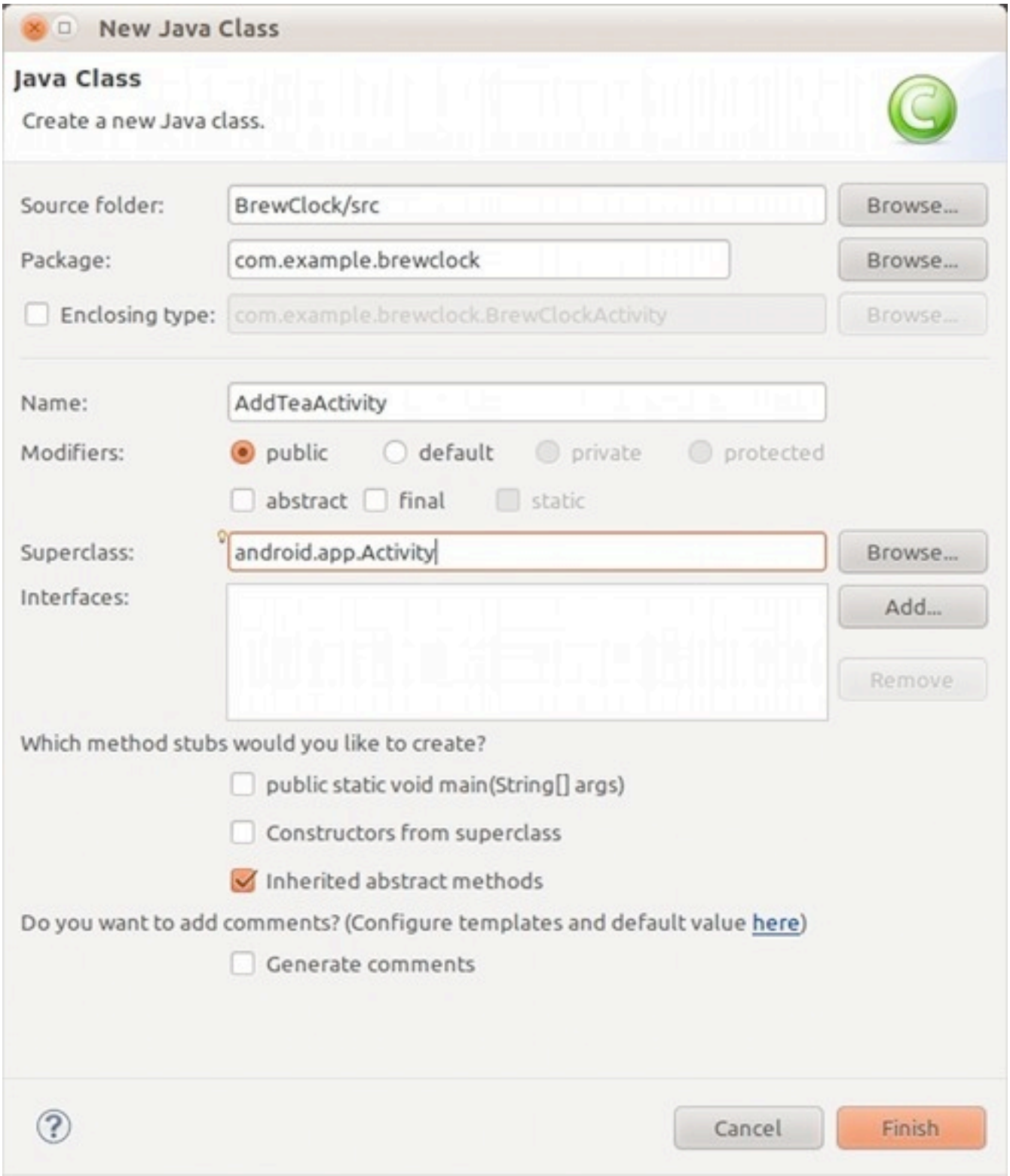
When the user presses the "Menu" button on their device, Android will now call **`onCreateOptionsMenu`**. In this method, we create a **`MenuInflater`**, which loads a menu resource from your application's package. Just like the buttons and text fields that make up your application's layout, the *main.xml* resource is available via the global `R` object, so we use that to supply the **`MenuInflater`** with our menu resource.

To test the menu, save and run the application in the Android emulator. While it's running, press the "Menu" button, and you'll see the options menu pop up with an "Add Tea" option.

If you tap the "Add Tea" option, Android automatically detects the tap and closes the menu. In the background, Android will notify the application that the option was tapped.

HANDLING MENU TAPS

When the user taps the "Add Tea" menu option, we want to display a new Activity so that they can enter the details of the tea to be added. Start by creating that new Activity by selecting File ➜ New ➜ Class.


New Java Class
Java Class
Create a new Java class.
Source folder:
BrewClock/src
Browse...
Package:
com.example.brewclock
Browse...
Enclosing type:
com.example.brewclock.BrewClockActivity
Browse...
Name:
AddTeaActivity
Modifiers:
public
default
private
protected
abstract
final
static
Superclass:
android.app.Activity
Browse...
Interfaces:
Add...
Remove
Which method stubs would you like to create?
public static void main(String[] args)
Constructors from superclass
Inherited abstract methods
Do you want to add comments? (Configure templates and default value here)
Generate comments
Cancel
Finish

Name the new class **`AddTeaActivity`**, and make sure it inherits from the **`android.app.Activity`** class. It should also be in the **`com.example.brewclock`** package:

```
// src/com/example/brewclock/AddTeaActivity.java
package com.example.brewclock;

import android.app.Activity;
import android.os.Bundle;

public class AddTeaActivity extends Activity {
  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
  }
}
```

This simple, blank Activity won't do anything yet, but it gives us enough to finish our options menu.

Add the **`onOptionsItemSelected`** override method to **`BrewClockActivity`**. This is the method that Android calls when you tap on a `MenuItem` (notice it receives the tapped **`MenuItem`** in the item parameter):

```
// src/com/example/brewclock/BrewClockActivity.java
@Override
public boolean onOptionsItemSelected(MenuItem item) {
  switch(item.getItemId()) {
    case R.id.add_tea:
      Intent intent = new Intent(this, AddTeaActivity.class);
      startActivity(intent);
      return true;

    default:
      return super.onOptionsItemSelected(item);
```

```
    }
}
```

With this code, we've told Android that when the "Add Tea" menu item is tapped, we want to start a new Activity; in this case, `AddTeaActivity`. However, rather than directly creating an instance of `AddTeaActivity`, notice that we've used an `Intent`. Intents are a powerful feature of the Android framework: they bind Activities together to make up an application and allow data to be passed between them.

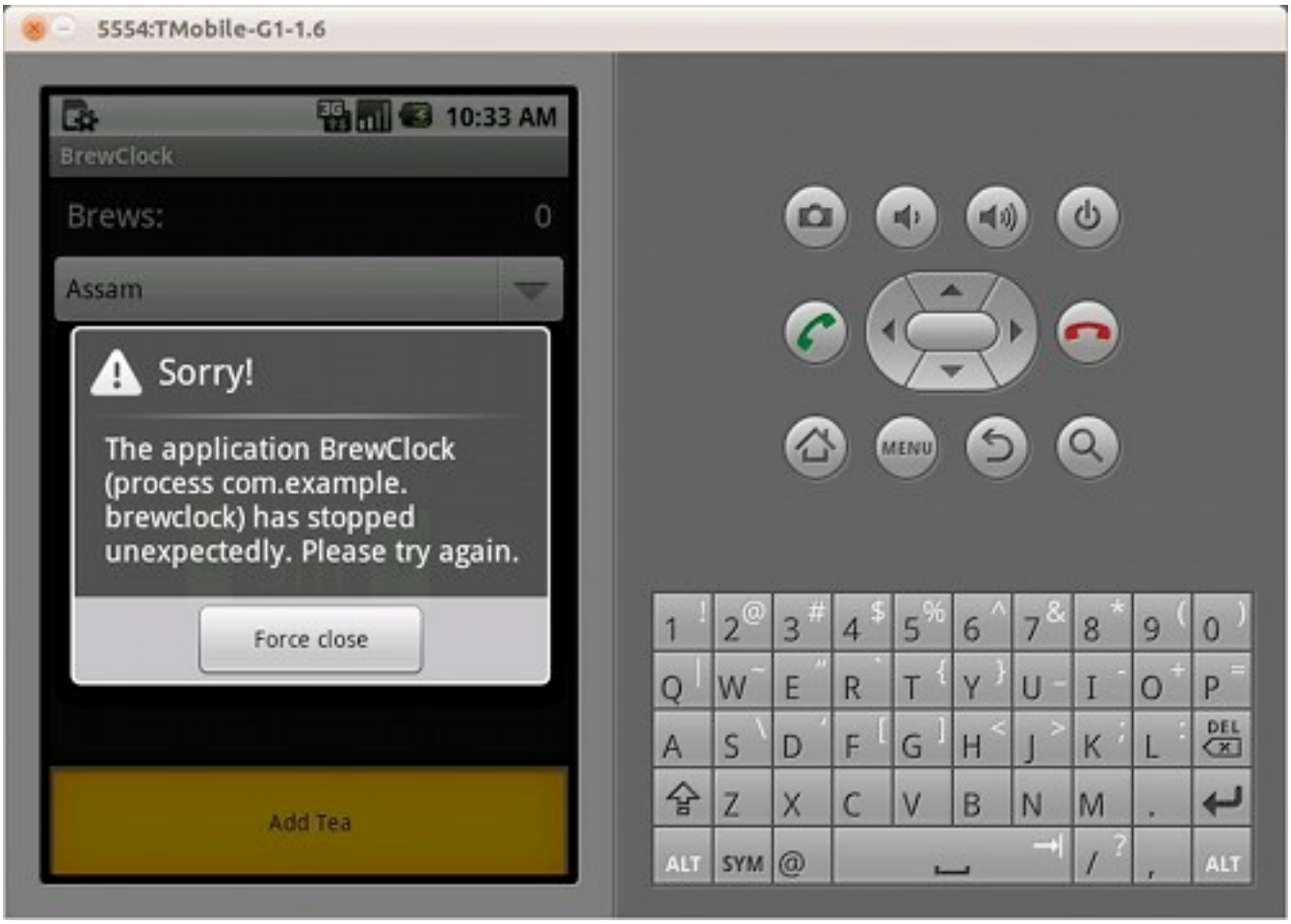
Intents even let your application take advantage of any Activities within other applications that the user has installed. For example, when the user asks to display a picture from a gallery, Android automatically displays a dialogue to the user allowing them to pick the application that displays the image. Any applications that are registered to handle image display will be shown in the dialogue.

Intents are a powerful and complex topic, so it's worth reading about them in detail in the official Android SDK documentation.

Let's try running our application to test out the new "Add Tea" screen.

Run your project, tap the "Menu" button and then tap "Add Tea."

Instead of seeing your "Add Tea" Activity as expected, you'll be presented with a dialogue that is all too common for Android developers:


5554:TMobile-G1-1.6
10:33 AM
BrewClock
Brews:
0
Assam
Sorry!
The application BrewClock (process com.example.brewclock) has stopped unexpectedly. Please try again.
Force close
Add Tea
MENU

Although we created the **`Intent`** and told it to start our **`AddTeaActivity`** Activity, the application crashed because we haven't yet registered it within Android. The system doesn't know where to find the Activity we're trying to run (remember that Intents can start Activities from any application installed on the device). Let's remedy this by registering our Activity within the application manifest file.

Open your application's manifest file, *AndroidManifest.xml* in Eclipse, and switch to the code view by selecting the "AndroidManifest.xml" tab along the bottom of the window.

The application's manifest file is where you define global settings and information about your application. You'll see that it already declares **`.BrewClockActivity`** as the Activity to run when the application is launched.

Within **`<application>`**, add a new **`<activity>`** node to describe the "Add Tea" Activity. Use the same **`add_tea_label`** string that we declared earlier in *strings.xml* for the Activity's title:

```
<!-- AndroidManifest.xml -->
<application …>
  …
  <activity android:name=".AddTeaActivity"
android:label="@string/add_tea_label" />
</application>
```

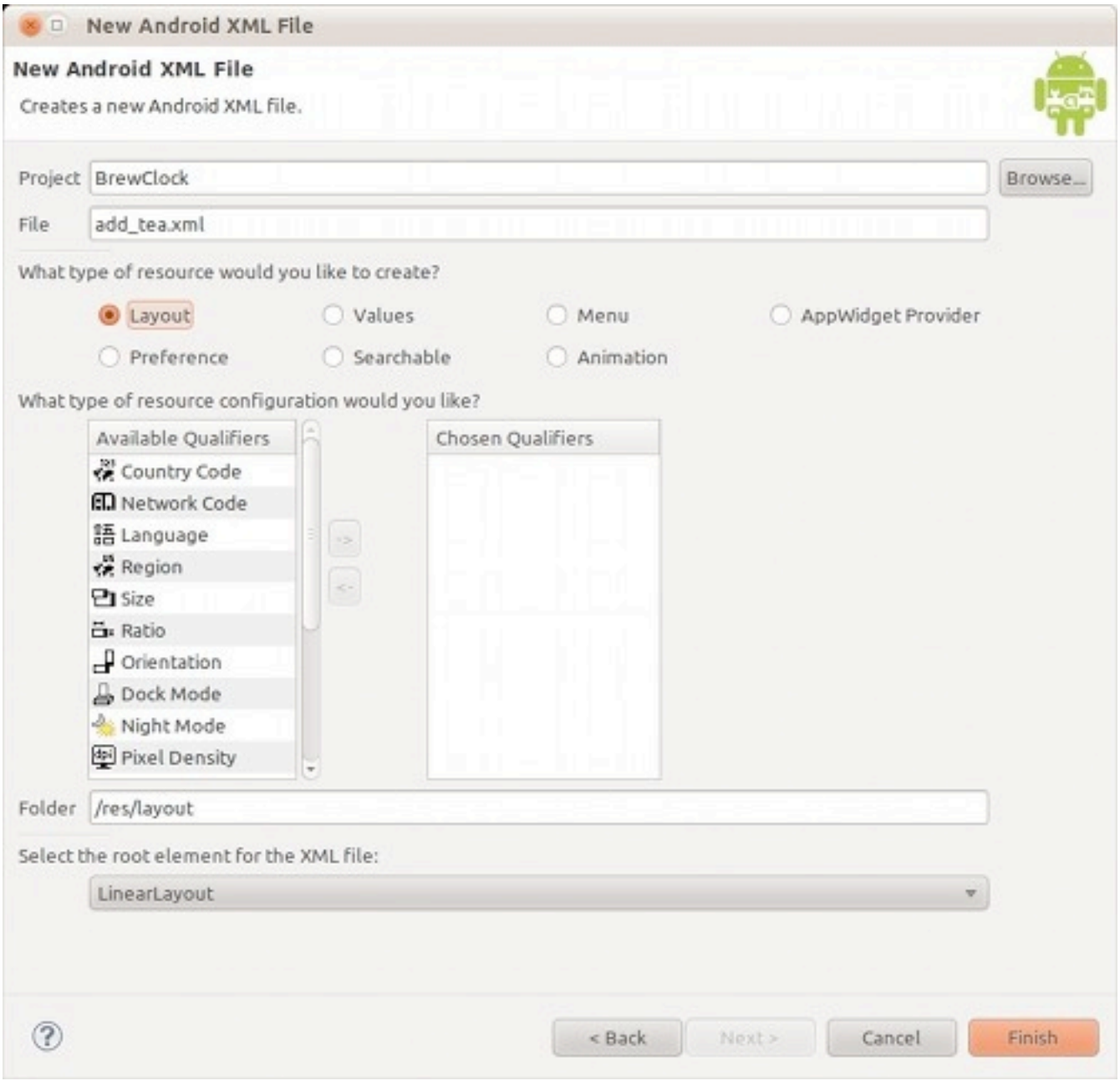
Save the manifest file before running BrewClock again. This time, when you open the menu and tap “Add Tea,” Android will start the **`AddTeaActivity`**. Hit the “Back” button to go back to the main screen.

With the Activities hooked together, it’s time to build an interface for adding tea!

Building The Tea Editor Interface

Building the interface to add a tea is very similar to how we built the main BrewClock interface in the previous tutorial. Start by creating a new layout file, and then add the appropriate XML, as below.

Alternatively, you could use Android’s recently improved layout editor in Eclipse to build a suitable interface. Create a new XML file in which to define the layout. Go to File ➜ New, then select “Android XML File,” and select a “Layout” type. Name the file *add_tea.xml*.



Replace the contents of *add_tea.xml* with the following layout:

```
<!-- res/layouts/add_tea.xml -->
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
  xmlns:android="http://schemas.android.com/apk/res/android"
  android:layout_width="fill_parent"
  android:layout_height="fill_parent"
```

```
  android:orientation="vertical"
  android:padding="10dip">

  <TextView
    android:text="@string/tea_name_label"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content" />

  <EditText
    android:id="@+id/tea_name"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"/>

  <TextView
    android:text="@string/brew_time_label"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"/>

  <SeekBar
    android:id="@+id/brew_time_seekbar"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:progress="2"
    android:max="9" />

  <TextView
    android:id="@+id/brew_time_value"
    android:text="3 m"
    android:textSize="20dip"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:gravity="center_horizontal" />
</LinearLayout>
```

We'll also need to add some new strings to *strings.xml* for the labels used in this interface:

```
<!-- res/values/strings.xml -->
<resources>
  <!-- … -->
  <string name="tea_name_label">Tea Name</string>

  <string name="brew_time_label">Brew Time</string>
</resources>
```

In this layout, we've added a new type of interface widget, the SeekBar. This lets the user easily specify a brew time by dragging a thumb from left to right. The range of values that the SeekBar produces always runs from zero (0) to the value of **`android:max`**.

In this interface, we've used a scale of 0 to 9, which we will map to brew times of 1 to 10 minutes (brewing for 0 minutes would be a waste of good tea!). First, though, we need to make sure that **`AddTeaActivity`** loads our new interface:

Add the following line of code to the Activity's **`onCreate()`** method that loads and displays the **`add_tea`** layout file:

```
// src/com/example/brewclock/AddTeaActivity.java
public void onCreate(Bundle savedInstanceState) {
  super.onCreate(savedInstanceState);
  setContentView(R.layout.add_tea);
}
```

Now test your application by running it, pressing the “Menu” button and tapping “Add Tea” from the menu.

You’ll see your new interface on the “Add Tea” screen. You can enter text and slide the SeekBar left and right. But as you’d expect, nothing works yet because we haven’t hooked up any code.

Declare some properties in **`AddTeaActivity`** to reference our interface elements:

```
// src/com/example/brewclock/AddTeaActivity.java
public class AddTeaActivity {
  // …
```

```
  /** Properties **/
  protected EditText teaName;
  protected SeekBar brewTimeSeekBar;
  protected TextView brewTimeLabel;

  // …
```

Next, connect those properties to your interface:

```
public void onCreate(Bundle savedInstanceState) {
  // …
  // Connect interface elements to properties
  teaName = (EditText) findViewById(R.id.tea_name);
  brewTimeSeekBar = (SeekBar)
findViewById(R.id.brew_time_seekbar);
  brewTimeLabel = (TextView)
findViewById(R.id.brew_time_value);
}
```

The interface is fairly simple, and the only events we need to listen for are changes to the SeekBar. When the user moves the SeekBar thumb left or right, our application will need to read the new value and update the label below with the selected brew time. We'll use a **`Listener`** to detect when the SeekBar is changed:

Add an **`onSeekBarChangedListener`** interface to the **`AddTeaActivity`** class declaration, and add the required methods:

```
// src/com/example/brewclock/AddTeaActivity.java
public class AddTeaActivity
extends Activity
implements OnSeekBarChangeListener {
  // …

  public void onProgressChanged(SeekBar seekBar, int progress,
boolean fromUser) {
    // TODO Detect change in progress
```

```
    }

    public void onStartTrackingTouch(SeekBar seekBar) {}

    public void onStopTrackingTouch(SeekBar seekBar) {}
  }
```

The only event we're interested in is **`onProgressChanged`**, so we need to add the code below to that method to update the brew time label with the selected value. Remember that our SeekBar values range from 0 to 9, so we'll add 1 to the supplied value so that it makes more sense to the user:

Add the following code to **`onProgressChanged()`** in *AddTeaActivity.java*:

```
// src/com/example/brewclock/AddTeaActivity.java
public void onProgressChanged(SeekBar seekBar, int progress,
boolean fromUser) {
  if(seekBar == brewTimeSeekBar) {
    // Update the brew time label with the chosen value.
    brewTimeLabel.setText((progress + 1) + " m");
  }
}
```

Set the SeekBar's listener to be our **`AddTeaActivity`** in **`onCreate`**:

```
// src/com/example/brewclock/AddTeaActivity.java
public void onCreate(Bundle savedInstanceState) {
  // …

  // Setup Listeners
  brewTimeSeekBar.setOnSeekBarChangeListener(this);
}
```

Now when run the application and slide the SeekBar left to right, the brew time label will be updated with the correct value:

SAVING TEA

With a working interface for adding teas, all that's left is to give the user the option to save their new tea to the database. We'll also add a little validation to the interface so that the user cannot save an empty tea to the database!

Start by opening *strings.xml* in the editor and adding some new labels for our application:

```
<!-- res/values/strings.xml -->
<string name="save_tea_label">Save Tea</string>
```

```
<string name="invalid_tea_title">Tea could not be saved.</
string>

<string name="invalid_tea_no_name">Enter a name for your
tea.</string>
```

Just like before, we'll need to create a new options menu for **`AddTeaActivity`** so that the user can save their favorite tea:

Create a new XML file, *add_tea.xml*, in the *res/menus* folder by choosing File ➜ New and then Other ➜ Android XML File. Remember to select "Menu" as the resource type.

Add an item to the new menu for saving the tea:

```
<!-- res/menus/add_tea.xml -->
<?xml version="1.0" encoding="utf-8"?>
<menu xmlns:android="http://schemas.android.com/apk/res/
android">
  <item android:title="@string/save_tea_label" android:id="@
+id/save_tea" />
</menu>
```

Back in **`AddTeaActivity`**, add the override methods for **`onCreateOptionsMenu`** and **`onOptionsItemSelected`**, just like you did in **`BrewClockActivity`**. However, this time, you'll supply the *add_tea.xml* resource file to the **`MenuInflater`**:

```
// src/com/example/brewclock/AddTeaActivity.java
@Override
public boolean onCreateOptionsMenu(Menu menu) {
  MenuInflater inflater = getMenuInflater();
  inflater.inflate(R.menu.add_tea, menu);

  return true;
}
```

```
@Override
public boolean onOptionsItemSelected(MenuItem item) {
  switch(item.getItemId()) {
    case R.id.save_tea:
      saveTea();

    default:
      return super.onOptionsItemSelected(item);
  }
}
```

Next, we'll add a new method, **`saveTea()`**, to handle saving the tea. The **`saveTea`** method first reads the name and brew time values chosen by the user, validates them and (if all is well) saves them to the database:

```
// src/com/example/brewclock/AddTeaActivity.java
public boolean saveTea() {
  // Read values from the interface
  String teaNameText = teaName.getText().toString();
  int brewTimeValue = brewTimeSeekBar.getProgress() + 1;

  // Validate a name has been entered for the tea
  if(teaNameText.length() < 2) {
    AlertDialog.Builder dialog = new
AlertDialog.Builder(this);
    dialog.setTitle(R.string.invalid_tea_title);
    dialog.setMessage(R.string.invalid_tea_no_name);
    dialog.show();

    return false;
  }

  // The tea is valid, so connect to the tea database and
insert the tea
  TeaData teaData = new TeaData(this);
  teaData.insert(teaNameText, brewTimeValue);
```

```
    teaData.close();

    return true;
  }
```

This is quite a hefty chunk of code, so let's go over the logic.

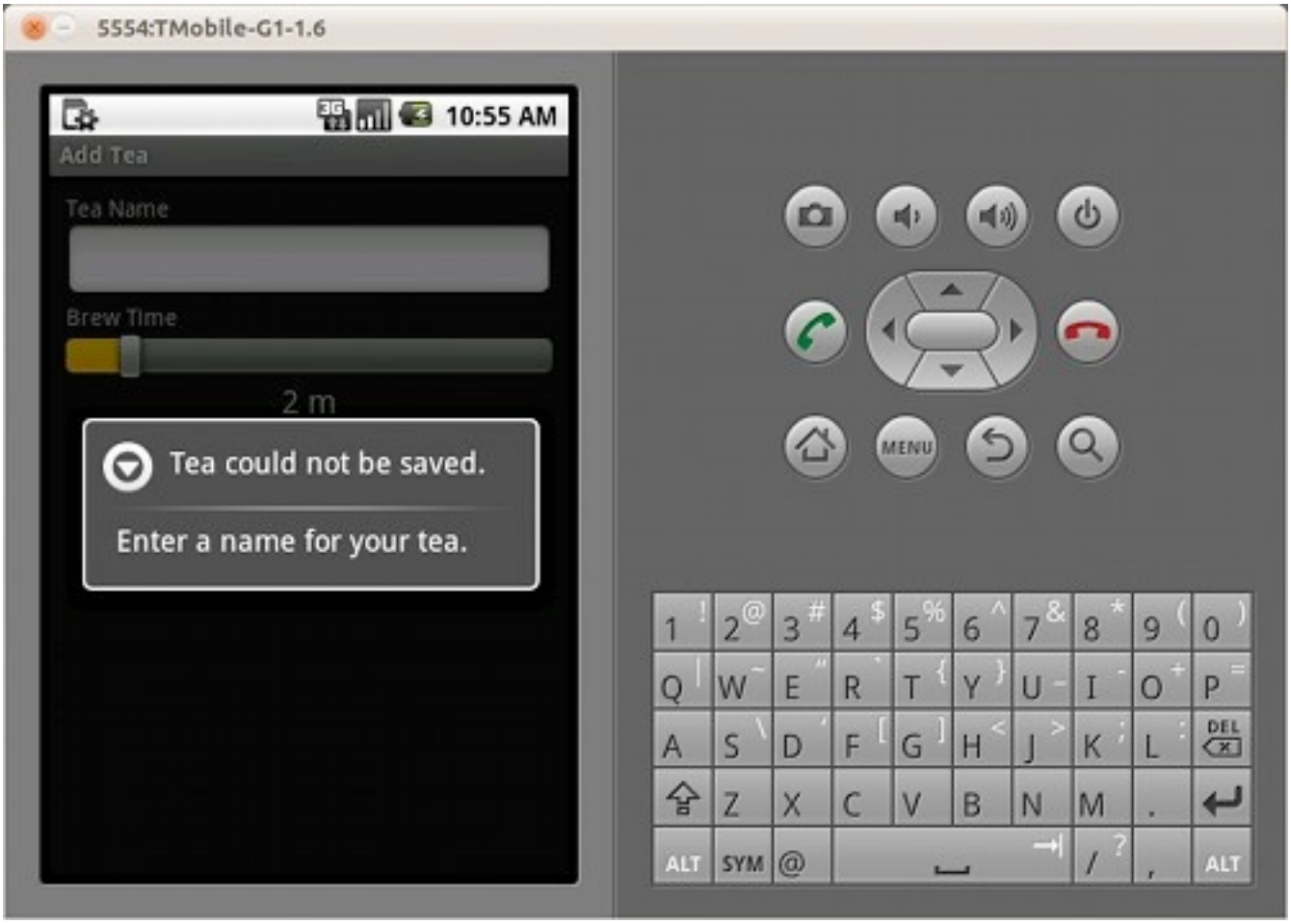
First, we read the values of the EditText **`teaName`** and the SeekBar **`brewTimeSeekBar`** (remembering to add 1 to the value to ensure a brew time of between 1 and 10 minutes). Next, we validate that a name has been entered that is two or more characters (this is really simple validation; you might want to experiment doing something more elaborate, such as using regular expressions).

If the tea name is not valid, we need to let the user know. We make use of one of Android's helper classes, **`AlertDialog.Builder`**, which gives us a handy shortcut for creating and displaying a modal dialog window. After setting the title and error message (using our string resources), the dialogue is displayed by calling its **`show()`** method. This dialogue is modal, so the user will have to dismiss it by pressing the "Back" key. At this point, we don't want to save any data, so just return **`false`** out of the method.

If the tea is valid, we create a new temporary connection to our tea database using the **`TeaData`** class. This demonstrates the advantage of abstracting your database access to a separate class: you can access it from anywhere in the application!

After calling **`teaData.insert()`** to add our tea to the database, we no longer need this database connection, so we close it before returning **`true`** to indicate that the save was successful.

Try this out by running the application in the emulator, pressing “Menu” and tapping “Add Tea.” Once on the “Add Tea” screen, try saving an empty tea by pressing “Menu” again and tapping “Save Tea.” With your validation in place, you’ll be presented with an error message:



Next, try entering a name for your tea, choosing a suitable brew time, and choosing “Save Tea” from the menu again. This time, you won’t see an error message. In fact, you’ll see nothing at all.

IMPROVING THE USER EXPERIENCE

While functional, this isn’t a great user experience. The user doesn’t know that their tea has been successfully saved. In fact, the only way to check is to go back from the “Add Tea” Activity and check the list of teas. Not great. Letting the user know that their tea was successfully saved would be much better. Let’s show a message on the screen when a tea has been added successfully.

We want the message to be passive, or non-modal, so using an **`AlertDialog`** like before won’t help. Instead, we’ll make use of another popular Android feature, the **`Toast`**.

Toasts display a short message near the bottom of the screen but do not interrupt the user. They’re often used for non-critical notifications and status updates.

Start by adding a new string to the *strings.xml* resource file. Notice the `%s` in the string? We’ll use this in the next step to interpolate the name of the saved tea into the message!

```
<!-- res/values/strings.xml -->
<string name="save_tea_success">%s tea has been saved.</
string>
```

Modify the code in **`onOptionsItemSelected`** to create and show a **`Toast`** pop-up if the result of **`saveTea()`** is **`true`**. The second parameter uses of **`getString()`** interpolate the name of our tea into the **`Toast`** message. Finally, we clear the “Tea Name” text so that the user can quickly add more teas!

```
// src/com/example/brewclock/AddTeaActivity.java
// …
switch(item.getItemId()) {
 case R.id.save_tea:
   if(saveTea()) {
     Toast.makeText(this, getString(R.string.save_tea_success,
teaName.getText().toString()), Toast.LENGTH_SHORT).show();
     teaName.setText("");
   }
// …
```

Now re-run your application and try adding and saving a new tea. You'll see a nice **`Toast`** pop up to let you know the tea has been saved. The **`getString()`** method interpolates the name of the tea that was saved into the XML string, where we placed the `%s`.

Click the "Back" button to return to the application's main screen, and tap the tea spinner. The new teas you added in the database now show up as options in the spinner!

User Preferences

BrewClock is now fully functional. Users can add their favorite teas and the respective brewing times to the database, and they can quickly select them to start a new brew. Any teas added to BrewClock are saved in the database, so even if we quit the application and come back to it later, our list of teas is still available.

One thing you might notice when restarting BrewClock, though, is that the brew counter is reset to 0. This makes keeping track of our daily tea intake (a vital statistic!) difficult. As a final exercise, let's save the total brew count to the device.

Rather than adding another table to our teas database, we'll make use of Android's "Shared Preferences," a simple database that Android provides to your application for storing simple data (strings, numbers, etc.), such as high scores in games and user preferences.

Start by adding a couple of constants to the top of *BrewClockActivity.java*. These will store the name of your shared preferences file and the name of the key we'll use to access the brew count. Android takes care of saving and persisting our shared preferences file.

```
// src/com/example/brewclock/BrewClockActivity.java
protected static final String SHARED_PREFS_NAME =
"brew_count_preferences";
protected static final String BREW_COUNT_SHARED_PREF =
"brew_count";
```

Next, we'll need to make some changes to the code so that we can read and write the brew count to the user preferences, rather than relying on an initial value in our code. In **`BrewClockActivity`'s `onCreate`** method, change the lines around **`setBrewCount(0)`** to the following:

```
// src/com/example/brewclock/BrewClockActivity.java
public void onCreate() {
  // …

  // Set the initial brew values
  SharedPreferences sharedPreferences =
getSharedPreferences(SHARED_PREFS_NAME, MODE_PRIVATE);
  brewCount = sharedPreferences.getInt(BREW_COUNT_SHARED_PREF,
0);
  setBrewCount(brewCount);

  // …
}
```

Here we're retrieving an instance of the application's shared preferences using **`SharedPreferences`**, and asking for the value of the **`brew_count`** key (identified by the **`BREW_COUNT_SHARED_PREF`** constant that was declared earlier). If a value is found, it will be returned; if not, we'll use the default value in the second parameter of **`getInt`** (in this case, 0).

Now that we can retrieve the stored value of brew count, we need to ensure its value is saved to **`SharedPreferences`** whenever the count is updated.

Add the following code to **`setBrewCount`** in **`BrewClockActivity`**:

```
// src/com/example/brewclock/BrewClockActivity.java
 public void setBrewCount(int count) {
   brewCount = count;
   brewCountLabel.setText(String.valueOf(brewCount));
```

```
    // Update the brewCount and write the value to the shared
preferences.
    SharedPreferences.Editor editor =
getSharedPreferences(SHARED_PREFS_NAME, MODE_PRIVATE).edit();
    editor.putInt(BREW_COUNT_SHARED_PREF, brewCount);
    editor.commit();
  }
```

Shared preferences are never saved directly. Instead, we make use of Android's **`SharedPreferences.Editor`** class. Calling **`edit()`** on **`SharedPreferences`** returns an **`editor`** instance, which can then be used to set values in our preferences. When the values are ready to be saved to the shared preferences file, we just call **`commit()`**.

With our application's code all wrapped up, it's time to test everything!

Run the application on the emulator, and time a few brews (this is the perfect excuse to go and make a well-deserved tea or two!), and then quit the application. Try running another application that is installed on the emulator to ensure BrewClock is terminated. Remember that Android doesn't terminate an Activity until it starts to run out of memory.

When you next run the application, you'll see that your previous brew count is maintained, and all your existing teas are saved!

Summary

Congratulations! You've built a fully working Android application that makes use of a number of core components of the Android SDK. In this tutorial, you have seen how to:

- Create a simple SQLite database to store your application's data;
- Make use of Android's database classes and write a custom class to abstract the data access;
- Add option menus to your application;
- Create and register new Activities within your application and bind them together into a coherent interface using Intents;
- Store and retrieve simple user data and settings using the built-in "Shared Preferences" database.

Data storage and persistence is an important topic, no matter what type of application you're building. From utilities and business tools to 3-D games, nearly every application will need to make use of the data tools provided by Android.

ACTIVITIES

BrewClock is now on its way to being a fully functional application. However, we could still implement a few more features to improve the user experience. For example, you might like to develop your skills by trying any of the following:

- Checking for duplicate tea name entries before saving a tea,
- Adding a menu option to reset the brew counter to 0,
- Storing the last-chosen brew time in a shared preference so that the application defaults to that value when restarted,
- Adding an option for the user to delete teas from the database.

Solutions for the Activities will be included in a future branch on the GitHub repository, where you'll find the full source-code listings. You can download the working tutorial code by switching your copy of the code to the **`tutorial_2`** branch:

```
# If you've not already cloned the repository,
# you'll need to do that first:
# $ git clone git://github.com/cblunt/BrewClock.git
# $ cd BrewClock
$ git checkout tutorial_2
```

I hope you've enjoyed working through this tutorial and that it helps you in designing and building your great Android applications.

About The Authors

Chris Blunt

Chris is a software developer working with Ruby, Rails and Android. In 2010, he founded Plymouth Software where he designs and builds applications for the web and mobile devices. As well as a fondness for travel and drinking tea, Chris writes about code, design and business on his blog at chrisblunt.com.
Website: chrisblunt.com
Twitter: Follow the author on Twitter

Dan McKenzie

Born and raised in Silicon Valley, Daniel McKenzie is a digital product designer helping startups and companies strategize and design products that matter. He also likes to write and tweet (@danielmckenzie) on various design and innovation topics.
Twitter: Follow the author on Twitter

Jamie McDonald

Jamie is an Android Developer at Novoda, where he drives a focus on usability and works closely with visual designers to create beautiful apps. Novoda is a London-based start-up that specializes on the Android platform, producing high-quality apps with design partners, start-ups and device manufacturers.
Twitter: Follow the author on Twitter

Juhani Lehtimaki

Juhani is an Android developer with more than ten years of experience in Java development in different business domains. He currently works as head of Android and Google TV development at Snapp TV (snapp.tv). He writes a blog about Android UI Design Patterns at www.androiduipatterns.com and tweets with focus on Android and Android design under @lehtimaeki twitter handle.
Website: Android UI Design Patterns
Twitter: Follow the author on Twitter
Google Profile: https://plus.google.com/u/0/102272971619910906878

Sue Smith

Sue Smith lives and works in Glasgow, Scotland doing Web, multimedia and mobile development plus a bit of writing. Sue began developing Android applications a short while ago, writes technical and educational material on a freelance basis, and has a number of comedy websites/ blogs, including Brain Dead Air Magazine. Started tweeting recently @BrainDeadAir.
Website: BeNormal
Twitter: Follow the author on Twitter