ANDROID

Android

Android Developer Toolbox

If you've read Getting Started and Developing Apps, then you know how to build an Android application. (If you haven't read those sections you should now.)

Android is a large system though, and there's a lot to learn. The best apps always make the most of the system's features. The links below tell you how to write code that bends the system to your will, allowing you to create cool custom components and do amazing things with the many available APIs.

Design Philosophy

A manifesto explaining a technical philosophy and perspective that Android developers will find useful. By reading this page, you'll learn how to write applications that perform well on embedded devices (such as phone handsets), and that play nicely with other parts of the system.

Building Custom Components

Explains how to create custom Android components, such as Views, Services, and Content Providers. Read this, and you'll soon be knocking out great-looking, efficient, and useful components. You can use these to make your own application great, or you can share them with other applications.

Optional APIs

Describes the APIs that provide access to optional Android components, such as GPS and BlueTooth. Android aims to be more than just another OS, and so the system supports all the features you could hope for in a phone. This section will show you how to use the Location-Based Services (such as GPS, compass, etc.), OpenGL 3D graphics, Bluetooth, and accelerometer.

Note that the APIs described in this section are all optional; if your application truly requires one of these APIs, you should be sure that it fails gracefully if the features are not present on a given phone.

Android

Android Application Design Philosophy

The process of learning how to build applications for a new API is pretty similar, even if the platforms themselves are wildly different. Generally, there are two phases: first, you learn how to use the APIs to do what you want to do; later, you learn the nuances of the platform. Put another way, first you learn how you *can* build applications; later, you learn how you *should* build them.

That second phase — learning the right way to build applications — can often take a long time, and frequently means "paying your dues", making mistakes, and learning from them. Well, that's not a very efficient process, so this page and the links below aim to give you a helping hand.

Before we dive into it, a quick word. Successful applications will offer an outstanding end-user experience. While the Android team has built a robust core system, the vast majority of the user experience will come from users interacting with your applications. As a result, we encourage you to take the time to build an outstanding user experience.

An outstanding user experience has three key characteristics: it is *fast*; it is *responsive*; and it is *seamless*. Of course every platform since the dawn of computing has probably cited those same three qualities at one time or another. However, each platform achieves them in different ways; the information below explains how your apps can achieve them on Android.

Fast

An Android application should be fast. Well, it's probably more accurate to say that it should be *efficient*. There is a tendency in the computing world these days to assume that Moore's Law will solve all our problems — eventually. When it comes to embedded applications, though, Moore's Law is a bit more complicated.

Moore's Law doesn't really apply to mobile devices in the same way as to desktop and server applications. Moore's Law is actually a law about transistor density — that is, it says that you can pack more circuitry into a given chip size, over time. For desktop and server applications, this means you can pack more "speed" into a chip of roughly the same size, resulting in the well-known performance increases. For embedded applications like cell phones, however, Moore's Law is usually exploited to make chips *smaller*. That is, the tendency is to use the increased density to make the same chip smaller and consume less power, to make phones smaller and make batteries last longer. As a result, embedded devices like phones are increasing in actual, raw speed much more slowly than desktop systems. For embedded devices, Moore's Law means more features and better battery life; increased speed is only an afterthought.

That's why it's important to write efficient code: you can't assume that phones will see the same speed increases as desktops and servers. Generally speaking, writing fast code means keeping memory allocations to a minimum, writing tight code, and avoiding certain language and programming idioms that can subtly cripple performance. In object-oriented terms, most of this work takes place at the *method* level, on the order of actual lines of code, loops, and so on.

The article on Writing Efficient Android Code will give you all the detail you need to write fast, efficient code for Android.

Responsive

It's possible to write code that wins every performance test in the world, but that still sends users in a fiery rage when they try to use it. These are the applications that aren't *responsive* enough — the ones that feel sluggish, hang or freeze for significant periods, or take too long to process input. In Android terms, applications that are insufficiently responsive will frequently cause the system to pop up the dreaded "Application Not Responding" (ANR) message.

Generally, this happens if your application cannot respond to user input. For example, if your application blocks on some I/O operation (frequently a network access), then the main application thread won't be able to process incoming user input events. After a time the system will conclude that your application has hung, and give the user the option to kill it. Similarly, if your application spends too much time building an elaborate in-memory structure, or perhaps computing the next move in a game, then again the system will conclude that your application has hung. It's always important to make sure these computations are efficient using the techniques above, but even the most efficient code still takes time to run.

In both of these cases, the fix is usually to create a child thread, and do most of your work there. This keeps the main thread (which drives the user interface event loop) running, and prevents the system from concluding your code has frozen. Since such threading usually is accomplished at the class level, you can think of responsiveness as a *class* problem. (Compare this with basic performance, which was described above as a *method*-level concern.)

The article on Building Responsive Android Applications discusses responsiveness in detail.

Seamless

Even if your application is fast and responsive, it can still annoy users. A common example is a background process (such as an Android Service or BroadcastReceiver) that pops up a UI in response to some event. This may seem harmless, and frequently developers assume that this is okay because they spend most of their time testing and using their own application. However, Android's application model is constructed explicitly to allow users to fluidly switch between applications. This means that when your background process actually fires up that UI, the user could be way over in another part of the system, doing something else — such as taking a phone call. Imagine if the SMS service popped up a dialog box every time a text message came in; this would annoy users in no time. That's why the Android standard is to use Notifications for such events; this leaves the user in control.

That's just one example; there are many more. For example, if Activities don't correctly implement the onPause() and other life-cycle methods, this will frequently result in data loss. Or, if your application exposes data intended to be used by other applications, you should expose it via a ContentProvider, rather than (for example) using a world-readable raw file or database.

What those examples have in common is that they involve cooperating nicely with the system and other applications. The Android system is designed to treat applications as a sort of federation of loosely-coupled components, rather than chunks of black-box code. This allows you as the developer to view the entire system as just an even-larger federation of these components. This benefits you by allowing you to integrate cleanly and seamlessly with other applications, and so you should design your own code to return the favor.

This is a *component*-level concept (as opposed to the *class*- and *method*-level concepts of performance and responsiveness, described above.) The article on Integrating with the System provides tips and best practices for writing code that cooperates nicely with the rest of the system.

Android

Building Custom Android Components

Android comes with a solid collection of View components that you can use to construct your applications, e.g. Button, TextView, EditText, ListView, CheckBox, RadioButton, Gallery, Spinner, and even some much more advanced and special purpose Views like AutoCompleteTextView, ImageSwitcher, and TextSwitcher. The various layout managers like LinearLayout, FrameLayout, and so forth are also considered Views and are descendents of the View class hierarchy.

You can combine these layouts and controls into a screen to display in your application, and much of the time this may be enough for you, but you should also be aware that you can create custom components by extending Views, Layouts, and even the advanced controls using inheritance. Some typical reasons for doing this might include:

- To create a completely custom-rendered component, for example a "volume control" knob rendered using 2D graphics, and which resembles an analog electronic control.
- Combine a group of View components into a new single component, perhaps to make something like a ComboBox (a combination of popup list and free entry text field), a dual-pane selector control (a left and right pane with a list in each where you can re-assign which item is in which list), and so on.
- To create your own kind of layout. The layouts provided in the SDK provide a good set of options for designing your own applications, but advanced developers may find the need to provide a new layout that extends one of the existing ones, or perhaps is entirely new.
- Override the display or behavior of an existing component; for example, change the way that an EditText component is rendered on the screen (the Notepad sample uses this to good effect to create a lined-notepad page).
- Capture other events like key presses and handle them in some custom way (e.g. for a game).

There are many more reasons why you might want to extend an existing View to achieve some goal. This page will give you some starting points on how to do it, and back it up with some examples.

Contents

The Basic Approach

These steps provide a high level overview of what you need to know to get started in creating your own components:

1. Extend an existing View class or subclass with your own class.
2. Override some of the methods from the superclass: the superclass methods to override start with '`on`', for example, onDraw(), onMeasure(), and onKeyDown().
    - This is similar to the `on...` events in Activity or ListActivity that you override for life cycle and other functionality hooks.
3. Use your new extension class: once completed, your new extension class can be used in place of the view upon which it was based, but now with the new functionality.

Extension classes can be defined as inner classes inside the activities that use them. This is useful because it controls access to them but isn't necessary (perhaps you want to create a new public component for wider use in your application).

Fully Customized Components

Fully customized components can be used to create graphical components that appear however you wish. Perhaps a graphical VU meter that looks like an old analog gauge, or a sing-a-long text view where a bouncing ball moves along the words so you can sing along with a karaoke machine. Either way, you want something that the built-in components just won't do, no matter how you combine them.

Fortunately, you can easily create components that look and behave in any way you like, limited perhaps only by your imagination, the size of the screen, and the available processing power (remember that ultimately your application might have to run on something with significantly less power than your desktop workstation).

To create a fully customized component:

1. The most generic view you can extend is, unsurprisingly, View, so you will usually start by extending this to create your new super component.
2. You can supply a constructor which can take attributes and parameters from the XML, and you can also consume your own such attributes and parameters (perhaps the color and range of the VU meter, or the width and damping of the needle, etc.)
3. You will probably want to create your own event listeners, property accessors and modifiers, and possibly more sophisticated behavior in your component class as well.
4. You will almost certainly want to override `onMeasure()` and are also likely to need to override `onDraw()` if you want the component to show something. While both have default behavior, the default `onDraw()` will do nothing, and the default `onMeasure()` will always set a size of 100x100 — which is probably not what you want.
5. Other `on...` methods may also be overridden as required.

`onDraw()` and `onMeasure()`

`onDraw()` delivers you a Canvas upon which you can implement anything you want: 2D graphics, other standard or custom components, styled text, or anything else you can think of.

Note: Except for 3D graphics. If you want to use 3D graphics, you must extend SurfaceView instead of View, and draw from a seperate thread. See the GLSurfaceViewActivity sample for details.

`onMeasure()` is a little more involved. `onMeasure()` is a critical piece of the rendering contract between your component and its container. `onMeasure()` should be overridden to efficiently and accurately report the measurements of its contained parts. This is made slightly more complex by the requirements of limits from the parent (which are passed in to the `onMeasure()` method) and by the requirement to call the `setMeasuredDimension()` method with the measured width and height once they have been calculated. If you fail to call this method from an overridden `onMeasure()` method, the result will be an exception at measurement time.

At a high level, implementing `onMeasure()` looks something like this:

1. The overridden `onMeasure()` method is called with width and height measure specifications (`widthMeasureSpec` and `heighMeasureSpec` parameters, both are integer codes representing dimensions) which should be treated as requirements for the restrictions on the width and height measurements you should produce. A full reference to the kind of restrictions these specifications can require can be found in the reference documentation under View.onMeasure(int, int) (this reference documentation does a pretty good job of explaining the whole measurement operation as well).
2. Your component's `onMeasure()` method should calculate a measurement width and height which will be required to render the component. It should try to stay within the specifications passed in, although it can choose to exceed them (in this case, the parent can choose what to do, including clipping, scrolling, throwing an exception, or asking the `onMeasure()` to try again, perhaps with different measurement specifications).
3. Once the width and height are calculated, the `setMeasuredDimension(int width, int height)` method must be called with the calculated measurements. Failure to do this will result in an exception being thrown.

A Customized Component Example

The CustomView sample in the API Demos provides an example of a customized component. The custom component is defined in the LabelView class.

The LabelView sample demonstrates a number of different aspects of custom components:

- Extending the View class for a completely custom component.
- Parameterized constructor that takes the view inflation parameters (parameters defined in the XML). Some of these are

passed through to the View superclass, but more importantly, there are some custom attributes defined and used for LabelView.
- Standard public methods of the type you would expect to see for a label component, for example `setText()`, `setTextSize()`, `setTextColor()` and so on.
- An overridden `onMeasure` method to determine and set the rendering size of the component. (Note that in LabelView, the real work is done by a private `measureWidth()` method.)
- An overridden `onDraw()` method to draw the label onto the provided canvas.

You can see some sample usages of the LabelView custom component in custom_view_1.xml from the samples. In particular, you can see a mix of both `android:` namespace parameters and custom `app:` namespace parameters. These `app:` parameters are the custom ones that the LabelView recognizes and works with, and are defined in a styleable inner class inside of the samples R resources definition class.

Compound Components (or Compound Controls)

If you don't want to create a completely customized component, but instead are looking to put together a reusable component that consists of a group of existing controls, then creating a Compound Component (or Compound Control) might fit the bill. In a nutshell, this brings together a number of more atomic controls (or views) into a logical group of items that can be treated as a single thing. For example, a Combo Box can be thought of as a combination of a single line EditText field and an adjacent button with an attached PopupList. If you press the button and select something from the list, it populates the EditText field, but the user can also type something directly into the EditText if they prefer.

In Android, there are actually two other Views readily available to do this: Spinner and AutoCompleteTextView, but regardless, the concept of a Combo Box makes an easy-to-understand example.

To create a Compound Component:

1. The usual starting point is a Layout of some kind, so create a class that extends a Layout. Perhaps in the case of a Combo box we might use a LinearLayout with horizontal orientation. Remember that other layouts can be nested inside, so the compound component can be arbitrarily complex and structured. Note that just like with an Activity, you can use either the declarative (XML-based) approach to creating the contained components, or you can nest them programmatically from your code.
2. In the constructor for the new class, take whatever parameters the superclass expects, and pass them through to the superclass constructor first. Then you can set up the other views to use within your new component; this is where you would create the EditText field and the PopupList. Note that you also might introduce your own attributes and parameters into the XML that can be pulled out and used by your constructor.
3. You can also create listeners for events that your contained views might generate, for example, a listener method for the List Item Click Listener to update the contents of the EditText if a list selection is made.
4. You might also create your own properties with accessors and modifiers, for example, allow the EditText value to be set initially in the component and query for its contents when needed.
5. In the case of extending a Layout, you don't need to override the `onDraw()` and `onMeasure()` methods since the layout will have default behavior that will likely work just fine. However, you can still override them if you need to.
6. You might override other `on...` methods, like `onKeyDown()`, to perhaps choose certain default values from the popup list of a combo box when a certain key is pressed.

To summarize, the use of a Layout as the basis for a Custom Control has a number of advantages, including:

- You can specify the layout using the declarative XML files just like with an activity screen, or you can create views programmatically and nest them into the layout from your code.
- The `onDraw()` and `onMeasure()` methods (plus most of the other `on...` methods) will likely have suitable behavior so you don't have to override them.
- In the end, you can very quickly construct arbitrarily complex compound views and re-use them as if they were a single component.

Examples of Compound Controls

In the API Demos project that comes with the SDK, there are two List examples — Example 4 and Example 6 under Views/Lists demonstrate a SpeechView which extends LinearLayout to make a component for displaying Speech quotes. The corresponding classes in the sample code are `List4.java` and `List6.java`.

Tweaking an Existing Component

There is an even easier option for creating a custom component which is useful in certain circumstances. If there is a component that is already very similar to what you want, you can simply extend that component and just override the behavior that you want to change. You can do all of the things you would do with a fully customized component, but by starting with a more specialized class in the View heirarchy, you can also get a lot of behavior for free that probably does exactly what you want.

For example, the SDK includes a NotePad application in the samples. This demonstrates many aspects of using the Android platform, among them is extending an EditText View to make a lined notepad. This is not a perfect example, and the APIs for doing this might change from this early preview, but it does demonstrate the principles.

If you haven't done so already, import the NotePad sample into Eclipse (or just look at the source using the link provided). In particular look at the definition of `MyEditText` in the NoteEditor.java file.

Some points to note here

1. **The Definition**

   The class is defined with the following line:

   ```
   public static class MyEditText extends EditText
   ```

   - It is defined as an inner class within the `NoteEditor` activity, but it is public so that it could be accessed as `NoteEditor.MyEditText` from outside of the `NoteEditor` class if desired.
   - It is `static`, meaning it does not generate the so-called "synthetic methods" that allow it to access data from the parent class, which in turn means that it really behaves as a separate class rather than something strongly related to `NoteEditor`. This is a cleaner way to create inner classes if they do not need access to state from the outer class, keeps the generated class small, and allows it to be used easily from other classes.
   - It extends `EditText`, which is the View we have chosen to customize in this case. When we are finished, the new class will be able to substitute for a normal `EditText` view.

2. **Class Initialization**

   As always, the super is called first. Furthermore, this is not a default constructor, but a parameterized one. The EditText is created with these parameters when it is inflated from an XML layout file, thus, our constructor needs to both take them and pass them to the superclass constructor as well.

3. **Overridden Methods**

   In this example, there is only one method to be overridden: `onDraw()` — but there could easily be others needed when you create your own custom components.

   For the NotePad sample, overriding the `onDraw()` method allows us to paint the blue lines on the `EditText` view canvas (the canvas is passed into the overridden `onDraw()` method). The super.onDraw() method is called before the method ends. The superclass method should be invoked, but in this case, we do it at the end after we have painted the lines we want to include.

4. **Use the Custom Component**

   We now have our custom component, but how can we use it? In the NotePad example, the custom component is used directly from the declarative layout, so take a look at `note_editor.xml` in the `res/layout` folder.

   ```
   <view xmlns:android="http://schemas.android.com/apk/res/android"
     class="com.android.notepad.NoteEditor$MyEditText"
     id="@+id/note"
     android:layout_width="fill_parent"
     android:layout_height="fill_parent"
     android:background="@android:drawable/empty"
     android:padding="10dip"
     android:scrollbars="vertical"
     android:fadingEdge="vertical" />
   ```

   - The custom component is created as a generic view in the XML, and the class is specified using the full package. Note also that the inner class we defined is referenced using the `NoteEditor$MyEditText` notation which is a standard way to refer to inner classes in the Java programming language.
   - The other attributes and parameters in the definition are the ones passed into the custom component constructor, and then passed through to the EditText constructor, so they are the same parameters that you would use for an

EditText view. Note that it is possible to add your own parameters as well, and we will touch on this again below.

And that's all there is to it. Admittedly this is a simple case, but that's the point — creating custom components is only as complicated as you need it to be.

A more sophisticated component may override even more `on...` methods and introduce some of its own helper methods, substantially customizing its properties and behavior. The only limit is your imagination and what you need the component to do.

Go Forth and Componentize

As you can see, Android offers a sophisticated and powerful component model where just about anything is possible, from simple tweaking of existing Views, to compound controls, to fully customized components. Combining these techniques, you should be able to achieve the exact look you want for your Android application.

Android

Optional APIs in Android

Android is suitable for a wide variety of phones, from high-end smartphones on down. The core Android APIs will be available on every Android phone, but there are a few APIs which have special concerns: the "optional" APIs.

These are "optional" in the sense that a given handset may not support them fully, or even at all. For instance, a given handset may not have GPS or Wi-Fi hardware. In this case, the APIs for accessing these features will still be present, but they may not work in the same way. For instance, the Location API will still exist on devices without GPS, but there may simply be no installed provider, meaning that the API can't be usefully used.

Your application won't have trouble running or linking on a device that doesn't support an API you use, because the classes will be present on the device. However, the implementations may not do anything, or may throw exceptions when you actually try to use them. What exactly each API does on unsupported devices is described in the documentation for that API; you should be sure to code your application to gracefully handle such cases.

Wi-Fi APIs

The Wi-Fi APIs provide a means by which applications can communicate with the lower-level wireless stack that provides Wi-Fi network access. Almost all information from the device supplicant is available, including the connected network's link speed, IP address, negotiation state, and more, plus information about all other available networks. Some of the available interactions include the ability to scan, add, save, terminate and initiate connections.

The Wi-Fi APIs are in the android.net.wifi package.

Location-Based Services

Location-Based Services (LBS) allow software to obtain the phone's current location. This includes location obtained from the Global Positioning System (GPS) satellite constellation, but it's not limited to that. For instance, other location-based systems may come online in the future, and as they do, support for them can be added to this API.

The Location-Based Services are in the android.location package.

Click here for an introduction to the Android LBS APIs.

Media APIs

The Media APIs are used to play media files. This includes both audio (such as playing MP3s or other music files, as well as game sound effects) and video (such as playing a video downloaded over the web.) Support is included for "playing URIs" — that is, streaming media data over the network. Technically the Media APIs are not optional since they'll always be present; however there may be differences in the specific sets of supported codecs across devices.

The Media APIs are in the android.media package.

Click here for an introduction to the Android Media APIs.

3D Graphics with OpenGL

Android's primary user interface framework is a typical widget-oriented class hierarchy. Don't let that fool you, though — sitting underneath that is a very fast 2D and 3D compositing engine, with support for hardware acceleration. The API used to access the 3D capabilities of the platform is the OpenGL ES API. Like the Media APIs, OpenGL is actually not strictly optional, since the API will always be present and will always function. However some devices may not have hardware acceleration, and thus use software rendering, which may affect the performance of your application.

The OpenGL utilities are in the android.opengl package.

Click here for an introduction to the Android OpenGL API.

Android

Writing Efficient Android Code

There's no way around it: Android-powered devices are embedded devices. Modern handsets may be more like small handheld computers than mere phones these days, but even the fastest, highest-end handset doesn't even come close to the capabilities of even a modest desktop system.

That's why it's very important to consider performance when you write Android applications. These systems are not that fast to begin with and they are also constrained by their battery life. This means that there's not a lot of horsepower to spare, so when you write Android code it's important to write it as efficiently as possible.

This page describes a number of things that developers can do to make their Android code run more efficiently. By following the tips on this page, you can help make sure your code runs as efficiently as possible.

Contents

Introduction

There are two basic rules for resource-constrained systems:

- Don't do work that you don't need to do.
- Don't allocate memory if you can avoid it.

All the tips below follow from these two basic tenets.

Some would argue that much of the advice on this page amounts to "premature optimization." While it's true that micro-optimizations sometimes make it harder to develop efficient data structures and algorithms, on embedded devices like handsets you often simply have no choice. For instance, if you bring your assumptions about VM performance on desktop machines to Android, you're quite likely to write code that exhausts system memory. This will bring your application to a crawl — let alone what it will do to other programs running on the system!

That's why these guidelines are important. Android's success depends on the user experience that your applications provide, and that user experience depends in part on whether your code is responsive and snappy, or slow and aggravating. Since all our applications will run on the same devices, we're all in this together, in a way. Think of this document as like the rules of the road you had to learn when you got your driver's license: things run smoothly when everybody follows them, but when you don't, you get your car smashed up.

Before we get down to brass tacks, a brief observation: nearly all issues described below are valid whether or not the VM features a JIT compiler. If I have two methods that accomplish the same thing, and the interpreted execution of foo() is faster than bar(), then the compiled version of foo() will probably be as fast or faster than compiled bar(). It is unwise to rely on a compiler to "save" you and make your code fast enough.

Avoid Creating Objects

Object creation is never free. A generational GC with per-thread allocation pools for temporary objects can make allocation cheaper, but allocating memory is always more expensive than not allocating memory.

If you allocate objects in a user interface loop, you will force a periodic garbage collection, creating little "hiccups" in the user experience.

Thus, you should avoid creating object instances you don't need to. Some examples of things that can help:

- When extracting strings from a set of input data, try to return a substring of the original data, instead of creating a copy. You will create a new String object, but it will share the char[] with the data.
- If you have a method returning a string, and you know that its result will always be appended to a StringBuffer anyway, change your signature and implementation so that the function does the append directly, instead of creating a short-lived temporary object.

A somewhat more radical idea is to slice up multidimensional arrays into parallel single one-dimension arrays:

- An array of ints is a much better than an array of Integers, but this also generalizes to the fact that two parallel arrays of ints are also a **lot** more efficient than an array of (int,int) objects. The same goes for any combination of primitive types.
- If you need to implement a container that stores tuples of (Foo,Bar) objects, try to remember that two parallel Foo[] and Bar[] arrays are generally much better than a single array of custom (Foo,Bar) objects. (The exception to this, of course, is when you're designing an API for other code to access; in those cases, it's usually better to trade correct API design for a small hit in speed. But in your own internal code, you should try and be as efficient as possible.)

Generally speaking, avoid creating short-term temporary objects if you can. Fewer objects created mean less-frequent garbage collection, which has a direct impact on user experience.

Use Native Methods

When processing strings, don't hesitate to use specialty methods like String.indexOf(), String.lastIndexOf(), and their cousins. These are typically implemented in C/C++ code that easily runs 10-100x faster than doing the same thing in a Java loop.

The flip side of that advice is that punching through to a native method is more expensive than calling an interpreted method. Don't use native methods for trivial computation, if you can avoid it.

Prefer Virtual Over Interface

Suppose you have a HashMap object. You can declare it as a HashMap or as a generic Map:

```
Map myMap1 = new HashMap();
HashMap myMap2 = new HashMap();
```

Which is better?

Conventional wisdom says that you should prefer Map, because it allows you to change the underlying implementation to anything that implements the Map interface. Conventional wisdom is correct for conventional programming, but isn't so great for embedded systems. Calling through an interface reference can take 2x longer than a virtual method call through a concrete reference.

If you have chosen a HashMap because it fits what you're doing, there is little value in calling it a Map. Given the availability of IDEs that refactor your code for you, there's not much value in calling it a Map even if you're not sure where the code is headed. (Again, though, public APIs are an exception: a good API usually trumps small performance concerns.)

Prefer Static Over Virtual

If you don't need to access an object's fields, make your method static. It can be called faster, because it doesn't require a virtual method table indirection. It's also good practice, because you can tell from the method signature that calling the method can't alter the object's state.

Avoid Internal Getters/Setters

In native languages like C++ it's common practice to use getters (e.g. `i = getCount()`) instead of accessing the field directly (`i = mCount`). This is an excellent habit for C++, because the compiler can usually inline the access, and if you need to restrict or debug field access you can add the code at any time.

On Android, this is a bad idea. Virtual method calls are expensive, much more so than instance field lookups. It's reasonable to follow common object-oriented programming practices and have getters and setters in the public interface, but within a class you should always access fields directly.

Cache Field Lookups

Accessing object fields is much slower than accessing local variables. Instead of writing:

```
for (int i = 0; i < this.mCount; i++)
      dumpItem(this.mItems[i]);
```

You should write:

```
  int count = this.mCount;
  Item[] items = this.mItems;

  for (int i = 0; i < count; i++)
      dumpItems(items[i]);
```

(We're using an explicit "this" to make it clear that these are member variables.)

A similar guideline is never call a method in the second clause of a "for" statement. For example, the following code will execute the getCount() method once per iteration, which is a huge waste when you could have simply cached the value as an int:

```
for (int i = 0; i < this.getCount(); i++)
    dumpItems(this.getItem(i));
```

It's also usually a good idea to create a local variable if you're going to be accessing an instance field more than once. For example:

```
    protected void drawHorizontalScrollBar(Canvas canvas, int width, int height) {
        if (isHorizontalScrollBarEnabled()) {
            int size = mScrollBar.getSize(false);
            if (size <= 0) {
                size = mScrollBarSize;
            }
            mScrollBar.setBounds(0, height - size, width, height);
            mScrollBar.setParams(
                    computeHorizontalScrollRange(),
                    computeHorizontalScrollOffset(),
                    computeHorizontalScrollExtent(), false);
            mScrollBar.draw(canvas);
        }
    }
```

That's four separate lookups of the member field `mScrollBar`. By caching mScrollBar in a local stack variable, the four member field lookups become four stack variable references, which are much more efficient.

Incidentally, method arguments have the same performance characteristics as local variables.

Declare Constants Final

Consider the following declaration at the top of a class:

```
static int intVal = 42;
static String strVal = "Hello, world!";
```

The compiler generates a class initializer method, called `<clinit>`, that is executed when the class is first used. The method stores the value 42 into `intVal`, and extracts a reference from the classfile string constant table for `strVal`. When these values are referenced later on, they are accessed with field lookups.

We can improve matters with the "final" keyword:

```
static final int intVal = 42;
static final String strVal = "Hello, world!";
```

The class no longer requires a `<clinit>` method, because the constants go into classfile static field initializers, which are handled directly by the VM. Code accessing `intVal` will use the integer value 42 directly, and accesses to `strVal` will use a relatively inexpensive "string constant" instruction instead of a field lookup.

Declaring a method or class "final" does not confer any immediate performance benefits, but it does allow certain optimizations. For example, if the compiler knows that a "getter" method can't be overridden by a sub-class, it can inline the method call.

You can also declare local variables final. However, this has no definitive performance benefits. For local variables, only use "final" if it makes the code clearer (or you have to, e.g. for use in an anonymous inner class).

Use Enhanced For Loop Syntax With Caution

The enhanced for loop (also sometimes known as "for-each" loop) can be used for collections that implement the Iterable interface. With these objects, an iterator is allocated to make interface calls to hasNext() and next(). With an ArrayList, you're better off walking through it directly, but for other collections the enhanced for loop syntax will be equivalent to explicit iterator usage.

Nevertheless, the following code shows an acceptable use of the enhanced for loop:

```
public class Foo {
    int mSplat;
    static Foo mArray[] = new Foo[27];

    public static void zero() {
        int sum = 0;
        for (int i = 0; i < mArray.length; i++) {
            sum += mArray[i].mSplat;
        }
    }

    public static void one() {
        int sum = 0;
        Foo[] localArray = mArray;
        int len = localArray.length;

        for (int i = 0; i < len; i++) {
            sum += localArray[i].mSplat;
        }
    }

    public static void two() {
        int sum = 0;
        for (Foo a: mArray) {
            sum += a.mSplat;
        }
    }
}
```

zero() retrieves the static field twice and gets the array length once for every iteration through the loop.

one() pulls everything out into local variables, avoiding the lookups.

two() uses the enhanced for loop syntax introduced in version 1.5 of the Java programming language. The code generated by the compiler takes care of copying the array reference and the array length to local variables, making it a good choice for walking through all elements of an array. It does generate an extra local load/store in the main loop (apparently preserving "a"), making it a teensy bit slower and 4 bytes longer than one().

To summarize all that a bit more clearly: enhanced for loop syntax performs well with arrays, but be cautious when using it with Iterable objects since there is additional object creation.

Avoid Enums

Enums are very convenient, but unfortunately can be painful when size and speed matter. For example, this:

```
public class Foo {
   public enum Shrubbery { GROUND, CRAWLING, HANGING }
}
```

turns into a 900 byte .class file (Foo$Shrubbery.class). On first use, the class initializer invokes the <init> method on objects representing each of the enumerated values. Each object gets its own static field, and the full set is stored in an array (a static field called "$VALUES"). That's a lot of code and data, just for three integers.

This:

```
Shrubbery shrub = Shrubbery.GROUND;
```

causes a static field lookup. If "GROUND" were a static final int, the compiler would treat it as a known constant and inline it.

The flip side, of course, is that with enums you get nicer APIs and some compile-time value checking. So, the usual trade-off applies: you should by all means use enums for public APIs, but try to avoid them when performance matters.

In some circumstances it can be helpful to get enum integer values through the `ordinal()` method. For example, replace:

```
for (int n = 0; n < list.size(); n++) {
    if (list.items[n].e == MyEnum.VAL_X)
       // do stuff 1
    else if (list.items[n].e == MyEnum.VAL_Y)
       // do stuff 2
}
```

with:

```
   int valX = MyEnum.VAL_X.ordinal();
   int valY = MyEnum.VAL_Y.ordinal();
   int count = list.size();
   MyItem items = list.items();

   for (int  n = 0; n < count; n++)
   {
        int  valItem = items[n].e.ordinal();

        if (valItem == valX)
          // do stuff 1
        else if (valItem == valY)
          // do stuff 2
   }
```

In some cases, this will be faster, though this is not guaranteed.

Use Package Scope with Inner Classes

Consider the following class definition:

```
public class Foo {
    private int mValue;

    public void run() {
        Inner in = new Inner();
        mValue = 27;
        in.stuff();
    }

    private void doStuff(int value) {
        System.out.println("Value is " + value);
    }

    private class Inner {
        void stuff() {
            Foo.this.doStuff(Foo.this.mValue);
        }
    }
}
```

The key things to note here are that we define an inner class (Foo$Inner) that directly accesses a private method and a private instance field in the outer class. This is legal, and the code prints "Value is 27" as expected.

The problem is that Foo$Inner is technically (behind the scenes) a totally separate class, which makes direct access to Foo's private members illegal. To bridge that gap, the compiler generates a couple of synthetic methods:

```
/*package*/ static int Foo.access$100(Foo foo) {
    return foo.mValue;
}
/*package*/ static void Foo.access$200(Foo foo, int value) {
    foo.doStuff(value);
}
```

The inner-class code calls these static methods whenever it needs to access the "mValue" field or invoke the "doStuff" method in the outer class. What this means is that the code above really boils down to a case where you're accessing member fields through accessor methods instead of directly. Earlier we talked about how accessors are slower than direct field accesses, so this is an example of a certain language idiom resulting in an "invisible" performance hit.

We can avoid this problem by declaring fields and methods accessed by inner classes to have package scope, rather than private scope. This runs faster and removes the overhead of the generated methods. (Unfortunately it also means the fields could be accessed directly by other classes in the same package, which runs counter to the standard OO practice of making all fields private. Once again, if you're designing a public API you might want to carefully consider using this optimization.)

Avoid Float

Before the release of the Pentium CPU, it was common for game authors to do as much as possible with integer math. With the Pentium, the floating point math co-processor became a built-in feature, and by interleaving integer and floating-point operations your game would actually go faster than it would with purely integer math. The common practice on desktop systems is to use floating point freely.

Unfortunately, embedded processors frequently do not have hardware floating point support, so all operations on "float" and "double" are performed in software. Some basic floating point operations can take on the order of a millisecond to complete.

Also, even for integers, some chips have hardware multiply but lack hardware divide. In such cases, integer division and modulus operations are performed in software — something to think about if you're designing a hash table or doing lots of math.

Some Sample Performance Numbers

To illustrate some of our ideas, here is a table listing the approximate run times for a few basic actions. Note that these values should NOT be taken as absolute numbers: they are a combination of CPU and wall clock time, and will change as improvements are made to the system. However, it is worth noting how these values apply relative to each other — for example, adding a member variable currently takes about four times as long as adding a local variable.

Action	Time
Add a local variable	1
Add a member variable	4
Call String.length()	5
Call empty static native method	5
Call empty static method	12
Call empty virtual method	12.5
Call empty interface method	15
Call Iterator:next() on a HashMap	165
Call put() on a HashMap	600
Inflate 1 View from XML	22,000
Inflate 1 LinearLayout containing 1 TextView	25,000
Inflate 1 LinearLayout containing 6 View objects	100,000
Inflate 1 LinearLayout containing 6 TextView objects	135,000
Launch an empty activity	3,000,000

Closing Notes

The best way to write good, efficient code for embedded systems is to understand what the code you write really does. If you really want to allocate an iterator, by all means use enhanced for loop syntax on a List; just make it a deliberate choice, not an inadvertent side effect.

Forewarned is forearmed! Know what you're getting into! Insert your favorite maxim here, but always think carefully about what your code is doing, and be on the lookout for ways to speed it up.

Android

Developing Responsive Applications

In this article, we'll cover how Android determines if an application is not responding (hereafter called ANR), the causes of ANR, and guidelines for ensuring that your application is responsive. There are a set of best practices — in addition to writing efficient Android code — that will help ensure that your application's user interface is responsive. But before delving into the details, here's a screenshot of what the dialog box created by Android when an application is not responding looks like:

When Good Applications Go Bad

In Android, application responsiveness is monitored by the Activity Manager and Window Manager system services. Android will display the ANR dialog for a particular application when it detects one of the following conditions:

- No response to an input event (e.g. key press, screen touch) within 5 seconds
- A BroadcastReceiver hasn't finished executing within 10 seconds

Screenshot of ANR dialog box

An ANR dialog displayed to the user.

Avoiding ANR

Given the above definition for ANR, let's examine why this can occur in Android applications and how best to structure your application to avoid ANR.

Android applications normally run entirely on a single (i.e. main) thread. This means that anything your application is doing in the main thread that takes a long time to complete can trigger the ANR dialog because your application is not giving itself a chance to handle the input event or Intent broadcast.

Therefore any method that runs in the main thread should do as little work as possible. In particular, Activities should do as little as possible to set up in key life-cycle methods such as `onCreate()` and `onResume()`. Potentially long running operations such as network or database operations, or computationally expensive calculations such as resizing bitmaps should be done in a child thread (or in the case of databases operations, via an asynchronous request). However, this does not mean that your main thread should block while waiting for the child thread to complete — nor should you call `Thread.wait()` or `Thread.sleep()`. Instead of blocking while waiting for a child thread to complete, your main thread should provide a Handler for child threads to post back to upon completion. Designing your application in this way will allow your main thread to remain responsive to input and thus avoid ANR dialogs caused by the 5 second input event timeout. These same practices should be followed for any other threads that display UI, as they are also subject to the same timeouts.

The specific constraint on IntentReciever execution time emphasizes what they were meant to do: small, discrete amounts of work in the background such as saving a setting or registering a Notification. So as with other methods called in the main thread, applications should avoid potentially long-running operations or calculations in BroadcastReceivers. But instead of doing intensive tasks via child threads (as the life of a BroadcastReceiver is short), your application should start a Service if a potentially long running action needs to be taken in response to an Intent broadcast. As a side note, you should also avoid starting an Activity from an Intent Receiver, as it will spawn a new screen that will steal focus from whatever application the user is currently has running. If your application has something to show the user in response to an Intent broadcast, it should do so using the Notification Manager.

Reinforcing Responsiveness

Generally, 100 to 200ms is the threshold beyond which users will perceive lag (or lack of "snappiness," if you will) in an application. As such, here are some additional tips beyond what you should do to avoid ANR that will help make your application seem responsive to users.

- If your application is doing work in the background in response to user input, show that progress is being made (ProgressBar and ProgressDialog are useful for this).
- For games specifically, do calculations for moves in a child thread.
- If your application has a time-consuming initial setup phase, consider showing a splash screen or rendering the main view as quickly as possible and filling in the information asynchronously. In either case, you should indicate somehow that progress is being made, lest the user perceive that the application is frozen.

Android

Writing Seamless Android Applications

Don't Drop Data

Always keep in mind that Android is a mobile platform. It may seem obvious to say it, but it's important to remember that another Activity (such as the "Incoming Phone Call" app) can pop up over your own Activity at any moment. This will fire the onSaveInstanceState() and onPause() methods, and will likely result in your application being killed.

If the user was editing data in your application when the other Activity appeared, your application will likely lose that data when your application is killed. Unless, of course, you save the work in progress first. The "Android Way" is to do just that: Android applications that accept or edit input should override the onSaveInstanceState() method and save their state in some appropriate fashion. When the user revisits the application, she should be able to retrieve her data.

A classic example of a good use of this behavior is a mail application. If the user was composing an email when another Activity started up, the application should save the in-process email as a draft.

No One Wants to See Your Data Naked

If you wouldn't walk down the street in your underwear, neither should your data. While it's possible to expose certain kinds of application to the world to read, this is usually not the best idea. Exposing raw data requires other applications to understand your data format; if you change that format, you'll break any other applications that aren't similarly updated.

The "Android Way" is to create a ContentProvider to expose your data to other applications via a clean, well-thought-out, and maintainable API. Using a ContentProvider is much like inserting a Java language interface to split up and componentize two tightly-coupled pieces of code. This means you'll be able to modify the internal format of your data without changing the interface exposed by the ContentProvider, and this without affecting other applications.

Don't Interrupt the User When He's Talking

If the user is running an application (such as the Phone application during a call) it's a pretty safe bet he did it on purpose. That's why you should avoid spawning activities except in direct response to user input from the current Activity.

That is, don't call startActivity() from BroadcastReceivers or Services running in the background. Doing so will interrupt whatever application is currently running, and result in an annoyed user. Perhaps even worse, your Activity may become a "keystroke bandit" and receive some of the input the user was in the middle of providing to the previous Activity. Depending on what your application does, this could be bad news.

Instead of spawning Activity UIs directly from the background, you should instead use the NotificationManager to set Notifications. These will appear in the status bar, and the user can then click on them at his leisure, to see what your application has to show him.

(Note that all this doesn't apply to cases where your own Activity is already in the foreground: in that case, the user expects to see your next Activity in response to input.)

Got a Lot to Do? Take it to a Thread

If your application needs to perform some expensive or long-running computation, you should probably move it to a thread. This will prevent the dreaded "Application Not Responding" dialog from being displayed to the user, with the ultimate result being the fiery demise of your application.

By default, all code in an Activity as well as all its Views run in the same thread. This is the same thread that also handles UI events. For example, when the user presses a key, a key-down event is added to the Activity's main thread's queue. The event handler system needs to dequeue and handle that event quickly; if it doesn't, the system concludes after a few seconds that the application is hung and offers to kill it for the user.

If you have long-running code, running it inline in your Activity will run it on the event handler thread, effectively blocking the event handler. This will delay input processing, and result in the ANR dialogs. To avoid this, move your computations to a thread; click here to learn how.

Avoid Monster Activity Screens

Any application worth using will probably have several different screens. When partitioning your UI, be sure to use the Activity class liberally.

Depending on your development background, you may interpret an Activity as similar to something like a Java Applet, in that it is the entry point for your application. However, that's not quite accurate: where an Applet subclass is the single entry point for a Java Applet, an Activity should be thought of as one of potentially several entry points to your application. The only difference between your "main" Activity and any others you might have is that the "main" one just happens to be the only one that expressed an interest in the "android.intent.action.MAIN" action in your AndroidManifest..xml file.

So, when designing your application, think of your application as a federation of Activity objects. This will make your code a lot more maintainable in the long run, and as a nice side effect also plays nicely with Android's application history and "backstack" model.

Extend Themes

When it comes to the look-and-feel of the user interface, it's important to blend in nicely. Users are jarred by applications which contrast with the user interface they've come to expect. When designing your UIs, you should try and avoid rolling your own as much as possible. Instead, use a Theme. You can override or extend those parts of the theme that you need to, but at least you're starting from the same UI base as all the other applications. For all the details, click here.

Make Being Flexible part of your Resolutions

Different Android-powered devices will sport different resolutions. Some will even be able to change resolutions on the fly, such as by switching to landscape mode. It's important to make sure your layouts and drawables are flexible.

Fortunately, this is very easy to do. Check out Implementing a User Interface for the full details, but in brief what you must do is provide different versions of your artwork (if you use any) for the key resolutions, and then design your layout to accommodate various dimensions. (For example, avoid using hard-coded positions and instead use relative layouts.) If you do that much, the system handles the rest, and your application looks great on any device.

Assume the Network is Slow

Android devices will come with a variety of network-connectivity options. All will have some data-access provision, though some will be faster than others. The lowest common denominator, however, is GPRS, the non-3G data service for GSM networks. Even 3G-capable devices will spend lots of time on non-3G networks, so slow networks will remain a reality for quite a long time to come.

That's why you should always code your applications to minimize network accesses and bandwidth. You can't assume the network is fast, so you should always plan for it to be slow. If your users happen to be on faster networks, then that's great — their experience will only improve. You want to avoid the inverse case though: applications that are usable some of the time, but frustratingly slow the rest based on where the user is at any given moment are likely to be unpopular.

One potential gotcha here is that it's very easy to fall into this trap if you're using the emulator, since the emulator uses your desktop computer's network connection. That's almost guaranteed to be much faster than a cell network, so you'll want to change the settings on the emulator that simulate slower network speeds. You can do this in Eclipse, in the "Emulator Settings" tab of your launch configuration or via a command line option when starting the emulator.

Different Keystrokes for Different Folks

Android will support a variety of handset form-factors. That's a fancy way of saying that some Android devices will have full "QWERTY" keyboards, while others will have 40-key, 12-key, or even other key configurations. Similarly, some devices will have touch-screens, but many won't.

When building your applications, keep that in mind. Don't make assumptions about specific keyboard layouts -- unless, of course, you're really interested in restricting your application so that it can only be used on those devices.

Don't Assault the Battery

A wireless device isn't very wireless if it's constantly plugged into the wall. Handheld devices are battery-powered, and the longer we can make that battery last on a charge, the happier everyone is -- especially the user. Two of the biggest consumers of battery power are the processor, and the radio; that's why it's important to write your applications to do as little work as possible, and use the network as infrequently as possible.

Minimizing the amount of processor time your application uses really comes down to writing efficient code. To minimize the power drain from using the radio, be sure to handle error conditions gracefully, and only fetch what you need. For example, don't constantly retry a network operation if one failed. If it failed once, it's likely because the user has no reception, so it's probably going to fail again if you try right away; all you'll do is waste battery power.

Users are pretty smart: if your program is power-hungry, you can count on them noticing. The only thing you can be sure of at that point is that your program won't stay installed very long.

Android

Location-based Service APIs

The Android SDK includes two packages that provide Android's primary support for building location-based services: android.location and com.google.android.maps. Please read on below for a brief introduction to each package.

android.location

This package contains several classes related to location services in the Android platform. Most importantly, it introduces the LocationManager service, which provides an API to determine location and bearing if the underlying device (if it supports the service). The LocationManager should **not** be instantiated directly; rather, a handle to it should be retrieved via getSystemService(Context.LOCATION_SERVICE).

Once your application has a handle to the LocationManager, your application will be able to do three things:

- Query for the list of all LocationProviders known to the LocationManager for its last known location.
- Register/unregister for periodic updates of current location from a LocationProvider (specified either by Criteria or name).
- Register/unregister for a given Intent to be fired if the device comes within a given proximity (specified by radius in meters) of a given lat/long.

However, during initial development, you may not have access to real data from a real location provider (Network or GPS). So it may be necessary to spoof some data for your application, with some mock location data.

Note: If you've used mock LocationProviders in previous versions of the SDK (m3/m5), you can no longer provide canned LocationProviders in the /system/etc/location directory. These directories will be wiped during boot-up. Please follow the new procedures below.

Providing Mock Location Data

When testing your application on the Android emulator, there are a couple different ways to send it some spoof location data: with the DDMS tool or the "geo" command.

Using DDMS

With the DDMS tool, you can simulate location data a few different ways:

- Manually send individual longitude/latitude coordinates to the device.
- Use a GPX file describing a route for playback to the device.
- Use a KML file describing individual placemarks for sequenced playback to the device.

For more information on using DDMS to spoof location data, see the Using DDMS guide.

Using the "geo" command

Launch your application in the Android emulator and open a terminal/console in your SDK's `/tools` directory. Now you can use:

- `geo fix` to send a fixed geo-location.

  This command accepts a longitude and latitude in decimal degrees, and an optional altitude in meters. For example:

  ```
  geo fix -121.45356 46.51119 4392
  ```

- `geo nmea` to send an NMEA 0183 sentence.

  This command accepts a single NMEA sentence of type '$GPGGA' (fix data) or '$GPRMC' (transit data). For example:

```
geo nmea $GPRMC,081836,A,3751.65,S,14507.36,E,000.0,360.0,130998,011.3,E*62
```

com.google.android.maps

This package introduces a number of classes related to rendering, controlling, and overlaying customized information on your own Google Mapified Activity. The most important of which is the MapView class, which automagically draws you a basic Google Map when you add a MapView to your layout. Note that, if you want to do so, then your Activity that handles the MapView must extend MapActivity.

Also note that you must obtain a MapView API Key from the Google Maps service, before your MapView can load maps data. For more information, see Obtaining a MapView API Key.

Once you've created a MapView, you'll probably want to use getController() to retrieve a MapController, for controlling and animating the map, and ItemizedOverlay to draw Overlays and other information on the Map.

This is not a standard package in the Android library. In order to use it, you must add the following node to your Android Manifest file, as a child of the `<application>` element:

```
<uses-library android:name="com.google.android.maps" />
```

Android

Android Media APIs

The Android Platform boasts strong multimedia capabilities, as you would expect from a modern embedded operating system. Using the multimedia capabilities is fairly straightforward, going through the same intents and activities mechanism that the rest of Android uses.

Multimedia Capabilities

The Android platform is capable of playing both audio and video media. It is also capable of playing media contained in the resources for an application, or a standalone file in the filesystem, or even streaming media over a data connection. Playback is achieved through the android.media.MediaPlayer class.

The Android platform can also record audio. Video recording capabilities are coming in the future. This is achieved through the android.media.MediaRecorder class. While the emulator obviously doesn't have hardware to capture and record audio and video, the eventual devices will have these capabilities, and the APIs are there already to develop against.

Playing Media Resources

Media can be played from anywhere: a raw resource, a file from the system, or from an available network (URL).

Note: You can only send audio files to the standard output device; currently, that is the device speaker or a Bluetooth headset. You cannot currently play sound files in the conversation audio.

Playing a Raw Resource

Perhaps the most common thing to want to do is play back media (notably sound) within your own applications. Doing this is easy:

1. Put the sound (or other media resource) file into the `res/raw` folder of your project, where the Eclipse plugin (or aapt) will find it and make it into a resource that can be referenced from your R class
2. Create an instance of `MediaPlayer`, referencing that resource using MediaPlayer.create, and then call start() on the instance:

```
    MediaPlayer mp = MediaPlayer.create(context, R.raw.sound_file_1);
    mp.start();
```

To stop playback, call stop(). If you wish to later replay the media, then you must reset() and prepare() the MediaPlayer object before calling start() again. (`create()` calls `prepare()` the first time.)

To pause playback, call pause(). Resume playback from where you paused with start().

Playing a File

You can play back media files from the filesystem or a web URL:

1. Create an instance of the `MediaPlayer` using `new`
2. Call setDataSource() with a String containing the path (local filesystem or URL) to the file you want to play
3. First prepare() then start() on the instance:

```
    MediaPlayer mp = new MediaPlayer();
    mp.setDataSource(PATH_TO_FILE);
    mp.prepare();
    mp.start();
```

stop() and pause() work the same as discussed above.

Note: It is possible that `mp` could be null, so good code should `null` check after the `new`. Also, `IllegalArgumentException` and `IOException` either need to be caught or passed on when using `setDataSource()`, since the file you are referencing may not exist.

Note: If you're passing a URL to an online media file, the file must be capable of progressive download.

Recording Media Resources

Recording media is a little more involved than playing it back, as you would probably expect, but it is still fairly simple. There is just a little more set up to do

1. Create a new instance of android.media.MediaRecorder using `new`
2. Create a new instance of android.content.ContentValues and put in some standard properties like `TITLE`, `TIMESTAMP`, and the all important `MIME_TYPE`
3. Create a file path for the data to go to (you can use android.content.ContentResolver to create an entry in the Content database and get it to assign a path automatically which you can then use)
4. Set the audio source using MediaRecorder.setAudioSource(). You will probably want to use `MediaRecorder.AudioSource.MIC`
5. Set output file format using MediaRecorder.setOutputFormat()
6. Set the audio encoder using MediaRecorder.setAudioEncoder()
7. Finally, prepare() and start() the recording. stop() and release() when you are done

Here is a code example that will hopefully help fill in the gaps:

Start Recording

```
    recorder = new MediaRecorder();
    ContentValues values = new ContentValues(3);

    values.put(MediaStore.MediaColumns.TITLE, SOME_NAME_HERE);
    values.put(MediaStore.MediaColumns.TIMESTAMP, System.currentTimeMillis());
    values.put(MediaStore.MediaColumns.MIME_TYPE, recorder.getMimeContentType());

    ContentResolver contentResolver = new ContentResolver();

    Uri base = MediaStore.Audio.INTERNAL_CONTENT_URI;
    Uri newUri = contentResolver.insert(base, values);

    if (newUri == null) {
        // need to handle exception here - we were not able to create a new
        // content entry
    }

    String path = contentResolver.getDataFilePath(newUri);

    // could use setPreviewDisplay() to display a preview to suitable View here

    recorder.setAudioSource(MediaRecorder.AudioSource.MIC);
    recorder.setOutputFormat(MediaRecorder.OutputFormat.THREE_GPP);
    recorder.setAudioEncoder(MediaRecorder.AudioEncoder.AMR_NB);
    recorder.setOutputFile(path);

    recorder.prepare();
    recorder.start();
```

Stop Recording

```
    recorder.stop();
    recorder.release();
```

Android

OpenGL in Android

Android includes support for 3D hardware acceleration. This functionality is accessed via the OpenGL API — specifically, the OpenGL ES API.

OpenGL ES is a flavor of the OpenGL specification intended for embedded devices. Versions of OpenGL ES are loosely peered to versions of the primary OpenGL standard. Android currently supports OpenGL ES 1.0, which corresponds to OpenGL 1.3. So, if the application you have in mind is possible with OpenGL 1.3 on a desktop system, it should be possible on Android.

The specific API provided by Android is similar to the J2ME JSR239 OpenGL ES API. However, it may not be identical, so watch out for deviations.

Using the API

Here's how to use the API at an extremely high level:

1. Write a custom View subclass.
2. Obtain a handle to an OpenGLContext, which provides access to the OpenGL functionality.
3. In your View's onDraw() method, get a handle to a GL object, and use its methods to perform GL operations.

For an example of this usage model (based on the classic GL ColorCube), see com.android.samples.graphics.GLView1.java in the ApiDemos sample code project. A slightly more sophisticated version showing how to use it with threads can be found in com.android.samples.graphics.GLSurfaceViewActivity.java.

Writing a summary of how to actually write 3D applications using OpenGL is beyond the scope of this text and is left as an exercise for the reader.

Links to Additional Information

Information about OpenGL ES can be found at http://www.khronos.org/opengles/.

Information specifically about OpenGL ES 1.0 (including a detailed specification) can be found at http://www.khronos.org/opengles/1_X/.

The documentation for the Android OpenGL ES implementations are also available.

Finally, note that though Android does include some basic support for OpenGL ES 1.1, the support is **not complete**, and should not be relied upon at this time.

Android

Obtaining a MapView API Key

Build 110632-110632 - 22 Sep 2008 13:34

MapView is a very useful class that lets you easily integrate Google Maps into your application. It provides built-in map downloading, rendering, and caching, as well as a variety of display options and controls. It provides a wrapper around the Google Maps API that lets your application request and manipulate Google Maps data through class methods, and it lets you work with Maps data as you would other types of Views.

Because MapView gives you access to Google Maps data, you need to register your application with the Google Maps service and agree to the applicable Terms of Service, before your MapView will be able to obtain data from Google Maps. This will apply whether you are developing your application on the emulator or preparing your application for deployment to mobile devices.

Registering your application is simple, and has two parts:

1. Registering a public key fingerprint from the certificate that you will use to sign the .apk. The registration service then provides you a Maps API Key that is associated with your application's signer certificate.
2. Adding the Maps API Key to a special attribute of the MapView element — `android:apiKey`. You can use the same Maps API Key for any MapView in any application, provided that the application's .apk is signed with the certificate whose fingerprint you registered with the service.

Once you have registered your application as described above, your MapView will be able to retrieve data from the Google Maps servers.

The MapView registration service is not yet active and Google Maps is not yet enforcing the Maps API Key requirement. The registration service will be activated soon, so that MapViews in any application deployed to a mobile device will require registration and a valid Maps API Key.
As soon as the registration service becomes available, this page (http://code.google.com/android/toolbox/apis/mapkey.html) will be updated with details about how and where to register and how to add your Maps API Key to your application.
In the meantime, you can continue developing your MapView without registration, provided that you:

a. Add the attribute "android:apiKey" to the MapView element in your layout XML, with any value. Or
b. Include an arbitrary string in the `apikey` parameter of the MapView constructor, if creating the MapView programmatically.

When the Maps API Key checking is activated in the service, any MapViews that do not have a properly registered apiKey will stop working. The map data (tile images) of the MapView will never load (even if the device is on the network). In this case, go to the page linked above and read about how to register your certificate fingerprint and obtain a Maps API Key.