Android

Developing Android Applications

You can develop Android applications with the same high-quality tools you use to develop Java applications. The Android core libraries provide the functionality needed to build some amazingly rich mobile applications, and the Android development tools make running, debugging, and testing your applications a snap.

This section explains the ins and outs of developing Android applications. It outlines the philosophy behind the system and then describes each of the key subsystems in detail. After reading this section, you'll have the knowledge and confidence to begin writing that real-world Android app you have in mind.

Before reading this section you should read the Getting Started Guide, which helps you get up and running with the Android SDK and shows you how to build a basic app. This section builds on the information in the Getting Started section.

Here's the content you'll find in this section:

Implementing a UI

Explains how to construct and interact with user interfaces for Android applications. After reading this page you'll have a solid understanding of how Android layouts are built, how they operate at runtime, and how you can make them pretty.

Building Blocks

Detailed descriptions of Android components. Covers the ins and outs of the components summarized in Anatomy of an Android App, plus more. This section goes into detail on each of the key Android components (Intents, Activities, Views, and events.)

Storing and Retrieving Data

How to read and write data to the various storage mechanisms provided by Android, and to network services. There are several different ways to read and write data from an Android application, each aimed at different needs. This page describes them all and explains how to pick the right one for your needs.

Security Model

Gaining access to secure system resources and features, and declaring permissions to control access to your own secure features. Permissions control whether a given application is able to access piece of functionality provided by another application (for example, which applications can dial the phone). This page describes how permissions work and how to request permissions as well as define your own.

Resources and i18n

Detailed descriptions of Android's application-resource management system, including how it's used for internationalization and localization. "Resources" are application assets (such as images, localized strings, and XML layouts) that need to be resolved at runtime. This page describes how Android resolves which resource to load from a selection of them, as well as how to create and use resources.

Build 110632-110632 - 22 Sep 2008 13:34

Android

Implementing a User Interface

This section describes the basics of how to implement the user interface of an Android screen. It covers the basic elements that make up a screen, how to define a screen in XML and load it in your code, and various other tasks you'll need to handle for your user interface.

Topics

Build 110632-110632 - 22 Sep 2008 13:34

Android

Android Building Blocks

You can think of an Android application as a collection of components, of various kinds. These components are for the most part quite loosely coupled, to the degree where you can accurately describe them as a federation of components rather than a single cohesive application.

Generally, these components all run in the same system process. It's possible (and quite common) to create multiple threads within that process, and it's also possible to create completely separate child processes if you need to. Such cases are pretty uncommon though, because Android tries very hard to make processes transparent to your code.

These are the most important parts of the Android APIs:

AndroidManifest.xml

The AndroidManifest.xml file is the control file that tells the system what to do with all the top-level components (specifically activities, services, intent receivers, and content providers described below) you've created. For instance, this is the "glue" that actually specifies which Intents your Activities receive.

Activities

An Activity is, fundamentally, an object that has a life cycle. An Activity is a chunk of code that does some work; if necessary, that work can include displaying a UI to the user. It doesn't have to, though - some Activities never display UIs. Typically, you'll designate one of your application's Activities as the entry point to your application.

Views

A View is an object that knows how to draw itself to the screen. Android user interfaces are comprised of trees of Views. If you want to perform some custom graphical technique (as you might if you're writing a game, or building some unusual new user interface widget) then you'd create a View.

Intents

An Intent is a simple message object that represents an "intention" to do something. For example, if your application wants to display a web page, it expresses its "Intent" to view the URI by creating an Intent instance and handing it off to the system. The system locates some other piece of code (in this case, the Browser) that knows how to handle that Intent, and runs it. Intents can also be used to broadcast interesting events (such as a notification) system-wide.

Services

A Service is a body of code that runs in the background. It can run in its own process, or in the context of another application's process, depending on its needs. Other components "bind" to a Service and invoke methods on it via remote procedure calls. An example of a Service is a media player; even when the user quits the media-selection UI, she probably still intends for her music to keep playing. A Service keeps the music going even when the UI has completed.

Notifications

A Notification is a small icon that appears in the status bar. Users can interact with this icon to receive information. The most well-known notifications are SMS messages, call history, and voicemail, but applications can create their own. Notifications are the strongly-preferred mechanism for alerting the user of something that needs their attention.

ContentProviders

A ContentProvider is a data storehouse that provides access to data on the device; the classic example is the ContentProvider that's used to access the user's list of contacts. Your application can access data that other applications have exposed via a ContentProvider, and you can also define your own ContentProviders to expose data of your own.

 Build 110632-110632 - 22 Sep 2008 13:34

Android

Storing, Retrieving and Exposing Data

A typical desktop operating system provides a common file system that any application can use to store and read files that can be read by other applications (perhaps with some access control settings). Android uses a different system: on Android, all application data (including files) are private to that application. However, Android also provides a standard way for an application to expose its private data to other applications. This section describes the many ways that an application can store and retrieve data, expose its data to other applications, and also how you can request data from other applications that expose their data.

Android provides the following mechanisms for storing and retrieving data:

Preferences

A lightweight mechanism to store and retrieve key/value pairs of primitive data types. This is typically used to store application preferences.

Files

You can store your files on the device or on a removable storage medium. By default, other applications cannot access these files.

Databases

The Android APIs contain support for SQLite. Your application can create and use a private SQLite database. Each database is private to the package that creates it.

Content Providers

A content provider is a optional component of an application that exposes read/write access to an application's private data, subject to whatever restrictions it wants to impose. Content providers implement a standard request syntax for data, and a standard access mechanism for the returned data. Android supplies a number of content providers for standard data types, such as personal contacts.

Network

Don't forget that you can also use the network to store and retrieve data.

Build 110632-110632 - 22 Sep 2008 13:34

Android

Security and Permissions in Android

Android is a multi-process system, where each application (and parts of the system) runs in its own process. Most security between applications and the system is enforced at the process level through standard Linux facilities, such as user and group IDs that are assigned to applications. Additional finer-grained security features are provided through a "permission" mechanism that enforces restrictions on the specific operations that a particular process can perform, and per-URI permissions for granting ad-hoc access to specific pieces of data.

Contents

Security Architecture

A central design point of the Android security architecture is that no application, by default, has permission to perform any operations that would adversely impact other applications, the operating system, or the user. This includes reading or writing the user's private data (such as contacts or e-mails), reading or writing another application's files, performing network access, keeping the device awake, etc.

An application's process is a secure sandbox. It can't disrupt other applications, except by explicitly declaring the *permissions* it needs for additional capabilities not provided by the basic sandbox. These permissions it requests can be handled by the operating in various ways, typically by automatically allowing or disallowing based on certificates or by prompting the user. The permissions required by an application are declared statically in that application, so they can be known up-front at install time and will not change after that.

Application Signing

All Android applications (.apk files) must be signed with a certificate whose private key is held by their developer. This certificate identifies the author of the application. The certificate does *not* need to be signed by a certificate authority: it is perfectly allowable, and typical, for Android applications to use self-signed certificates. The certificate is used only to establish trust relationships between applications, not for wholesale control over whether an application can be installed. The most significant ways that signatures impact security is by determining who can access signature-based permissions and who can share user IDs.

User IDs and File Access

Each Android package (.apk) file installed on the device is given its own unique Linux user ID, creating a sandbox for it and

preventing it from touching other applications (or other applications from touching it). This user ID is assigned to it when the application is installed on the device, and remains constant for the duration of its life on that device.

Because security enforcement happens at the process level, the code of any two packages can not normally run in the same process, since they need to run as different Linux users. You can use the sharedUserId attribute in the `AndroidManifest.xml`'s manifest tag of each package to have them assigned the same user ID. By doing this, for purposes of security the two packages are then treated as being the same application, with the same user ID and file permissions. Note that in order to retain security, only two applications signed with the same signature (and requesting the same sharedUserId) will be given the same user ID.

Any files created by an application will be assigned that application's user ID, and not normally accessible to other packages. When creating a new file with getSharedPreferences(String, int), openFileOutput(String, int), or openOrCreateDatabase(String, int, SQLiteDatabase.CursorFactory), you can use the MODE_WORLD_READABLE and/or MODE_WORLD_WRITEABLE flags to allow any other package to read/write the file. When setting these flags, the file is still owned by your application, but its global read and/or write permissions have been set appropriately so any other application can see it.

Using Permissions

A basic Android application has no permissions associated with it, meaning it can not do anything that would adversely impact the user experience or any data on the device. To make use of protected features of the device, you must include in your `AndroidManifest.xml` one or more `<uses-permission>` tags declaring the permissions that your application needs.

For example, an application that needs to monitor incoming SMS messages would specify:

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.android.app.myapp" >

    <uses-permission android:name="android.permission.RECEIVE_SMS" />

</manifest>
```

At application install time, permissions requested by the application are granted to it by the package installer, based on checks against the signatures of the applications declaring those permissions and/or interaction with the user. *No* checks with the user are done while an application is running: it either was granted a particular permission when installed, and can use that feature as desired, or the permission was not granted and any attempt to use the feature will fail without prompting the user.

Often times a permission failure will result in a SecurityException being thrown back to the application. However, this is not guaranteed to occur everywhere. For example, the sendBroadcast(Intent) method checks permissions as data is being delivered to each receiver, after the method call has returned, so you will not receive an exception if there are permission failures. In almost all cases, however, a permission failure will be printed to the system log.

The permissions provided by the Android system can be found at Manifest.permission. Any application may also define and enforce its own permissions, so this is not a comprehensive list of all possible permissions.

A particular permission may be enforced at a number of places during your program's operation:

- At the time of a call into the system, to prevent an application from executing certain functions.
- When starting an activity, to prevent applications from launching activities of other applications.
- Both sending and receiving broadcasts, to control who can receive your broadcast or who can send a broadcast to you.
- When accessing and operating on a content provider.
- Binding or starting a service.

Declaring and Enforcing Permissions

To enforce your own permissions, you must first declare them in your `AndroidManifest.xml` using one or more `<permission>` tags.

For example, an application that wants to control who can start one of its activities could declare a permission for this operation as follows:

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.me.app.myapp" >

    <permission android:name="com.me.app.myapp.permission.DEADLY_ACTIVITY"
        android:label="@string/permlab_deadlyActivity"
        android:description="@string/permdesc_deadlyActivity"
        android:permissionGroup="android.permission-group.COST_MONEY"
        android:protectionLevel="dangerous" />

</manifest>
```

The <protectionLevel> attribute is required, telling the system how the user is to be informed of applications requiring the permission, or who is allowed to hold that permission, as described in the linked documentation.

The <permissionGroup> attribute is optional, and only used to help the system display permissions to the user. You will usually want to set this to either a standard system group (listed in android.Manifest.permission_group) or in more rare cases to one defined by yourself. It is preferred to use an existing group, as this simplifies the permission UI shown to the user.

Note that both a label and description should be supplied for the permission. These are string resources that can be displayed to the user when they are viewing a list of permissions (`android:label`) or details on a single permission (`android:description`). The label should be short, a few words describing the key piece of functionality the permission is protecting. The description should be a couple sentences describing what the permission allows a holder to do. Our convention for the description is two sentences, the first describing the permission, the second warning the user of what bad things can happen if an application is granted the permission.

Here is an example of a label and description for the CALL_PHONE permission:

```
    <string name="permlab_callPhone">directly call phone numbers</string>
    <string name="permdesc_callPhone">Allows the application to call
        phone numbers without your intervention. Malicious applications may
        cause unexpected calls on your phone bill. Note that this does not
        allow the application to call emergency numbers.</string>
```

You can look at the permissions currently defined in the system with the shell command `adb shell pm list permissions`. In particular, the '-s' option displays the permissions in a form roughly similar to how the user will see them:

```
$ adb shell pm list permissions -s
All Permissions:

Network communication: view Wi-Fi state, create Bluetooth connections, full
Internet access, view network state

Your location: access extra location provider commands, fine (GPS) location,
mock location sources for testing, coarse (network-based) location

Services that cost you money: send SMS messages, directly call phone numbers

...
```

Enforcing Permissions in AndroidManifest.xml

High-level permissions restricting access to entire components of the system or application can be applied through your `AndroidManifest.xml`. All that this requires is including an android:permission attribute on the desired component, naming the permission that will be used to control access to it.

Activity permissions (applied to the <activity> tag) restrict who can start the associated activity. The permission is checked during Context.startActivity() and Activity.startActivityForResult(); if the caller does not have the required permission then SecurityException is thrown from the call.

Service permissions (applied to the <service> tag) restrict who can start or bind to the associated service. The permission is checked during Context.startService(), Context.stopService() and Context.bindService(); if the caller does not have the required permission then SecurityException is thrown from the call.

BroadcastReceiver permissions (applied to the <receiver> tag) restrict who can send broadcasts to the associated receiver. The permission is checked *after* Context.sendBroadcast() returns, as the system tries to deliver the submitted broadcast to the

given receiver. As a result, a permission failure will not result in an exception being thrown back to the caller; it will just not deliver the intent. In the same way, a permission can be supplied to Context.registerReceiver() to control who can broadcast to a programmatically registered receiver. Going the other way, a permission can be supplied when calling Context.sendBroadcast() to restrict which BroadcastReceiver objects are allowed to receive the broadcast (see below).

ContentProvider permissions (applied to the <provider> tag) restrict who can access the data in a ContentProvider. (Content providers have an important additional security facility available to them called URI permissions which is described later.) Unlike the other components, there are two separate permission attributes you can set: android:readPermission restricts who can read from the provider, and android:writePermission restricts who can write to it. Note that if a provider is protected with both a read and write permission, holding only the write permission does not mean you can read from a provider. The permissions are checked when you first retrieve a provider (if you don't have either permission, a SecurityException will be thrown), and as you perform operations on the provider. Using ContentResolver.query() requires holding the read permission; using ContentResolver.insert(), ContentResolver.update(), ContentResolver.delete() requires the write permission. In all of these cases, not holding the required permission results in a SecurityException being thrown from the call.

Enforcing Permissions when Sending Broadcasts

In addition to the permission enforcing who can send Intents to a registered BroadcastReceiver (as described above), you can also specify a required permission when sending a broadcast. By calling Context.sendBroadcast() with a permission string, you require that a receiver's application must hold that permission in order to receive your broadcast.

Note that both a receiver and a broadcaster can require a permission. When this happens, both permission checks must pass for the Intent to be delivered to the associated target.

Other Permission Enforcement

Arbitrarily fine-grained permissions can be enforced at any call into a service. This is accomplished with the Context.checkCallingPermission() method. Call with a desired permission string and it will return an integer indicating whether that permission has been granted to the current calling process. Note that this can only be used when you are executing a call coming in from another process, usually through an IDL interface published from a service or in some other way given to another process.

There are a number of other useful ways to check permissions. If you have the pid of another process, you can use the Context method Context.checkPermission(String, int, int) to check a permission against that pid. If you have the package name of another application, you can use the direct PackageManager method PackageManager.checkPermission(String, String) to find out whether that particular package has been granted a specific permission.

URI Permissions

The standard permission system described so far is often not sufficient when used with content providers. A content provider may want to protect itself with read and write permissions, while its direct clients also need to hand specific URIs to other applications for them to operate on. A typical example is attachments in a mail application. Access to the mail should be protected by permissions, since this is sensitive user data. However, if a URI to an image attachment is given to an image viewer, that image viewer will not have permission to open the attachment since it has no reason to hold a permission to access all e-mail.

The solution to this problem is per-URI permissions: when starting an activity or returning a result to an activity, the caller can set Intent.FLAG_GRANT_READ_URI_PERMISSION and/or Intent.FLAG_GRANT_WRITE_URI_PERMISSION. This grants the receiving activity permission access the specific data URI in the Intent, regardless of whether it has any permission to access data in the content provider corresponding to the Intent.

This mechanism allows a common capability-style model where user interaction (opening an attachment, selecting a contact from a list, etc) drives ad-hoc granting of fine-grained permission. This can be a key facility for reducing the permissions needed by applications to only those directly related to their behavior.

The granting of fine-grained URI permissions does, however, require some cooperation with the content provider holding those URIs. It is strongly recommended that content providers implement this facility, and declare that they support it through the android:grantUriPermissions attribute or <grant-uri-permissions> tag.

More information can be found in the Context.grantUriPermission(), Context.revokeUriPermission(), and Context.checkUriPermission() methods.

Build 110632-110632 - 22 Sep 2008 13:34

Android

Resources and Internationalization

Resources are external files (that is, non-code files) that are used by your code and compiled into your application at build time. Android supports a number of different kinds of resource files, including XML, PNG, and JPEG files. The XML files have very different formats depending on what they describe. This document describes what kinds of files are supported, and the syntax or format of each.

Resources are externalized from source code, and XML files are compiled into a binary, fast loading format for efficiency reasons. Strings, likewise are compressed into a more efficient storage form. It is for these reasons that we have these different resource types in the Android platform.

This document contains the following sections:

This is a fairly technically dense document, and together with the Resource Reference document, they cover a lot of information about resources. It is not necessary to know this document by heart to use Android, but rather to know that the information is here when you need it.

Resources

This topic includes a terminology list associated with resources, and a series of examples of using resources in code. For a complete guide to the supported Android resource types, see Resources.

The Android resource system keeps track of all non-code assets associated with an application. You use the Resources class to access your application's resources; the Resources instance associated with your application can generally be found through Context.getResources().

An application's resources are compiled into the application binary at build time for you by the build system. To use a resource, you must install it correctly in the source tree and build your application. As part of the build process, symbols for each of the resources are generated that you can use in your source code -- this allows the compiler to verify that your application code matches up with the resources you defined.

The rest of this section is organized as a tutorial on how to use resources in an application.

Creating Resources

Android supports string, bitmap, and many other types of resource. The syntax and format of each, and where they're stored, depends upon the type of object. In general, though, you create resources from three types of files: XML files (everything but bitmaps and raw), bitmap files(for images) and Raw files (anything else, for example sound files, etc.). In fact, there are two different types of XML file as well, those that get compiled as-is into the package, and those that are used to generate resources by aapt. Here is a list of each resource type, the format of the file, a description of the file, and details of any XML files.

You will create and store your resource files under the appropriate subdirectory under the `res/` directory in your project. Android has a resource compiler (aapt) that compiles resources according to which subfolder they are in, and the format of the file. Here is a list of the file types for each resource. See the resource reference for descriptions of each type of object, the syntax, and the format or syntax of the containing file.

Directory	Resource Types
`res/anim/`	XML files that are compiled into frame by frame animation or tweened animation objects
`res/drawable/`	.png, .9.png, .jpg files that are compiled into the following Drawable resource subtypes: To get a resource of this type, use `Resource.getDrawable(id)` • bitmap files • 9-patches (resizable bitmaps)
`res/layout/`	XML files that are compiled into screen layouts (or part of a screen). See layouts
`res/values/`	XML files that can be compiled into many kinds of resource. **Note:** unlike the other res/ folders, this one can hold any number of files that hold descriptions of resources to create rather than the resources themselves. The XML element types control where these resources are placed under the R class. While the files can be named anything, these are the typical files in this folder (the convention is to name the file after the type of elements defined within): • **arrays.xml** to define arrays • **colors.xml** to define color drawables and color string values. Use `Resources.getDrawable()` and `Resources.getColor(), respectively,` to get these resources. • **dimens.xml** to define dimension value. Use `Resources.getDimension()` to get these resources. • **strings.xml** to define string values (use either `Resources.getString` or preferably `Resources.getText()` to get these resources. `getText()` will retain any rich text styling which is usually desirable for UI strings. • **styles.xml** to define style objects.
`res/xml/`	Arbitrary XML files that are compiled and can be read at run time by calling Resources.getXML().
`res/raw/`	Arbitrary files to copy directly to the device. They are added uncompiled to the compressed file that your application build produces. To use these resources in your application, call Resources.openRawResource() with the resource ID, which is R.raw.*somefilename*.

Resources are compiled into the final APK file. Android creates a wrapper class, called R, that you can use to refer to these resources in your code. R contains subclasses named according to the path and file name of the source file

Global Resource Notes

- Several resources allow you to define colors. Android accepts color values written in various web-style formats -- a hexadecimal constant in any of the following forms: #RGB, #ARGB, #RRGGBB, #AARRGGBB.
- All color values support setting an alpha channel value, where the first two hexadecimal numbers specify the transparency. Zero in the alpha channel means transparent. The default value is opaque.

Using Resources

This section describes how to use the resources you've created. It includes the following topics:

- Using resources in code - How to call resources in your code to instantiate them.

- Referring to resources from other resources - You can reference resources from other resources. This lets you reuse common resource values inside resources.
- Supporting Alternate Resources for Alternate Configurations - You can specify different resources to load, depending on the language or display configuration of the host hardware.

At compile time, Android generates a class named R that contains resource identifiers to all the resources in your program. This class contains several subclasses, one for each type of resource supported by Android, and for which you provided a resource file. Each class contains one or more identifiers for the compiled resources, that you use in your code to load the resource. Here is a small resource file that contains string, layout (screens or parts of screens), and image resources.

Note: the R class is an auto-generated file and is not designed to be edited by hand. It will be automatically re-created as needed when the resources are updated.

```
package com.android.samples;
public final class R {
    public static final class string {
        public static final int greeting=0x0204000e;
        public static final int start_button_text=0x02040001;
        public static final int submit_button_text=0x02040008;
        public static final int main_screen_title=0x0204000a;
    };
    public static final class layout {
        public static final int start_screen=0x02070000;
        public static final int new_user_pane=0x02070001;
        public static final int select_user_list=0x02070002;

    };
    public static final class drawable {
        public static final int company_logo=0x02020005;
        public static final int smiling_cat=0x02020006;
        public static final int yellow_fade_background=0x02020007;
        public static final int stretch_button_1=0x02020008;

    };
};
```

Using Resources in Code

Using resources in code is just a matter of knowing the full resource ID and what type of object your resource has been compiled into. Here is the syntax for referring to a resource:

`R.resource_type.resource_name`

or

`android.R.resource_type.resource_name`

Where `resource_type` is the R subclass that holds a specific type of resource. `resource_name` is the *name* attribute for resources defined in XML files, or the file name (without the extension) for resources defined by other file types. Each type of resource will be added to a specific R subclass, depending on the type of resource it is; to learn which R subclass hosts your compiled resource type, consult the resource reference document. Resources compiled by your own application can be referred to without a package name (simply as `R.resource_type.resource_name`). Android contains a number of standard resources, such as screen styles and button backgrounds. To refer to these in code, you must qualify them with `android`, as in `android.R.drawable.button_background`.

Here are some good and bad examples of using compiled resources in code:

```
// Load a background for the current screen from a drawable resource.
this.getWindow().setBackgroundDrawableResource(R.drawable.my_background_image);

// WRONG Sending a string resource reference into a
// method that expects a string.
this.getWindow().setTitle(R.string.main_title);

// RIGHT Need to get the title from the Resources wrapper.
this.getWindow().setTitle(Resources.getText(R.string.main_title));
```

```
// Load a custom layout for the current screen.
setContentView(R.layout.main_screen);

// Set a slide in animation for a ViewFlipper object.
mFlipper.setInAnimation(AnimationUtils.loadAnimation(this,
        R.anim.hyperspace_in));

// Set the text on a TextView object.
TextView msgTextView = (TextView)findViewByID(R.id.msg);
msgTextView.setText(R.string.hello_message);
```

References to Resources

A value supplied in an attribute (or resource) can also be a reference to a resource. This is often used in layout files to supply strings (so they can be localized) and images (which exist in another file), though a reference can be any resource type including colors and integers.

For example, if we have color resources, we can write a layout file that sets the text color size to be the value contained in one of those resources:

```
<?xml version="1.0" encoding="utf-8"?>
<EditText id="text"
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="fill_parent" android:layout_height="fill_parent"
    android:textColor="@color/opaque_red"
    android:text="Hello, World!" />
```

Note here the use of the '@' prefix to introduce a resource reference -- the text following that is the name of a resource in the form of `@[package:]type/name`. In this case we didn't need to specify the package because we are referencing a resource in our own package. To reference a system resource, you would need to write:

```
<?xml version="1.0" encoding="utf-8"?>
<EditText id="text"
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="fill_parent" android:layout_height="fill_parent"
    android:textColor="@android:color/opaque_red"
    android:text="Hello, World!" />
```

As another example, you should always use resource references when supplying strings in a layout file so that they can be localized:

```
<?xml version="1.0" encoding="utf-8"?>
<EditText id="text"
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="fill_parent" android:layout_height="fill_parent"
    android:textColor="@android:color/opaque_red"
    android:text="@string/hello_world" />
```

This facility can also be used to create references between resources. For example, we can create new drawable resources that are aliases for existing images:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <drawable id="my_background">@android:drawable/theme2_background</drawable>
</resources>
```

References to Theme Attributes

Another kind of resource value allows you to reference the value of an attribute in the current theme. This attribute reference can *only* be used in style resources and XML attributes; it allows you to customize the look of UI elements by changing them to standard variations supplied by the current theme, instead of supplying more concrete values.

As an example, we can use this in our layout to set the text color to one of the standard colors defined in the base system theme:

```
<?xml version="1.0" encoding="utf-8"?>
<EditText id="text"
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="fill_parent" android:layout_height="fill_parent"
    android:textColor="?android:textDisabledColor"
    android:text="@string/hello_world" />
```

Note that this is very similar to a resource reference, except we are using an '?' prefix instead of '@'. When you use this markup, you are supplying the name of an attribute resource that will be looked up in the theme -- because the resource tool knows that an attribute resource is expected, you do not need to explicitly state the type (which would be `?android:attr/android:textDisabledColor`).

Other than using this resource identifier to find the value in the theme instead of raw resources, the name syntax is identical to the '@' format: `?[namespace:]type/name` with the type here being optional.

Using System Resources

Many resources included with the system are available to applications. All such resources are defined under the class "android.R". For example, you can display the standard application icon in a screen with the following code:

```
public class MyActivity extends Activity
{
    public void onStart()
    {
        requestScreenFeatures(FEATURE_BADGE_IMAGE);

        super.onStart();

        setBadgeResource(android.R.drawable.sym_def_app_icon);
    }
}
```

In a similar way, this code will apply to your screen the standard "green background" visual treatment defined by the system:

```
public class MyActivity extends Activity
{
    public void onStart()
    {
        super.onStart();

        setTheme(android.R.style.Theme_Black);
    }
}
```

Supporting Alternate Resources for Alternate Languages and Configurations

You can supply different resources for your product according to the UI language or hardware configuration on the device. Note that although you can include different string, layout, and other resources, the SDK does not expose methods to let you specify which alternate resource set to load. Android detects the proper set for the hardware and location, and loads them as appropriate. Users can select alternate language settings using the settings panel on the device.

To include alternate resources, create parallel resource folders with qualifiers appended to the folder names, indicating the configuration it applies to (language, screen orientation, and so on). For example, here is a project that holds one string resource file for English, and another for French:

```
MyApp/
    res/
        values-en/
            strings.xml
```

```
        values-fr/
            strings.xml
```

Android supports several types of qualifiers, with various values for each. Append these to the end of the resource folder name, separated by dashes. You can add multiple qualifiers to each folder name, but they must appear in the order they are listed here. For example, a folder containing drawable resources for a fully specified configuration would look like:

```
MyApp/
    res/
        drawable-en-rUS-port-160dpi-finger-keysexposed-qwerty-dpad-480x320/
```

More typically, you will only specify a few specific configuration options that a resource is defined for. You may drop any of the values from the complete list, as long as the remaining values are still in the same order:

```
MyApp/
    res/
        drawable-en-rUS-finger/
        drawable-port/
        drawable-port-160dpi/
        drawable-qwerty/
```

Qualifier	Values
Language	The two letter ISO 639-1 language code in lowercase. For example: `en`, `fr`, `es`
Region	The two letter ISO 3166-1-alpha-2 language code in uppercase preceded by a lowercase "r". For example: `rUS`, `rFR`, `rES`
Screen orientation	`port`, `land`, `square`
Screen pixel density	`92dpi`, `108dpi`, etc.
Touchscreen type	`notouch`, `stylus`, `finger`
Whether the keyboard is available to the user	`keysexposed`, `keyshidden`
Primary text input method	`nokeys`, `qwerty`, `12key`
Primary non-touchscreen navigation method	`notouch`, `dpad`, `trackball`, `wheel`
Screen dimensions	`320x240`, `640x480`, etc. The larger dimension must be specified first.

This list does not include device-specific parameters such as carrier, branding, device/hardware, or manufacturer. Everything that an application needs to know about the device that it is running on is encoded via the resource qualifiers in the table above.

Here are some general guidelines on qualified resource directory names:

- Values are separated by a dash (as well as a dash after the base directory name)
- Values are case-sensitive (even though they must be unique across all folder names in a case-insensitive way) For example,
  - A portrait-specific `drawable` directory must be named `drawable-port`, not `drawable-PORT`.
  - You may not have two directories named `drawable-port` and `drawable-PORT`, even if you had intended "port" and "PORT" to refer to different parameter values.
- Only one value for each qualifier type is supported (that is, you cannot specify `drawable-rEN-rFR/`)
- You can specify multiple parameters to define specific configurations, but they must always be in the order listed above. For example, `drawable-en-rUS-land` will apply to landscape view, US-English devices.

- Android will try to find the most specific matching directory for the current configuration, as described below
- The order of parameters listed in this table is used to break a tie in case of multiple qualified directories (see the example given below)
- All directories, both qualified and unqualified, live under the `res/` folder. Qualified directories cannot be nested (you cannot have `res/drawable/drawable-en`)
- All resources will be referenced in code or resource reference syntax by their simple, undecorated name. So if a resource is named this:
  `MyApp/res/drawable-port-92dp/myimage.png`
  It would be referenced as this:
  `R.drawable.myimage` (code)
  `@drawable/myimage` (XML)

How Android finds the best matching directory

Android will pick which of the various underlying resource files should be used at runtime, depending on the current configuration. The selection process is as follows:

1. Eliminate any resources whose configuration does not match the current device configuration. For example, if the screen pixel density is 108dpi, this would eliminate only `MyApp/res/drawable-port-92dpi/`.

```
MyApp/res/drawable/myimage.png
MyApp/res/drawable-en/myimage.png
MyApp/res/drawable-port/myimage.png
~~MyApp/res/drawable-port-92dpi/myimage.png~~
```

2. Pick the resources with the highest number of matching configurations. For example, if our locale is en-GB and orientation is port, then we have two candidates with one matching configuration each: `MyApp/res/drawable-en/` and `MyApp/res/drawable-port/`. The directory `MyApp/res/drawable/` is eliminated because it has zero matching configurations, while the others have one matching configuration.

```
~~MyApp/res/drawable/myimage.png~~
MyApp/res/drawable-en/myimage.png
MyApp/res/drawable-port/myimage.png
```

3. Pick the final matching file based on configuration precedence, which is the order of parameters listed in the table above. That is, it is more important to match the language than the orientation, so we break the tie by picking the language-specific file, `MyApp/res/drawable-en/`.

```
MyApp/res/drawable-en/myimage.png
~~MyApp/res/drawable-port/myimage.png~~
```

Terminology

The resource system brings a number of different pieces together to form the final complete resource functionality. To help understand the overall system, here are some brief definitions of the core concepts and components you will encounter in using it:

Asset: A single blob of data associated with an application. This includes object files compiled from the Java source code, graphics (such as PNG images), XML files, etc. These files are organized in a directory hierarchy that, during final packaging of the application, is bundled together into a single ZIP file.

aapt: Android Asset Packaging Tool. The tool that generates the final ZIP file of application assets. In addition to collecting raw assets together, it also parses resource definitions into binary asset data.

Resource Table: A special asset that aapt generates for you, describing all of the resources contained in an application/package. This file is accessed for you by the Resources class; it is not touched directly by applications.

Resource: An entry in the Resource Table describing a single named value. Broadly, there are two types of resources: primitives and bags.

Resource Identifier: In the Resource Table all resources are identified by a unique integer number. In source code (resource descriptions, XML files, Java source code) you can use symbolic names that stand as constants for the actual resource identifier integer.

Primitive Resource: All primitive resources can be written as a simple string, using formatting to describe a variety of primitive types included in the resource system: integers, colors, strings, references to other resources, etc. Complex resources, such as bitmaps and XML describes, are stored as a primitive string resource whose value is the path of the underlying Asset holding its actual data.

Bag Resource: A special kind of resource entry that, instead of a simple string, holds an arbitrary list of name/value pairs. Each name is itself a resource identifier, and each value can hold the same kinds of string formatted data as a normal resource. Bags also support inheritance: a bag can inherit the values from another bag, selectively replacing or extending them to generate its own contents.

Kind: The resource kind is a way to organize resource identifiers for various purposes. For example, drawable resources are used to instantiate Drawable objects, so their data is a primitive resource containing either a color constant or string path to a bitmap or XML asset. Other common resource kinds are string (localized string primitives), color (color primitives), layout (a string path to an XML asset describing a view layout), and style (a bag resource describing user interface attributes). There is also a standard "attr" resource kind, which defines the resource identifiers to be used for naming bag items and XML attributes

Style: The name of the resource kind containing bags that are used to supply a set of user interface attributes. For example, a TextView class may be given a style resource that defines its text size, color, and alignment. In a layout XML file, you associate a style with a bag using the "style" attribute, whose value is the name of the style resource.

Style Class: Specifies a related set of attribute resources. This data is not placed in the resource table itself, but used to generate constants in the source code that make it easier for you to retrieve values out of a style resource and/or XML tag's attributes. For example, the Android platform defines a "View" style class that contains all of the standard view attributes: padding, visibility, background, etc.; when View is inflated it uses this style class to retrieve those values from the XML file (at which point style and theme information is applied as approriate) and load them into its instance.

Configuration: For any particular resource identifier, there may be multiple different available values depending on the current configuration. The configuration includes the locale (language and country), screen orientation, screen density, etc. The current configuration is used to select which resource values are in effect when the resource table is loaded.

Theme: A standard style resource that supplies global attribute values for a particular context. For example, when writing an Activity the application developer can select a standard theme to use, such as the Theme.White or Theme.Black styles; this style supplies information such as the screen background image/color, default text color, button style, text editor style, text size, etc. When inflating a layout resource, most values for widgets (the text color, selector, background) if not explicitly set will come from the current theme; style and attribute values supplied in the layout can also assign their value from explicitly named values in the theme attributes if desired.

Overlay: A resource table that does not define a new set of resources, but instead replaces the values of resources that are in another resource table. Like a configuration, this is applied at load time to the resource data; it can add new configuration values (for example strings in a new locale), replace existing values (for example change the standard white background image to a "Hello Kitty" background image), and modify resource bags (for example change the font size of the Theme.White style to have an 18 pt font size). This is the facility that allows the user to select between different global appearances of their device, or download files with new appearances.

Resource Reference

The Resource Reference document provides a detailed list of the various types of resource and how to use them from within the Java source code, or from other references.

Internationalization and Localization

Coming Soon: Internationalization and Localization are critical, but are also not quite ready yet in the current SDK. As the SDK matures, this section will contain information on the Internationalization and Localization features of the Android platform. In the meantime, it is a good idea to start by externalizing all strings, and practicing good structure in creating and using resources.

 Build 110632-110632 - 22 Sep 2008 13:34

Android

Hierarchy of Screen Elements

The basic functional unit of an Android application is the *activity*--an object of the class android.app.Activity. An activity can do many things, but by itself it does not have a presence on the screen. To give your activity a screen presence and design its UI, you work with *views* and *viewgroups* -- basic units of user interface expression on the Android platform.

Views

A view is an object of base class android.view.View. It's a data structure whose properties store the layout and content for a specific rectangular area of the screen. A View object handles measuring and layout, drawing, focus change, scrolling, and key/gestures for the screen area it represents.

The View class serves as a base class for *widgets* -- a set of fully implemented subclasses that draw interactive screen elements. Widgets handle their own measuring and drawing, so you can use them to build your UI more quickly. The list of widgets available includes Text, EditText, InputMethod, MovementMethod, Button, RadioButton, Checkbox, and ScrollView.

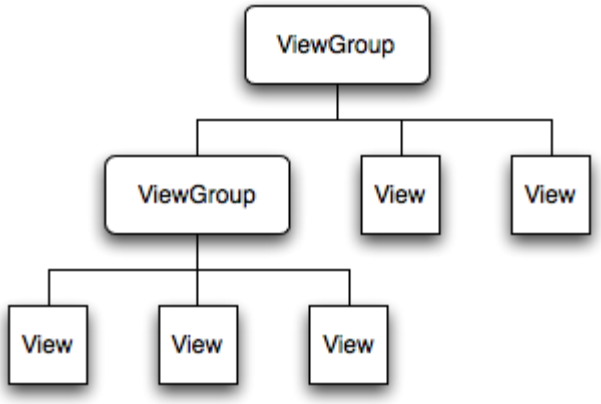
Viewgroups

A viewgroup is an object of class android.view.Viewgroup. As its name indicates, a viewgroup is a special type of view object whose function is to contain and manage a subordinate set of views and other viewgroups, Viewgroups let you add structure to your UI and build up complex screen elements that can be addressed as a single entity.

The Viewgroup class serves as a base class for *layouts* -- a set of fully implemented subclasses that provide common types of screen layout. The layouts give you a way to build a structure for a set of views.

A Tree-Structured UI

On the Android platform, you define an Activity's UI using a tree of view and viewgroup nodes, as shown in the diagram below. The tree can be as simple or complex as you need to make it, and you can build it up using Android's set of predefined widgets and layouts or custom view types that you create yourself.



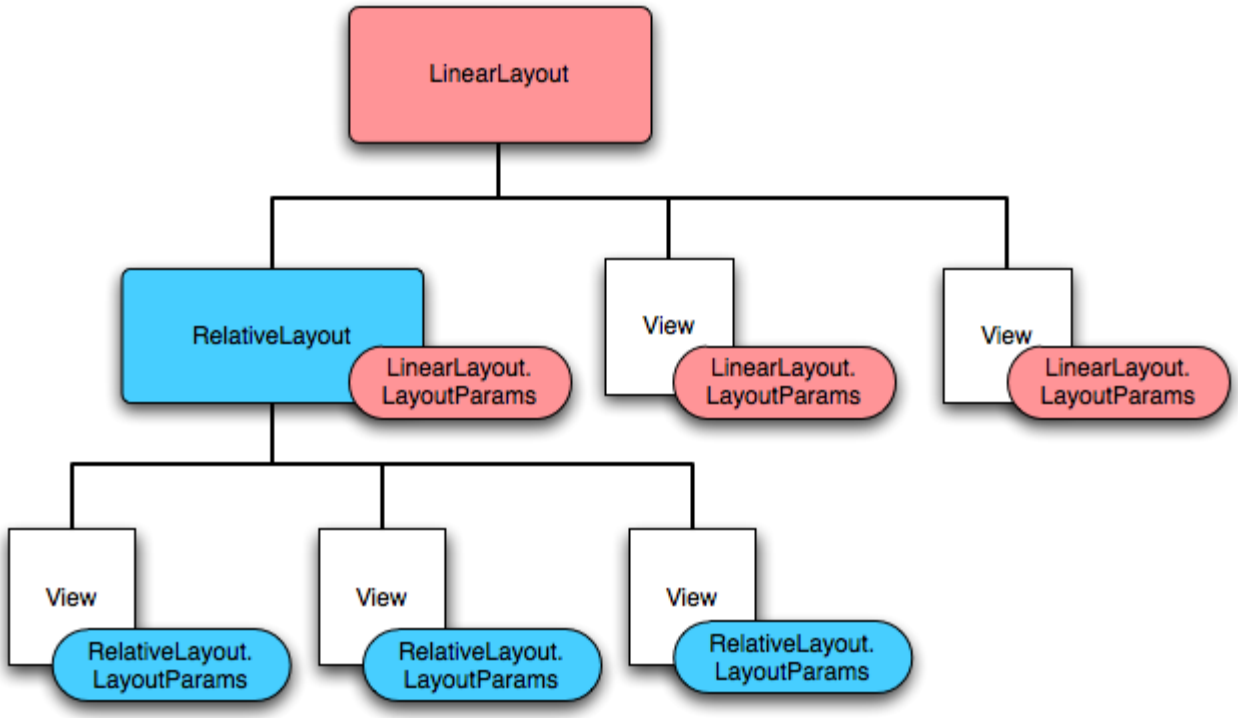
To attach the tree to the screen for rendering, your Activity calls its setContentView() method and passes a reference to the root node object. Once the Android system has the reference to the root node object, it can work directly with the node to invalidate, measure, and draw the tree. When your Activity becomes active and receives focus, the system notifies your activity and requests the root node to measure and draw the tree. The root node then requests that its child nodes draw themselves -- in turn, each viewgroup node in the tree is responsible for drawing its direct children.

As mentioned previously, each view group has the responsibility of measuring its available space, laying out its children, and

calling Draw() on each child to let it render itself. The children may request a size and location in the parent, but the parent object has the final decision on where how big each child can be.

LayoutParams: How a Child Specifies Its Position and Size

Every viewgroup class uses a nested class that extends ViewGroup.LayoutParams. This subclass contains property types that define a child's size and position, in properties appropriate for that view group class.



Note that every LayoutParams subclass has its own syntax for setting values. Each child element must define LayoutParams that are appropriate for its parent, although it may define different LayoutParams for its children.

All viewgroups include width and height. Many also include margins and borders. You can specify width and height exactly, though you probably won't want to do this often. More often you will tell your view to size itself either to the dimensions of its content, or to become as big as its containing object will allow.

Android

Common Layout Objects

The following are the most common view groups you'll use in your applications. This gives some basic information about each type; for in-depth detail, see the linked reference page topic for each.

FrameLayout

FrameLayout is the simplest layout object. It is intended as a blank reserved space on your screen that you can later fill with a single object — for example, a picture that you'll swap out. All child elements are pinned to the top left corner of the screen; you cannot specify a location for a child of a FrameLayout. Later children will simply be drawn over earlier objects, partially or totally obscuring them (unless the newer object is transparent).

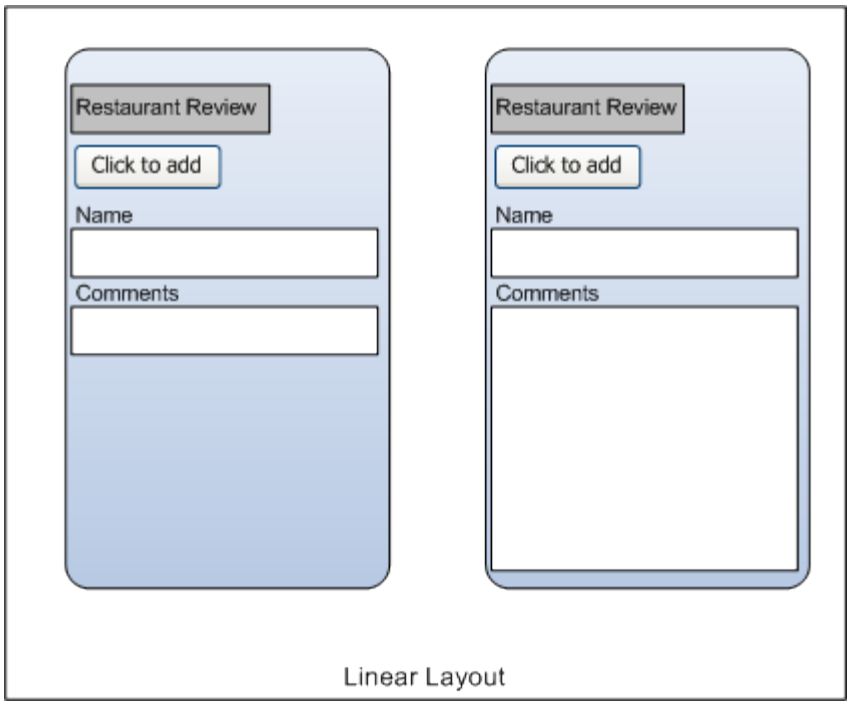
LinearLayout

A LinearLayout aligns all children in a single direction — vertically or horizontally, depending on what property you set on the LinearLayout. All children are stacked one after the other, so a vertical list will only have one child per row, no matter how wide they are, and a horizontal list will only be one row high (the height of the tallest child, plus padding). LinearLayout respects margins between children, and also *gravity* (right, center, or left alignment of a child).

LinearLayout also supports assigning a *weight* to individual children. This value allows children to expand to fill any remaining space on a screen. This prevents a list of small objects from being bunched to one end of a large screen, allowing them to expand to fill the space. Children specify a weight value, and any remaining space is assigned to children in the proportion of their declared weight. Default weight is zero. So, for example, if there are three text boxes, and two of them declare a weight of 1, two of them will expand equally to fill the remaining space, and the third will not grow any additional amount.

The following two forms represent a LinearLayout with a set of elements: a button, some labels, some text boxes. Both have padding values to adjust the padding nicely. The text boxes have their width set to `FILL_PARENT`; other elements are set to `WRAP_CONTENT`. The gravity, by default, is left. The form on the left has weight values unset (0 by default); the form on the right has the comments text box weight set to 1. If the Name textbox had also been set to 1, the Name and Comments text boxes would be the same height.

Tip: To create a proportionate size layout on the screen, create a container object that is fill_parent, assign the children heights or widths of zero, and then assign relative weight values to each child, depending on what proportion of the screen each should take.



Linear Layout

Within a horizontal LinearLayout, items are aligned by the position of their text base line (the first line of the first list element — topmost or leftmost — is considered the reference line). This is so that people scanning elements in a form shouldn't have to jump up and down to read element text in neighboring elements. This can be turned off by setting `android:baselineAligned="false"` in the layout XML.

TableLayout

TableLayout positions its children into rows and columns. A TableLayout consists of a number of TableRow objects, each defining a row (actually, you can have other children, which will be explained below). TableLayout containers do not display border lines for their rows, columns, or cells. Each row has zero or more cells; each cell can hold one View object. The table has as many columns as the row with the most cells. A table can leave cells empty. Cells cannot span columns, as they can in HTML. The following image shows a table layout, with the invisible cell borders displayed as dotted lines.

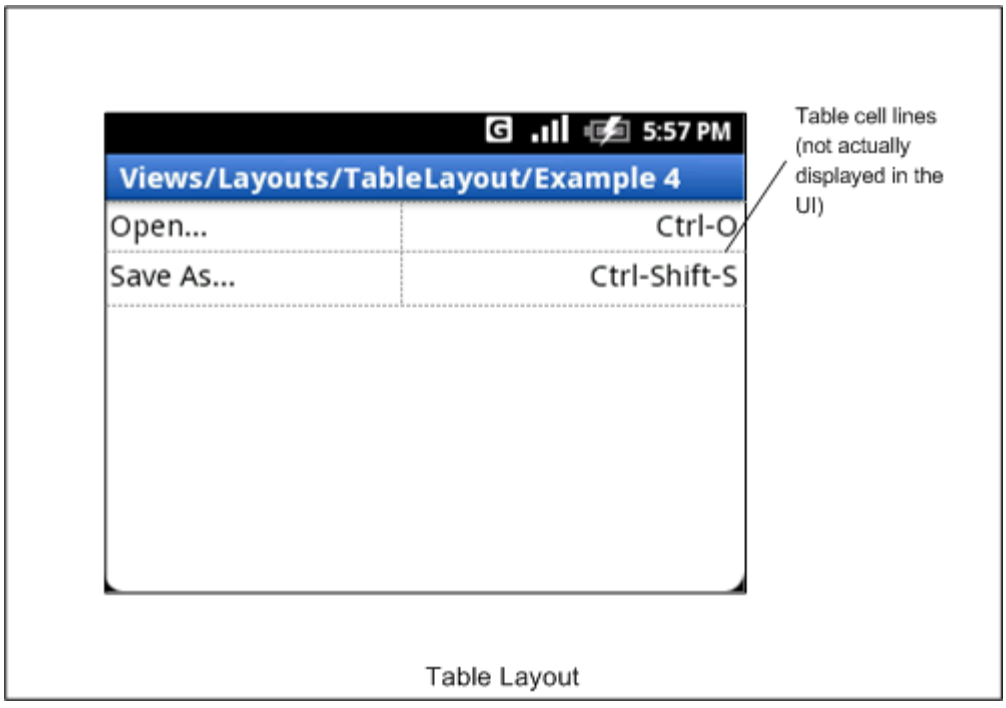


Table Layout

Columns can be hidden, can be marked to stretch to fill available screen space, or can be marked as shrinkable to force the column to shrink until the table fits the screen. See the reference documentation for this class for more details.

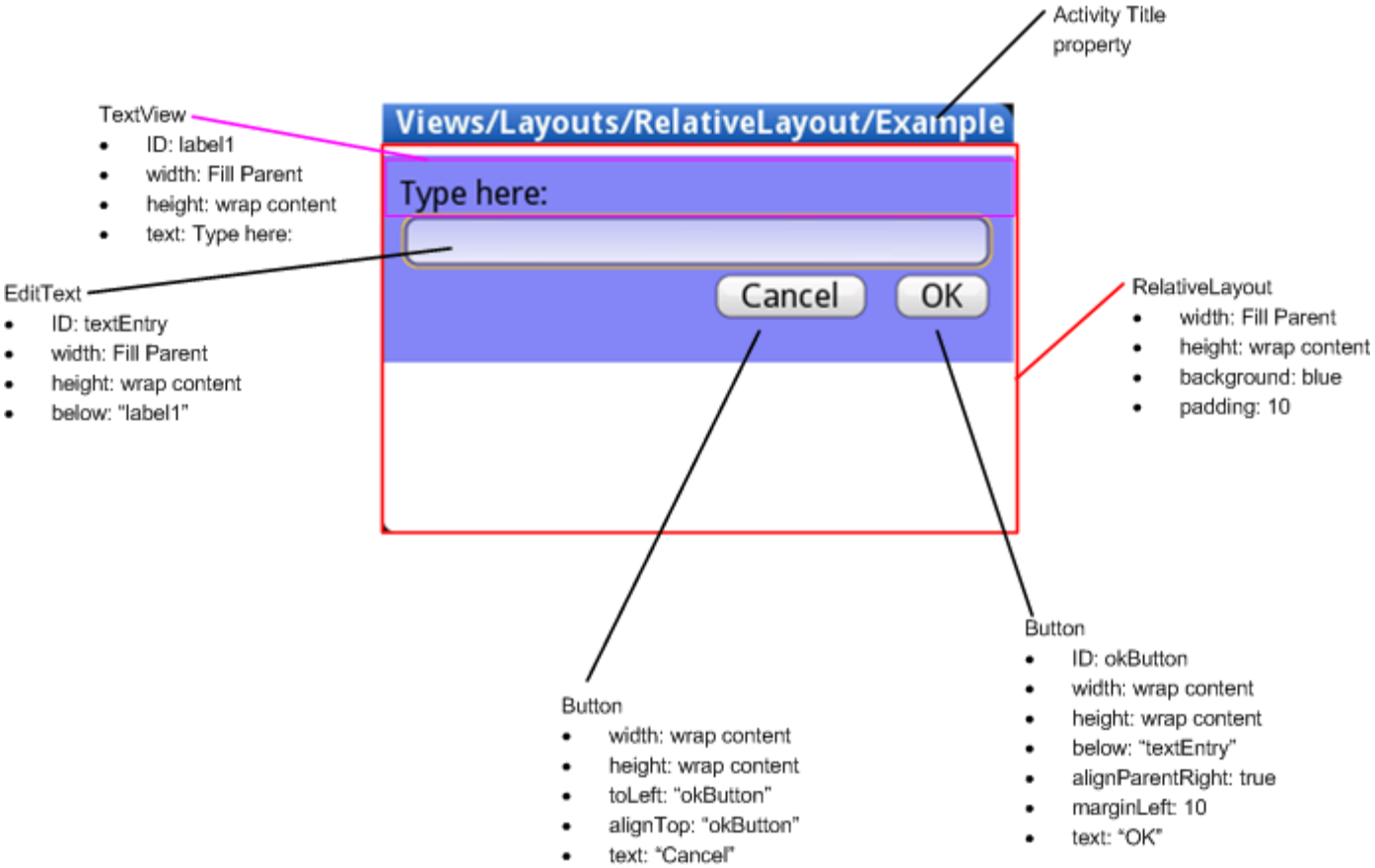
AbsoluteLayout

AbsoluteLayout enables children to specify exact x/y coordinates to display on the screen, where (0,0) is the upper left corner, and values increase as you move down or to the right. Margins are not supported, and overlapping elements are allowed (although not recommended). We generally recommend against using AbsoluteLayout unless you have good reasons to use it, because it is fairly rigid and does not work well with different device displays.

RelativeLayout

RelativeLayout lets children specify their position relative to each other (specified by ID), or to the parent. So you can align two elements by right border, or make one below another, or centered in the screen. Elements are rendered in the order given, so if the first element is centered in the screen, other elements aligning themselves to that element will be aligned relative to screen center. If using XML to specify this layout (as described later), a referenced element must be listed before you refer to it.

Here is an example relative layout with the visible and invisible elements outlined. The root screen layout object is a RelativeLayout object.



This diagram shows the class names of the screen elements, followed by a list of the properties of each. Some of these properties are supported directly by the element, and some are supported by its LayoutParams member (subclass RelativeLayout for all the elements in this screen, because all elements are children of a RelativeLayout parent object). The RelativeLayout parameters are width, height, below, alignTop, toLeft, padding, and marginLeft. Note that some of these parameters support values relative to other children — hence the name RelativeLayout. These include the toLeft, alignTop, and below properties, which indicate the object to the left, top, and below respectively.

Summary of Important View Groups

These objects all hold child UI elements. Some provide visible UI, and others only handle child layout.

Class	Description
AbsoluteLayout	Enables you to specify the location of child objects relative to the parent in exact measurements (for example, pixels).
FrameLayout	Layout that acts as a view frame to display a single object.
Gallery	A horizontal scrolling display of images, from a bound list.
GridView	Displays a scrolling grid of m columns and n rows.
LinearLayout	A layout that organizes its children into a single horizontal or vertical row. It creates a scrollbar if the length of the window exceeds the length of the screen.
ListView	Displays a scrolling single column list.
RelativeLayout	Enables you to specify the location of child objects relative to each other (child A to the left of child B) or to the parent (aligned to the top of the parent).
ScrollView	A vertically scrolling column of elements.
Spinner	Displays a single item at a time from a bound list, inside a one-row textbox. Rather like a one-row listbox that can scroll either horizontally or vertically.
SurfaceView	Provides direct access to a dedicated drawing surface. It can hold child views layered on top of the surface, but is intended for applications that need to draw pixels, rather than using widgets.
TabHost	Provides a tab selection list that monitors clicks and enables the application to change the screen whenever a tab is clicked.
TableLayout	A tabular layout with an arbitrary number of rows and columns, each cell holding the widget of your choice. The rows resize to fit the largest column. The cell borders are not visible.
ViewFlipper	A list that displays one item at a time, inside a one-row textbox. It can be set to swap items at timed intervals, like a slide show.
ViewSwitcher	Same as ViewFlipper.

Build 110632-110632 - 22 Sep 2008 13:34

Android

Working with AdapterViews (Binding to Data)

As we mentioned, some view groups have UI. These objects typically subclass AdapterView. Examples include such as Gallery (an image selection widget) and ListView (a list of views). These objects have two jobs in common:

- Filling the layout with data
- Handling user selections

Filling the layout with data

This is typically done by binding the class to an Adapter that gets its data from somewhere — either a list that the code supplies, or query results from the device's database.

```
// Get a Spinner and bind it to an ArrayAdapter that
// references a String array.
Spinner s1 = (Spinner) findViewById(R.id.spinner1);
ArrayAdapter adapter = ArrayAdapter.createFromResource(
    this, R.array.colors, android.R.layout.simple_spinner_item);
adapter.setDropDownViewResource(android.R.layout.simple_spinner_dropdown_item);
s1.setAdapter(adapter);

// Load a Spinner and bind it to a data query.
private static String[] PROJECTION = new String[] {
        People._ID, People.NAME
    };

Spinner s2 = (Spinner) findViewById(R.id.spinner2);
Cursor cur = managedQuery(People.CONTENT_URI, PROJECTION, null, null);

SimpleCursorAdapter adapter2 = new SimpleCursorAdapter(this,
    android.R.layout.simple_spinner_item, // Use a template
                                          // that displays a
                                          // text view
    cur, // Give the cursor to the list adatper
    new String[] {People.NAME}, // Map the NAME column in the
                                // people database to...
    new int[] {android.R.id.text1}); // The "text1" view defined in
                                     // the XML template

adapter2.setDropDownViewResource(android.R.layout.simple_spinner_dropdown_item);
s2.setAdapter(adapter2);
```

Note that it is necessary to have the People._ID column in projection used with CursorAdapter or else you will get an exception.

Handling user selections

This is done by setting the class's AdapterView.OnItemClickListener member to a listener and catching the selection changes.

```
// Create a message handling object as an anonymous class.
private OnItemClickListener mMessageClickedHandler = new OnItemClickListener() {
    public void onItemClick(AdapterView parent, View v, int position, long id)
    {
        // Display a messagebox.
        Toast.makeText(mContext,"You've got an event",Toast.LENGTH_SHORT).show();
    }
};

// Now hook into our object and set its onItemClickListener member
```

```
// to our class handler object.
mHistoryView = (ListView)findViewById(R.id.history);
mHistoryView.setOnItemClickListener(mMessageClickedHandler);
```


Build 110632-110632 - 22 Sep 2008 13:34

Android

Designing Your Screen in XML

Because designing a screen in code can be cumbersome, Android supports an XML syntax to design screens. Android defines a large number of custom elements, each representing a specific Android View subclass. You can design a screen the same way you create HTML files, as a series of nested tags, saved in an XML file inside the application's `res/layout/` directory. To learn what elements are exposed, and the format of the XML file, see Layout Resources. Each file describes a single android.view.View element, but this element can be either a simple visual element, or a layout element that contains a collection of child objects (a screen or a portion of a screen). When Android compiles your application, it compiles each file into an android.view.View resource that you can load in code by calling `setContentView(R.layout.layout_file_name)` in your Activity.onCreate() implementation.

Each XML file is made of tags that correspond to Android GUI classes. These tags have attributes that roughly correspond to methods in that class (for example, EditText has a *text* attribute that corresponds to EditText.setText).

Note that there is not an exact correspondence between class and method names, and element and attribute names — they're close, but not always 1:1.

Also note that Android tends to draw elements in the order in which they appear in the XML. Therefore, if elements overlap, the last one in the XML file will probably be drawn on top of any previously listed elements in that same space.

Each XML file is compiled into a tree rooted by single View or ViewGroup object, and so must contain a single root tag. In the following example, it evaluates to the outermost LinearLayout object.

Attributes named `layout_something` apply to that object's LayoutParams member. Layout Resources also describes how to learn the syntax for specifying LayoutParams properties.

The following values are supported for dimensions (described in TypedValue):

- px (pixels)
- dip (device independent pixels)
- sp (scaled pixels — best for text size)
- pt (points)
- in (inches)
- mm (millimeters)

Example: `android:layout_width="25px"`

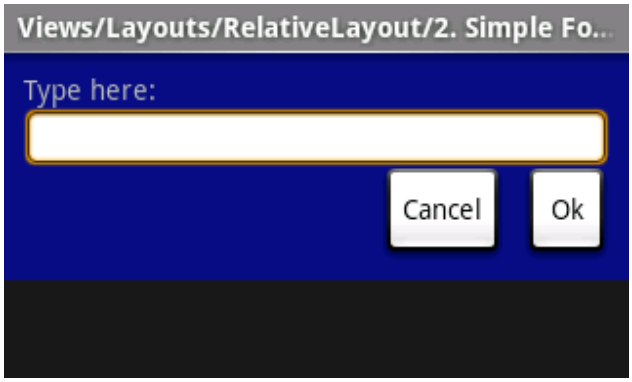
For more information about these dimensions, see Dimension Values.

The following XML file creates the screen shown. Note that the text on the top of the screen was set by calling Activity.setTitle. Note that the attributes that refer to relative elements (i.e., layout_toLeft) refer to the ID using the syntax of a relative resource (@id/*id_number*).

```
<?xml version="1.0" encoding="utf-8"?>
<!-- Demonstrates using a relative layout to create a
form -->
<RelativeLayout
xmlns:android="http://schemas.android.com/apk/res/android
                android:layout_width="fill_parent"
                android:layout_height="wrap_content"
                android:background="@drawable/blue"
                android:padding="10px">

    <TextView id="@+id/label"
              android:layout_width="fill_parent"
              android:layout_height="wrap_content"
              android:text="Type here:"/>

    <EditText id="@+id/entry"
              android:layout_width="fill_parent"
              android:layout_height="wrap_content"
```



```
android:background="@android:drawable/editbox_background"
                android:layout_below="@id/label"/>

    <Button id="@+id/ok"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_below="@id/entry"
            android:layout_alignParentRight="true"
            android:layout_marginLeft="10px"
            android:text="OK" />

    <Button android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_toLeftOf="@id/ok"
            android:layout_alignTop="@id/ok"
            android:text="Cancel" />
</RelativeLayout>
```

Loading the XML Resource

Loading the compiled layout resource is very easy, and done with a single call in the application's onCreate() method, as shown here:

```
protected void onCreate(Bundle savedValues)
{
   // Be sure to call the super class.
   super.onCreate(savedValues);

   // Load the compiled layout resource into the window's
   // default ViewGroup.
   // The source file is res/layout/hello_activity.xml
    setContentView(R.layout.hello_activity);

   // Retrieve any important stored values.
   restoreValues(savedValues);
}
```

Android

Hooking into a Screen Element

You can get a handle to a screen element by calling Activity.findViewById. You can use this handle to set or retrieve any values exposed by the object.

```
TextView msgTextView = (TextView)findViewById(R.id.msg);
msgTextView.setText(R.string.push_me);
```


Build 110632-110632 - 22 Sep 2008 13:34

Android

Listening for UI Notifications

Some UI notifications are automatically exposed and called by Android. For instance, Activity exposes overrideable methods onKeyDown and onKeyUp, and Widget exposes onFocusChanged(boolean, int, Rect). However, some important callbacks, such as button clicks, are not exposed natively, and must be registered for manually, as shown here.

```
public class SendResult extends Activity
{
    /**
     * Initialization of the Screen after it is first created.  Must at least
     * call setContentView() to
     * describe what is to be displayed in the screen.
     */
    protected void onCreate(Bundle savedValues)
    {
        ...

        // Listen for button clicks.
        Button button = (Button)findViewById(R.id.corky);
        button.setOnClickListener(mCorkyListener);
    }

    // Create an anonymous class to act as a button click listener.
    private OnClickListener mCorkyListener = new OnClickListener()
    {
        public void onClick(View v)
        {
            // To send a result, simply call setResult() before your
            // activity is finished, building an Intent with the data
            // you wish to send.
            Intent data = new Intent();
            data.setAction("Corky!");
            setResult(RESULT_OK, data);
            finish();
        }
    };
```


Build 110632-110632 - 22 Sep 2008 13:34

Android

Applying a Theme to your Application

If you do not explicitly specify a theme for your UI, Android will use the default theme defined by android.R.style.Theme. Many times you will want to use a different system theme (such as Theme.Light) or create your own theme (as described in Style and Theme Resources).

To set your theme in XML, simply specify the desired theme in your AndroidManifest.xml file with the `theme` attribute. This can be used with the `<application>` tag (shown here) to specify a default theme for all of your activities, and/or with the `<activity>` to control the theme of a particular activity.

```
<!-- AndroidManifest.xml-->
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
        package="com.android.home">
    <application android:theme="@android:style/Theme.Dark" >
        <activity class=".Home"
        ...
        </activity>
    </application>
</manifest>
```

You can also set the theme programmatically, if needed. When doing so, be sure to set the theme before creating any views so that the correct theme is used for all of your user-interface elements. Note that this approach should typically be avoided, especially from the main activities of your application, because the theme you set here may not be used for any animations the system uses to show the activity (which is done before your application starts).

```
 protected void onCreate(Bundle icicle) {
    super.onCreate(icicle);
    ...
    setTheme(android.R.style.Theme_Light);
    setContentView(R.layout.linear_layout_3);
}
```


Build 110632-110632 - 22 Sep 2008 13:34

Android

UI Elements and Concepts Glossary

Here is a list of common UI elements and concepts that you will see here and elsewhere in the SDK.

Activity

The standard screen in an Android application. Activity is a class that Android can start when a matching Intent is thrown by this or another application. Most commonly, it is visibly represented by a full screen window that can receive and handle UI events and perform complex tasks, because of the Window it uses to render its window. Though an Activity is typically full screen, it can also be floating or transparent.

View

A rectangular area on the screen that can be drawn to, handles click, keystroke, and other interaction events. A View is a base class for most components of an Activity or Dialog screen (text boxes, windows, and so on). It receives calls from its container object to draw itself, and informs its parent object about where and how big it would like to be (which may or may not be respected by the parent). It is represented by the base class View.

View Group

A container that holds multiple child View objects, deciding where they will be and how large they can be, and calling on them to draw themselves when appropriate. Some are invisible and for layout only, while others have a UI themselves (for instance, scrolling list boxes). View groups are all in the widget package, but extend ViewGroup.

Widget

A form element, such as a text box or popup menu. They have the ability to draw themselves and handle UI events. Widgets are all in the widget package.

Drawable

A visual element that is loaded into another UI element, typically as a background image. It does not receive events, but does assign various other properties such as "state" and scheduling to enable subclasses such as animation objects or image libraries. Many drawable objects are loaded from resource files — xml or bitmap files that describe the image. The base class is Drawable. See Resources.

Panel

A panel is a concept not backed by a specific class. It is a View of some sort that is tied in closely to a parent window, but can handle clicks and perform simple functions related to its parent. A panel floats in front of its parent, and is positioned relative to it. A common example of a panel (implemented by Android) is the options menu available to every screen. At present, there are no specific classes or methods for creating a panel — it's more of a general idea.

Dialog

A dialog is a floating window that can have buttons, and acts as a lightweight form that is intended to, at most, perform a simple action (such as click a button) and perhaps return a value. It is not intended to persist in the history stack, contain complex layout, or perform complex actions. Android provides a default simple dialog for you with optional buttons, though you can define a dialog layout yourself. The base class is Dialog.

Window

An abstract class that specifies the elements of a generic window, such as the look and feel (title bar text, location and content of menus, and so on). Dialog and Activity use an implementation of this class to render a window. You should not need to implement this class.

Surface

A block of memory that gets composited to the screen. A Surface holds a Canvas object for drawing, and provides various helper methods to draw layers and resize the surface. You should not use this class directly; use SurfaceView instead.

SurfaceView

A View object that wraps a Surface for drawing, and exposes methods to specify its size and format dynamically. The

camera app uses SurfaceView for its preview screen. A SurfaceView provides a way to draw independently of the UI thread for resource-intense operations (such as games or camera previews), but it uses extra memory as a result. SurfaceView supports both Canvas and OpenGL ES graphics.

Canvas

A drawing surface where the actual bits are composited. It has methods for standard computer drawing of bitmaps, lines, circles, rectangles, text, and so on. It is bound to a Bitmap or Surface. Canvas is the simplest, easiest way to draw 2D objects on the screen. However, it does not support hardware acceleration, as OpenGL ES does.

OpenGL ES

Android provides OpenGL ES libraries that you can use for fast, complex 3D images. It is harder to use than a Canvas object, but better for 3D objects. The opengl and javax.microedition.khronos.opengles packages expose OpenGL ES functionality.

 Build 110632-110632 - 22 Sep 2008 13:34

Android

The AndroidManifest.xml File

AndroidManifest.xml is a required file for every application. It sits in the root folder for an application, and describes global values for your package, including the application components (activities, services, etc) that the package exposes and the implementation classes for each component, what kind of data each can handle, and where they can be launched.

An important aspect of this file are the *intent filters* that it includes. These filters describe where and when that activity can be started. When an activity (or the operating system) wants to perform an action such as open a Web page or open a contact picker screen, it creates an Intent object. This object can hold several descriptors describing what you want to do, what data you want to do it to, the type of data, and other bits of information. Android compares the information in an Intent object with the intent filter exposed by every application and finds the activity most appropriate to handle the data or action specified by the caller. More details on intents is given in the Intent reference page.

Besides declaring your application's Activities, Content Providers, Services, and Intent Receivers, you can also specify permissions and instrumentation (security control and testing) in AndroidManifest.xml. For a reference of the tags and their attributes, please see AndroidManifest.

A simple AndroidManifest.xml looks like this:

```
<?xml version="1.0" encoding="utf-8"?>

<manifest xmlns:android="http://schemas.android.com/apk/res/android"
        package="com.my_domain.app.helloactivity">

    <application android:label="@string/app_name">

        <activity android:name=".HelloActivity">
            <intent-filter>
                <action android:name="android.intent.action.MAIN"/>
                <category android:name="android.intent.category.LAUNCHER"/>
            </intent-filter>
        </activity>

    </application>

</manifest>
```

Some general items to note:

- Almost every AndroidManifest.xml (as well as many other Android XML files) will include the namespace declaration `xmlns:android="http://schemas.android.com/apk/res/android"` in its first element. This makes a variety of standard Android attributes available in the file, which will be used to supply most of the data for elements in that file.
- Most manifests include a single `<application>` element, which defines all of the application-level components and properties that are available in the package.
- Any package that will be presented to the user as a top-level application available from the program launcher will need to include at least one Activity component that supports the MAIN action and LAUNCHER category as shown here.

Here is a detailed outline of the structure of an AndroidManifest.xml file, describing all tags that are available.

<manifest>

The root node of the file, describing the complete contents of the package. Under it you can place:

<uses-permission>

Requests a security permission that your package must be granted in order for it to operate correctly. See the Security Model document for more information on permissions. A manifest can contain zero or more of these elements.

<permission>

Declares a security permission that can be used to restrict which applications can access components or features in your (or another) package. See the Security Model document for more information on permissions. A manifest can contain zero or more of these elements.

<instrumentation>

Declares the code of an instrumentation component that is available to test the functionality of this *or another* package. See Instrumentation for more details. A manifest can contain zero or more of these elements.

<application>

Root element containing declarations of the application-level components contained in the package. This element can also include global and/or default attributes for the application, such as a label, icon, theme, required permission, etc. A manifest can contain zero or one of these elements (more than one application tag is not allowed). Under it you can place zero or more of each of the following component declarations:

<activity>

An Activity is the primary facility for an application to interact with the user. The initial screen the user sees when launching an application is an activity, and most other screens they use will be implemented as separate activities declared with additional activity tags.

Note: Every Activity must have an <activity> tag in the manifest whether it is exposed to the world or intended for use only within its own package. If an Activity has no matching tag in the manifest, you won't be able to launch it.

Optionally, to support late runtime lookup of your activity, you can include one or more <intent-filter> elements to describe the actions the activity supports.

<intent-filter>

Declares a specific set of Intent values that a component supports, in the form of an IntentFilter. In addition to the various kinds of values that can be specified under this element, attributes can be given here to supply a unique label, icon, and other information for the action being described.

<action>

An Intent action that the component supports.

<category>

An Intent category that the component supports.

<data>

An Intent data MIME type, Intent data URI scheme, Intent data URI authority, or Intent data URI path that the component supports.

You can also optionally associate one or more pieces of meta-data with your activity that other clients can retrieve to find additional arbitrary information about it:

<meta-data>

Adds a new piece of meta data to the activity, which clients can retrieve through ComponentInfo.metaData.

<receiver>

An BroadcastReceiver allows an application to be told about changes to data or actions that happen, even if it is not currently running. As with the activity tag, you can optionally include one or more <intent-filter> elements that the receiver supports or <meta-data> values; see the activity's <intent-filter> and <meta-data> descriptions for more information.

<service>

A Service is a component that can run in the background for an arbitrary amount of time. As with the activity tag, you can optionally include one or more <intent-filter> elements that the service supports or <meta-data> values; see the activity's <intent-filter> and <meta-data> descriptions for more information.

<provider>

A ContentProvider is a component that manages persistent data and publishes it for access by other applications. You can also optionally attach one or more <meta-data> values, as described in the activity's

<meta-data> description.

 Build 110632-110632 - 22 Sep 2008 13:34

Android

Using Application Preferences

You can store application preferences such as a default greeting or text font to be loaded whenever this application is started. Call `Context.getSharedPreferences()` to read and write values. Assign a name to your set of preferences if you want to share them with other components in the same package, or use `Activity.getPreferences()` with no name to keep them private to the calling activity. You cannot share preferences across packages. Here is an example of setting user preferences for silent keypress mode for a calculator.

```
public class Calc extends Activity {
public static final String PREFS_NAME = "MyPrefsFile";
     ...

    @Override
    protected void onCreate(Bundle state){
       super.onCreate(state);

    ...

       // Restore preferences
       SharedPreferences settings = getSharedPreferences(PREFS_NAME, 0);
       boolean silent = settings.getBoolean("silentMode", false);
       setSilent(silent);
    }

    @Override
    protected void onStop(){
       super.onStop();

      // Save user preferences. We need an Editor object to
      // make changes. All objects are from android.context.Context
      SharedPreferences settings = getSharedPreferences(PREFS_NAME, 0);
      SharedPreferences.Editor editor = settings.edit();
      editor.putBoolean("silentMode", mSilentMode);

      // Don't forget to commit your edits!!!
      editor.commit();
    }
}
```


Build 110632-110632 - 22 Sep 2008 13:34

Android

Using Files

Android provides access to read or write streams to files local to an application. Call Context.openFileOutput() and Context.openFileInput() with a local name and path to read and write files. Calling these methods with the same name and path strings from another application will not work; you can only access local files.

If you have static files to package with your application at compile time, you can save your file in your project in `res/raw/<mydatafile>`, and then get it with Resources.openRawResource (R.raw.*mydatafile*).

Build 110632-110632 - 22 Sep 2008 13:34

Android

Using SQLite Databases

Android supports a SQLite database system and exposes database management functions that let you store complex collections of data wrapped into useful objects. For example, Android defines a Contact data type that consists of many fields including string first and last names, string address and phone numbers, a bitmap image, and much other information describing the person. To create a database, use SQLiteOpenHelper and read and write this data as appropriate. (Note that file data such as a bitmap is typically stored as a string file path value in the database, with the location of the local file.)

Android ships with the sqlite3 database tool, which enables you to browse table contents, run SQL commands, and perform other useful functions on SQLite databases. See Examine databases (sqlite3) to learn how to run this program.

All databases, SQLite and others, are stored on the device in `/data/data/<package_name>/databases`

Discussion of how many tables to create, what fields they contain, and how they are linked, is beyond the scope of this document, but Android does not impose any limitations beyond the standard SQLite concepts. We do recommend including an autoincrement value key field that can be used as a unique ID to quickly find a record. This is not required for private data, but if you implement a content provider, you must include such a unique ID field. See the sample class NotePadProvider.java in the NotePad sample project for an example of creating and populating a new database. Any databases you create will be accessible by name to any other class in the application, but not outside the application.

Build 110632-110632 - 22 Sep 2008 13:34

Android

Accessing Content Providers

If you want to make your data public, you can create (or call) a content provider. This is an object that can store and retrieve data accessible by all applications. This is the only way to share data across packages; there is no common storage area that all packages can share. Android ships with a number of content providers for common data types (audio, video, images, personal contact information, and so on). You can see some of Android's native content providers in the provider package.

How a content provider actually stores its data under the covers is up to the implementation of the content provider, but all content providers must implement a common convention to query for data, and a common convention to return results. However, a content provider can implement custom helper functions to make data storage/retrieval simpler for the specific data that it exposes.

This document covers two topics related to Content Providers:

- Using a Content Provider
- Creating a Content Provider

Using a Content Provider to Store and Retrieve Data

This section describes how to store and retrieve data using a content provider implemented by you or anyone else. Android exposes a number of content providers for a wide range of data types, from music and image files to phone numbers. You can see a list of content providers exposed through the convenience classes in the android.provider package.

Android's content providers are loosely linked to their clients. Each content provider exposes a unique string (a URI) identifying the type of data that it will handle, and the client must use that string to store or retrieve data of that type. We'll explain this more in Querying for Data.

This section describes the following activities:

- Querying for Data
  - Making the query
  - What the query returns
  - Querying for files
  - Reading retrieved data
- Modifying Data
- Adding a Record
- Deleting a Record

Querying for Data

Each contact provider exposes a unique public URI (wrapped by Uri) that is used by a client to query/add/update/delete data on that content provider. This URI has two forms: one to indicate all values of that type (e.g., all personal contacts), and one form to indicate a specific record of that type (e.g., Joe Smith's contact information).

- **content://contacts/people/** is the URI that would return a list of all contact names on the device.
- **content://contacts/people/23** is the URI string that would return a single result row, the contact with ID = 23. .

An application sends a query to the device that specifies a general type of item (all phone numbers), or a specific item (Bob's phone number), to retrieve. Android then returns a Cursor over a recordset of results, with a specific set of columns. Let's look at a hypothetical query string and a result set (the results have been trimmed a bit for clarity):

query = `content://contacts/people/`

Results:

_ID	_COUNT	NUMBER	NUMBER_KEY	LABEL	NAME	TYPE

13	4	(425) 555 6677	425 555 6677	California office	Bully Pulpit	Work
44	4	(212) 555-1234	212 555 1234	NY apartment	Alan Vain	Home
45	4	(212) 555-6657	212 555 6657	Downtown office	Alan Vain	Work
53	4	201.555.4433	201 555 4433	Love Nest	Rex Cars	Home

Note that the query string isn't a standard SQL query string, but instead a URI string that describes the type of data to return. This URI consists of three parts: the string "content://"; a segment that describes what kind of data to retrieve; and finally an optional ID of a specific item of the specified content type. Here are a few more example query strings:

- **content://media/internal/images** is the URI string that would return a list of all the internal images on the device.
- **content://media/external/images** is the URI string that would return a list of all the images on the "primary" external storage (e.g., the SD card).
- **content://contacts/people/** is the URI that would return a list of all contact names on the device.
- **content://contacts/people/23** is the URI string that would return a single result row, the contact with ID = 23.

Although there is a general form, these query URIs are somewhat arbitrary and confusing. Therefore, Android provides a list of helper classes in the android.provider package that define these query strings so you should not need to know the actual URI value for different data types. These helper classes define a string (actually, a Uri object) called CONTENT_URI for a specific data type.

Typically you will use the defined CONTENT_URI object to make a query, instead of writing the full URI yourself. So, each of the example query strings listed above (except for the last one that specifies the record ID) can be acquired with the following Uri references:

- MediaStore.Images.Media.INTERNAL_CONTENT_URI
- MediaStore.Images.Media.EXTERNAL_CONTENT_URI
- Contacts.People.CONTENT_URI

To query a specific record ID (e.g., content://contacts/people/23), you'll use the same CONTENT_URI, but must append the specific ID value that you want. This is one of the few times you should need to examine or modify the URI string. So, for example, if you were looking for record 23 in the people contacts, you might run a query as shown here:

```
// Get the base URI for contact with _ID=23.
// This is same as Uri.parse("content://contacts/people/23");
Uri myPerson = ContentUris.withAppendedId(People.CONTENT_URI, 23);
// Query for this record.
Cursor cur = managedQuery(myPerson, null, null, null);
```

Tip: You can also append a string to a Uri, using withAppendedPath(Uri, String).

This query returns a cursor over a database query result set. What columns are returned, what they're called, and what they are named are discussed next. For now, though, know that you can specify that only certain columns be returned, the sort order, and a SQL WHERE clause.

You should use the Activity.managedQuery() method to retrieve a managed cursor. A managed cursor handles all the niceties such as unloading itself when the application pauses, and requerying itself when the application restarts. You can ask Android to manage an unmanaged cursor for you by calling Activity.startManagingCursor().

Let's look at an example query to retrieve a list of contact names and their primary phone numbers.

```
// An array specifying which columns to return.
string[] projection = new string[] {
    People._ID,
    People.NAME,
    People.NUMBER,
};

// Get the base URI for People table in Contacts content provider.
// ie. content://contacts/people/
Uri mContacts = People.CONTENT_URI;
```

```
// Best way to retrieve a query; returns a managed query.
Cursor managedCursor = managedQuery( mContacts,
                        projection, //Which columns to return.
                        null,       // WHERE clause--we won't specify.
                        People.NAME + " ASC"); // Order-by clause.
```

This query will retrieve data from the people table of the Contacts content provider. It will retrieve the name, primary phone number, and unique record ID for each contact.

What the query returns

A query returns a set of zero or more database records. The column names, order, and type are specific to the content provider, but every query includes a column called _id, which is the ID of the item in that row. If a query can return binary data, such as a bitmap or audio file, it will have a column with any name that holds a content:// URI that you can use to get this data (more information on how to get the file will be given later). Here is a tiresome example result set for the previous query:

_id	name	number
44	Alan Vain	212 555 1234
13	Bully Pulpit	425 555 6677
53	Rex Cars	201 555 4433

This result set demonstrates what is returned when we specified a subset of columns to return. The optional subset list is passed in the *projection* parameter of the query. A content manager should list which columns it supports either by implementing a set of interfaces describing each column (see Contacts.People.Phones, which extends BaseColumns, PhonesColumns, and PeopleColumns), or by listing the column names as constants. Note that you need to know the data type of a column exposed by a content provider in order to be able to read it; the field reading method is specific to the data type, and a column's data type is not exposed programmatically.

The retrieved data is exposed by a Cursor object that can be used to iterate backward or forward through the result set. You can use this cursor to read, modify, or delete rows. Adding new rows requires a different object described later.

Note that by convention, every recordset includes a field named _id, which is the ID of a specific record, and a _count field, which is a count of records in the current result set. These field names are defined by BaseColumns.

Querying for Files

The previous query result demonstrates how a file is returned in a data set. The file field is typically (but not required to be) a string path to the file. However, the caller should never try to read and open the file directly (permissions problems for one thing can make this fail). Instead, you should call ContentResolver.openInputStream() / ContentResolver.openOutputStream(), or one of the helper functions from a content provider.

Reading Retrieved Data

The Cursor object retrieved by the query provides access to a recordset of results. If you have queried for a specific record by ID, this set will contain only one value; otherwise, it can contain multiple values. You can read data from specific fields in the record, but you must know the data type of the field, because reading data requires a specialized method for each type of data. (If you call the string reading method on most types of columns, Android will give you the String representation of the data.) The Cursor lets you request the column name from the index, or the index number from the column name.

If you are reading binary data, such as an image file, you should call ContentResolver.openOutputStream() on the string content:// URI stored in a column name.

The following snippet demonstrates reading the name and phone number from our phone number query:

```
private void getColumnData(Cursor cur){
    if (cur.moveToFirst()) {

        String name;
        String phoneNumber;
        int nameColumn = cur.getColumnIndex(People.NAME);
        int phoneColumn = cur.getColumnIndex(People.NUMBER);
```

```
        String imagePath;

        do {
            // Get the field values
            name = cur.getString(nameColumn);
            phoneNumber = cur.getString(phoneColumn);

            // Do something with the values.
            ...

        } while (cur.moveToNext());

    }
}
```

Modifying Data

To batch update a group of records (for example, to change "NY" to "New York" in all contact fields), call the ContentResolver.update() method with the columns and values to change.

Adding a New Record

To add a new record, call ContentResolver.insert() with the URI of the type of item to add, and a Map of any values you want to set immediately on the new record. This will return the full URI of the new record, including record number, which you can then use to query and get a Cursor over the new record.

```
ContentValues values = new ContentValues();
Uri phoneUri = null;
Uri emailUri = null;

values.put(Contacts.People.NAME, "New Contact");
//1 = the new contact is added to favorites
//0 = the new contact is not added to favorites
values.put(Contacts.People.STARRED,1);

//Add Phone Numbers
Uri uri = getContentResolver().insert(Contacts.People.CONTENT_URI, values);

//The best way to add Contacts data like Phone, email, IM is to
//get the CONTENT_URI of the contact just inserted from People's table,
//and use withAppendedPath to construct the new Uri to insert into.
phoneUri = Uri.withAppendedPath(uri, Contacts.People.Phones.CONTENT_DIRECTORY);

values.clear();
values.put(Contacts.Phones.TYPE, Phones.TYPE_MOBILE);
values.put(Contacts.Phones.NUMBER, "1233214567");
getContentResolver().insert(phoneUri, values);

//Add Email
emailUri = Uri.withAppendedPath(uri, ContactMethods.CONTENT_DIRECTORY);

values.clear();
//ContactMethods.KIND is used to distinguish different kinds of
//contact data like email, im, etc.
values.put(ContactMethods.KIND, Contacts.KIND_EMAIL);
values.put(ContactMethods.DATA, "test@example.com");
values.put(ContactMethods.TYPE, ContactMethods.TYPE_HOME);
getContentResolver().insert(emailUri, values);
```

To save a file, you can call ContentResolver().openOutputStream() with the URI as shown in the following snippet:

```
// Save the name and description in a map. Key is the content provider's
// column name, value is the value to save in that record field.
ContentValues values = new ContentValues(3);
values.put(MediaStore.Images.Media.DISPLAY_NAME, "road_trip_1");
values.put(MediaStore.Images.Media.DESCRIPTION, "Day 1, trip to Los Angeles");
values.put(MediaStore.Images.Media.MIME_TYPE, "image/jpeg");
```

```
// Add a new record without the bitmap, but with the values.
// It returns the URI of the new record.
Uri uri = getContentResolver().insert(MediaStore.Images.Media.EXTERNAL_CONTENT_URI, values);

try {
    // Now get a handle to the file for that record, and save the data into it.
    // sourceBitmap is a Bitmap object representing the file to save to the database.
    OutputStream outStream = getContentResolver().openOutputStream(uri);
    sourceBitmap.compress(Bitmap.CompressFormat.JPEG, 50, outStream);
    outStream.close();
} catch (Exception e) {
    Log.e(TAG, "exception while writing image", e);
}
```

Deleting a Record

To delete a single record, call ContentResolver.delete() with the URI of a specific row.

To delete multiple rows, call ContentResolver.delete() with the URI of the type of record to delete (for example, android.provider.Contacts.People.CONTENT_URI) and a SQL WHERE clause defining which rows to delete (*Warning*: be sure to include a valid WHERE clause if deleting a general type using ContentResolver.delete(), or else you risk deleting more records than you intended!).

Creating a Content Provider

Here is how to create your own content provider to act as a public source for reading and writing a new data type:

1. Extend ContentProvider.
2. Define a `public static final Uri` named CONTENT_URI. This is the string that represents the full "content://" URI that your content provider handles. You must define a unique string for this value; the best solution is to use the fully-qualified class name of your content provider (lowercase). So, for example:
   `public static final Uri CONTENT_URI = Uri.parse( "content://com.google.codelab.rssprovider");`
3. Create your system for storing data. Most content providers store their data using Android's file storage methods or SQLite databases, but you can store your data any way you want, so long as you follow the calling and return value conventions.
4. Define the column names that you will return to your clients. If you are using an underlying database, these column names are typically identical to the SQL database column names they represent. In any case, you should include an integer column named `_id` to define a specific record number. If using the SQLite database, this should be type `INTEGER PRIMARY KEY AUTOINCREMENT`. The `AUTOINCREMENT` descriptor is optional, but by default, SQLite autoincrements an ID counter field to the next number above the largest existing number in the table. If you delete the last row, the next row added will have the same ID as the deleted row. To avoid this by having SQLite increment to the next largest value whether deleted or not, then assign your ID column the following type: INTEGER PRIMARY KEY AUTOINCREMENT. (**Note** You should have a unique _id field whether or not you have another field (such as a URL) that is also unique among all records.) Android provides the SQLiteOpenHelper class to help you create and manage versions of your database.
5. If you are exposing byte data, such as a bitmap file, the field that stores this data should actually be a string field with a content:// URI for that specific file. This is the field that clients will call to retrieve this data. The content provider for that content type (it can be the same content provider or another content provider — for example, if you're storing a photo you would use the media content provider) should implement a field named _data for that record. The _data field lists the exact file path on the device for that file. This field is not intended to be read by the client, but by the ContentResolver. The client will call ContentResolver.openOutputStream() on the user-facing field holding the URI for the item (for example, the column named photo might have a value content://media/images/4453). The ContentResolver will request the _data field for that record, and because it has higher permissions than a client, it should be able to access that file directly and return a read wrapper for that file to the client.
6. Declare public static Strings that clients can use to specify which columns to return, or to specify field values from the cursor. Carefully document the data type of each field. Remember that file fields, such as audio or bitmap fields, are typically returned as string path values
7. Return a Cursor object over a recordset in reply to a query. This means implementing the query(), update(), insert(), and delete() methods. As a courtesy, you might want to call ContentResolver.notifyChange() to notify listeners about updated information.
8. Add a `<provider>` tag to AndroidManifest.xml, and use its *authorities* attribute to define the authority part of the content

type it should handle. For example, if your content type is content://com.example.autos/auto to request a list of all autos, then *authorities* would be `com.example.autos`. Set the *multiprocess* attribute to true if data does not need to be synchronized between multiple running versions of the content provider.

9. If you are handling a new data type, you must define a new MIME type to return for your implementation of android.ContentProvider.getType(url). This type corresponds to the `content://` URI submitted to getType(), which will be one of the content types handled by the provider. The MIME type for each content type has two forms: one for a specific record, and one for multiple records. Use the Uri methods to help determine what is being requested. Here is the general format for each:
   - `vnd.android.cursor.item/vnd.`*`yourcompanyname.contenttype`* for a single row. For example, a request for train record 122 using

     ```
     content://com.example.transportationprovider/trains/122
     ```

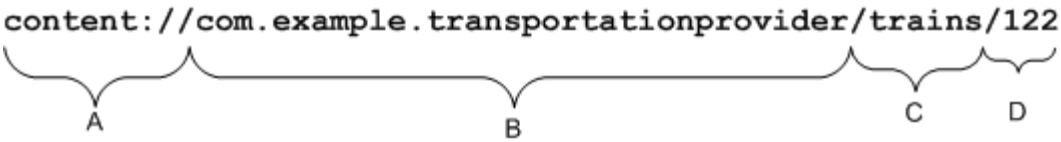
     might return the MIME type `vnd.android.cursor.item/vnd.example.rail`
   - `vnd.android.cursor.dir/vnd.`*`yourcompanyname.contenttype`* for multiple rows. For example, a request for all train records using

     ```
     content://com.example.transportationprovider/trains
     ```

     might return the MIME type `vnd.android.cursor.dir/vnd.example.rail`

For an example of a private content provider implementation, see the NodePadProvider class in the notepad sample application that ships with the SDK.

Here is a recap of the important parts of a content URI:



A. Standard required prefix. Never modified.
B. Authority part. For third-party applications, this should be a fully-qualified class to ensure uniqueness. This corresponds to the value in the `<provider>` element's *authorities* attribute: `<provider class="TransportationProvider" authorities="com.example.transportationprovider" />`
C. The path that the content provider uses to determine what kind of data is being requested. This can be zero or more segments: if the content provider exposes only one type of data (only trains, for example), this can be absent. If it provides several types, including subtypes, this can be several elements long: e.g., "`land/bus`, `land/train`, `sea/ship`, and `sea/submarine`" to give four possibilities.
D. A specific record being requested, if any. This is the _id value of a specific record being requested. If all records of a specific type are being requested, omit this and the trailing slash: `content://com.example.transportationprovider/trains`

 Build 110632-110632 - 22 Sep 2008 13:34

Android

Network Accesses with Android

In addition to all the on-device storage options, you can also store and retrieve data from the network (when available). To do network operations, you'll want to use the following packages:

- `java.net.*`
- `android.net.*`

Build 110632-110632 - 22 Sep 2008 13:34