안드로이드 애플리케이션 교육 자료

Android Application Development

2009. 2. 19. ~ 2. 20

www.kandroid.org 관리자 : 양정수 (yangjeongsoo@gmail.com), 닉네임:들풀

목 차

안드로이드 구조


APPLICATIONS
Alarm
Dialer
SMS/MMS
IM
Browser
Camera
Alarm
Home
Contacts
Voice Dial
Email
Calendar
Media Player
Albums
Clock
...
APPLICATION FRAMEWORK
Activity Manager
Window Manager
Content Provider
View System
Notification Manager
Package Manager
Telephony Manager
Resource Manager
Location Manager
...
LIBRARIES
Surface Manager
Media Framework
SQLite
OpenGL|ES
FreeType
WebKit
SGL
SSL
Libc
ANDROID RUNTIME
Core Libraries
Dalvik Virtual Machine
HARDWARE ABSTRACTION LAYER
Graphics
Audio
Camera
Bluetooth
GPS
Radio(RIL)
WiFi
...
LINUX KERNEL
Display Driver
Camera Driver
Bluetooth Driver
Shared Memory Driver
Binder (IPC) Driver
USB Driver
Keypad Driver
WiFi Driver
Audio Driver
Power Management

안드로이드 구조 : 리눅스 커널

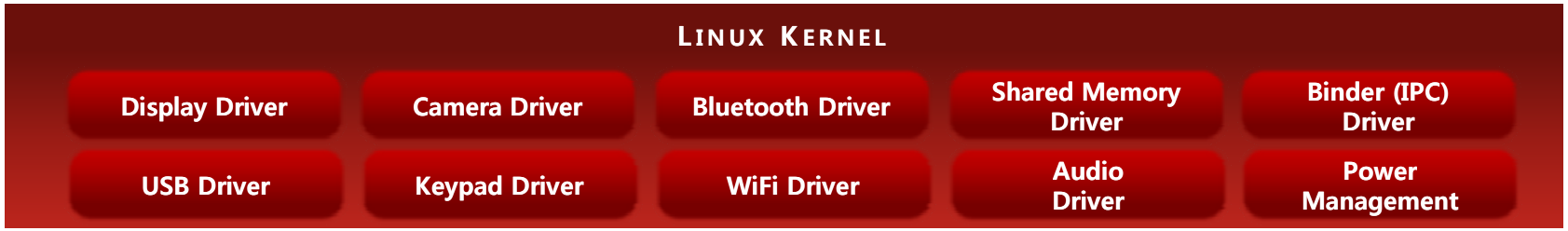
안드로이드는 리눅스 커널을 기반으로 하고 있으나, 안드로이드는 리눅스는 아니다.
안드로이드는 X-Window와 같은 내장 윈도우 시스템을 포함하지 않는다.
안드로이드는 glibc를 지원하지 않는다.
안드로이드는 표준 리눅스 유틸리티 전체를 포함하고 있지 않는다.

안드로이드는 리눅스 커널 버전 2.6.23, 2.6.24, 2.6.25, 2.6.27 을 사용해 왔다.
안드로이드를 지원하기 위해, 리눅스 커널 확장을 위한 패치를 포함하고 있다.

안드로이드에서 리눅스를 사용하는 이유는 메모리 및 프로세스 관리, 인가(Permission) 기반의 보안 모델, 검증된 드라이버 모델, 공유 라이브러리 지원, 오픈 소스 기반 등의 장점 때문이다.

안드로이드를 위해 확장된 리눅스 커널 영역은, Alarm, Ashmem, Binder, Power Management, Low Memory Killer, Kernel Debugger, Logger 이다.

안드로이드 리눅스 커널 소스는 현재 아래의 URL에서 배포되고 있다.
http://git.android.com



안드로이드 구조 : 내장 라이브러리

안드로이드 내장 라이브러리는, Bionic Libc, Function Library, Native Server, Hardware Abstraction Library로 구성된다.

Bionic 은 임베디드에서 사용을 위해 최적화된 직접 구현된 libc 이다. libc를 직접 구현한 이유는, user-space에서는 GPL 을 사용하지 않기 위한 라이선스 이유와, 개별 프로세스마다 포함되어야 하는 영역이기 때문에 크기를 최소화하기 위한 이유와, 제한적 CPU 파워하에서도 빠를 필요가 있기 때문이었다.

Function Library에는 웹 브라우저를 위한 WebKit, PacketVideo의 OpenCORE 플랫폼 기반의 미디어 프레임워크, 가벼운 데이타베이스인 SQLite 가 있다.

Native Server에는 Surface Flinger와 Audio Flinger가 있다. Surface Flinger는 2D와 3D Surface의 조합 및 다양한 애플리케이션에서 사용된 Surface들을 Frame buffer 디바이스로의 렌더링을 제어함으로써, 시스템 전역에 걸쳐서 surface 의 구성을 가능하게 한다. Audio Flinger는 모든 오디오 아웃풋 디바이스를 제어하는 것으로써, 다양한 오디오 스트림을 PCM 오디오 아웃풋 경로로 처리하며, 다양한 아웃풋으로 오디오를 제공하는 역할을 담당한다.

Hardware Abstraction Layer에 대해서는 다음 페이지를 통해 설명하기로 한다.

안드로이드 구조 : 하드웨어 추상화 계층

Hardware Abstraction Library는 User space의 C/C++ 라이브러리 계층으로써, 안드로드에서 요구되는 하드웨어 드라이버의 구현에 대한 인터페이스를 정의한다. 더불어 하드웨어 인터페이스로 부터 안드로이드 플랫폼의 로직을 분리하는데 사용된다.

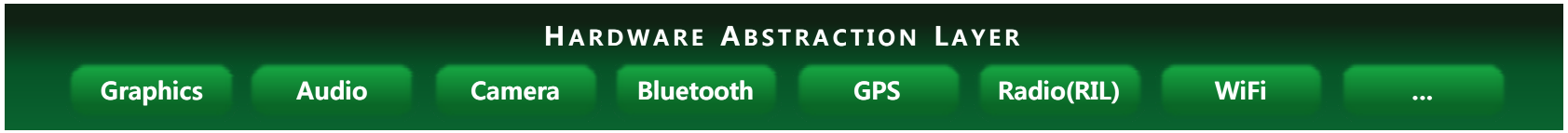
User-pace의 HAL이 필요한 이유는, 모든 컴포넌트들이 표준화된 리눅스 커널 드라이버 인터페이스를 가지고 있지 않기 때문이며, 리눅스 드라이버들은 사적인 지적소유권을 공개할 수 밖에 없는 GPL 기반이란 이유 때문이다. 또한 안드로이드는 하드웨어 드라이버들을 위한 별도의 요구사항을 가지고 있기 때문이기도 하다.

HAL Header는 다음과 같은 예로 존재한다

```
typedef struct {
        int  (*foo)( void );
        char (*bar)( void);
        …
} AcmeFunctions;
const AcmeFunctions *Acme_Init(const struct Env *env, int argc, char **argv);
```

더불어, HAL 라이브러리들은 필요에 따라 아래와 같이 런타임에서 다이나믹하게 로딩된다.

```
dlHandle = dlopen("/system/lib/libacme.so", RTLD_NOW);
…
acmeInit = (const AcmeFuctions *(*)(const struct Env *, init, char **)dlsym(dlHandle, "Acme_Init");
…
acmeFuncs = acmeInit(&env, argc, argv);
```



안드로이드 구조 : 안드로이드 런타임

안드로이드 런타임은, 안드로이드에서 사용되는 Dalvik 가상 머신과 Core 라이브러리들로 구성된다.

Dalvik 가상머신은 안드로이드에서 자체적으로 만든 것으로 clean-room(청정영역)을 제공하는 가상 머신이다. 이것은, 애플리케이션의 호환성(Portability)과 실행 일관성(runtime consistency)을 제공하며, 최적화된 파일 포맷(.dex)과 Dalvik 바이트 코드를 실행한다. 더불어 빌드 시점에서 Java .class / .jar 파일들은 .dex 파일로 변환된다.

Dalvik 가상머신은 임베디드 환경을 위해 디자인 되었다. Dalvik은 디바이스 별로 다양한 가상머신 프로세스들을 지원하며, 높은 수준으로 CPU에 최적화된 바이트코드 인터프리터에 기반하며, 실행시 메모리를 매우 효율적으로 사용한다.

Cora 라이브러리들은 강력하지만, 단순하고 익숙한 개발 플랫폼을 제공하는 Java 언어를 위한 Core API들을 포함하고 있다. 여기에는 Data structure, Utility, File Access, Network Access, Graphic 등이 포함되어 있다.

안드로이드 구조 : 안드로이드 런타임

Dalvik Virtual Machine

- Register 기반의 가상머신 (no JIT, interpreter-only)
- Optimized for low memory requirements
- Designed to allow multiple VM instances to run at one
- Relying on underlying OS for process isolation, memory management and threading support
- Executes Dalvik Executables (DEX) files which are zipped into an Android Package (APK)

Core libraries

- Bundled in android.jar
- Android platform library: android.*.
  - XML Parser implementations included (DOM, SAX, XMLPullParser)
- Apache Harmony (Standrard Java Library implementation)
- Several popular Open Source projects available out-of-the-box.
  - Apache Commons (HttpClients 2/4, Codec)
  - BouncyCastle JCE providers
  - SQLite, JUnit 3.8.x
- GData APIs partly included as wireless version
  - com.google.wireless.gdata

Dalvik Virtual Machine

Memory Efficiency
- Android Memory Usage
- Dex File Anatomy
- Shard Constant Pool
- Size Comparison
- 4 Kinds of Memory
- Enter the Zygote
- GC and Sharing

CPU Efficiency
- Android CPU Usage
- No JIT?
- Install-Time Work
- Register Machine
- Interpreters

Optimizing Your Code
- Time Scale
- Get Plenty Of Rest
- Loop Wisely
- Avoid Allocation

Dalvik이란 이름은, Dalvik의 창시자인 본스타인이 자신의 조상이 살던 아이슬란드의 한 어촌의 이름을 따서 만든 것임.

Core Libraries

안드로이드 Core 라이브러리는 Java Standard Edition과 Java Mobile Edition과는 다르지만, 중복되는 부분이 상당히 있다.

J2SE 5.0 Supported	J2SE 5.0 Not supported	3rd Party libraries	Android Specific libraries	
java.awt.font java.io java.lang java.math java.net java.nio java.security java.sql java.text java.util javax.crypto javax.microedition.khronos javax.net javax.security javax.sql javax.xml.parsers org.w3c.dom org.xml.sax	java.applet java.awt java.beans java.lang.management java.rmi javax.accessibility javax.activity javax.imageio javax.management javax.naming javax.print javax.rmi javax.security.auth.kerb eros javax.security.auth.spi javax.security.sasl javax.sound javax.swing javax.transaction javax.xml org.ietf.* org.omg.* org.w3c.dom.*	org.apache.http org.json org.xml.sax org.xmlpull.v1	android android.app android.content android.content.pm android.content.res android.database android.database.sqlite android.graphics Provides android.graphics.drawable android.graphics.drawable.shapes android.hardware android.location android.media android.net android.net.http android.net.wifi android.opengl android.os	android.preference android.provider android.sax android.telephony android.telephony.gsm android.test android.test.mock android.test.suitebuilder android.text android.text.method android.text.style android.text.util android.util android.view android.view.animation android.webkit android.widget com.google.android.maps dalvik.bytecode dalvik.system

안드로이드 구조 : 애플리케이션 프레임워크

안드로이드 애플리케이션 프레임워크는 Java 기반의 Framework 이며, 대부분이 JNI(Java Native Interface) 통해 native C/C++ 코드로 작성되어 있다. 더불어, 아래와 같이 핵심 시스템 서비스를 담당하는 Core 시스템 서비스들과 하드웨어와의 인터페이스를 담당하는 하드웨어 서비스들로 구성된다.

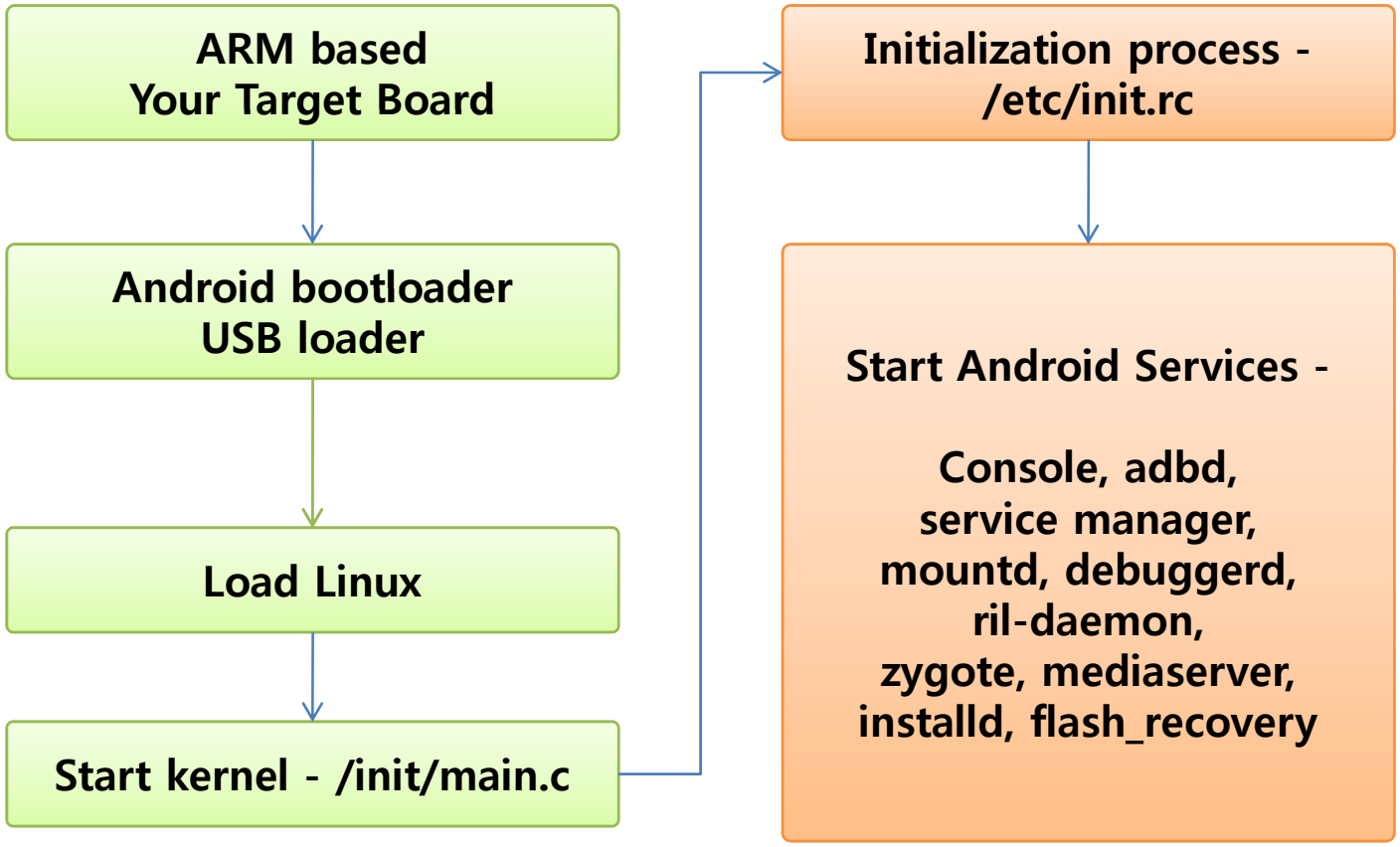
Core System services

- Activity manager (manages application lifecycle)
- Package manager (loads apk files)
- Window manager (handles applications window manager interaction with surface flinger)
- Resource manager (handles media resources)
- Content providers (provides data to application)
- View system (provides widgets, views, layouts to applications)

Hardware services

- Provides low-level access to hardware device
- Location manager
- Telephony manager
- Bluetooth service
- WiFi service
- USB service
- Sensor service

Overview


ARM based
Your Target Board
Android bootloader
USB loader
Load Linux
Start kernel - /init/main.c
Initialization process -
/etc/init.rc
Start Android Services -
Console, adbd,
service manager,
mountd, debuggerd,
ril-daemon,
zygote, mediaserver,
installd, flash_recovery

안드로이드 구조 : 안드로이드 구동

Zygote

App. Framework
• System Services
• Hardware Services

Dalvik VM

Native Server
• Audio Flinger
• Surface Flinger

System Server

libc

Home

Dalvik VM

Home

libc

daemons
• usbd
• adbd
• debuggerd
• rild

libc

Service Manager

runtime

libc

Zygote

libc

Init

Kernel

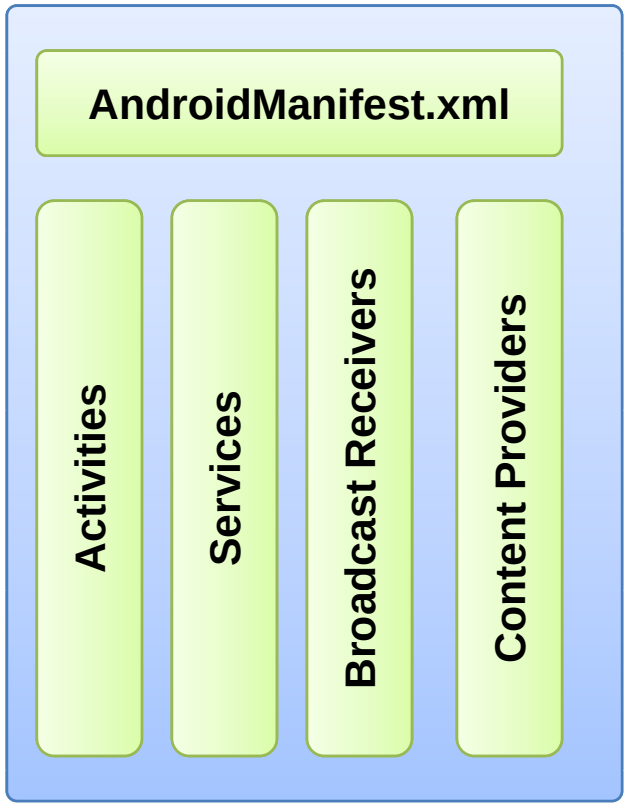
Binder Driver

zygote란 애플리케이션을 빠르게 구동하기 위해서 미리 fork 되어 있는 프로세스이다.
이것은 시스템에서 exec() 호출을 통해 특정 애플리케이션을 실행하고자 하기 전까지는 중립적인 상태, 즉 특정 애플리케이션과 합체되지 않는 상태를 유지한다.

목 차

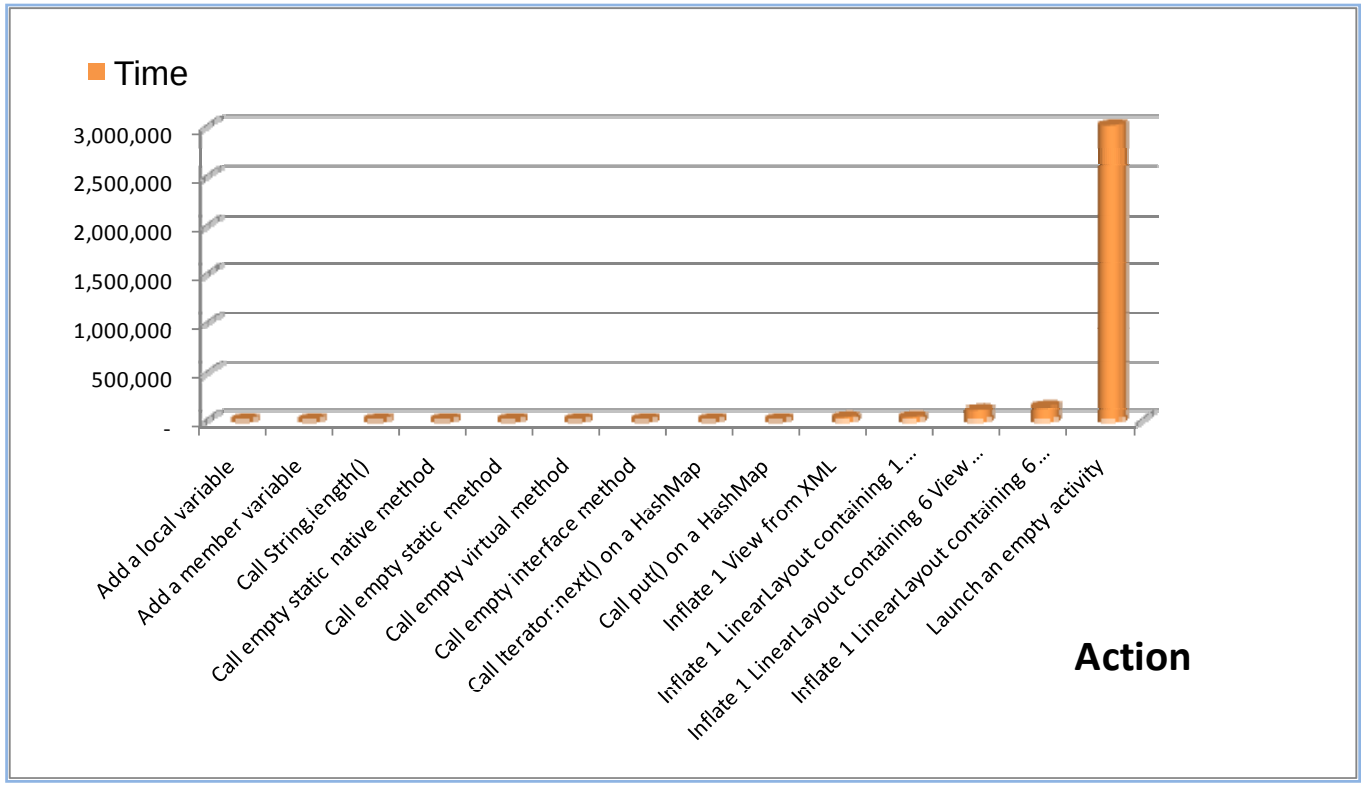
애플리케이션 개요 : 구성요소

안드로이드 애플리케이션은 신속성, 응답성, 무결절성을 디자인 철학으로 하고 있으며, 애플리케이션은 왼쪽 하단의 그림과 같은 구성요소를 가진다.



Performance

애플리케이션의 성능을 보장하기 위해서, 무엇보다 중요한 것은 효율적인 코드 작성임. 효율적인 코드란, 메모리 할당을 최소화, 코드 최소화, 프로그래밍 언어적 특성을 잘 반영하는 것임. 아래의 도표는 다른 모든 것보다 안드로이드에서 Activity를 생성하는 것이 오버헤드가 됨을 보여주는 통계표임.

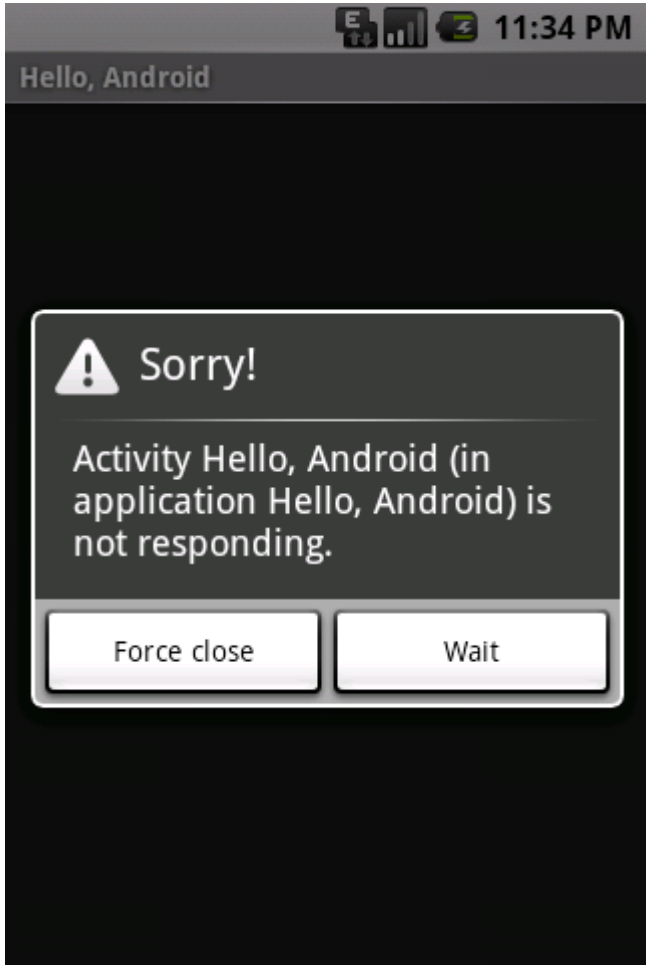


Responsiveness

안드로이드에서 빈번하게 볼 수 있는 "ANR(Application Not Responding)" 이란 메시지는 안드로이드 디자인 철학 중 하나인, 응답성과 관련되어 있음.

5초이내에 사용자 input event에 대한 response가 없거나, 10초 이내에 BroadcastReceiver가 종료되지 않을 때, 안드로이드는 ANR을 사용자에게 보여줌으로써, 해당 Job을 강제로 종료할 수 있는 기능을 제공함.

ANR의 발생을 최소화하기 위해서는 Thread를 효율적으로 활용할 수 있어야 함.



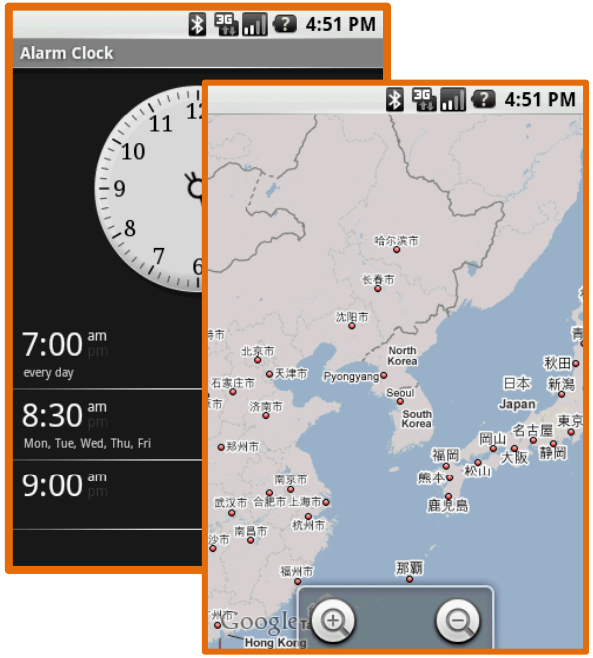
Seamlessness

안드로이드에서 무결절성 개념은 사용자들이 애플리케이션을 부드럽게 전환해 나갈 수 있도록 해주는 것이 목적이다. 이를 위해서 Notification 같은 구조를 사용하는 것과 더불어 Task를 중심으로 애플리케이션들간의 경계를 허물어 나갈 수 있는 구조를 지원하고 있는 것이 하나의 큰 특징에 속한다.

Don't Drop Data
Don't Expose Raw Data
Don't Interrupt the User
Got a Lot to Do? Do it in a Thread
Don't Overload a Single Activity Screen
Extend System Themes
Design Your UI to Work with Multiple Screen Resolutions
Assume the Network is Slow
Don't Assume Touchscreen or Keyboard
Do Conserve the Device Battery

Activities

Activity는 하나의 가상의 사용자 인터페이스에 대한 표현이다.



- Can be faceless
- Can be in a floating window
- Can return a value
- Can be embedded

Services

- Faceless classes that run in the background

  Example : Music player, network download, etc.

- Services run in your application's process or their own process
- Your code can bind to Services in your process or another process
- Once bound, you communicate with Services using a remote-able interface defined in IDL

Broadcast receivers

A broadcast receiver is a component that does nothing but receive and react to broadcast announcements. Many broadcasts originate in system code – for example, announcements that the timezone has changed, that the battery is low, that a picture has been taken, or that the user changed a language preference. Applications can also initiate broadcasts – for example, to let other applications know that some data has been downloaded to the device and is available for them to use.

An application can have any number of broadcast receivers to response to any announcements it considers important. All receivers extend the **BroadcastReceiver** base class.

Broadcast receivers do not display a user interface. However, they may start an activity in response to the information they receive, or they may use the **NotificationManager** to alert the user. Notifications can get the user's attention in various ways – flashing the backlight, vibrating the device, playing a sound, and so on. They typically place a persistent icon in the status bar, which user can open to get the message

Content Providers

- Enables sharing of data across applications

  Examples : address book, photo gallery, etc.

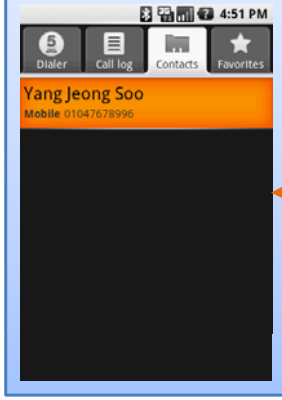
- Provides uniform API for :

  Querying (return a Cursor)

  delete, update, and insert rows

- Content is represented by URI and MIME type

다양한 Content Provider 가 존재함.



CalendarProvider
ContactsProvider
DownloadProvider
DrmProvider
GoogleContactsProvider
GoogleSubscribedFeedsProvider
ImProvider
MediaProvider
TelephonyProvider

Activating components : intents

Content providers are activated when they're targeted by a request from a ContentResolver. The other three components – activities, services, and broadcast receivers – are activated by asynchronous messages called intents. An intent is an Intent object that holds the content of the message. For activities and services, it names the action being requested and specifies the URI of the data to act on, among other things. For example, it might convey a request for an activity to present an image to the user of let the user edit some text. For broadcast receivers, the Intent object names the action being announced. For example, it might announce to interested parties that the camera button has been pressed.

There are separate methods for activating each type of component:

An activity is launched (or given something new to do) by passing an Intent object to Context.startActivity() or Activity.startActivityForResult().

A service is started (or new instructions are given to an ongoing service) by passing an Intent object to Context.startService(). Android calls the service's onStart() method and passes it the Intent Object.

An application can initiate a broadcast by passing an Intent object to methods like Context.sendBroadcast(), Context.sendOrderedBroadcast(), and Context.sendStickyBroadcast() in any of their variations.

Shutting down components

A content provider is active only while it's responding to a request form a ContentResolver. And a broadcast receiver is active only while it's responding to a broadcast message. So there's no need to explicitly shut down these components.

Activities, on the other hand, provide the user interface. They're in a long-running conversation with the user and may remain active, even when idle, as long as the conversation continues. Similarly, services may also remain running for a long time. So Android has methods to shut down activities and services in an orderly way:

An activity can be shut down by calling its finish() method. One activity can shut down another activity (one it start with startActivityForResult()) by calling finishActivity().

A service can be stopped by calling its stopSelf() method, or by calling Context.stopService().

The manifest file

Before Android can start application component, it must learn that the component exist. Therefore, applications declare their components in a manifest file that's bundled into the Android package, the .apk file that also holds the application's code, files, and resources.

The manifest is a structured XML file and is always named AndroidManifest.xml for all applications. It does a number of things in addition to declaring the application's components, such as naming any libraries the application needs to be linked against (besides the default Android library) and identifying any permissions the application expects to be granted.

But the principal task of the manifest is to inform Android about the application's components. For example, an activity might be declared as follows:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest . . . >
    <application . . . >
        <activity android:name="com.example.project.FreneticActivity"
                  android:icon="@drawable/small_pic.png"
                  android:label="@string/freneticLabel" . . . >
        </activity>
        . . .
    </application>
</manifest>
```

AndroidManifest.xml

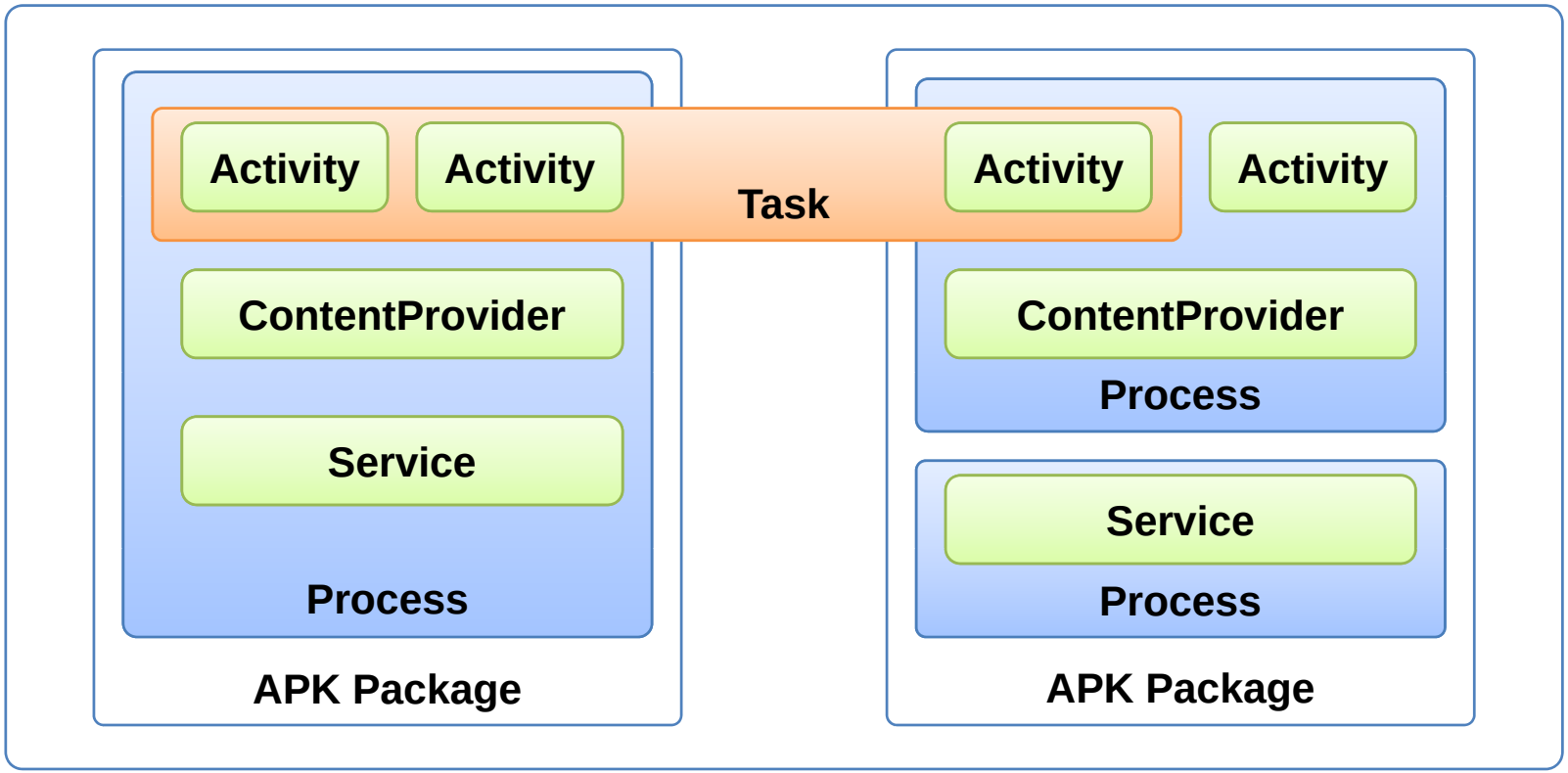
```
<manifest>
        <instrumentation>
        <uses-sdk>
        <uses-permission>
        <permission>
        <permission-group>
        <permission-tree>
        <application>
                <uses-library>
                <activity>
                <activity-alias>
                <provider>
                        <grant-uri-permission>
                <receiver>
                <service>
                        <intent-filter>
                                <action>
                                <category>
                                <data>
                        <meta-data>
```

Intent filters

```
<?xml version="1.0" encoding="utf-8"?>
<manifest . . . >
   <application . . . >
      <activity android:name="com.example.project.FreneticActivity"
                android:icon="@drawable/small_pic.png"
                android:label="@string/freneticLabel"
                . . . >
         <intent-filter . . . >
            <action android:name="android.intent.action.MAIN" />
            <category android:name="android.intent.category.LAUNCHER" />
         </intent-filter>
         <intent-filter . . . >
            <action android:name="com.example.project.BOUNCE" />
            <data android:type="image/jpeg" />
            <category android:name="android.intent.category.DEFAULT" />
         </intent-filter>
      </activity>
      . . .
   </application>
</manifest>
```

애플리케이션 개요 : Activities, Tasks, and Processes

An Activity is a “molecule”: a discrete chunk of functionality
A Task is a collection of Activities
A “processes” is a standard Linux process



Affinities and new tasks

The FLAG_ACTIVITY_NEW_TASK flag

As described earlier, a new activity is, by default, launched into the task of the activity that called startActivity(). It's pushed onto the same stack as the caller. However, if the Intent object passed to startActivity() contains the FLAG_ACTIVITY_NEW_TASK flag, the system looks for a different task to house the new activity. Often, as the name of the flag implies, it's a new task. However, it doesn't have to be. If there's already an existing task with the same affinity as the new activity, the activity is launched into that task. If not, it begins a new task.

The allowTaskReparenting attribute

If an activity has its allowTaskReparenting attribute set to "true", it can move from the task it starts in to the task it has an affinity for when that task comes to the fore. For example, suppose that an activity that reports weather conditions in selected cities is defined as part of a travel application. It has the same affinity as other activities in the same application (the default affinity) and it allows reparenting. One of your activities starts the weather reporter, so it initially belongs to the same task as your activity. However, when the travel application next comes forward, the weather reporter will be reassigned to and displayed with that task.

Launch modes

Which task will hold the activity that responds to the intent. For the "standard" and "singleTop" modes, it's the task that originated the intent (and called startActivity()) ? unless the Intent object contains the FLAG_ACTIVITY_NEW_TASK flag. In that case, a different task is chosen as described in the previous section, Affinities and new tasks.

Whether there can be multiple instances of the activity. A "standard" or "singleTop" activity can be instantiated many times. They can belong to multiple tasks, and a given task can have multiple instances of the same activity.

Whether the instance can have other activities in its task. A "singleInstance" activity stands alone as the only activity in its task. If it starts another activity, that activity will be launched into a different task regardless of its launch mode ? as if FLAG_ACTIVITY_NEW_TASK was in the intent. In all other respects, the "singleInstance" mode is identical to "singleTask".

Whether a new instance of the class will be launched to handle a new intent. For the default "standard" mode, a new instance is created to respond to every new intent. Each instance handles just one intent. For the "singleTop" mode, an existing instance of the class is re-used to handle a new intent if it resides at the top of the activity stack of the target task. If it does not reside at the top, it is not re-used. Instead, a new instance is created for the new intent and pushed on the stack.

Clearing the stack

The alwaysRetainTaskState attribute

If this attribute is set to "true" in the root activity of a task, the default behavior just described does not happen. The task retains all activities in its stack even after a long period.

The clearTaskOnLaunch attribute

If this attribute is set to "true" in the root activity of a task, the stack is cleared down to the root activity whenever the user leaves the task and returns to it. In other words, it's the polar opposite of alwaysRetainTaskState. The user always returns to the task in its initial state, even after a momentary absence.

The finishOnTaskLaunch attribute

This attribute is like clearTaskOnLaunch, but it operates on a single activity, not an entire task. And it can cause any activity to go away, including the root activity. When it's set to "true", the activity remains part of the task only for the current session. If the user leaves and then returns to the task, it no longer is present.

Starting tasks

An activity is set up as the entry point for a task by giving it an intent filter with "android.intent.action.MAIN" as the specified action and "android.intent.category.LAUNCHER" as the specified category. A filter of this kind causes an icon and label for the activity to be displayed in the application launcher, giving users a way both to launch the task and to return to it at any time after it has been launched.

This second ability is important: Users must be able to leave a task and then come back to it later. For this reason, the two launch modes that mark activities as always initiating a task, "singleTask" and "singleInstance", should be used only when the activity has a MAIN and LAUNCHER filter.

A similar difficulty attends the FLAG_ACTIVITY_NEW_TASK flag. If this flag causes an activity to begin a new task and the user presses the HOME key to leave it, there must be some way for the user to navigate back to it again. Some entities (such as the notification manager) always start activities in an external task, never as part of their own, so they always put FLAG_ACTIVITY_NEW_TASK in the intents they pass to startActivity(). If you have an activity that can be invoked by an external entity that might use this flag, take care that the user has a independent way to get back to the task that's started.

For those cases where you don't want the user to be able to return to an activity, set the <activity> element's finishOnTaskLaunch to "true".

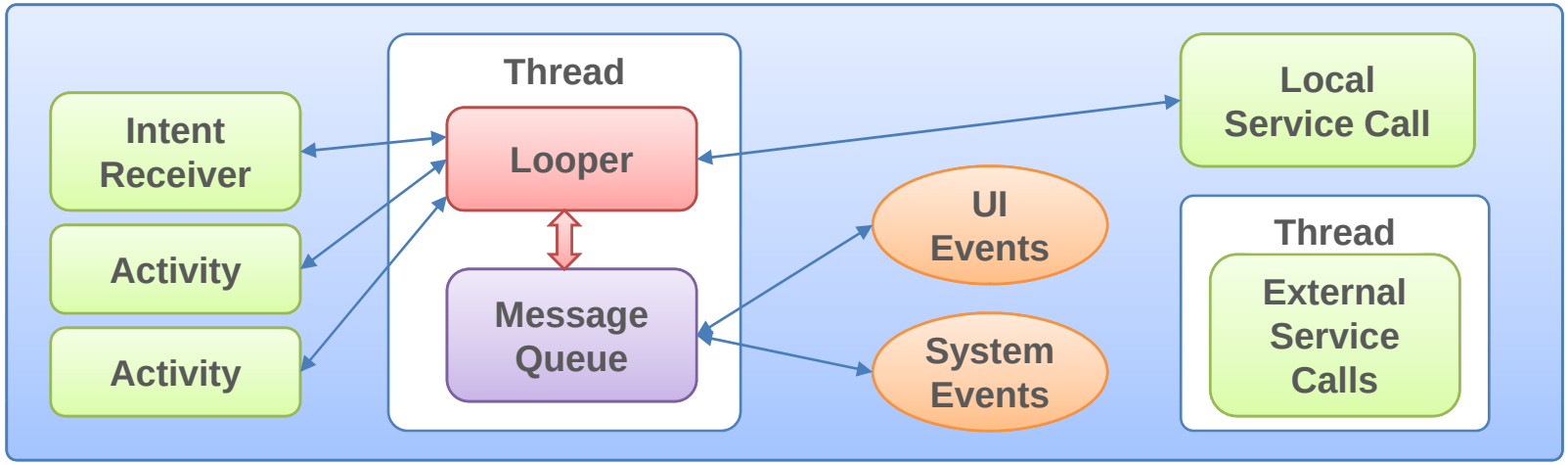
Processes

The process where a component runs is controlled by the manifest file.
The component elements - <activity>, <service>, <receiver>, and <provider> - each have a process attribute that can specify a process where that component should run. These attributes can be set so that each component runs in its own process, or so that some components share a process while others do not. They can also be set so that components of different applications run in the same process - provided that the applications share the same Linux user ID and are signed by the same authorities. The <application> element also has a process attribute, for setting a default value that applies to all components.

Threads

Even though you may confine your application to a single process, there will likely be times when you will need to spawn a thread to do some background work. Since the user interface must always be quick to respond to user actions, the thread that hosts an activity should not also host time-consuming operations like network downloads. Anything that may not be completed quickly should be assigned to a different thread.

Threads are created in code using standard Java Thread objects. Android provides a number of convenience classes for managing threads - Looper for running a message loop within a thread, Handler for processing messages, and HandlerThread for setting up a thread with a message loop.

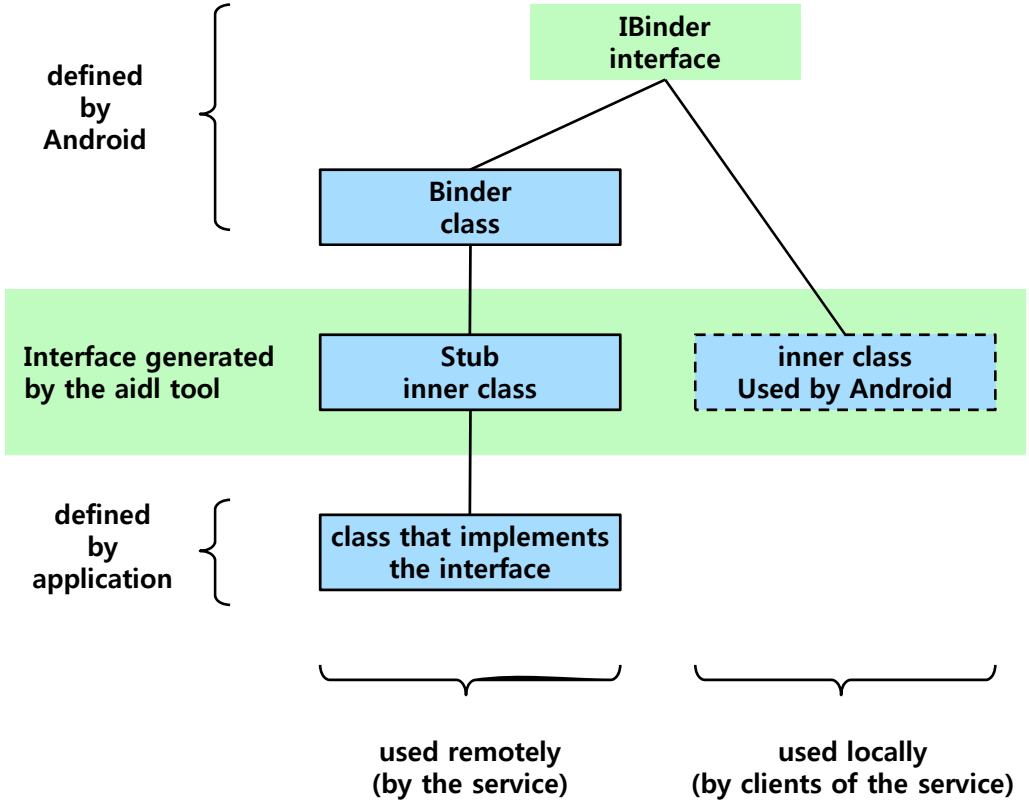


Remote procedure calls

Android has a lightweight mechanism for remote procedure calls (RPCs) - where a method is called locally, but executed remotely (in another process), with any result returned back to the caller.

This entails decomposing the method call and all its attendant data to a level the operating system can understand, transmitting it from the local process and address space to the remote process and address space, and reassembling and reenacting the call there.

...



Thread-safe methods

In a few contexts, the methods you implement may be called from more than one thread, and therefore must be written to be thread-safe.

This is primarily true for methods that can be called remotely - as in the RPC mechanism discussed in the previous section. When a call on a method implemented in an IBinder object originates in the same process as the IBinder, the method is executed in the caller's thread. However, when the call originates in another process, the method is executed in a thread chosen from a pool of threads that Android maintains in the same process as the IBinder; it's not executed in the main thread of the process. For example, whereas a service's onBind() method would be called from the main thread of the service's process, methods implemented in the object that onBind() returns (for example, a Stub subclass that implements RPC methods) would be called from threads in the pool. Since services can have more than one client, more than one pool thread can engage the same IBinder method at the same time. IBinder methods must, therefore, be implemented to be thread-safe.

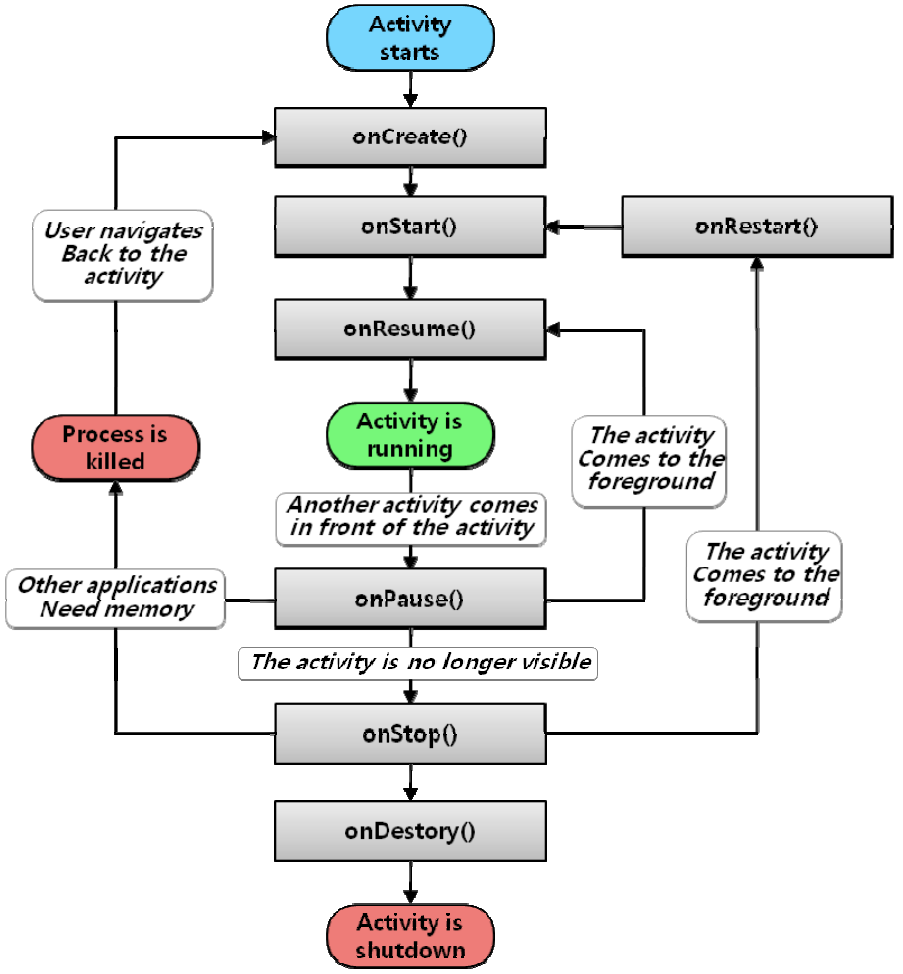
Activity 생명주기

An activity has essentially three states:

It is active or running when it is in the foreground of the screen. This is the activity that is the focus for the user's actions.

It is paused if it has lost focus but is still visible to the user. That is, another activity lies on top of it and that activity either is transparent or doesn't cover the full screen, so some of the paused activity can show through. A paused activity is completely alive, but can be killed by the system in extreme low memory situations.

It is stopped if it is completely obscured by another activity. It still retains all state and member information.



애플리케이션 개요 : 컴포넌트 생명주기

Method	Killable?	Description
onCreate()	No	Activity가 처음 시작될 때 호출되며, 사용자 인터페이스를 만드는 것과 같은 일회적 초기화 작업에 사용된다.
onRestart()	No	Activity가 Stop 되었다가 다시 시작할 때 호출된다.
onStart()	No	Activity가 사용자에게 곧 보여지게 될 것임을 나타낸다
onResume()	No	Activity가 사용자와의 상호작용이 가능할 때 호출되며, 시작 애니메이션이나 음악 등을 시작하기 좋은 위치이다.
onPause()	Yes	Activity가 background로 전환될 때 호출되며, 주로 아직 저장되지 않는 변경 정보들을 persistent data에 저장하는데 사용될 수 있다.
onStop()	Yes	Activity가 더 이상 사용자에게 보여지지 않을 때 호출된다.
onDestory()	Yes	Activity가 소멸되기 직전에 호출된다. 사용자가 finish()를 호출하는 하거나, 시스템이 메모리 공간을 절약하기 위해 Activity를 임시로 종료시키는 과정에서 호출된다.

Saving activity state : onSaveInstanceState(), onRestoreInstanceState()

Service 생명주기

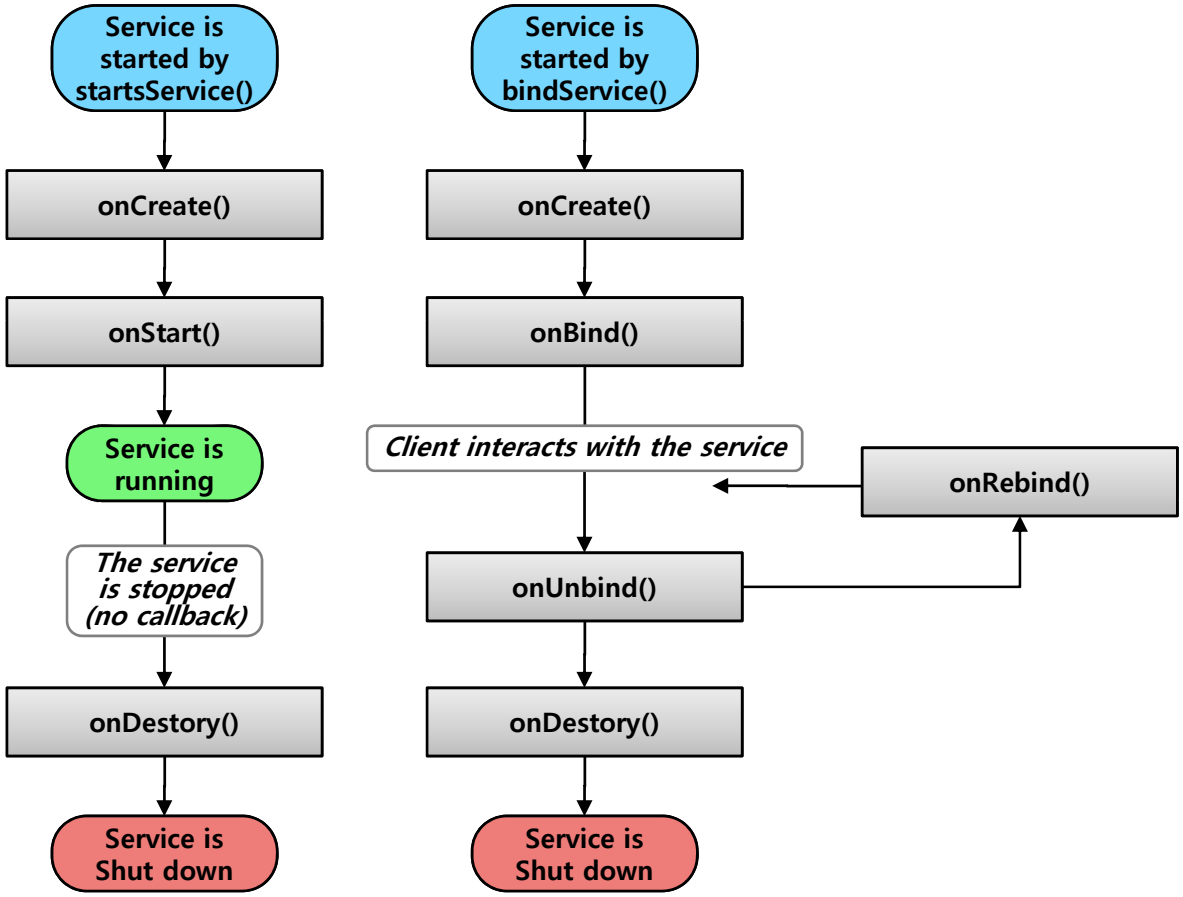
A service can be used in two ways:

Context.startService()

It can be started and allowed to run until someone stops it or it stops itself.

Context.bindService()

It can be operated programmatically using an interface that it defines and exports.



Broadcast receiver 생명주기

A broadcast receiver has single callback method:

void onReceive(Context curContext, Intent broadcastMsg)

When a broadcast message arrives for the receiver, Android calls its onReceive() method and passes it the Intent object containing the message. The broadcast receiver is considered to be active only while it is executing this method. When onReceive() returns, it is inactive.

A process with an active broadcast receiver is protected from being killed. But a process with only inactive components can be killed by the system at any time, when the memory it consumes is needed by other processes.

This presents a problem when the response to a broadcast message is time consuming and, therefore, something that should be done in a separate thread, away from the main thread where other components of the user interface run. If onReceive() spawns the thread and then returns, the entire process, including the new thread, is judged to be inactive (unless other application components are active in the process), putting it in jeopardy of being killed. The solution to this problem is for onReceive() to start a service and let the service do the job, so the system knows that there is still active work being done in the process.

The next section has more on the vulnerability of processes to being killed.

Process와 생명주기

A foreground process is one that is required for what the user is currently doing.

A visible process is one that doesn't have any foreground components, but still can affect what the user sees on screen.

A service process is one that is running a service that has been started with the startService() method and that does not fall into either of the two higher categories.

A background process is one holding an activity that's not currently visible to the user (the Activity object's onStop() method has been called).

An empty process is one that doesn't hold any active application components.

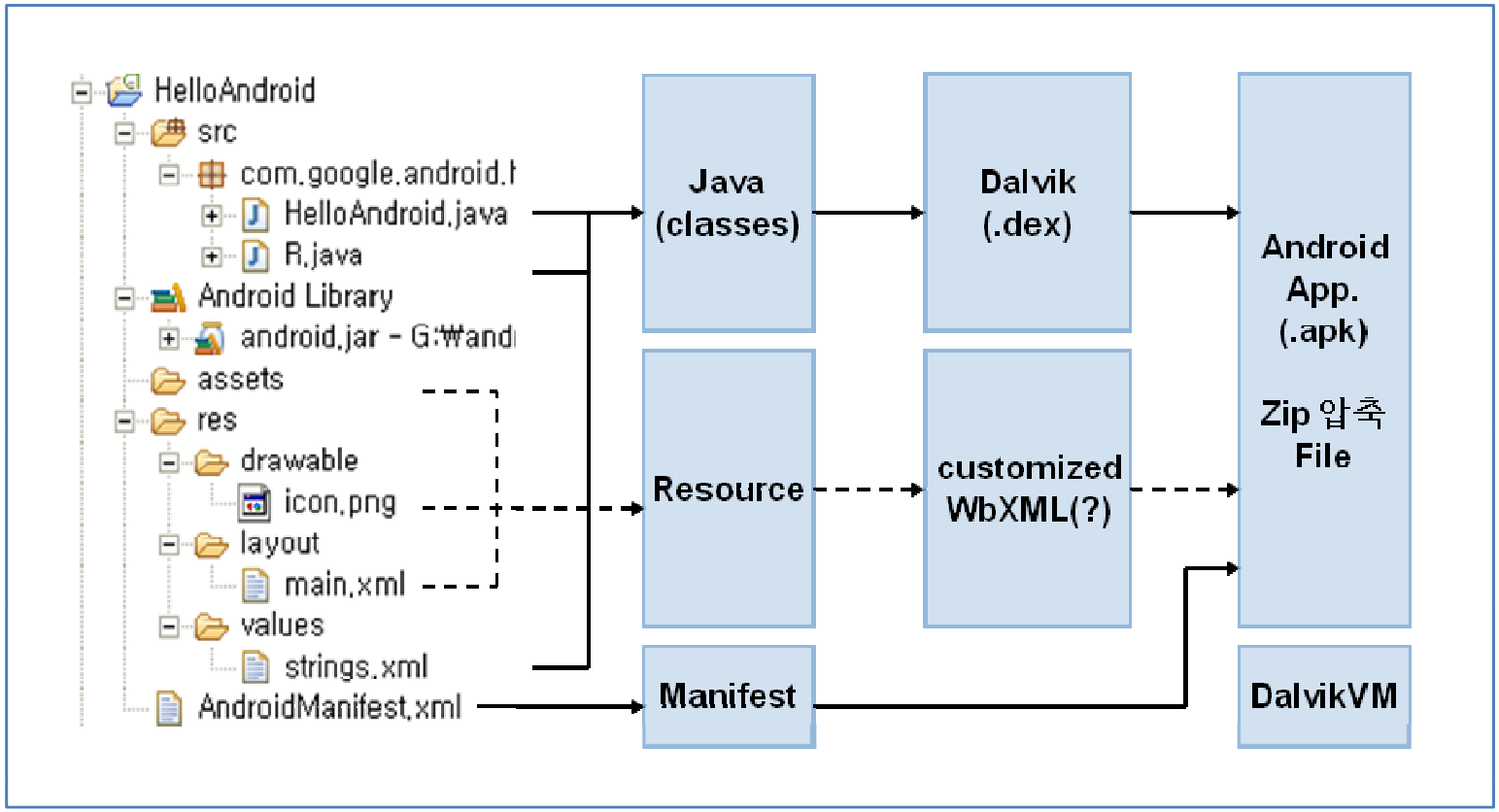
애플리케이션 개요 : 컴포넌트 생명주기

foreground process	foreground 프로세스는 사용자와 상호작용을 하고 있는 스크린의 최상위에 있는 Activity나 현재 수행되고 있는 IntentReceiver를 점유하고 있는 프로세스이다. 시스템에는 매우 작은 수의 그러한 프로세스들이 존재할 뿐이며, 이런 프로세스가 계속 실행 되기조차 어려운 최후의 메모리 부족 상태에서만 종료된다. 일반적으로 디바이스가 메모리 페이징 상태에 도달하는 시점에, 사용자 인터페이스에 대한 응답을 처리하기 위해서 그러한 행위가 요구된다.
visible process	visible 프로세스는 사용자의 화면상에는 나타나지만 foreground 상태는 아닌 Activity를 점유하는 프로세스이다. 예를 들어 foreground activity 다이얼로그 형태로 그 뒤에 이전에 보여졌던 activity를 허용하면서 표시될 때 이러한 것은 발생하게 된다. 그러한 프로세스는 극도록 중요하게 고려되며 더 이상 그것을 수행하지 않을 때까지 종료되지 않으며, 모든 foreground 프로세스들을 실행 상태로 유지하는 것이 요구된다.
service process	service 프로세스는 startService() 메쏘드를 가지고 시작된 Service를 점유하고 있는 프로세스이다. 이러한 프로세스는 사용자에게 직접적으로 보여지는 않지만, 이것은 일반적으로 사용자와 관련된 어떤 일을 일반적으로 수행하며, 시스템이 모든 foreground와 visible 프로세스를 보유하기에 충분한 메모리가 존재하는 한, 시스템은 그러한 프로세스들은 항상 실행상태로 유지할 것이다.
background process	background 프로세스는 사용자에게는 현재 보여지지 않는 Activity를 점유하는 프로세스이다. 이러한 프로세스는 사용자에게 어떤 것도 직접적으로 영향을 미치지 않는다. activity 생명주기를 정확하게 구현하기 위해서 마련된 것이며, 시스템은 위의 3가지 프로세스 타입 중 한 가지를 위한 메모리 반환 요청이 있을 시에만 그러한 프로세스를 종료시킬 것이다. 일반적으로 많은 수의 이런 프로세스가 실행되고 있으며, 해당 프로세스들은 메모리 결핍 시 사용자에게 가장 최근에 보여진 것이 가장 마지막에 종료되는 절차를 확립하기 위해 LRU 리스트 상에서 유지된다.
empty process	empty 프로세스는 어떤 활성화 된 애플리케이션 컴포넌트도 점유하지 않는 프로세스이다. 이러한 프로세스를 유지하고 있는 유일한 이유는 다음번에 해당 애플리케이션을 실행할 필요가 있을 때 시동(startup) 시간을 개선하기 위한 캐쉬로써 사용하기 위함이다. 그런 이유로, 시스템은 이러한 캐쉬화된 empty 프로세스들과 기반에 있는 커널 캐쉬들 사이에서 전반적인 시스템 균형을 유지하기 위해 이러한 프로세스들을 가끔 종료하게 된다.

목 차

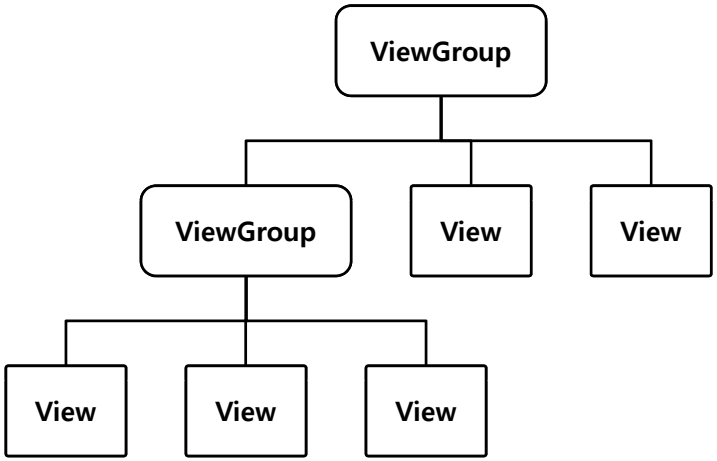
애플리케이션 개발 : 개발, 빌드, 배포 절차

안드로이드 app. 즉, apk는 아래와 같은 과정을 통해서 개발 및 빌드되며, 최종적으로는 apk 형태로 배포되며, 해당 apk는 안드로이드 package manager에 의해 관리되며, dalvik 가상 머신 상에서 실행된다.



View 계층구조

On the Android platform, you define an Activity's UI using a hierarchy of View and ViewGroup nodes, as shown in the diagram below. This hierarchy tree can be as simple or complex as you need it to be, and you can build it up using Android's set of predefined widgets and layouts, or with custom Views that you create yourself.



Layout

The most common way to define your layout and express the view hierarchy is with an XML layout file. XML offers a human-readable structure for the layout, much like HTML. Each element in XML is either a View or ViewGroup object (or descendent thereof). View objects are leaves in the tree, ViewGroup objects are branches in the tree (see the View Hierarchy figure above).

For example, a simple vertical layout with a text view and a button looks like this:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android=http://schemas.android.com/apk/res/android
                android:layout_width="fill_parent"
                android:layout_height="fill_parent"
                android:orientation="vertical" >
        <TextView android:id="@+id/text"
                android:layout_width="wrap_content"
                android:layout_height="wrap_content"
                android:text="Hello, I am a TextView" />
        <Button android:id="@+id/button"
                android:layout_width="wrap_content"
                android:layout_height="wrap_content"
                android:text="Hello, I am a Button" />
</LinearLayout>
```

Declaring Layout

Declare UI elements in XML. Android provides a straightforward XML vocabulary that corresponds to the View classes and subclasses, such as those for widgets and layouts.

Instantiate layout elements at runtime. Your application can create View and ViewGroup objects (and manipulate their properties) programmatically.

Write the XML

Load the XML

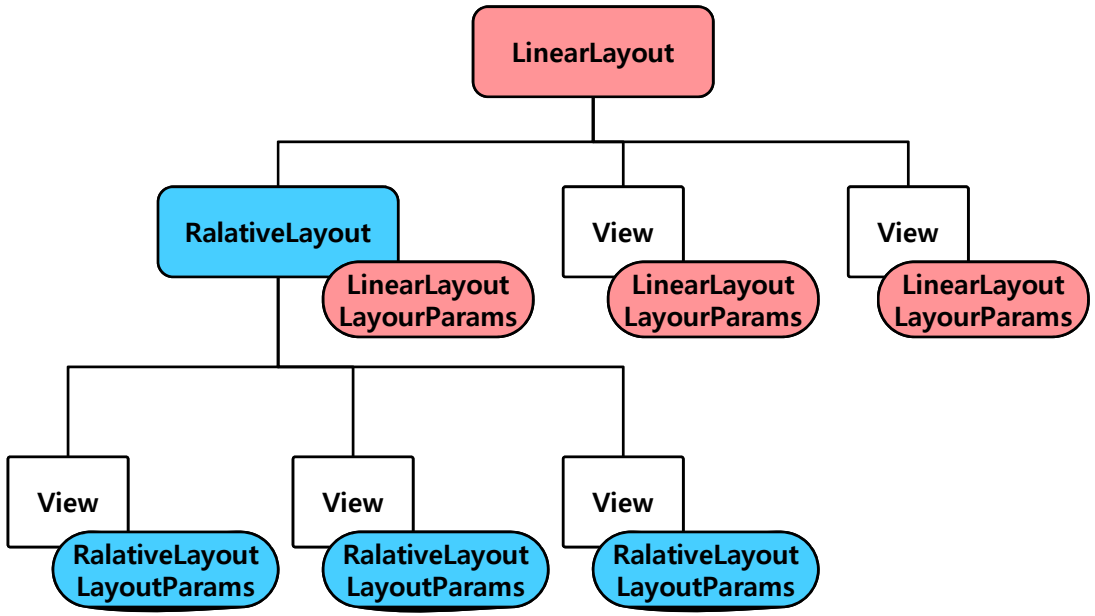
Resource

Attributes
- ID
- Layout

Parameters

Layout Position

Size, Padding and Margins



Common Layout Object

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.
ListView	Displays a scrolling single column list.
RelativeLayout	Enables you to specify the location of child objects relative to each other or to 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.
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 scre en 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 t imed intervals, like a slide show.
ViewSwitcher	Same as ViewFlipper.

Wigets

A widget is a View object that serves as an interface for interaction with the user. Android provides a set of fully implemented widgets, like buttons, checkboxes, and text-entry fields, so you can quickly build your UI. Some widgets provided by Android are more complex, like a date picker, a clock, and zoom controls. But you're not limited to the kinds of widgets provided by the Android platform. If you'd like to do something more customized and create your own actionable elements, you can, by defining your own View object or by extending and combining existing widgets.

UI Events

Once you've added some Views/widgets to the UI, you probably want to know about the user's interaction with them, so you can perform actions. To be informed of UI events, you need to do one of two things:

Define an event listener and register it with the View.

More often than not, this is how you'll listen for events. The View class contains a collection of nested interfaces named On<something>Listener, each with a callback method called On<something>(). For example, View.OnClickListener (for handling "clicks" on a View), View.OnTouchListener (for handling touch screen events in a View), and View.OnKeyListener (for handling device key presses within a View). So if you want your View to be notified when it is "clicked" (such as when a button is selected), implement OnClickListener and define its onClick() callback method (where you perform the action upon click), and register it to the View with setOnClickListener().

Override an existing callback method for the View.

This is what you should do when you've implemented your own View class and want to listen for specific events that occur within it. Example events you can handle include when the screen is touched (onTouchEvent()), when the trackball is moved (onTrackballEvent()), or when a key on the device is pressed (onKeyDown()). This allows you to define the default behavior for each event inside your custom View and determine whether the event should be passed on to some other child View. Again, these are callbacks to the View class, so your only chance to define them is when you build a custom component.

Menus

Application menus are another important part of an application's UI. Menus offers a reliable interface that reveals application functions and settings. The most common application menu is revealed by pressing the MENU key on the device. However, you can also add Context Menus, which may be revealed when the user presses and holds down on an item.

Menus are also structured using a View hierarchy, but you don't define this structure yourself. Instead, you define the onCreateOptionsMenu() or onCreateContextMenu() callback methods for your Activity and declare the items that you want to include in your menu. At the appropriate time, Android will automatically create the necessary View hierarchy for the menu and draw each of your menu items in it.

Menus also handle their own events, so there's no need to register event listeners on the items in your menu. When an item in your menu is selected, the onOptionsItemSelected() or onContextItemSelected() method will be called by the framework.

And just like your application layout, you have the option to declare the items for you menu in an XML file.

Menus

Options Menu

This is the primary set of menu items for an Activity. It is revealed by pressing the device MENU key. Within the Options Menu are two groups of menu items:

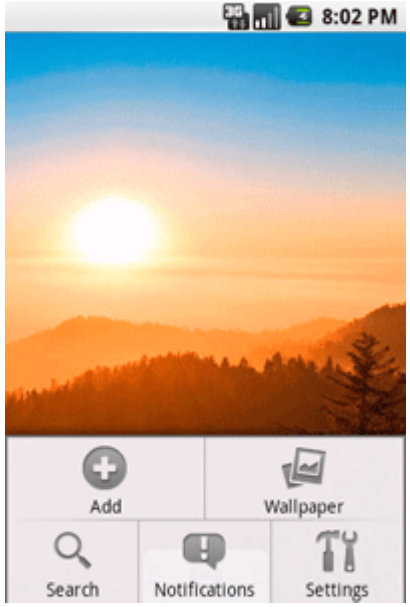
- Icon Menu
- Expanded Menu

Context Menu

This is a floating list of menu items that may appear when you perform a long-press on a View (such as a list item).

Submenu

This is a floating list of menu items that is revealed by an item in the Options Menu or a Context Menu. A Submenu item cannot support nested Submenus.



Adapters

Sometimes you'll want to populate a view group with some information that can't be hard-coded, instead, you want to bind your view to an external source of data. To do this, you use an AdapterView as your view group and each child View is initialized and populated with data from the Adapter.

The AdapterView object is an implementation of ViewGroup that determines its child views based on a given Adapter object. The Adapter acts like a courier between your data source (perhaps an array of external strings) and the AdapterView, which displays it. There are several implementations of the Adapter class, for specific tasks, such as the CursorAdapter for reading database data from a Cursor, or an ArrayAdapter for reading from an arbitrary array.

Styles and Themes

Perhaps you're not satisfied with the look of the standard widgets. To revise them, you can create some of your own styles and themes.

- A style is a set of one or more formatting attributes that you can apply as a unit to individual elements in your layout. For example, you could define a style that specifies a certain text size and color, then apply it to only specific View elements.

- A theme is a set of one or more formatting attributes that you can apply as a unit to all activities in an application, or just a single activity. For example, you could define a theme that sets specific colors for the window frame and the panel background, and sets text sizes and colors for menus. This theme can then be applied to specific activities or the entire application.

Styles and themes are resources. Android provides some default style and theme resources that you can use, or you can declare your own custom style and theme resources.

Building Custom Components

The Basic Approach

Here is a high level overview of what you need to know to get started in creating your own View components:

1. Extend an existing View class or subclass with your own class.

2. Override some of the methods from the superclass. The superclass methods to override start with 'on', for example, onDraw(), onMeasure(), and onKeyDown(). This is similar to the on... events in Activity or ListActivity that you override for lifecycle and other functionality hooks.

3. Use your new extension class. Once completed, your new extension class can be used in place of the view upon which it was based.

Fully Customized Components

Compound Controls

Modifying an Existing View Type

How Android Draws View

Drawing the layout is a two pass process: a measure pass and a layout pass.

The measuring pass is implemented in measure(int, int) and is a top-down traversal of the View tree. Each View pushes dimension specifications down the tree during the recursion. At the end of the measure pass, every View has stored its measurements.

The second pass happens in layout(int, int, int, int) and is also top-down. During this pass each parent is responsible for positioning all of its children using the sizes computed in the measure pass.

How Android Draws View

Preferences

Preferences is a lightweight mechanism to store and retrieve key-value pairs of primitive data types. It is typically used to store application preferences, such as a default greeting or a text font to be loaded whenever the 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 application, or use Activity.getPreferences() with no name to keep them private to the calling activity. You cannot share preferences across applications (except by using a content provider).

Files

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

Databases

The Android API contains support for creating and using SQLite databases. Each database is private to the application that creates it.

Network

You can also use the network to store and retrieve data (when it's available). To do network operations, use the classes in the following packages:

java.net.* android.net.*

목 차

애플리케이션 개발 : 실습 (Simple Note)

목 차

Overview

Three of the core components of an application - activities, services, and broadcast receivers - are activated through messages, called intents. Intent messaging is a facility for late run-time binding between components in the same or different applications. The intent itself, an Intent object, is a passive data structure holding an abstract description of an operation to be performed - or, in the case of broadcasts, a description of something that has happened and is being announced.

There are separate mechanisms for delivering intents to each type of component:

1. An Intent object is passed to Context.startActivity() or Activity.startActivityForResult() to launch an activity or get an existing activity to do something new.

2. An Intent object is passed to Context.startService() to initiate a service or deliver new instructions to an ongoing service. Similarly, an intent can be passed to Context.bindService() to establish a connection between the calling component and a target service. It can optionally initiate the service if it's not already running.

3. Intent objects passed to any of the broadcast methods (such as Context.sendBroadcast(), Context.sendOrderedBroadcast(), or Context.sendStickyBroadcast()) are delivered to all interested broadcast receivers. Many kinds of broadcasts originate in system code.

Intent Objects : Component name

An Intent object is a bundle of information. It contains information of interest to the component that receives the intent (such as the action to be taken and the data to act on) plus information of interest to the Android system (such as the category of component that should handle the intent and instructions on how to launch a target activity). Principally, it can contain the following:

Component name

The name of the component that should handle the intent. This field is a ComponentName object - a combination of the fully qualified class name of the target component (for example "com.example.project.app.FreneticActivity") and the package name set in the manifest file of the application where the component resides (for example, "com.example.project"). The package part of the component name and the package name set in the manifest do not necessarily have to match.

The component name is optional. If it is set, the Intent object is delivered to an instance of the designated class. If it is not set, Android uses other information in the Intent object to locate a suitable target.

The component name is set by setComponent(), setClass(), or setClassName() and read by getComponent().

Intent Objects : Action

A string naming the action to be performed - or, in the case of broadcast intents, the action that took place and is being reported. The Intent class defines a number of action constants, including these:

Constant	Target component	Action
ACTION_CALL	activity	Initiate a phone call.
ACTION_EDIT	activity	Display data for the user to edit.
ACTION_MAIN	activity	Start up as the initial activity of a task, with n o data input and no returned output.
ACTION_SYNC	activity	Synchronize data on a server with data on the mobile device.
ACTION_BATTERY_LOW	broadcast receiver	A warning that the battery is low.
ACTION_HEADSET_PLUG	broadcast receiver	A headset has been plugged into the device, or unplugged from it.
ACTION_SCREEN_ON	broadcast receiver	The screen has been turned on.
ACTION_TIMEZONE_CHANGED	broadcast receiver	The setting for the time zone has changed.

Intent Objects : Data

The URI of the data to be acted on and the MIME type of that data. Different actions are paired with different kinds of data specifications. For example, if the action field is ACTION_EDIT, the data field would contain the URI of the document to be displayed for editing. If the action is ACTION_CALL, the data field would be a tel: URI with the number to call. Similarly, if the action is ACTION_VIEW and the data field is an http: URI, the receiving activity would be called upon to download and display whatever data the URI refers to.

When matching an intent to a component that is capable of handling the data, it's often important to know the type of data (its MIME type) in addition to its URI. For example, a component able to display image data should not be called upon to play an audio file.

In many cases, the data type can be inferred from the URI - particularly content: URIs, which indicate that the data is located on the device and controlled by a content provider (see the separate discussion on content providers). But the type can also be explicitly set in the Intent object. The setData() method specifies data only as a URI, setType() specifies it only as a MIME type, and setDataAndType() specifies it as both a URI and a MIME type. The URI is read by getData() and the type by getType().

Intent Objects : Category

A string containing additional information about the kind of component that should handle the intent. Any number of category descriptions can be placed in an Intent object. As it does for actions, the Intent class defines several category constants, including these:

Constant	Meaning
CATEGORY_BROWSABLE	The target activity can be safely invoked by the browser to display data referenced by a link — for example, an image or an e-mail message.
CATEGORY_GADGET	The activity can be embedded inside of another activity that hosts gadgets.
CATEGORY_HOME	The activity displays the home screen, the first screen the user sees when the device is turned on or when the HOME key is pressed.
CATEGORY_LAUNCHER	The activity can be the initial activity of a task and is listed in the top-level application launcher.
CATEGORY_PREFERENCE	The target activity is a preference panel.

Intent Objects : Extras

Key-value pairs for additional information that should be delivered to the component handling the intent. Just as some actions are paired with particular kinds of data URIs, some are paired with particular extras. For example, an ACTION_TIMEZONE_CHANGED intent has a "time-zone" extra that identifies the new time zone, and ACTION_HEADSET_PLUG has a "state" extra indicating whether the headset is now plugged in or unplugged, as well as a "name" extra for the type of headset. If you were to invent a SHOW_COLOR action, the color value would be set in an extra key-value pair.

The Intent object has a series of put...() methods for inserting various types of extra data and a similar set of get...() methods for reading the data. These methods parallel those for Bundle objects. In fact, the extras can be installed and read as a Bundle using the putExtras() and getExtras() methods.

Intent Objects : Flags

Flags of various sorts. Many instruct the Android system how to launch an activity (for example, which task the activity should belong to) and how to treat it after it's launched (for example, whether it belongs in the list of recent activities). All these flags are defined in the Intent class.

Intent Resolution

Intents can be divided into two groups:

- **Explicit intents** designate the target component by its name (the component name field, mentioned earlier, has a value set). Since component names would generally not be known to developers of other applications, explicit intents are typically used for application-internal messages - such as an activity starting a subordinate service or launching a sister activity.

- **Implicit intents** do not name a target (the field for the component name is blank). Implicit intents are often used to activate components in other applications.

Only three aspects of an Intent object are consulted when the object is tested against an intent filter:

- action
- data (both URI and data type)
- category

Intent Resolution : Intent filters

An intent filter is an instance of the IntentFilter class. However, since the Android system must know about the capabilities of a component before it can launch that component, intent filters are generally not set up in Java code, but in the application's manifest file (AndroidManifest.xml) as <intent-filter> elements. (The one exception would be filters for broadcast receivers that are registered dynamically by calling Context.registerReceiver(); they are directly created as IntentFilter objects.)

A filter has fields that parallel the action, data, and category fields of an Intent object. An implicit intent is tested against the filter in all three areas. To be delivered to the component that owns the filter, it must pass all three tests. If it fails even one of them, the Android system won't deliver it to the component ? at least not on the basis of that filter. However, since a component can have multiple intent filters, an intent that does not pass through one of a component's filters might make it through on another.

Intent Resolution : Intent filters – Category test

An <intent-filter> element in the manifest file lists actions as <action> subelements.
For example:

```
<intent-filter . . . >
        <category android:name="android.intent.category.DEFAULT" />
        <category android:name="android.intent.category.BROWSABLE" />
        . . .
</intent-filter>
```

For an intent to pass the category test, every category in the Intent object must match a category in the filter. The filter can list additional categories, but it cannot omit any that are in the intent.

In principle, therefore, an Intent object with no categories should always pass this test, regardless of what's in the filter. That's mostly true. However, with one exception, Android treats all implicit intents passed to startActivity() as if they contained at least one category: "android.intent.category.DEFAULT" (the CATEGORY_DEFAULT constant). Therefore, activities that are willing to receive implicit intents must include "android.intent.category.DEFAULT" in their intent filters.

Intent Resolution : Intent filters – Action test

An <intent-filter> element also lists categories as subelements.
For example:

```
<intent-filter . . . >
        <action android:name="com.example.project.SHOW_CURRENT" />
        <action android:name="com.example.project.SHOW_RECENT" />
        <action android:name="com.example.project.SHOW_PENDING" />
        . . .
</intent-filter>
```

To pass this test, the action specified in the Intent object must match one of the actions listed in the filter. If the object or the filter does not specify an action, the results are as follows:

- If the filter fails to list any actions, there is nothing for an intent to match, so all intents fail the test. No intents can get through the filter.

- On the other hand, an Intent object that doesn't specify an action automatically passes the test ? as long as the filter contains at least one action.

Intent Resolution : Intent filters – Data test

Like the action and categories, the data specification for an intent filter is contained in a subelement. And, as in those cases, the subelement can appear multiple times, or not at all. For example:

```
<intent-filter . . . >
    <data android:type="video/mpeg" android:scheme="http" . . . />
    <data android:type="audio/mpeg" android:scheme="http" . . . />
    . . .
</intent-filter>
```

Each <data> element can specify a URI and a data type (MIME media type). There are separate attributes - scheme, host, port, and path - for each part of the URI:

scheme://host:port/path

For example, in the following URI,

content://com.example.project:200/folder/subfolder/etc

the scheme is "content", the host is "com.example.project", the port is "200", and the path is "folder/subfolder/etc". The host and port together constitute the URI authority; if a host is not specified, the port is ignored.

Intent Resolution : Intent filters – Data test

The data test compares both the URI and the data type in the Intent object to a URI and data type specified in the filter. The rules are as follows:

A. An Intent object that contains neither a URI nor a data type passes the test only if the filter likewise does not specify any URIs or data types.

B. An Intent object that contains a URI but no data type (and a type cannot be inferred from the URI) passes the test only if its URI matches a URI in the filter and the filter likewise does not specify a type. This will be the case only for URIs like mailto: and tel: that do not refer to actual data.

C. An Intent object that contains a data type but not a URI passes the test only if the filter lists the same data type and similarly does not specify a URI.

D. An Intent object that contains both a URI and a data type (or a data type can be inferred from the URI) passes the data type part of the test only if its type matches a type listed in the filter. It passes the URI part of the test either if its URI matches a URI in the filter or if it has a content: or file: URI and the filter does not specify a URI. In other words, a component is presumed to support content: and file: data if its filter lists only a data type.

Intent Resolution : Intent filters – Common cases

The last rule shown above for the data test, rule (d), reflects the expectation that components are able to get local data from a file or content provider. Therefore, their filters can list just a data type and do not need to explicitly name the content: and file: schemes. This is a typical case. A <data> element like the following, for example, tells Android that the component can get image data from a content provider and display it:

```
<data android:type="image/*" />
```

Another common configuration is filters with a scheme and a data type. For example, a <data> element like the following tells Android that the component can get video data from the network and display it:

```
<data android:scheme="http" android:type="video/*" />
```

Most applications also have a way to start fresh, without a reference to any particular data.

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

Intent Resolution : Intent filters – Using intent matching

Intents are matched against intent filters not only to discover a target component to activate, but also to discover something about the set of components on the device. For example, the Android system populates the application launcher, the top-level screen that shows the applications that are available for the user to launch, by finding all the activities with intent filters that specify the "android.intent.action.MAIN" action and "android.intent.category.LAUNCHER" category (as illustrated in the previous section). It then displays the icons and labels of those activities in the launcher. Similarly, it discovers the home screen by looking for the activity with "android.intent.category.HOME" in its filter.

Your application can use intent matching is a similar way. The PackageManager has a set of query...() methods that return all components that can accept a particular intent, and a similar series of resolve...() methods that determine the best component to respond to an intent. For example, queryIntentActivities() returns a list of all activities that can perform the intent passed as an argument, and queryIntentServices() returns a similar list of services. Neither method activates the components; they just list the ones that can respond. There's a similar method, queryBroadcastReceivers(), for broadcast receivers.

목 차

애플리케이션 개발 : 실습 (API Demos)

목 차

Designing a Remote Interface Using AIDL

Since each application runs in its own process, and you can write a service that runs in a different process from your Application's UI, sometimes you need to pass objects between processes. On the Android platform, one process can not normally access the memory of another process. So to talk, they need to decompose their objects into primitives that the operating system can understand, and "marshall" the object across that boundary for you.

The code to do that marshalling is tedious to write, so we provide the AIDL tool to do it for you.

AIDL (Android Interface Definition Language) is an IDL language used to generate code that enables two processes on an Android-powered device to talk using interprocess communication (IPC). If you have code in one process (for example, in an Activity) that needs to call methods on an object in another process (for example, a Service), you would use AIDL to generate code to marshall the parameters.

The AIDL IPC mechanism is interface-based, similar to COM or Corba, but lighter weight. It uses a proxy class to pass values between the client and the implementation.

- Implementing IPC Using AIDL
- Calling an .aidl (IPC) Class

Implementing IPC Using AIDL

Follow these steps to implement an IPC service using AIDL.

1. **Create your .aidl file - This file defines an interface (YourInterface.aidl) that defines the methods and fields available to a client.**

2. **Add the .aidl file to your makefile - (the ADT Plugin for Eclipse manages this for you). Android includes the compiler, called AIDL, in the tools/ directory.**

3. **Implement your interface methods - The AIDL compiler creates an interface in the Java programming language from your AIDL interface. This interface has an inner abstract class named Stub that inherits the interface (and implements a few additional methods necessary for the IPC call). You must create a class that extends YourInterface.Stub and implements the methods you declared in your .aidl file.**

4. **Expose your interface to clients - If you're writing a service, you should extend Service and override Service.onBind(Intent) to return an instance of your class that implements your interface.**

Implementing IPC Using AIDL : Creating an .aidl File

AIDL is a simple syntax that lets you declare an interface with one or more methods, that can take parameters and return values. These parameters and return values can be of any type, even other AIDL-generated interfaces. However, it is important to note that you must import all non-built-in types, even if they are defined in the same package as your interface. Here are the data types that AIDL can support:

- Primitive Java programming language types (int, boolean, etc) - No import statement is needed.
- One of the following classes (no import statements needed):
  - String
  - List - All elements in the List must be one of the types in this list, including other AIDL-generated interfaces and parcelables. List may optionally be used as a "generic" class (e.g. List<String>). The actual concrete class that the other side will receive will always be an ArrayList, although the method will be generated to use the List interface.
  - Map - All elements in the Map must be of one of the types in this list, including other AIDL-generated interfaces and parcelables. Generic maps, (e.g. of the form Map<String,Integer> are not supported. The actual concrete class that the other side will receive will always be a HashMap, although the method will be generated to use the Map interface.
  - CharSequence - This is useful for the CharSequence types used by TextView and other widget objects.
- Other AIDL-generated interfaces, which are always passed by reference. An import statement is always needed for these.
- Custom classes that implement the Parcelable protocol and are passed by value. An import statement is always needed for these.

Implementing IPC Using AIDL : Implementing the Interface

AIDL generates an interface file for you with the same name as your .aidl file. If you are using the Eclipse plugin, AIDL will automatically be run as part of the build process (you don't need to run AIDL first and then build your project). If you are not using the plugin, you should run AIDL first.

The generated interface includes an abstract inner class named Stub that declares all the methods that you declared in your .aidl file. Stub also defines a few helper methods, most notably asInterface(), which takes an IBinder (passed to a client's onServiceConnected() implementation when applicationContext.bindService() succeeds), and returns an instance of the interface used to call the IPC methods. See the section Calling an IPC Method for more details on how to make this cast.

To implement your interface, extend YourInterface.Stub, and implement the methods. (You can create the .aidl file and implement the stub methods without building between--the Android build process will process .aidl files before .java files.)

Here is an example of implementing an interface called IRemoteService, which exposes a single method, getPid(), using an anonymous instance:

```
private final IRemoteService.Stub mBinder = new IRemoteService.Stub(){
    public int getPid(){
        return Process.myPid();
    }
}
```

Implementing IPC Using AIDL : Exposing Your Interface to Clients

Now that you've got your interface implementation, you need to expose it to clients. This is known as "publishing your service." To publish a service, inherit Service and implement Service.onBind(Intent) to return an instance of the class that implements your interface. Here's a code snippet of a service that exposes the IRemoteService interface to clients.

```
public class RemoteService extends Service {
...
    @Override
    public IBinder onBind(Intent intent) {
        if (IRemoteService.class.getName().equals(intent.getAction())) {            return mBinder;        }
        if (ISecondary.class.getName().equals(intent.getAction())) {            return mSecondaryBinder;        }
        return null;
    }
    private final IRemoteService.Stub mBinder = new IRemoteService.Stub() {
        public void registerCallback(IRemoteServiceCallback cb) {
            if (cb != null) mCallbacks.register(cb);
        }
        public void unregisterCallback(IRemoteServiceCallback cb) {
            if (cb != null) mCallbacks.unregister(cb);
        }
    };
    private final ISecondary.Stub mSecondaryBinder = new ISecondary.Stub() {
        public int getPid() {            return Process.myPid();        }
        public void basicTypes(int anInt, long aLong, boolean aBoolean,
                float aFloat, double aDouble, String aString) {
        }
    };
}
```

Implementing IPC Using AIDL : Pass by value Parameters using Parcelables

If you have a class that you would like to send from one process to another through an AIDL interface, you can do that. You must ensure that the code for your class is available to the other side of the IPC. Generally, that means that you're talking to a service that you started.

There are five parts to making a class support the Parcelable protocol:

1. Make your class implement the Parcelable interface.
2. Implement the method public void writeToParcel(Parcel out) that takes the current state of the object and writes it to a parcel.
3. Implement the method public void readFromParcel(Parcel in) that reads the value in a parcel into your object.
4. Add a static field called CREATOR to your class which is an object implementing the Parcelable.Creator interface.
5. Last but not least:
    - If you are developing with Eclipse/ADT, follow these steps:
        - In the Package Explorer view, right-click on the project.
        - Choose Android Tools > Create Aidl preprocess file for Parcelable classes.
        - This will create a file called "project.aidl" in the root of the project. The file will be automatically used when compiling an aidl file that uses the parcelable classes.
    - If you are developing with Ant or are using a custom build process, create an aidl file that declares your parcelable class (as shown below). If you are using a custom build process, do not add the aidl file to your build. Similar to a header file in C, the aidl file isn't compiled.

Implementing IPC Using AIDL : Pass by value Parameters using Parcelables

AIDL will use these methods and fields in the code it generates to marshall and unmarshall your objects.

Here is an example of how the Rect class implements the Parcelable protocol.

```
import android.os.Parcel;
import android.os.Parcelable;

public final class Rect implements Parcelable {
    public int left;
    public int top;
    public int right;
    public int bottom;

    public static final Parcelable.Creator<Rect> CREATOR = new Parcelable.Creator<Rect>() {
        public Rect createFromParcel(Parcel in) {            return new Rect(in);        }
        public Rect[] newArray(int size) {            return new Rect[size];        }
    };

    public Rect() {    }

    private Rect(Parcel in) {        readFromParcel(in);    }
    public void writeToParcel(Parcel out) {
        out.writeInt(left);
        out.writeInt(top);
        out.writeInt(right);
        out.writeInt(bottom);
    }
    public void readFromParcel(Parcel in) {
        left = in.readInt();
        top = in.readInt();
        right = in.readInt();
        bottom = in.readInt();
    }
}
```

```
// Rect.aidl
package android.graphics;

// Declare Rect so AIDL can find it and knows that it
implements
// the parcelable protocol.
parcelable Rect;
```

Calling an IPC Method

Here are the steps a calling class should make to call your remote interface:

1. Declare a variable of the interface type that your .aidl file defined.
2. Implement ServiceConnection.
3. Call Context.bindService(), passing in your ServiceConnection implementation.
4. In your implementation of ServiceConnection.onServiceConnected(), you will receive an IBinder instance (called service). Call YourInterfaceName.Stub.asInterface((IBinder)service) to cast the returned parameter to YourInterface type.
5. Call the methods that you defined on your interface. You should always trap DeadObjectException exceptions, which are thrown when the connection has broken; this will be the only exception thrown by remote methods.
6. To disconnect, call Context.unbindService() with the instance of your interface.

A few comments on calling an IPC service:

- Objects are reference counted across processes.
- You can send anonymous objects as method arguments.

Calling an IPC Method : Example

```
public class RemoteServiceBinding extends Activity {
    /** The primary interface we will be calling on the service. */
    IRemoteService mService = null;
    /** Another interface we use on the service. */
    ISecondary mSecondaryService = null;

    Button mKillButton;
    TextView mCallbackText;

    private boolean mIsBound;

    // Standard initialization of this activity.  Set up the UI, then wait  for the user to poke it before doing anything.

    @Override
    protected void onCreate(Bundle savedInstanceState) {

        super.onCreate(savedInstanceState);

        setContentView(R.layout.remote_service_binding);

        // Watch for button clicks.

        Button button = (Button)findViewById(R.id.bind);
        button.setOnClickListener(mBindListener);
        button = (Button)findViewById(R.id.unbind);
        button.setOnClickListener(mUnbindListener);
        mKillButton = (Button)findViewById(R.id.kill);
        mKillButton.setOnClickListener(mKillListener);
        mKillButton.setEnabled(false);

        mCallbackText = (TextView)findViewById(R.id.callback);
        mCallbackText.setText("Not attached.");

    }
```

Calling an IPC Method : Example

```
/**
 * Class for interacting with the main interface of the service.
 */
private ServiceConnection mConnection = new ServiceConnection() {

    public void onServiceConnected(ComponentName className,IBinder service) {
        // This is called when the connection with the service has been
        // established, giving us the service object we can use to
        // interact with the service.  We are communicating with our
        // service through an IDL interface, so get a client-side
        // representation of that from the raw service object.

        mService = IRemoteService.Stub.asInterface(service);
        mKillButton.setEnabled(true);
        mCallbackText.setText("Attached.");

        // We want to monitor the service for as long as we are
        // connected to it.

        try {
            mService.registerCallback(mCallback);
        } catch (RemoteException e) {

            // In this case the service has crashed before we could even
            // do anything with it; we can count on soon being
            // disconnected (and then reconnected if it can be restarted)
            // so there is no need to do anything here.
        }

        // As part of the sample, tell the user what happened.
        Toast.makeText(RemoteServiceBinding.this, R.string.remote_service_connected,
                Toast.LENGTH_SHORT).show();
    }
```

Calling an IPC Method : Example

```
public void onServiceDisconnected(ComponentName className) {
        // This is called when the connection with the service has been
        // unexpectedly disconnected -- that is, its process crashed.
        mService = null;
        mKillButton.setEnabled(false);
        mCallbackText.setText("Disconnected.");

        // As part of the sample, tell the user what happened.
        Toast.makeText(RemoteServiceBinding.this, R.string.remote_service_disconnected,
                Toast.LENGTH_SHORT).show();
    }
};

/**
 * Class for interacting with the secondary interface of the service.
 */
private ServiceConnection mSecondaryConnection = new ServiceConnection() {
    public void onServiceConnected(ComponentName className,
            IBinder service) {
        // Connecting to a secondary interface is the same as any
        // other interface.
        mSecondaryService = ISecondary.Stub.asInterface(service);
        mKillButton.setEnabled(true);
    }

    public void onServiceDisconnected(ComponentName className) {
        mSecondaryService = null;
        mKillButton.setEnabled(false);
    }
};
```

Calling an IPC Method : Example

```
private OnClickListener mBindListener = new OnClickListener() {
    public void onClick(View v) {
        // Establish a couple connections with the service, binding  by interface names.
        // This allows other applications to be installed that replace the remote service by implementing the same interface.
        bindService(new Intent(IRemoteService.class.getName()), mConnection, Context.BIND_AUTO_CREATE);
        bindService(new Intent(ISecondary.class.getName()), mSecondaryConnection, Context.BIND_AUTO_CREATE);
        mIsBound = true;
        mCallbackText.setText("Binding.");
    }
};

private OnClickListener mUnbindListener = new OnClickListener() {
    public void onClick(View v) {
        if (mIsBound) {
            // If we have received the service, and hence registered with it, then now is the time to unregister.
            if (mService != null) {
                try {
                    mService.unregisterCallback(mCallback);
                } catch (RemoteException e) {
                    // There is nothing special we need to do if the service has crashed.
                }
            }

            // Detach our existing connection.
            unbindService(mConnection);
            unbindService(mSecondaryConnection);
            mKillButton.setEnabled(false);
            mIsBound = false;
            mCallbackText.setText("Unbinding.");
        }
    }
};
```

Calling an IPC Method : Example

```
private OnClickListener mKillListener = new OnClickListener() {

    public void onClick(View v) {

        // To kill the process hosting our service, we need to know its
        // PID.  Conveniently our service has a call that will return
        // to us that information.

        if (mSecondaryService != null) {

            try {

                int pid = mSecondaryService.getPid();
                // Note that, though this API allows us to request to kill any process based on its PID, the kernel will
                // still impose standard restrictions on which PIDs you are actually able to kill.  Typically this means only
                // the process running your application and any additional processes created by that app as shown here; packages
                // sharing a common UID will also be able to kill each  other's processes.

                Process.killProcess(pid);
                mCallbackText.setText("Killed service process.");

            } catch (RemoteException ex) {
                // Recover gracefully from the process hosting the server dying.
                // Just for purposes of the sample, put up a notification.

                Toast.makeText(RemoteServiceBinding.this,
                        R.string.remote_call_failed,
                        Toast.LENGTH_SHORT).show();
            }
        }
    }
};
```

Calling an IPC Method : Example

```
// --------------------------------------------------------------------
// Code showing how to deal with callbacks.
// --------------------------------------------------------------------

//  This implementation is used to receive callbacks from the remote service.

private IRemoteServiceCallback mCallback = new IRemoteServiceCallback.Stub() {
    /**
     * This is called by the remote service regularly to tell us about
     * new values.  Note that IPC calls are dispatched through a thread
     * pool running in each process, so the code executing here will
     * NOT be running in our main thread like most other things -- so,
     * to update the UI, we need to use a Handler to hop over there.
     */
    public void valueChanged(int value) {
        mHandler.sendMessage(mHandler.obtainMessage(BUMP_MSG, value, 0));
    }
};

private static final int BUMP_MSG = 1;

private Handler mHandler = new Handler() {
    @Override public void handleMessage(Message msg) {
        switch (msg.what) {
            case BUMP_MSG:
                mCallbackText.setText("Received from service: " + msg.arg1);
                break;
            default:
                super.handleMessage(msg);
        }
    }

};
}
```

목 차

Content Provider Basics : Overview

How a content provider actually stores its data under the covers is up to its designer. But all content providers implement a common interface for querying the provider and returning results ? as well as for adding, altering, and deleting data.

It's an interface that clients use indirectly, most generally through ContentResolver objects. You get a ContentResolver by calling getContentResolver() from within the implementation of an Activity or other application component:

```
ContentResolver cr = getContentResolver();
```

You can then use the ContentResolver's methods to interact with whatever content providers you're interested in.

When a query is initiated, the Android system identifies the content provider that's the target of the query and makes sure that it is up and running. The system instantiates all ContentProvider objects; you never need to do it on your own. In fact, you never deal directly with ContentProvider objects at all. Typically, there's just a single instance of each type of ContentProvider. But it can communicate with multiple ContentResolver objects in different applications and processes. The interaction between processes is handled by the ContentResolver and ContentProvider classes.

Content Provider Basics : The data model

Content providers expose their data as a simple table on a database model, where each row is a record and each column is data of a particular type and meaning.
For example, information about people and their phone numbers might be exposed as follows:

_ID	NUMBER	NUMBER_KEY	LABEL	NAME	TYPE
13	(425) 555 6677	425 555 6677	Kirkland office	Bully Pulpit	TYPE_WORK
44	(212) 555-1234	212 555 1234	NY apartment	Alan Vain	TYPE_HOME
45	(212) 555-6657	212 555 6657	Downtown office	Alan Vain	TYPE_MOBILE
53	201.555.4433	201 555 4433	Love Nest	Rex Cars	TYPE_HOME

Every record includes a numeric _ID field that uniquely identifies the record within the table. IDs can be used to match records in related tables - for example, to find a person's phone number in one table and pictures of that person in another.

A query returns a Cursor object that can move from record to record and column to column to read the contents of each field. It has specialized methods for reading each type of data. So, to read a field, you must know what type of data the field contains. (There's more on query results and Cursor objects later.)

Content Provider Basics : URIs

Each content provider exposes a public URI (wrapped as a Uri object) that uniquely identifies its data set. A content provider that controls multiple data sets (multiple tables) exposes a separate URI for each one. All URIs for providers begin with the string "content://". The content: scheme identifies the data as being controlled by a content provider.

If you're defining a content provider, it's a good idea to also define a constant for its URI, to simplify client code and make future updates cleaner. Android defines CONTENT_URI constants for all the providers that come with the platform. For example, the URI for the table that matches phone numbers to people and the URI for the table that holds pictures of people (both controlled by the Contacts content provider) are:

android.provider.Contacts.Phones.CONTENT_URI
android.provider.Contacts.Photos.CONTENT_URI

Similarly, the URIs for the table of recent phone calls and the table of calendar entries are:

android.provider.CallLog.Calls.CONTENT_URI
android.provider.Calendar.CONTENT_URI

The URI constant is used in all interactions with the content provider. Every ContentResolver method takes the URI as its first argument. It's what identifies which provider the ContentResolver should talk to and which table of the provider is being targeted.

Querying a Content Provider

You need three pieces of information to query a content provider:

- The URI that identifies the provider
- The names of the data fields you want to receive
- The data types for those fields

If you're querying a particular record, you also need the ID for that record.

Querying a Content Provider : Making the query

To query a content provider, you can use either the ContentResolver.query() method or the Activity.managedQuery() method. Both methods take the same set of arguments, and both return a Cursor object. However, managedQuery() causes the activity to manage the life cycle of the Cursor. A managed Cursor handles all of the niceties, such as unloading itself when the activity pauses, and requerying itself when the activity restarts. You can ask an Activity to begin managing an unmanaged Cursor object for you by calling Activity.startManagingCursor().

The first argument to either query() or managedQuery() is the provider URI - the CONTENT_URI constant that identifies a particular ContentProvider and data set. To restrict a - query to just one record, you can append the _ID value for that record to the URI that is, place a string matching the ID as the last segment of the path part of the URI. For example, if the ID is 23, the URI would be:

content://. . . ./23

Querying a Content Provider : What a query returns

A query returns a set of zero or more database records. The names of the columns, their default order, and their data types are specific to each content provider. But every provider has an _ID column, which holds a unique numeric ID for each record. Every provider can also report the number of records returned as the _COUNT column; its value is the same for all rows.

Here is an example result set for the query in the previous section:

_ID	_COUNT	NAME	NUMBER
44	3	Alan Vain	212 555 1234
13	3	Bully Pulpit	425 555 6677
53	3	Rex Cars	201 555 4433

Querying a Content Provider : Reading retrieved data

The Cursor object returned by a 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. (If there are no matches, it can also be empty.) You can read data from specific fields in the record, but you must know the data type of the field, because the Cursor object has a separate method for reading each type of data - such as getString(), getInt(), and getFloat(). (However, for most types, if you call the method for reading strings, the Cursor object will give you the String representation of the data.) The Cursor lets you request the column name from the index of the column, or the index number from the column name.

Querying a Content Provider : Reading retrieved data

```
import android.provider.Contacts.People;

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: Overview

Data kept by a content provider can be modified by:

- Adding new records
- Adding new values to existing records
- Batch updating existing records
- Deleting records

All data modification is accomplished using ContentResolver methods. Some content providers require a more restrictive permission for writing data than they do for reading it. If you don't have permission to write to a content provider, the ContentResolver methods will fail.

Modifying Data: Adding records

To add a new record to a content provider, first set up a map of key-value pairs in a ContentValues object, where each key matches the name of a column in the content provider and the value is the desired value for the new record in that column. Then call ContentResolver.insert() and pass it the URI of the provider and the ContentValues map. This method returns the full URI of the new record ? that is, the provider's URI with the appended ID for the new record. You can then use this URI to query and get a Cursor over the new record, and to further modify the record. Here's an example:

```
import android.provider.Contacts.People;
import android.content.ContentResolver;
import android.content.ContentValues;

ContentValues values = new ContentValues();

// Add Abraham Lincoln to contacts and make him a favorite.
values.put(People.NAME, "Abraham Lincoln");
// 1 = the new contact is added to favorites
// 0 = the new contact is not added to favorites
values.put(People.STARRED, 1);

Uri uri = getContentResolver().insert(People.CONTENT_URI, values);
```

Modifying Data: Adding new values

Once a record exists, you can add new information to it or modify existing information. For example, the next step in the example above would be to add contact information ? like a phone number or an IM or e-mail address - to the new entry.

The best way to add to a record in the Contacts database is to append the name of the table where the new data goes to the URI for the record, then use the amended URI to add the new data values. Each Contacts table exposes a name for this purpose as a CONTENT_DIRECTORY constant. The following code continues the previous example by adding a phone number and e-mail address for the record just created:

```
Uri phoneUri = null; Uri emailUri = null;
phoneUri = Uri.withAppendedPath(uri, People.Phones.CONTENT_DIRECTORY);
values.clear();
values.put(People.Phones.TYPE, People.Phones.TYPE_MOBILE);
values.put(People.Phones.NUMBER, "1233214567");
getContentResolver().insert(phoneUri, values);
emailUri = Uri.withAppendedPath(uri, People.ContactMethods.CONTENT_DIRECTORY);
values.clear();
values.put(People.ContactMethods.KIND, Contacts.KIND_EMAIL);
values.put(People.ContactMethods.DATA, "test@example.com");
values.put(People.ContactMethods.TYPE, People.ContactMethods.TYPE_HOME);
getContentResolver().insert(emailUri, values);
```

Modifying Data: Adding new values

In this regard, the MediaStore content provider, the main provider that dispenses image, audio, and video data, employs a special convention: The same URI that is used with query() or managedQuery() to get meta-information about the binary data (such as, the caption of a photograph or the date it was taken) is used with openInputStream() to get the data itself. Similarly, the same URI that is used with insert() to put meta-information into a MediaStore record is used with openOutputStream() to place the binary data there. The following code snippet illustrates this convention:

```
ContentValues values = new ContentValues(3);
values.put(Media.DISPLAY_NAME, "road_trip_1");
values.put(Media.DESCRIPTION, "Day 1, trip to Los Angeles");
values.put(Media.MIME_TYPE, "image/jpeg");

Uri uri = getContentResolver().insert(Media.EXTERNAL_CONTENT_URI, values);
 try {
    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);
}
```

Modifying Data: Batch updating records

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

Modifying Data: 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 an SQL WHERE clause defining which rows to delete. (Caution: Be sure to include a valid WHERE clause if you're deleting a general type, or you risk deleting more records than you intended!).

Creating a Content Provider : Overview

To create a content provider, you must:

- Set up a system for storing the 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. Android provides the SQLiteOpenHelper class to help you create a database and SQLiteDatabase to manage it.
- Extend the ContentProvider class to provide access to the data.
- Declare the content provider in the manifest file for your application (AndroidManifest.xml).

The following sections have notes on the last two of these tasks.

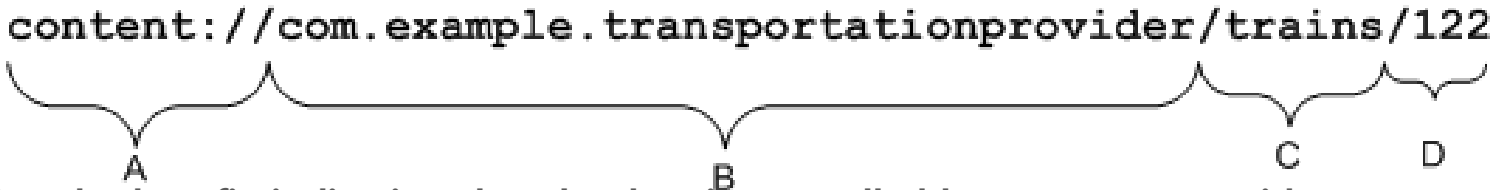
Creating a Content Provider : Extending the ContentProvider class

You define a ContentProvider subclass to expose your data to others using the conventions expected by ContentResolver and Cursor objects. Principally, this means implementing six abstract methods declared in the ContentProvider class:

- query()
- insert()
- update()
- delete()
- getType()
- onCreate()

Declaring the content provider

Content URI Summary



A. Standard prefix indicating that the data is controlled by a content provider. It's never modified.

B. The authority part of the URI; it identifies the content provider. For third-party applications, this should be a fully-qualified class name (reduced to lowercase) to ensure uniqueness.

```
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 long. If the content provider exposes only one type of data (only trains, for example), it can be absent. If the provider exposes several types, including subtypes, it can be several segments long - for example, "land/bus", "land/train", "sea/ship", and "sea/submarine" to give four possibilities.The ID of the specific record being requested, if any.

D. This is the _ID value of the requested record. If the request is not limited to a single record, this segment and the trailing slash are omitted:

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

목 차

Overview

Resources are an integral part of an Android application. In general, these are external elements that you want to include and reference within your application, like images, audio, video, text strings, layouts, themes, etc. Every Android application contains a directory for resources (res/) and a directory for assets (assets/). Assets are used less often, because their applications are far fewer. You only need to save data as an asset when you need to read the raw bites. The directories for resources and assets both reside at the top of your project directory, alongside your source code directory (src/).

The difference between "resources" and "assets" isn't much on the surface, but in general, you'll use resources to store your external content much more often than you'll use assets. The real difference is that anything placed in the resources directory will be easily accessible from your application from the R class, which is compiled by Android. Whereas, anything placed in the assets directory will maintain its raw file format and, in order to read it, you must use the AssetManager to read the file as a stream of bytes. So keeping files and data in resources (res/) makes them easily accessible.

Assets
- Similar to Resources, but...
- InputStream access to Assets
- Placed under assets folder
- Looks like a “root” folder to app
- Read only access
- Any kind of file
- Stored on device - watch the size

Resources and i18n : Introduction

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 Available 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.

Resources and i18n : Creating Resources

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 Declaring Layout
res/values/	XML files that can be compiled into many kinds of resource. 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. •dimens.xml to define dimension value. •strings.xml to define string values •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 and i18n : 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.

Resources and i18n : Using Resources –

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.

Resources and i18n : Using Resources –

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!" />
```

Resources and i18n : Using Resources –

References to Theme Attribute

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" />
```

Resources and i18n : Using Resources –

Using System Resource

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);
    }
}
```

Resources and i18n : Alternate Resources

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	nonav, dpad, trackball, wheel
Screen dimensions	320x240, 640x480, etc. The larger dimension must be specified first.

Resources and i18n : Alternate Resources

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/.

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.

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/.

Available Resource Types – Simple Values

All simple resource values can be expressed as a string, using various formats to unambiguously indicate the type of resource being created. For this reason, these values can be defined both as standard resources (under res/values/), as well as direct values supplied for mappings in styles and themes, and attributes in XML files such as layouts.

- Color Values
  - #RGB
  - #ARGB
  - #RRGGBB
  - #AARRGGBB
- Strings and Styled Text
  - Using Styled Text as a Format String
- Dimension Values
  - px
  - in
  - mm
  - pt
  - dp
  - sp

Available Resource Types – Drawables

A Drawable is a type of resource that you retrieve with Resources.getDrawable() and use to draw to the screen. There are a number of drawable resources that can be created.

Bitmap Files

Android supports bitmap resource files in a few different formats: png (preferred), jpg (acceptable), gif (discouraged). The bitmap file itself is compiled and referenced by the file name without the extension (so res/drawable/my_picture.png would be referenced as R.drawable.my_picture).

Color Drawables

You can create a PaintDrawable object that is a rectangle of color, with optionally rounded corners. This element can be defined in any of the files inside res/values/.

Nine-Patch (stretchable) Images

Android supports a stretchable bitmap image, called a NinePatch graphic. This is a PNG image in which you define stretchable sections that Android will resize to fit the object at display time to accommodate variable sized sections, such as text strings. You typically assign this resource to the View's background. An example use of a stretchable image is the button backgrounds that Android uses; buttons must stretch to accommodate strings of various lengths.

Available Resource Types – Animation

Tweened Animation

Android can perform simple animation on a graphic, or a series of graphics. These include rotations, fading, moving, and stretching.

Source file format: XML file, one resource per file, one root tag with no <?xml> declaration

Resource file location: res/anim/some_file.xml

Compiled resource datatype: Resource pointer to an Animation.

Resource reference name:

- Java: R.anim.some_file
- XML: @[package:]anim/some_file

Syntax

The file must have a single root element: this will be either a single <alpha>, <scale>, <translate>, <rotate>, interpolator element, or <set> element that holds groups of these elements (which may include another <set>). By default, all elements are applied simultaneously. To have them occur sequentially, you must specify the startOffset attribute.

Available Resource Types – Menus

Application menus (Options Menu, Context Menu, or Sub Menu) can be defined as XML resources and inflated by your application using MenuInflater.

Source file format: XML file, one resource per file, one root tag, <?xml> declaration not required.

Resource file location: res/menu/some_file.xml

Compiled resource datatype: Resource pointer to a Menu (or subclass) resource.

Resource reference name:

- Java: R.menu.some_file

Syntax

The file must have a single root element: a <menu> element. In all, there are three valid elements: <menu>, <group> and <item>. The <item> and <group> elements must be the children of a <menu>, but <item> elements can also be the children of a <group>, and another <menu> element may be the child of an <item> (to create a Sub Menu).

Available Resource Types – Layout

Android lets you specify screen layouts using XML elements inside an XML file, similar to design ing screen layout for a webpage in an HTML file. Files are saved in the res/layout/ folder of your p roject, and compiled by the Android resource compiler, aapt.

Resource file location: res/layout/some_file.xml.

Compiled resource datatype: Resource pointer to a View (or subclass) resource.

Resource reference name:

Java: R.layout.some_file

XML: @[package:]layout/some_file

Syntax

```
<ViewGroupClass xmlns:android="http://schemas.android.com/apk/res/android"
        id="@+id/string_name" (attributes)>
  <widget or other nested ViewGroupClass>+
  <requestFocus/>(0 or 1 per layout file, assigned to any element)
</ViewGroupClass>
```

Available Resource Types – Layout (Custom Layout Resources)

You can define custom elements to use in layout resources. These custom elements can then be used the same as any Android layout elements: that is, you can use them and specify their attributes in other resources. The ApiDemos sample application has an example of creating a custom layout XML tag, LabelView. To create a custom element, you will need the following files:

- Java implementation file - The implementation file. The class must extend View or a subclass. See LabelView.java in ApiDemos.
- res/values/attrs.xml - Defines the XML element, and the attributes that it supports, for clients to use to instantiate your object in their layout XML file. Define your element in a <declare-styleable id=your_java_class_name>. See res/layout/attrs.xml in ApiDemos.
- res/layout/your_class.xml [optional] - An optional XML file to describe the layout of your object. This could also be done in Java. See custom_view_1.xml in ApiDemos.

Source file format: XML file without an <?xml> declaration, and a <resources> root element containing one or more custom element tags.

Resource file location: res/values/attrs.xml (file name is arbitrary).

Compiled resource datatype: Resource pointer to a View (or subclass) resource.

Resource reference name: R.styleable.some_file (Java).

Available Resource Types – Styles and Themes

A style is one or more attributes applied to a single element (for example, 10 point red Arial font, applied to a TextView). A style is applied as an attribute to an element in a layout XML file.

A theme is one or more attributes applied to a whole screen - for example, you might apply the stock Android Theme.dialog theme to an activity designed to be a floating dialog box. A theme is assigned as an attribute to an Activity in the manifest file.

Resource source file location: res/values/styles.xml (file name is arbitrary). The file name is arbit rary, but standard practice is to put all styles into a file named styles.xml.

Compiled resource datatype: Resource pointer to a Java CharSequence.

Resource reference name:

- Java: R.style.styleID for the whole style, R.style.styleID.itemID for an individual setting
- XML: @[package:]style/styleID for a whole style, @[package:]style/styleID/itemID for an in dividual item. Note: to refer to a value in the currently applied theme, use "?" instead of "@ " as described below (XML).

Syntax

```
<style name=string [parent=string] >
  <item name=string>Hex value | string value | reference</item>+
</style>
```

목 차

Overview

Android is a multi-process system, in which 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.

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 data stored 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.

Using Permissions (cont.)

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>
```

Declaring and Enforcing Permissions - protectionLevel

Constant	V	Description
normal	0	A lower-risk permission that gives an application access to isolated application-level features, with minimal risk to other applications, the system, or the user. The system automatically grants this type of permission to a requesting application at installation, with out asking for the user's explicit approval (though the user always has the option to review these permissions before installing).
dangerous	1	A higher-risk permission that would give a requesting application access to private user data or control over the device that can negatively impact the user. Because this type of permission introduces potential risk, the system may not automatically grant it to the requesting application. For example, any dangerous permissions requested by an application may be displayed to the user and require confirmation before proceeding, or some other approach may be taken to avoid the user automatically allowing the use of such facilities.
signature	2	A permission that the system is to grant only if the requesting application is signed with the same certificate as the application that declared the permission. If the certificates match, the system automatically grants the permission without notifying the user or asking for the user's explicit approval.
Signature OrSystem	3	A permission that the system is to grant only to packages in the Android system image *or* that are signed with the same certificates. Please avoid using this option, as the signature protection level should be sufficient for most needs and works regardless of exactly where applications are installed. This permission is used for certain special situations where multiple vendors have applications built in to a system image which need to share specific features explicitly because they are being built together.

Declaring and 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).

Declaring and Enforcing Permissions in AndroidManifest.xml (cont.)

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 Permissions 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.

목 차

Graphics Overview

Android graphics are powered by a custom 2D graphics library and OpenGL ES 1.0 for high performance 3D graphics. The most common 2D graphics APIs can be found in the drawable package. OpenGL APIs are available from the Khronos OpenGL ES package, plus some Android OpenGL utilities.

When starting a project, it's important to consider exactly what your graphical demands will be. Varying graphical tasks are best accomplished with varying techniques. For example, graphics and animations for a rather static application should be implemented much differently than graphics and animations for an interactive game or 3D rendering.

Here, we'll discuss a few of the options you have for drawing graphics on Android, and which tasks they're best suited for.

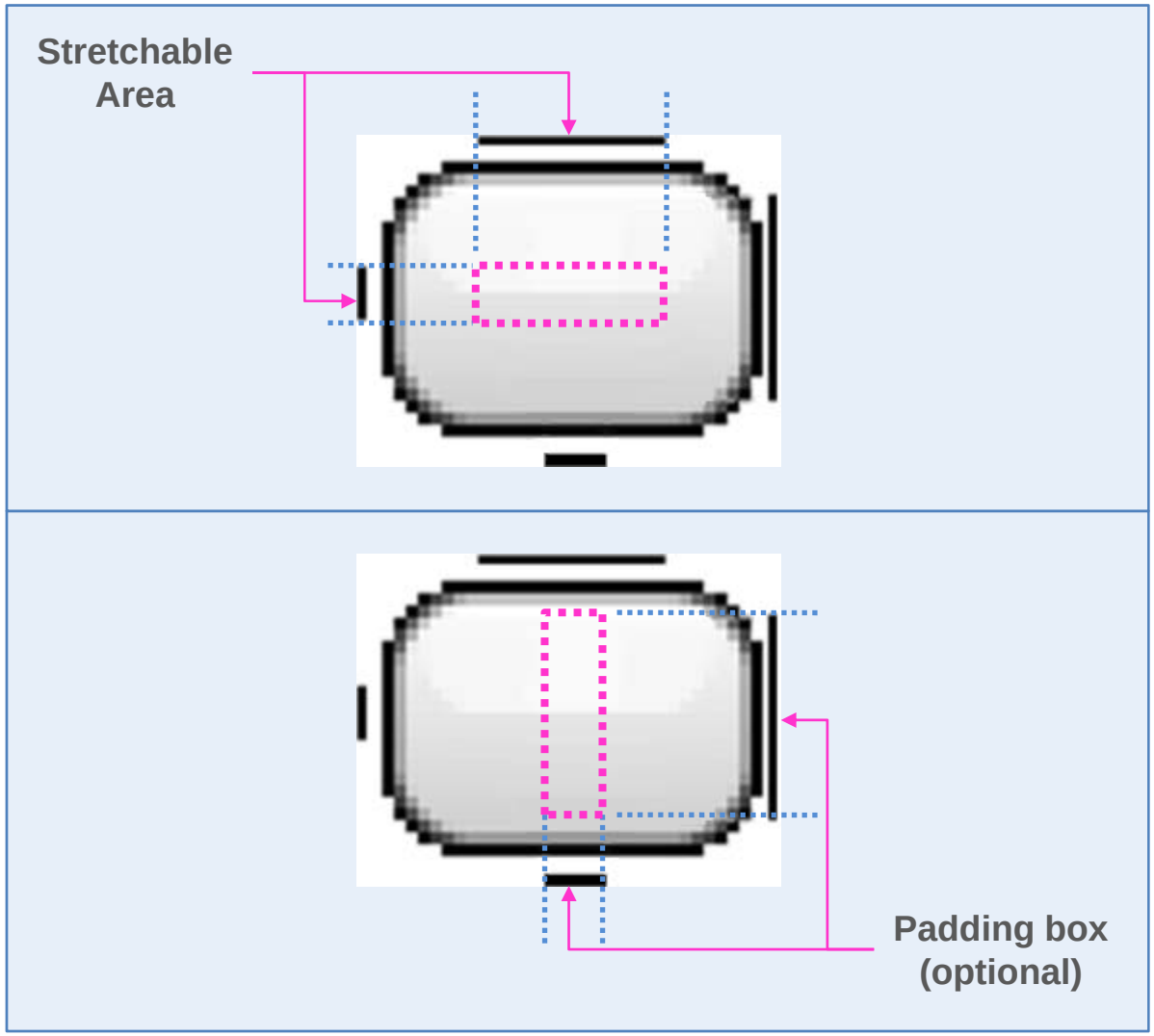
If you're specifically looking for information on drawing 3D graphics, this page won't help a lot. However, the information below, on Drawing with a Canvas (and the section on SurfaceView), will give you a quick idea of how you should draw to the View hierarchy. For more information on Android's 3D graphic utilities (provided by the OpenGL ES API), read 3D with OpenGL and refer to other OpenGL documentation.

2D Graphics

Android offers a custom 2D graphics library for drawing and animating shapes and images. The android.graphics.drawable and android.view.animation packages are where you'll find the common classes used for drawing and animating in two-dimensions.

- Drawables
  - Creating from resource images
  - Creating from resource XML
- ShapeDrawable
- NinePatchDrawable
- Tween Animation
- Frame Animation

2D Graphics (cont.)



Tiny Biiiiiiig text!

3D with OpenGL

Android includes support for high performance 3D graphics via the OpenGL API ? specifically, the OpenGL ES API.

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

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

Using the API

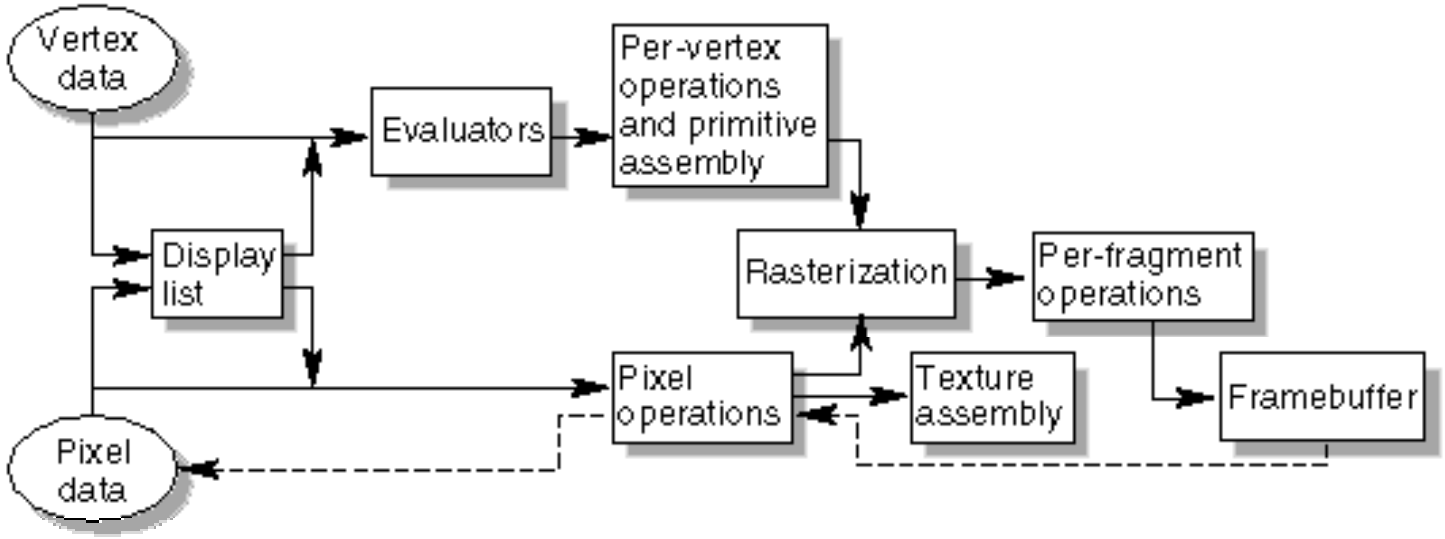
Here's how to use the API at an extremely high level:

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

3D with OpenGL (cont.)

What is OpenGL ES?

- **Royalty-free, cross-platform API for full-function 2D and 3D graphics on embedded system**
- **3D API for Symbian, Android, iPhone, etc**
- **Stripped down version of OpenGL**
  - **Standard specification for cross-language, cross-platform API for 2D & 3D graphics**
  - **About 150 commands specifying objects and operations**



Audio and Video

The Android platform offers built-in encoding/decoding for a variety of common media types, so that you can easily integrate audio, video, and images into your applications. Accessing the platform's media capabilities is fairly straightforward ? you do so using the same intents and activities mechanism that the rest of Android uses.

Android lets you play audio and video from several types of data sources. You can play audio or video from media files stored in the application's resources (raw resources), from standalone files in the filesystem, or from a data stream arriving over a network connection. To play audio or video from your application, use the MediaPlayer class.

The platform also lets you record audio, where supported by the mobile device hardware. Recording of video is not currently supported, but is planned for a future release. To record audio, use the MediaRecorder class. Note that the emulator doesn't have hardware to capture audio, but actual mobile devices are likely to provide these capabilities that you can access through MediaRecorder.

Audio and Video (cont.) – Core Media Formats

Type	Format	E	D	Details	File Type(s) Supported
Audio	AAC LC/LTP		O	Mono/Stereo content in any combination of standard bit rates up to 160 kbps and sampling rates from 8 to 48kHz	3GPP (.3gp) and MPEG-4 (.mp4, .m4a). No support for raw AAC (.aac)
	HE-AACv1 (AAC+)		O		
	HE-AACv2 (enhanced AAC+)		O		
	AMR-NB	O	O	4.75 to 12.2 kbps sampled @ 8kHz	3GPP (.3gp)
	AMR-WB		O	9 rates from 6.60 kbit/s to 23.85 kbit/s sampled @ 16kHz	3GPP (.3gp)
	MP3		O	Mono/Stereo 8-320Kbps constant (CBR) or variable bit-rate (VBR)	MP3 (.mp3)
	MIDI		O	MIDI Type 0 and 1. DLS Version 1 and 2. XMF and Mobile XMF. Support for ringtone formats RTTTL/RTX, OTA, and iMelody	Type 0 and 1 (.mid, .xmf, .mxmf) . Also RTTTL/RTX (.rtttl, .rtx), OTA (.ota), and iMelody (.imy)
	Ogg Vorbis		O		Ogg (.ogg)
	PCM/WAVE		O	8- and 16-bit linear PCM (rates up to limit of hardware)	WAVE (.wav)
Video	H.263	O	O		3GPP (.3gp)
	H.264	O	O		3GPP (.3gp) and MPEG-4 (.mp4)
	MPEG4 SP				3GPP (.3gp) and MPEG-4 (.mp4)
Image	JPEG	O	O	base+progressive	JPEG (.jpg)
	GIF,PNG,BMP		O		GIF (.gif) PNG (.png) BMP (.bmp)

Location - android.location

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

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

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

Providing Mock Location Data

- Using DDMS
- Using the "geo" command

Location – com.google.maps

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

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

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

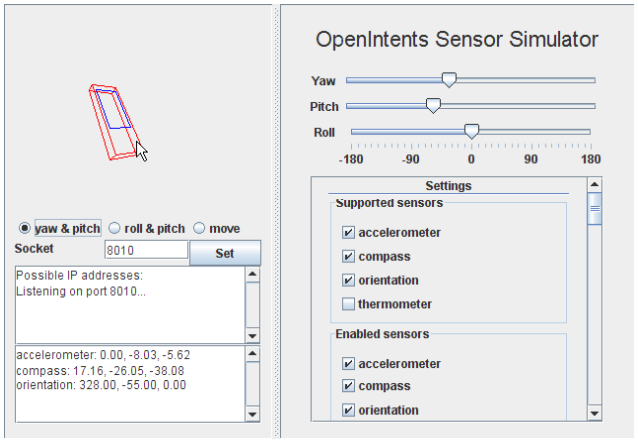
This is not a standard package in the Android library. In order to use it, you must add the following node to your Android Manifest file, as a child of the <application> element:

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

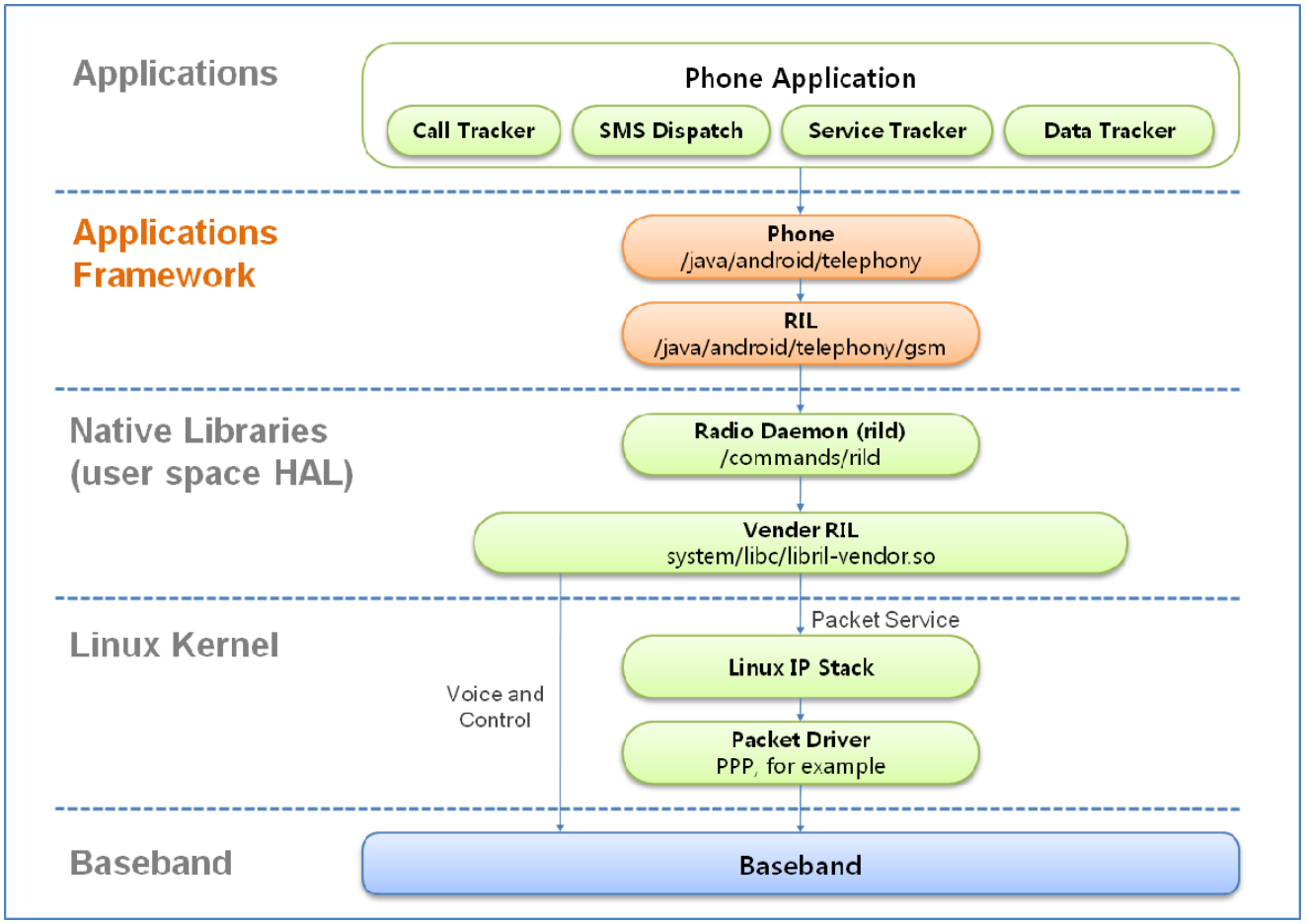
Sensor

The Android SDK supports many different types of sensor devices, including:

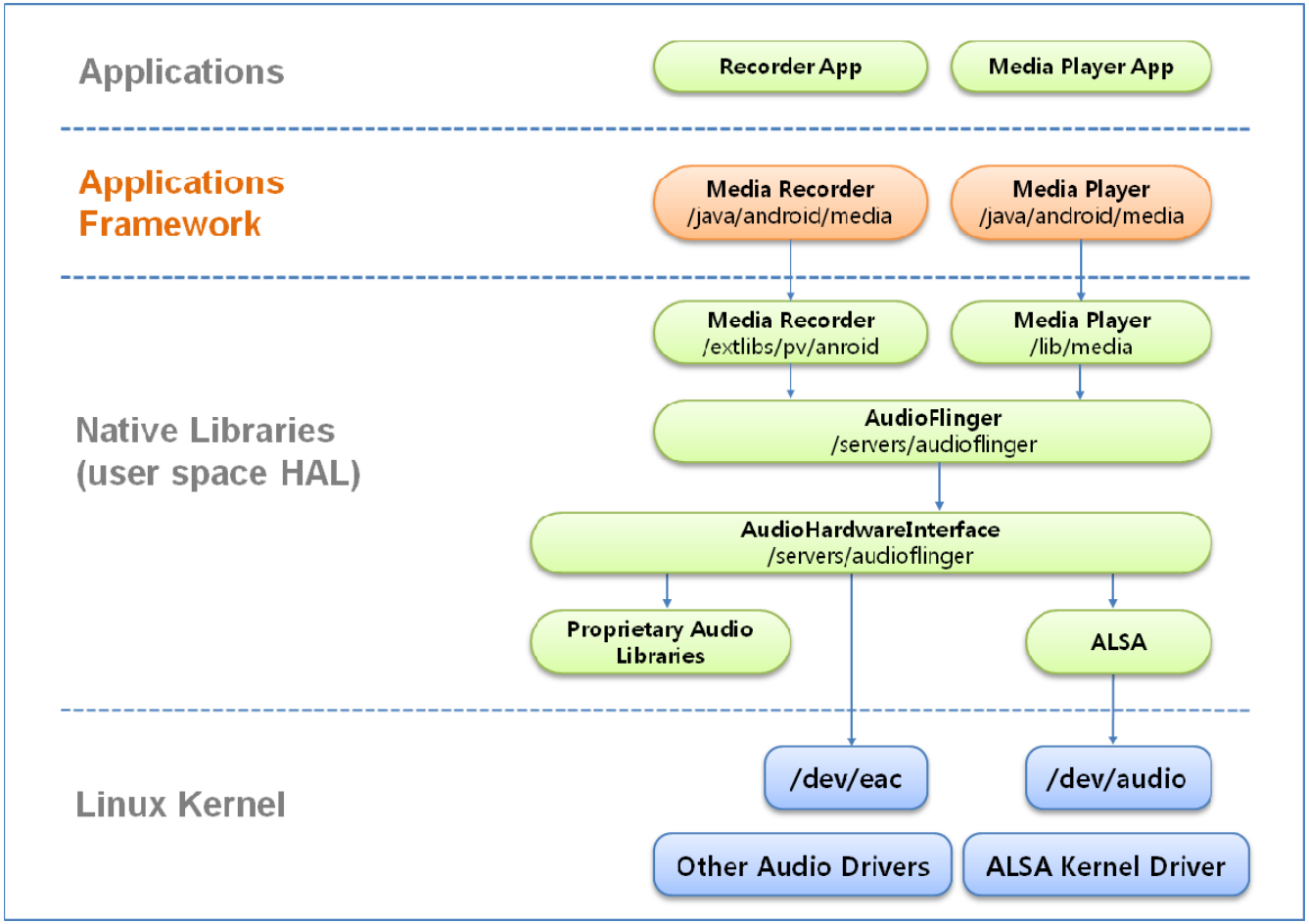
- SENSOR_ACCELEROMETER - measures acceleration in the X, Y, and Z axis.
- SENSOR_LIGHT - tells you how bright your surrounding area is.
- SENSOR_MAGNETIC_FIELD - returns magnetic attraction in the X, Y,and Z axis.
- SENSOR_ORIENTATION - measures the yaw, pitch, and roll of the device.
- SENSOR_ORIENTATION_RAW - same thing without filtering.
- SENSOR_PROXIMITY - provides the distance between the sensor and some object.
- SENSOR_TEMPERATURE - measures the temperature of the surrounding area.
- SENSOR_TRICORDER - turns your device into a fully functional Star Trek Tricorder.



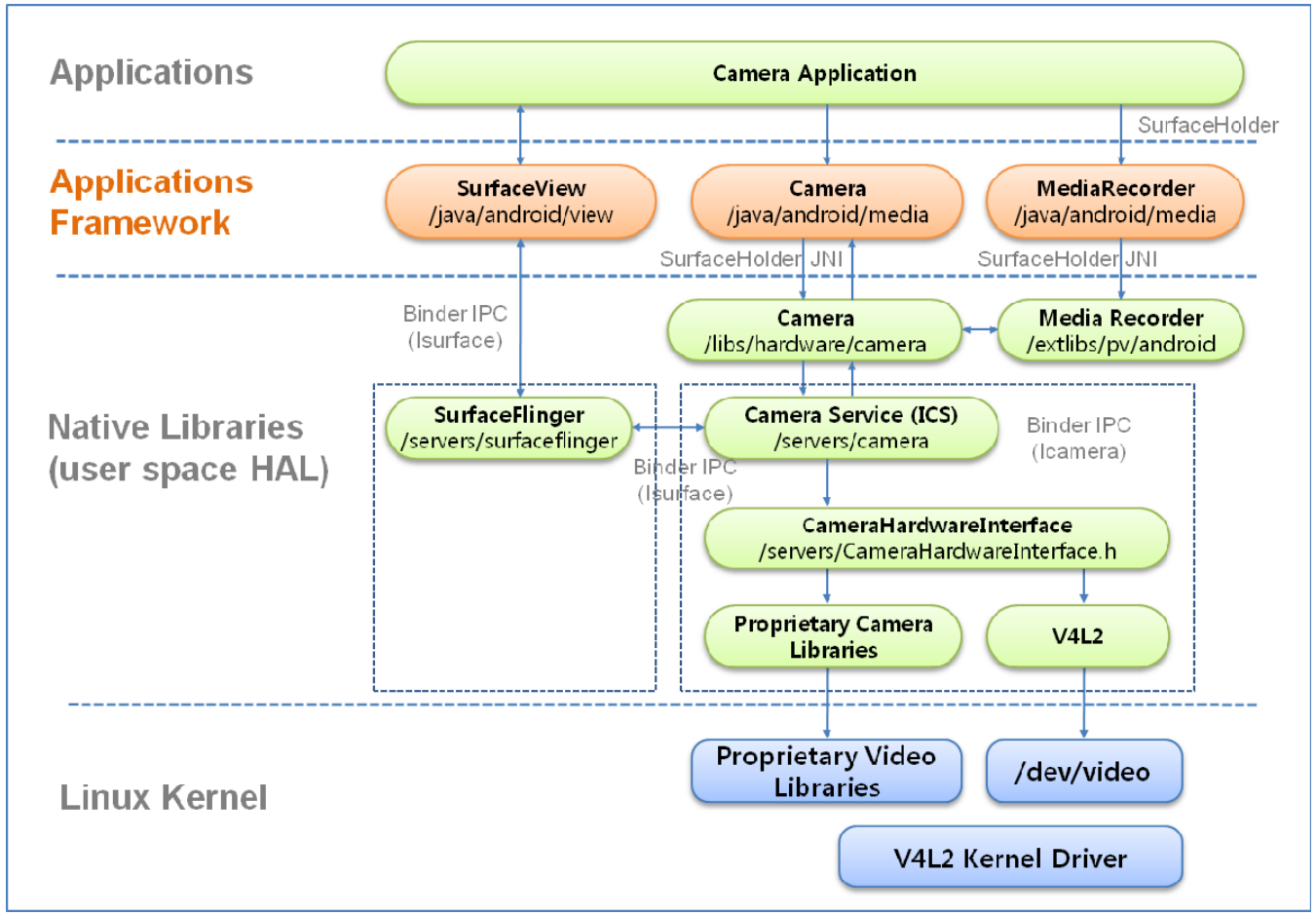
Hardware Service : Telephony



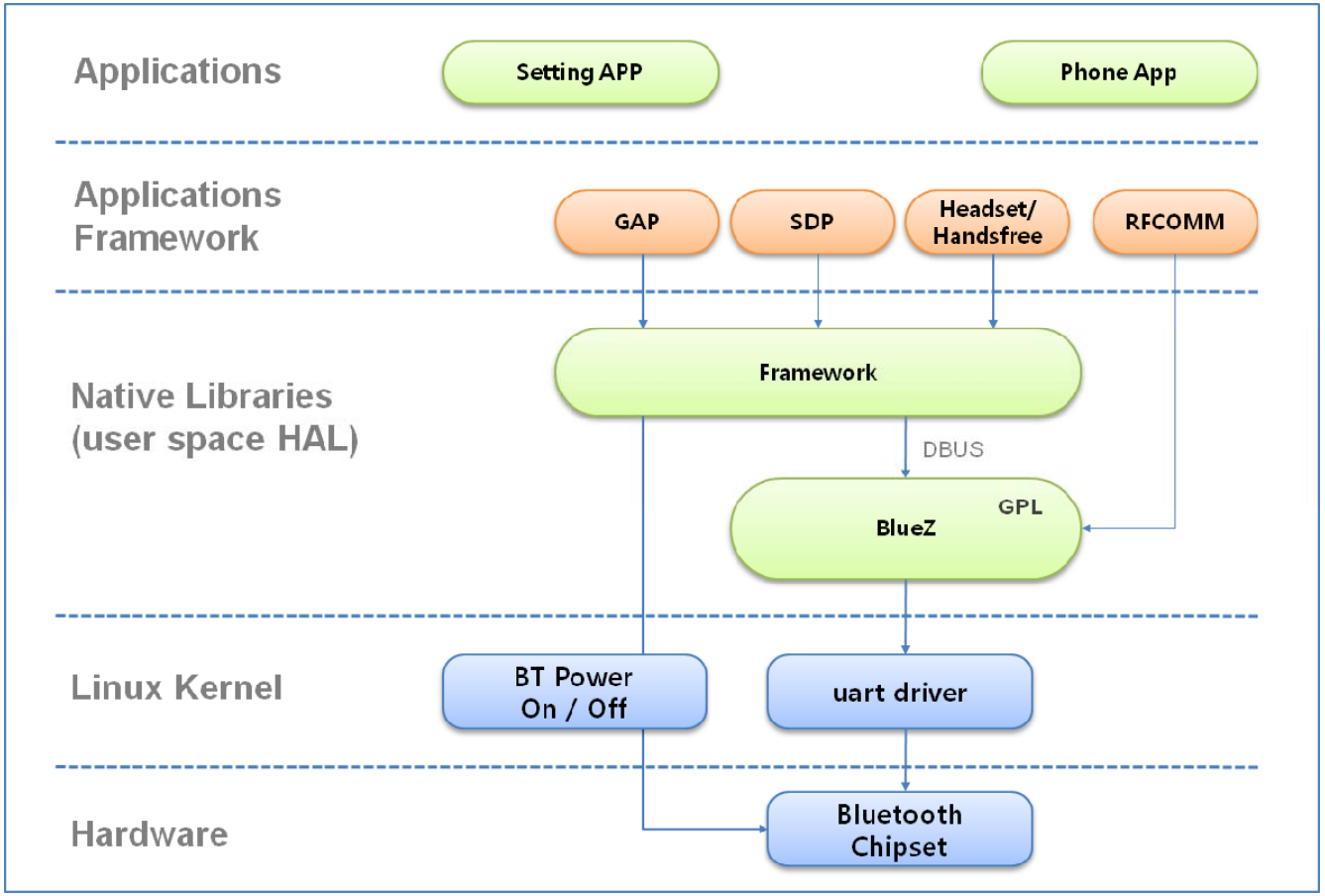
Hardware Service : Audio



Hardware Service : Camera



Hardware Service : Bluetooth



목 차

Signing Your Applications :

map 관련 apikey와 apk signing 관련 요약

> "c:Program FilesJava\jdk1.6.0_10\bin\keytool.exe" -genkey -keystore keystore -validity 10000 -alias kandroid
실행후...입력할 것들을 모두 입력함.

> "c:Program Files\Java\jdk1.6.0_10\bin\keytool.exe" -list -keystore keystore
실행 후, 인증서 지문(MD5) : xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx 을 http://code.google.com/android/maps-api-signup.html 에 접속하여 입력한 후, Map apikey를 받아서 mapview layout에서 사용함.

안드로이드 **mapview** 관련 **apk**를 빌드한 후,
다음과 같은 방식으로 **apk**에 **signing**을 함.

> "c:Program Files\Java\jdk1.6.0_10\bin\jarsigner.exe" -verbose -keystore keystore 안드로이드.apk kandroid

Android Market : Overview

http://www.android.com/market/

개발자 등록비 : 25$

수익배분 : 30(통신사 또는 결제대행 수수료) 대 70 (개발자)

현재 무료 애플리케이션만 실제 디바이스에 다운로드 서비스 중

2009년 부터는 상용 애플리케이션 서비스 개시 예정

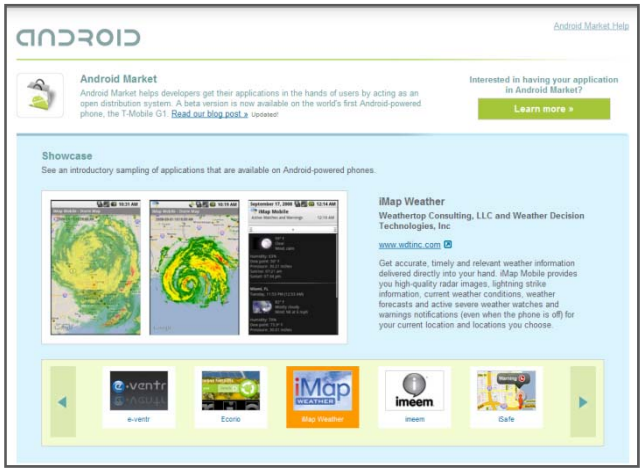
애플리케이션 등록에 제한을 가하지 않음

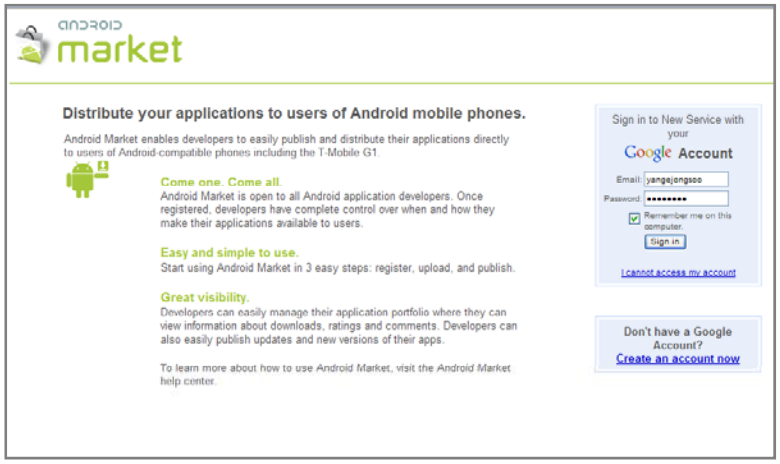
구입전 미리 사용해 보기 기능 제공

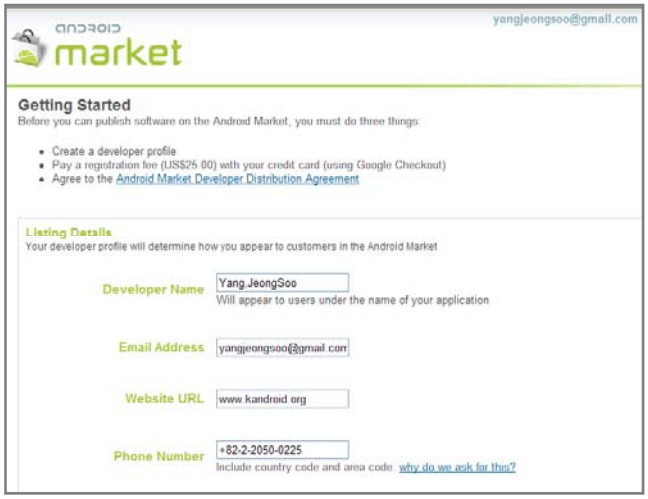
“킬 스위치”

구글이 사용자 폰에 저장된 어플 삭제 가능
삭제된 어플에 대해 24시간이내 환불
제한없는 어플 등록에 따르는 필요악

Android Market : 개발자 등록과정

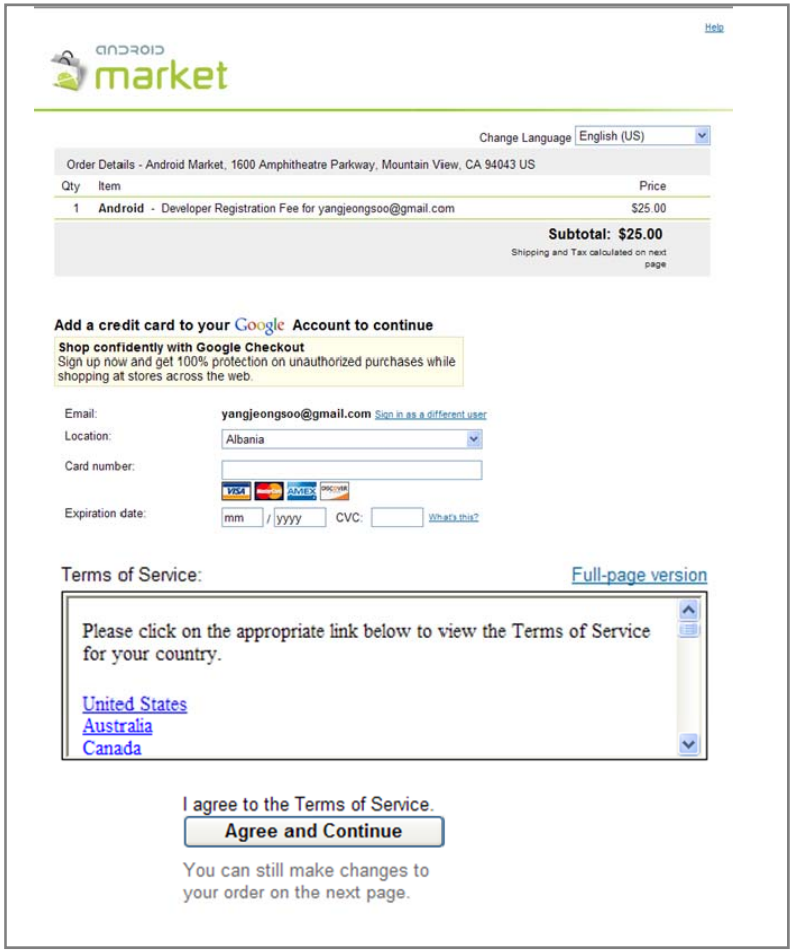


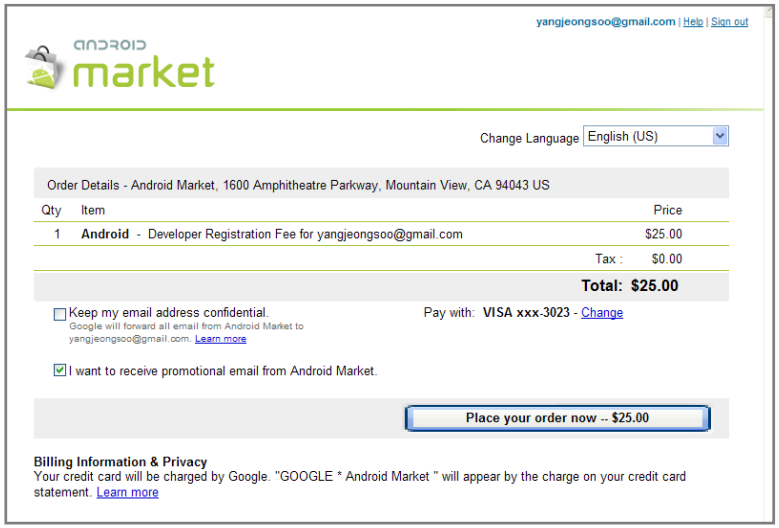


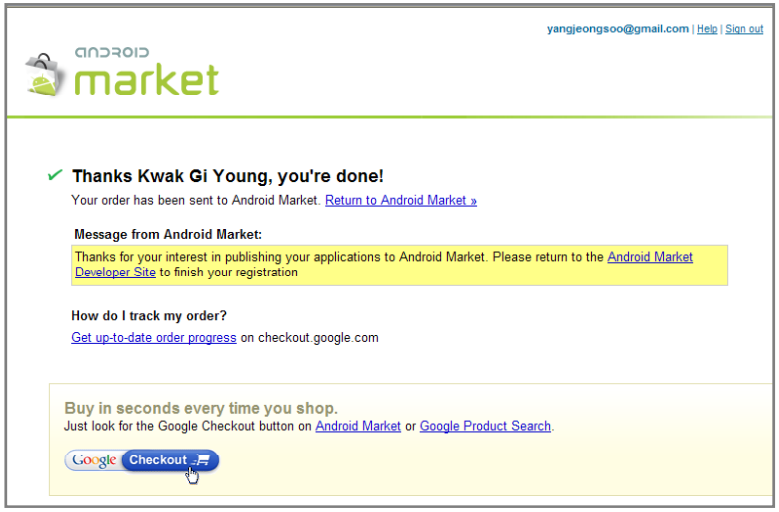


정리

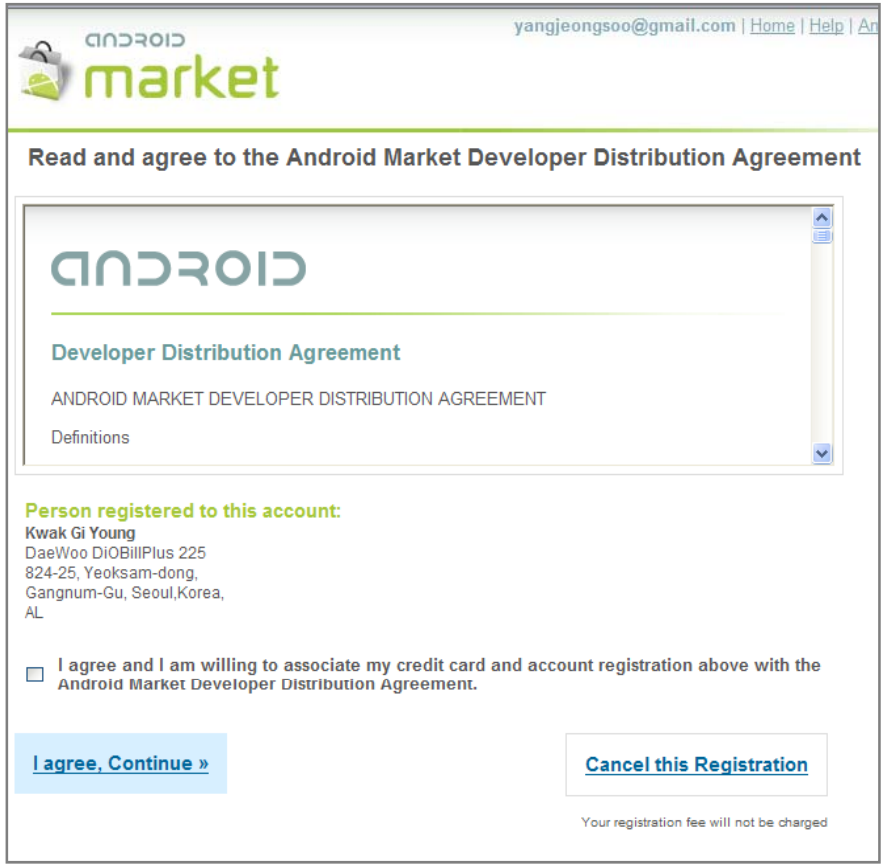
Android Market : 개발자 등록과정

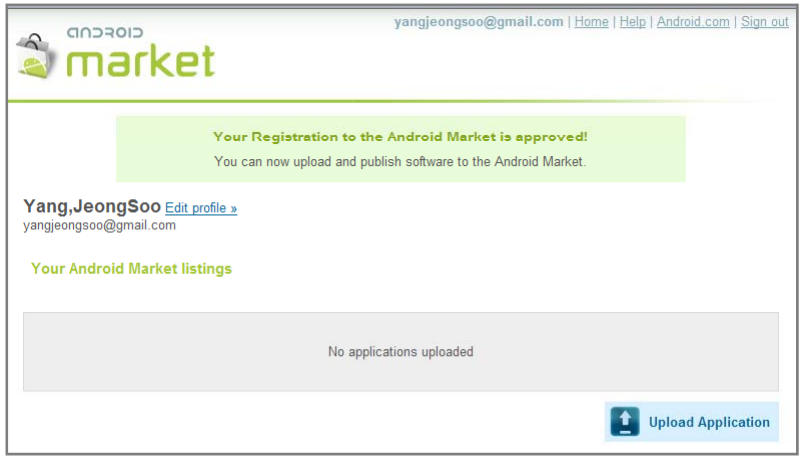




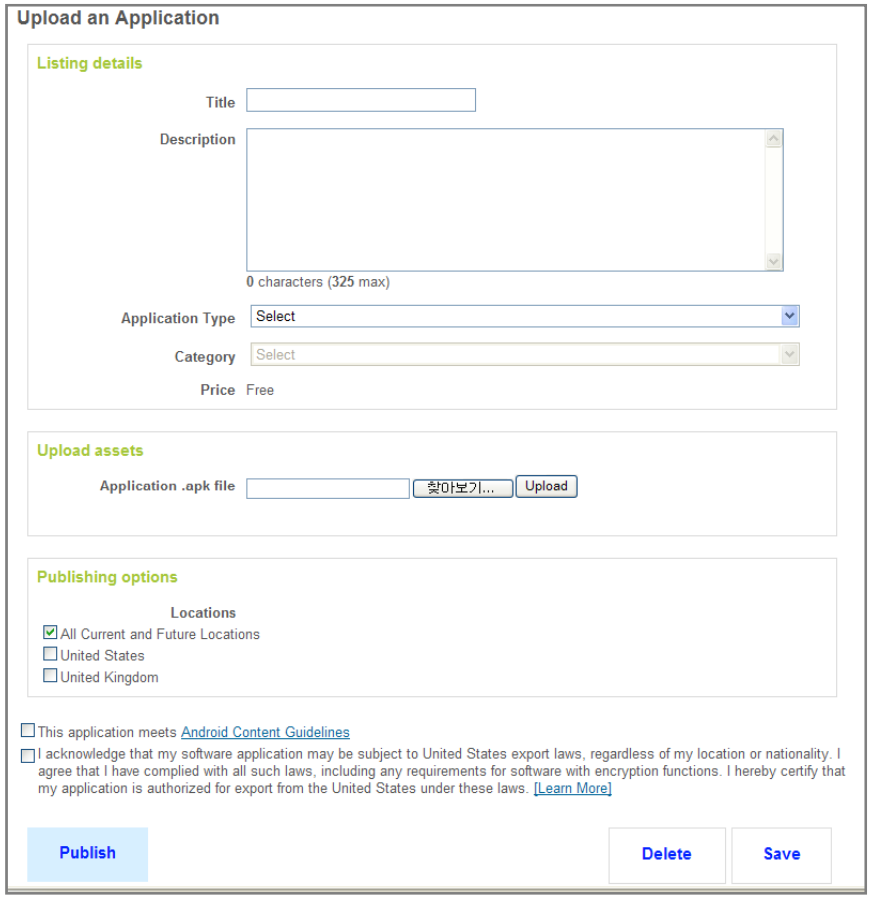


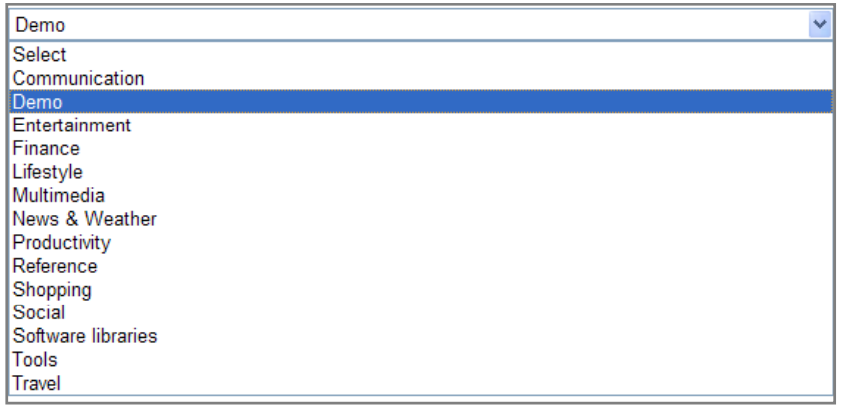
Android Market : 개발자 등록과정



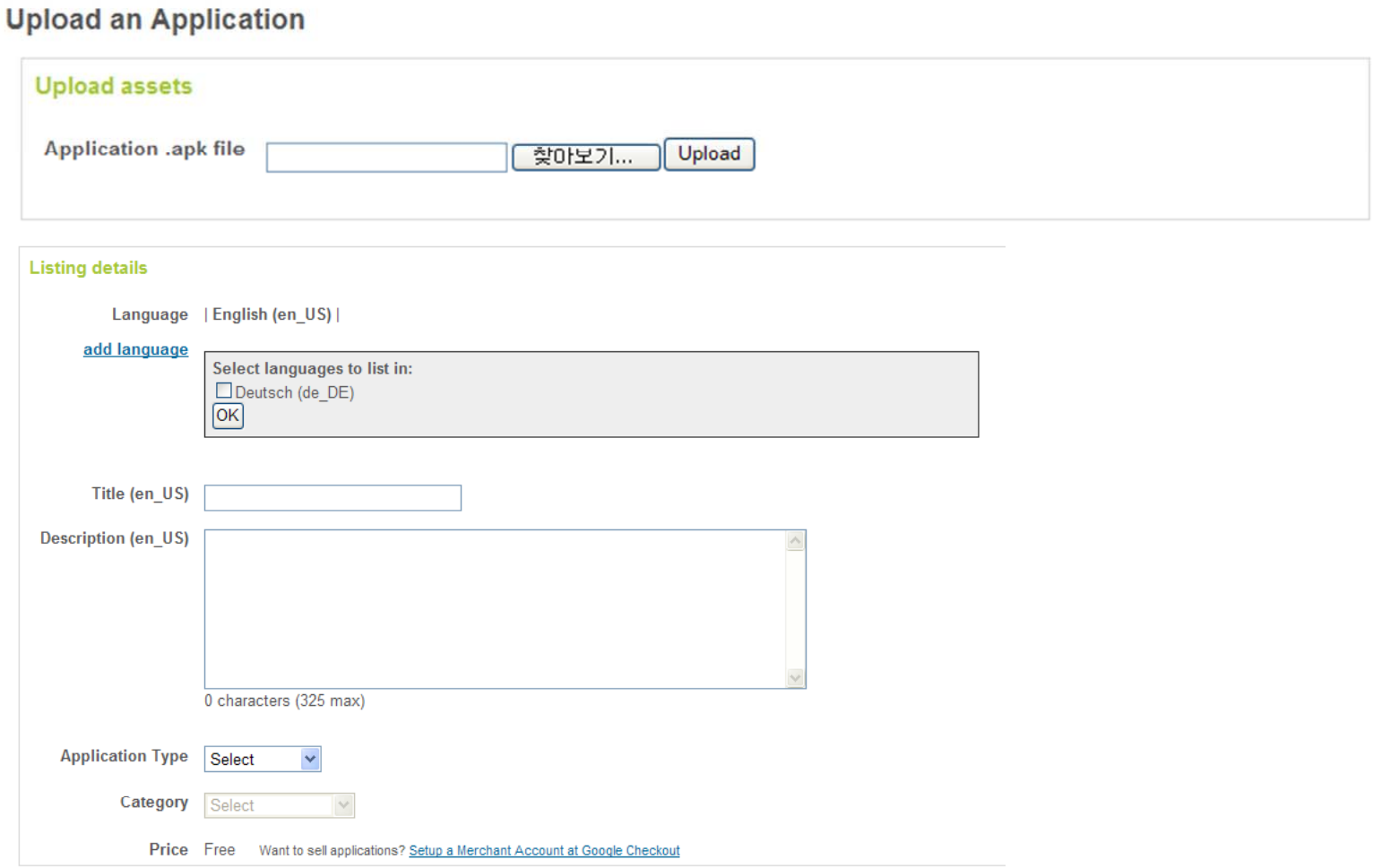


Android Market : 개발자 등록과정





Android Market : Publishing



Android Market : Publishing

Publishing options

Copy Protection
- (•) Off (Application can be copied from the device)
- ( ) On (Helps prevent copying of this application from the device. Increases the amount of memory required by users to install the application)

Locations
- [x] All Current and Future Locations
- [ ] Australia
- [ ] Austria
- [ ] Czech Republic
- [ ] Germany
- [ ] Netherlands
- [ ] Poland
- [ ] Singapore
- [ ] United Kingdom
- [ ] United States

Contact information

Website: www.kandroid.org

Email: yangjeongsoo@gmail.com

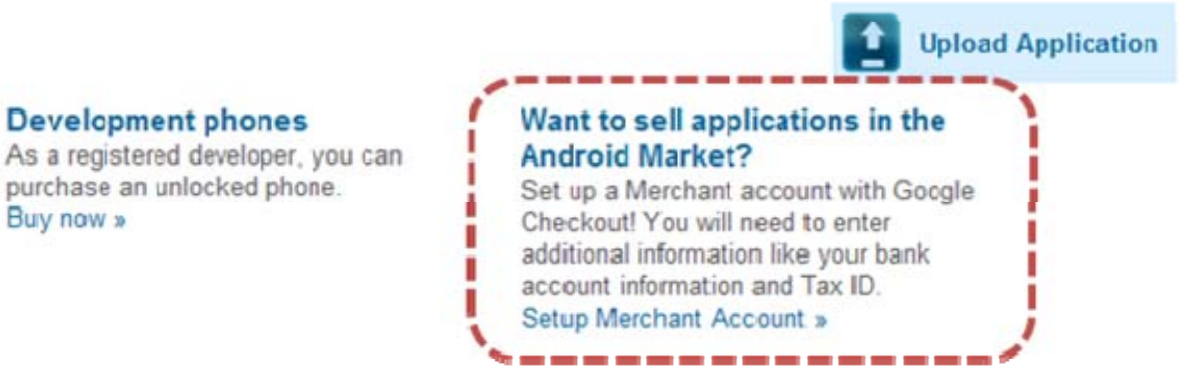
Phone: +82-2-2050-0225

- [ ] This application meets Android Content Guidelines
- [ ] I acknowledge that my software application may be subject to United States export laws, regardless of my location or nationality. I agree that I have complied with all such laws, including any requirements for software with encryption functions. I hereby certify that my application is authorized for export from the United States under these laws. [Learn More]

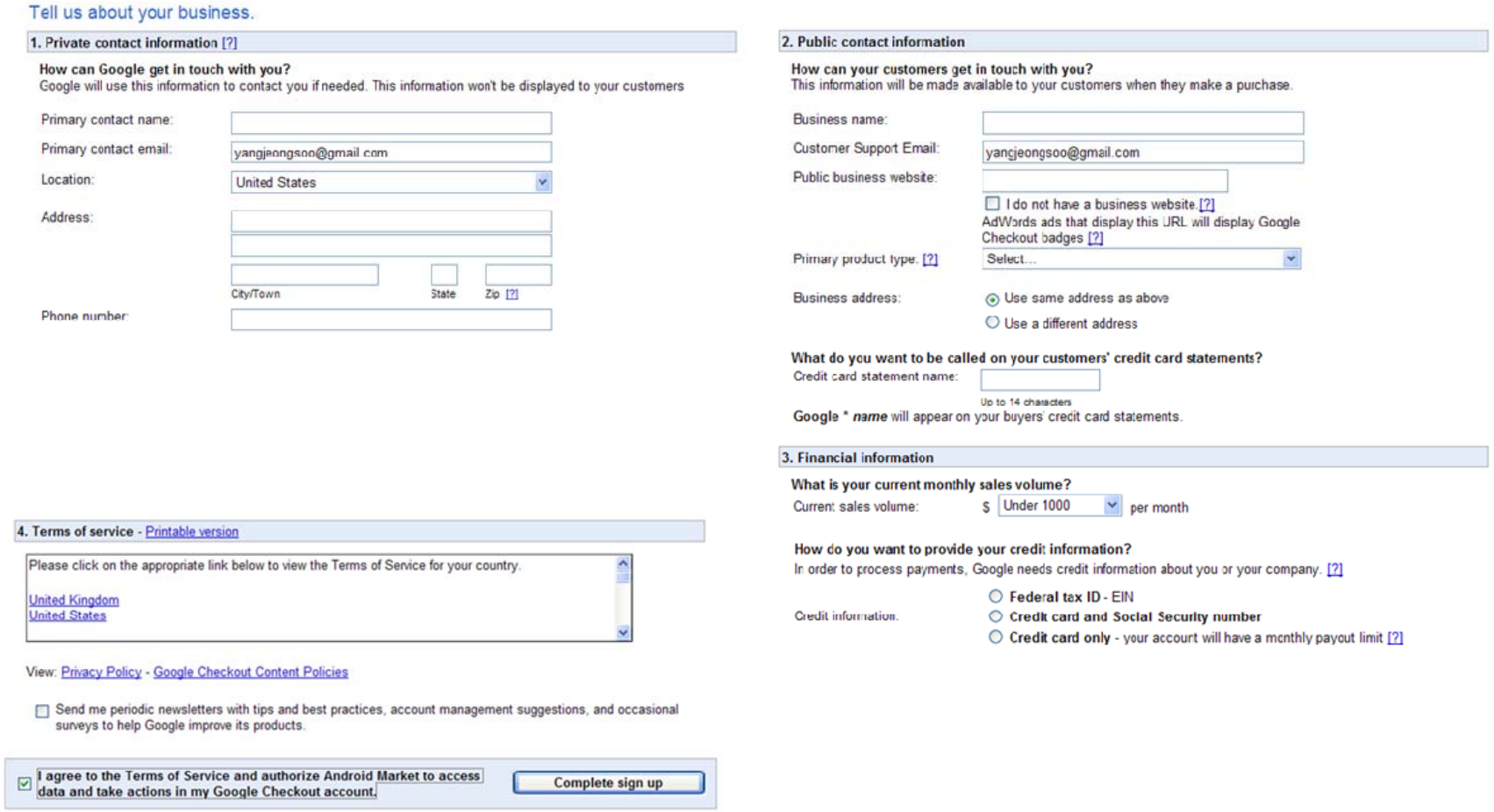
Publish | Delete | Save

Android Market : Commercial Publishing

http://market.android.com/publish



Android Market : Commercial Publishing



Android Roadmap

최근 Android 1.5 Cupcake를 통해 위의 로드맵이 대부분 완성되어 가고 있으며,
예측컨대, ADCII 는 Cupcake기반의 firmware Update 및 SDK 배포 후 이루어 질 것으로 예상됨.

2007.11.05 : 오픈 핸드셋 얼라이언스(OHA)에서 안드로이드 발표
2007.11.12 : 최초의 안드로이드 SDK 배포
2008.04.17 : 안드로이드 개발자 챌린지 I 종료
2008.08.28 : 안드로이드 마켓 발표
2008.09.23 : T-Mobile G1 발표
2008.09.23 : 안드로이드 1.0 SDK Release 1 배포
2008.10.21 : 안드로이드 전체 소스 공개

2008. 4/4 분기
• Localization : 문자열, 사용자 인터페이스, 날짜 및 숫자 표기
• Support for multiple APNs
• SIM application toolkit (STK)

Q4 2008 - Key Announcement on Android Developer Challenge II

2009. 1/4 분기
• Input method framework (IMF)
• Input method engines (IME)

정리

Q & A