Notification

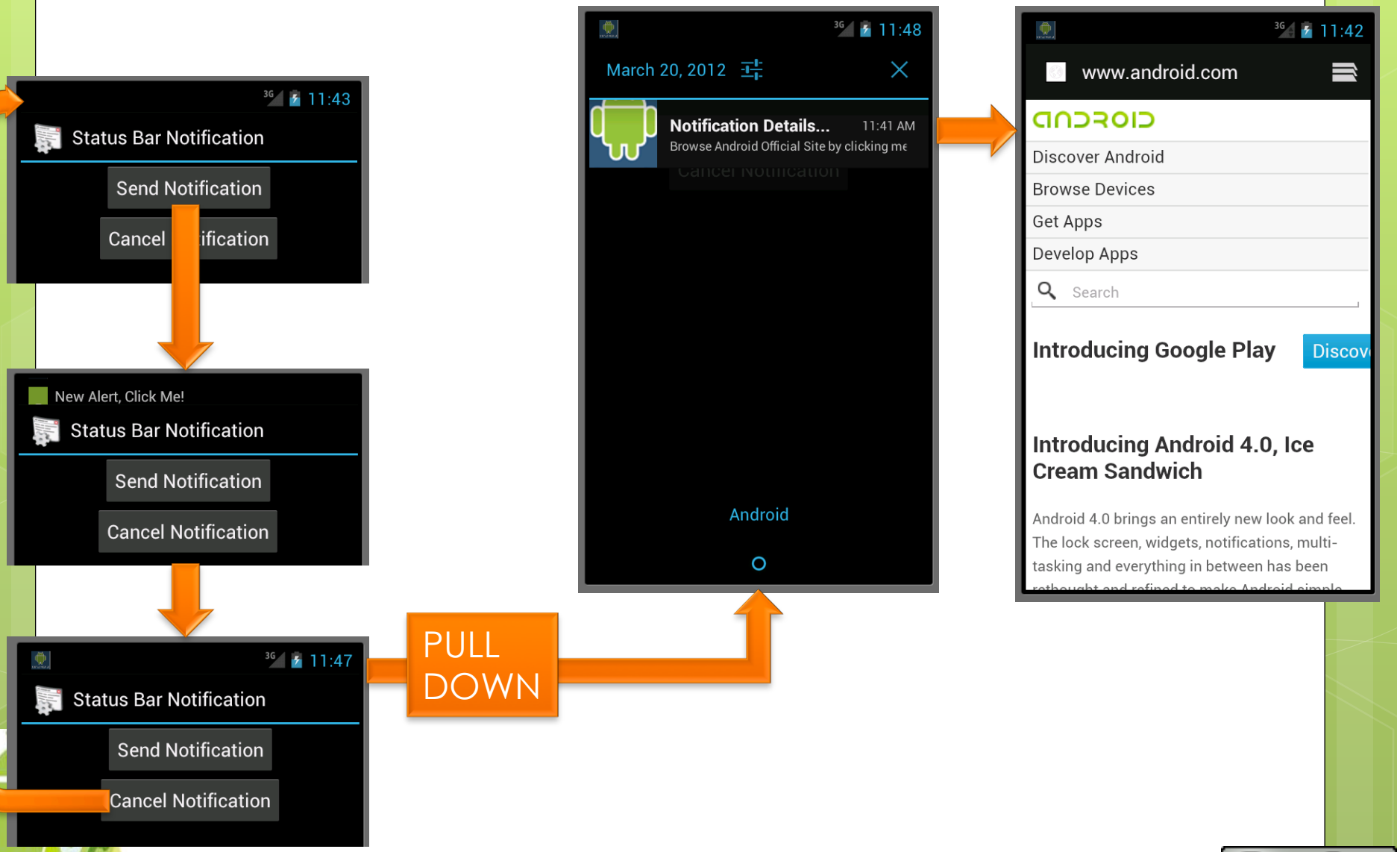
Marco Ronchetti
Università degli Studi di Trento

Pending Intent

A PendingIntent is a token that you give to another application, which allows it to use the permissions of your application to execute a predefined piece of code.

```
Intent notificationIntent = new Intent(this, MyClass.class);
PendingIntent contentIntent =
PendingIntent.getActivity(this, 0, notificationIntent, 0);
notification.setLatestEventInfo(context, "Title",
            "something went wrong", contentIntent);
```

Notification Bar



SimpleNotification

```
public class SimpleNotification extends Activity {
   private NotificationManager nm;
   private int SIMPLE_NOTIFICATION_ID;
   @Override
   public void onCreate(Bundle savedInstanceState) {
      super.onCreate(savedInstanceState);
      setContentView(R.layout.main);
      nm = (NotificationManager)getSystemService(NOTIFICATION_SERVICE);
      final Notification notifyDetails = new Notification(
            R.drawable.android,"New Alert, Click Me!",
            System.currentTimeMillis());
      Button cancel = (Button)findViewById(R.id.cancelButton);
      cancel.setOnClickListener(new OnClickListener() {
         public void onClick(View v) {
            nm.cancel(SIMPLE_NOTIFICATION_ID);
         }});}
```

Adapted from http://saigeethamn.blogspot.it

SimpleNotification – part 2

```
Button start = (Button)findViewById(R.id.notifyButton);
start.setOnClickListener(new OnClickListener() {
   public void onClick(View v) {
      Context context = getApplicationContext();
      CharSequence contentTitle = "Notification Details...";
      CharSequence contentText = "Browse Android Site by clicking me";
      Intent notifyIntent = new Intent
         (android.content.Intent.ACTION_VIEW,
         Uri.parse("http://www.android.com"));
      PendingIntent intent =
      PendingIntent.getActivity(SimpleNotification.this, 0, notifyIntent,
           android.content.Intent.FLAG_ACTIVITY_NEW_TASK);
      notifyDetails.setLatestEventInfo(context, contentTitle,
          contentText, intent);
      nm.notify(SIMPLE_NOTIFICATION_ID, notifyDetails);
      }
   });
}}
```

Flags

int DEFAULT_ALL	Use all default values (where applicable).
int DEFAULT_LIGHTS	Use the default notification lights.
int DEFAULT_SOUND	Use the default notification sound.
int DEFAULT_VIBRATE	Use the default notification vibrate.

Bit to be bitwise-ored into the flags field

int FLAG_AUTO_CANCEL

- should be set if the notification should be canceled when it is clicked by the user.

int FLAG_FOREGROUND_SERVICE

- should be set if this notification represents a currently running service.

int FLAG_HIGH_PRIORITY

- should be set if this notification represents a high-priority event that may be shown to the user even if notifications are otherwise unavailable (that is, when the status bar is hidden).

int FLAG_INSISTENT

- if set, the audio will be repeated until the notification is cancelled or the notification window is opened.

int FLAG_NO_CLEAR

- should be set if the notification should not be canceled when the user clicks the Clear all button.

int FLAG_ONGOING_EVENT

- should be set if this notification is in reference to something that is ongoing, like a phone call.

int FLAG_ONLY_ALERT_ONCE

- should be set if you want the sound and/or vibration play each time the notification is sent, even if it has not been canceled before that.

int FLAG_SHOW_LIGHTS

- should be set if you want the LED on for this notification.

7

Broadcast receivers

Marco Ronchetti
Università degli Studi di Trento

Bradcast receiver

a component that responds to system-wide broadcast announcements.

Many broadcasts originate from the system—for example, a broadcast announcing that the screen has turned off, the battery is low, or a picture was captured.

Applications can initiate broadcasts—e.g. to let other applications know that some data has been downloaded to the device and is available for them to use.

Broadcast receivers don't display a user interface, but they can crate a status bar notification.

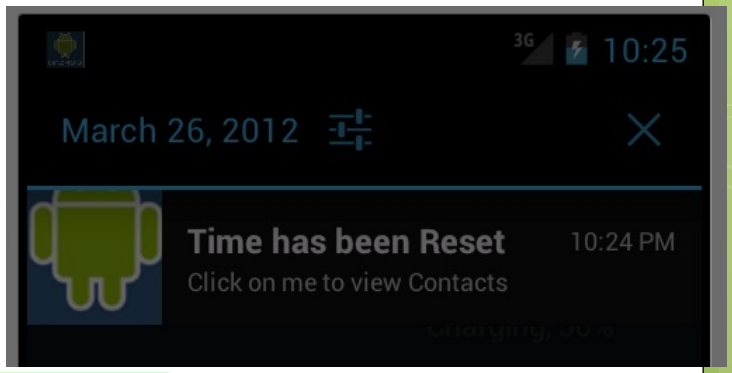
More commonly, a broadcast receiver is just a "gateway" to other components and is intended to do a very minimal amount of work e.g. it might initiate a service.

Broadcast receiver

```
public class MyBroadcastReceiver extends BroadcastReceiver {
    …
    public void onReceive(Context context, Intent intent) {
            …
    }
}
```

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="…I" android:versionCode="1"  android:versionName="1.0">
    <application android:icon="@drawable/icon" android:label="@string/app_name">
        <receiver android:name=".MyBroadcastReceiver">
            <intent-filter>
            <action android:name="android.intent.action.TIME_SET"/>
            </intent-filter>
        </receiver>
    </application>
    <uses-sdk android:minSdkVersion="13" />
</manifest>
```

```
>adb shell
# date +%s
1332793443
# date -s +%s 1332793443
time 1332793443 -> 1332793443.0
settimeofday failed Invalid argument
```



Adapted from saigeethamn.blogspot.it

Broadcast receiver

```
public class MyBroadcastReceiver extends BroadcastReceiver {
   private NotificationManager nm;
   private int SIMPLE_NOTFICATION_ID;

   @Override
   public void onReceive(Context context, Intent intent) {
    nm = (NotificationManager) context.getSystemService
                        (Context.NOTIFICATION_SERVICE);
    Notification n= new Notification(R.drawable.android,"Time Reset!",
                        System.currentTimeMillis());
    PendingIntent myIntent = PendingIntent.getActivity(context, 0,
            new Intent(Intent.ACTION_VIEW, People.CONTENT_URI), 0);
    n.setLatestEventInfo(context, "Time has been Reset",
                        "Click on me to view Contacts", myIntent);
     n|= Notification.FLAG_AUTO_CANCEL;
     n|= Notification.DEFAULT_SOUND;
     nm.notify(SIMPLE_NOTFICATION_ID, n);
     Log.i(getClass().getSimpleName(),"Sucessfully Changed Time");
   }
}
```

Adapted from saigeethamn.blogspot.it

Sending broadcast events

(in Context)

sendBroadcast (Intent intent, String receiverPermission)

Broadcast the given intent to all interested BroadcastReceivers, allowing an optional required permission to be enforced.

This call is asynchronous; it returns immediately, and you will continue executing while the receivers are run.

No results are propagated from receivers and receivers can not abort the broadcast.

Sending ordered broadcast events

(in Context)

sendOrderedBroadcast (Intent intent, String receiverPermission)

Broadcast the given intent to all interested BroadcastReceivers, delivering them one at a time to allow more preferred receivers to consume the broadcast before it is delivered to less preferred receivers.

This call is asynchronous; it returns immediately, and you will continue executing while the receivers are run.

LocalBroadcastManager

Helper to register for and send broadcasts of Intents to local objects within your process.

Advantages of Local vs Global B.M.:

- the data you are broadcasting will not leave your app
  - (you don't need to worry about leaking private data).
- it is not possible for other applications to send these broadcasts to your app
  - (you don't need to worry about having security holes)
- it is more efficient than sending a global broadcast through the system.

14

Services

Marco Ronchetti
Università degli Studi di Trento

Service

An application component that can perform long-running operations in the background and does not provide a user interface.

So, what's different from a Thread?

a) Services are declared in the Manifest
b) Services can be exposed to other processes
c) Services do not need to be connected with an Activity

Service

A service can essentially take two forms:

Started

- A service is "started" when an application component (such as an activity) starts it by calling startService(). Once started, a service can run in the background indefinitely, even if the component that started it is destroyed. Usually, a started service performs a single operation and does not return a result to the caller. For example, it might download or upload a file over the network. When the operation is done, the service should stop itself.

Bound

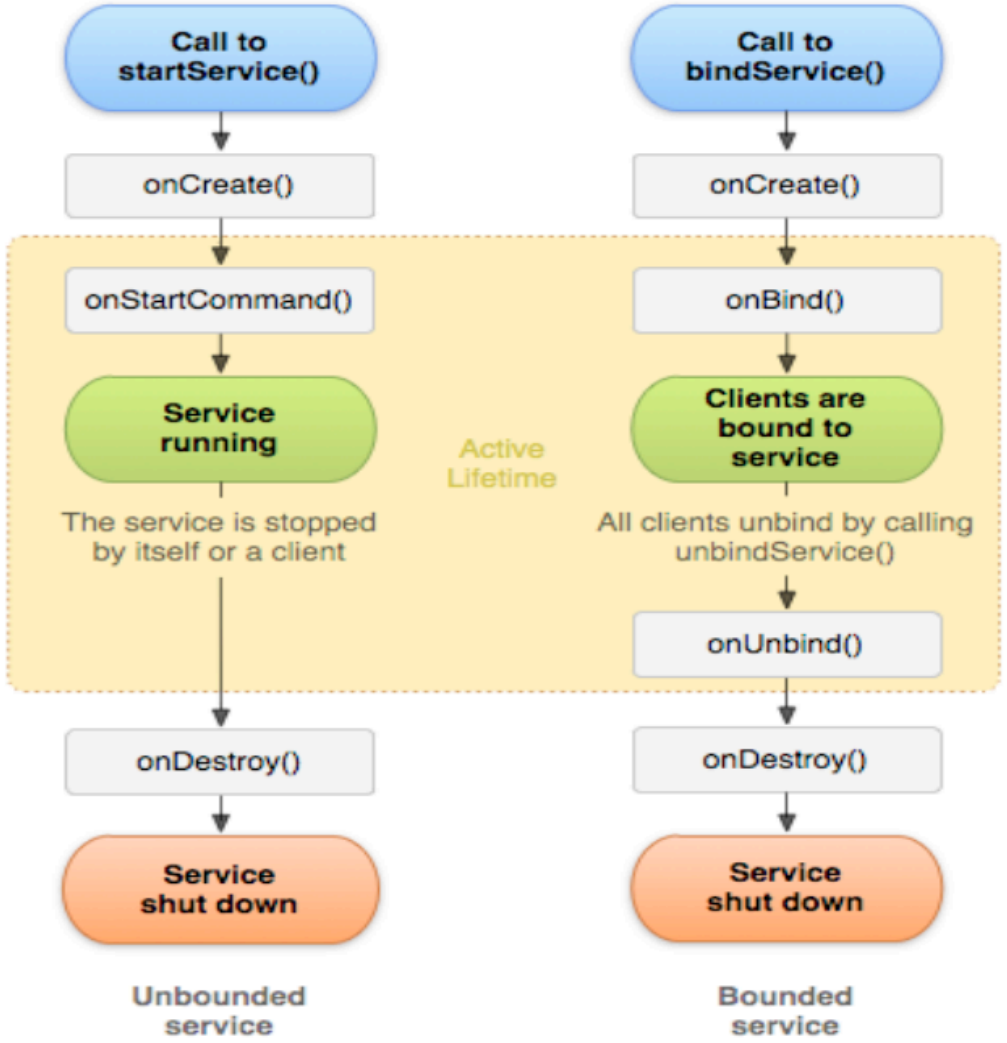
- A service is "bound" when an application component binds to it by calling bindService(). A bound service offers a client-server interface that allows components to interact with the service, send requests, get results, and even do so across processes with interprocess communication (IPC). A bound service runs only as long as another application component is bound to it. Multiple components can bind to the service at once, but when all of them unbind, the service is destroyed.

Although this documentation generally discusses these two types of services separately, your service can work both ways—it can be started (to run indefinitely) and also allow binding. It's simply a matter of whether you implement a couple callback methods: onStartCommand() to allow components to start it and onBind() to allow binding.

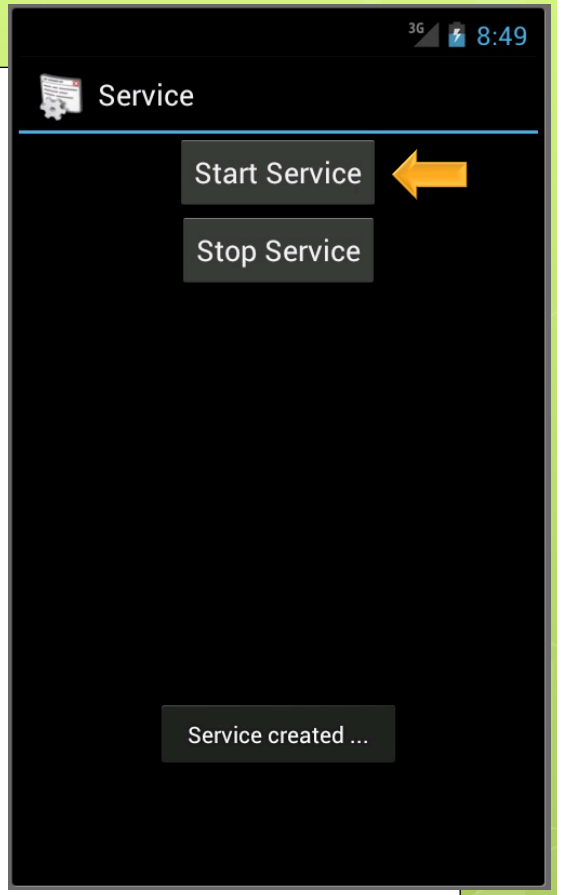
Regardless of whether your application is started, bound, or both, any application component can use the service (even from a separate application), in the same way that any component can use an activity—by starting it with an Intent.

However, you can declare the service as private, in the manifest file, and block access from other applications.

Service lifecycle



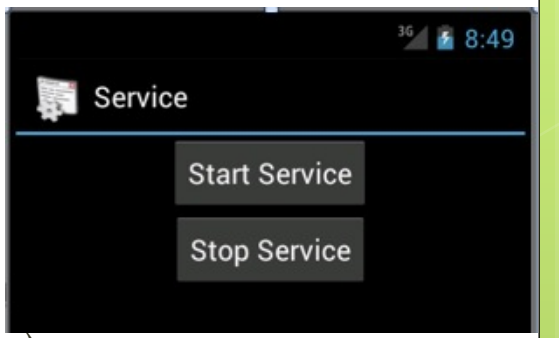
A simple service skeleton




```
import ...
public class SimpleService extends Service {
  public IBinder onBind(Intent arg0) {
        return null;
  }
  public void onCreate() {
        super.onCreate();
        Toast.makeText(this,"Service created ...", Toast.LENGTH_LONG).show();
  }
  public void onDestroy() {
        super.onDestroy();
        Toast.makeText(this, "Service destroyed ...", Toast.LENGTH_LONG).show();
  }
}
```

Adapted from saigeethamn.blogspot.it

Using our simple service




```
import ...
public class SimpleServiceController extends Activity {
  protected void onCreate(Bundle savedInstanceState) {
      super.onCreate(savedInstanceState);
      setContentView(R.layout.main);
      Button start = (Button)findViewById(R.id.serviceButton);
      Button stop = (Button)findViewById(R.id.cancelButton);
      start.setOnClickListener(startListener);
      stop.setOnClickListener(stopListener);
  }
  private OnClickListener startListener = new OnClickListener() {
      public void onClick(View v){
            startService(new Intent(SimpleServiceController.this,SimpleService.class));
   }};
   private OnClickListener stopListener = new OnClickListener() {
       public void onClick(View v){
            stopService(new Intent(SimpleServiceController.this,SimpleService.class));
    }};
}
```

Adapted from saigeethamn.blogspot.it

Service methods and IVs

int mStartMode; // indicates how to behave if the service is killed
IBinder mBinder; // interface for clients that bind
boolean mAllowRebind; // indicates whether onRebind should be used

public void onCreate()

- The service is being created

public int onStartCommand(Intent intent, int flags, int startId) {

- The service is starting, due to a call to startService()

public IBinder onBind(Intent intent) {

- A client is binding to the service with bindService()

public boolean onUnbind(Intent intent) {

- All clients have unbound with unbindService()

public void onRebind(Intent intent) {

- A client is binding to the service with bindService() after onUnbind() has been called

public void onDestroy() {

- The service is no longer used and is being destroyed

IntentService

a subclass for Services that handle asynchronous requests (expressed as Intents) on demand.

Clients send requests through startService(Intent) calls; the service is started as needed, handles each Intent in turn using a worker thread, and stops itself when it runs out of work.

"work queue processor" pattern

To use it, extend IntentService and implement onHandleIntent(Intent). IntentService will receive the Intents, launch a worker thread, and stop the service as appropriate.

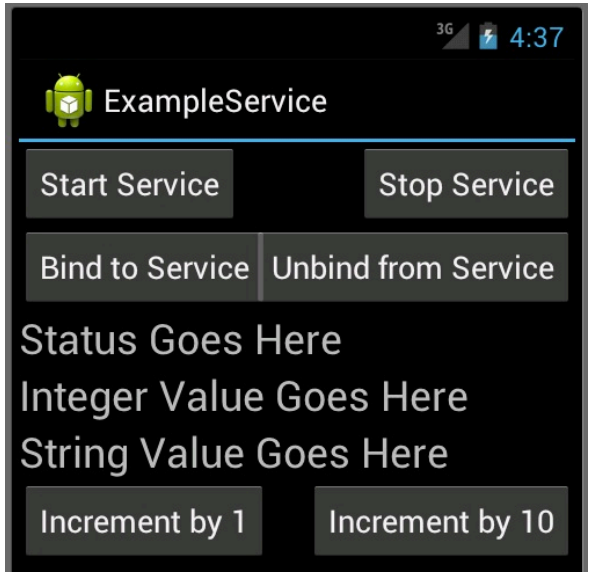
All requests are handled on a single worker thread -- they may take as long as necessary (and will not block the application's main loop), but only one request will be processed at a time.

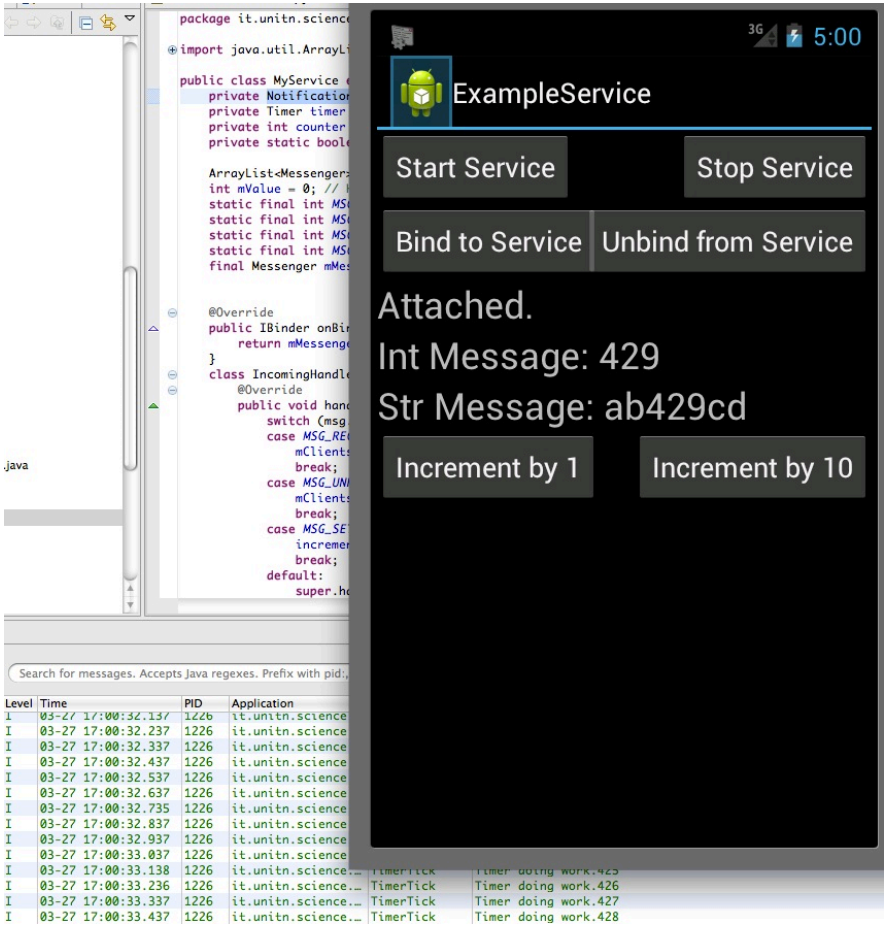
A full example part 1: introduction and essential classes

Marco Ronchetti
Università degli Studi di Trento

A full example

Get the Eclipse project from
http://latemar.science.unitn.it/segue_userFiles/2012Mobile/ServiceFullDemo.zip
(project declares Api version 13, even though an older one, like 8, would be fine)





Code adapted from an example on StackOverflow

Binder

Base class for a remotable object.

the core part of a lightweight remote procedure call mechanism defined by the interface IBinder.

This class is an implementation of IBinder that provides the standard support creating a local implementation of such an object.

Parcel

A Parcel is a serialized object. It can contain both flattened data that will be unflattened on the other side of the IPC, and references to live iBinder objects that will result in the other side receiving a proxy IBinder connected with the original IBinder in the Parcel.

Parcel is **not** a general-purpose serialization mechanism. This class (and the corresponding Parcelable API for placing arbitrary objects into a Parcel) is designed as a high-performance IPC transport.

Bundle

A mapping from String values to various Parcelable types.

Handler

allows you to send and process Message and Runnable objects associated with a thread's MessageQueue.

Each Handler instance is associated with a single thread and that thread's message queue.

When you create a new Handler, it is bound to the thread/ message queue of the thread that is creating it -- from that point on, it will deliver messages and runnables to that message queue and execute them as they come out of the message queue.

Handlers are used to

(1) to schedule messages and runnables to be executed as some point in the future;

(2) to enqueue an action to be performed on a different thread than your own.

void handleMessage(Message msg)

Message

Defines a message containing a description and arbitrary data object that can be sent to a Handler.

It contains two extra int fields and an extra object field that allow you to not do allocations in many cases.

To create one, it's best to use a factory method: Message.obtain()

Fields

Object obj

- An arbitrary object to send to the recipient.

Messenger replyTo

- Optional Messenger where replies to this message can be sent.

int what

- User-defined message code so that the recipient can identify what this message is about.

setData(Bundle b)

Bundle: a type of Parcel,

- Sets a Bundle of arbitrary data values.

Messenger

Reference to a Handler, which others can use to send messages to it. This allows for the implementation of message-based communication across processes, by creating a Messenger pointing to a Handler in one process, and handing that Messenger to another process.

ServiceConnection

Interface for monitoring the state of an application service.

void onServiceConnected(ComponentName name, IBinder service)

- Called when a connection to the Service has been established, with the IBinder of the communication channel to the Service.

void onServiceDisconnected(ComponentName name)

- Called when a connection to the Service has been lost.

java.util.Timer and TimerTask

Timer

facility for threads to schedule tasks for future execution in a background thread. Tasks may be scheduled for one-time execution, or for repeated execution at regular intervals.

schedule(TimerTask task, Date time)

scheduleAtFixedRate(TimerTask task, Date firstTime, long period)

TimerTask

A task that can be scheduled for one-time or repeated execution by a Timer.

public abstract void run()

A full example part 2: service implementation

Marco Ronchetti
Università degli Studi di Trento

MyService

```
public class MyService extends Service {
   private Timer timer = new Timer();
   private int counter = 0, incrementby = 1;
   final Messenger mMessenger = new Messenger(new Handler() {...});
   ArrayList<Messenger> mClients = new ArrayList<Messenger>();
   static final int MSG_REGISTER_CLIENT = 1;
   static final int MSG_UNREGISTER_CLIENT = 2;
   static final int MSG_SET_INT_VALUE = 3;
   static final int MSG_SET_STRING_VALUE = 4;
   private static boolean isRunning = false;
   public static boolean isRunning() {return isRunning; }
    public void onCreate() {...}
   public int onStartCommand(Intent intent, int flags, int startId) {...}
    public IBinder onBind(Intent intent){...}
   public void onDestroy() {...}
}
```

Create Messenger – MyService

```
final Messenger mMessenger = new Messenger(new Handler() {
    // Handler of incoming messages from clients.
    public void handleMessage(Message msg) {
        switch (msg.what) {
        case MSG_REGISTER_CLIENT:
            mClients.add(msg.replyTo);
            break;
        case MSG_UNREGISTER_CLIENT:
            mClients.remove(msg.replyTo);
            break;
        case MSG_SET_INT_VALUE:
            incrementby = msg.arg1;
            break;
        default:
            super.handleMessage(msg);
        }
    }
});
```

onCreate - MyService

```
public void onCreate() {
    super.onCreate();
    Log.i("MyService", "Service Created.");
    Toast.makeText(this,"Service Created", Toast.LENGTH_LONG).show();
    timer.scheduleAtFixedRate(new TimerTask(){ public void run() {
      Log.i("TimerTick", "Timer doing work." + counter);
       try {
          counter += incrementby;
          sendMessageToUI(counter);
       } catch (Throwable t) {Log.e("TimerTick", "Timer Tick Failed.", t); }
    }}, 0, 100L);
    isRunning = true;
  }
```

sendMessageToUI - onCreate - MyService

```
private void sendMessageToUI(int intvaluetosend) {
    for (int i=mClients.size()-1; i>=0; i--) {
        try {
            // Send data as an Integer
            mClients.get(i).send(Message.obtain(null,
                                    MSG_SET_INT_VALUE, intvaluetosend, 0));
            //Send data as a String
            Bundle b = new Bundle();
            b.putString("str1", "ab" + intvaluetosend + "cd");
            Message msg = Message.obtain(null, MSG_SET_STRING_VALUE);
            msg.setData(b);
            mClients.get(i).send(msg);
        } catch (RemoteException e) {
            // The client is dead. Remove it from the list;
            //we are going through the list from back to front so this is safe
            //to do inside the loop.
            mClients.remove(i);
        }
    }
}
```

onStartCommand – onDestroy - My Service

```
public int onStartCommand(Intent intent, int flags, int startId) {
    Log.i("MyService", "Received start id " + startId + ": " + intent);
    Toast.makeText(this,"Service started"+ startId ,
                                    Toast.LENGTH_LONG).show();
    return START_STICKY; // run until explicitly stopped.
  }

public void onDestroy() {
    super.onDestroy();
    if (timer != null) {timer.cancel();}
    counter=0;
   Log.i("MyService", "Service Stopped.");
   Toast.makeText(this,"Service Stopped ", Toast.LENGTH_LONG).show();
    isRunning = false;
  }

public IBinder onBind(Intent intent) {
    return mMessenger.getBinder();
  }
```

ServiceFullDemoActivity

```
public class ServiceFullDemoActivity extends Activity {
   Button btnStart, btnStop, btnBind, btnUnbind, btnUpby1, btnUpby10;
   TextView textStatus, textIntValue, textStrValue;
   Messenger mService = null;
   boolean mIsBound;
   final Messenger mMessenger = new Messenger(new Handler() {…});
   private ServiceConnection mConnection = new ServiceConnection() {…};
   public void onCreate(Bundle savedInstanceState) {…}
   protected void onDestroy() {…}
}
```

A full example part 3: activity implementation

Marco Ronchetti
Università degli Studi di Trento

Create Messenger – Activity

```
final Messenger mMessenger = new Messenger(new Handler() {
    @Override
    public void handleMessage(Message msg) {
      switch (msg.what) {
      case MyService.MSG_SET_INT_VALUE:
        textIntValue.setText("Int Message: " + msg.arg1);
        break;
      case MyService.MSG_SET_STRING_VALUE:
        String str1 = msg.getData().getString("str1");
        textStrValue.setText("Str Message: " + str1);
        break;
      default:
        super.handleMessage(msg);
      }
    }
});
```

Create ServiceConnection

```
private ServiceConnection mConnection = new ServiceConnection() {
    public void onServiceConnected(ComponentName className, IBinder service) {
        mService = new Messenger(service);
        textStatus.setText("Attached.");
        try {
            Message msg = Message.obtain(null, MyService.MSG_REGISTER_CLIENT);
            msg.replyTo = mMessenger;
            mService.send(msg);
        } catch (RemoteException e) {
            // In this case the service has crashed before we could even do anything with it
        }
    }

    public void onServiceDisconnected(ComponentName className) {
        // This is called when the connection with the service has been
        // unexpectedly disconnected - process crashed.
        mService = null;
        textStatus.setText("Disconnected.");
    }
};
```

bind - unbind

```
void doBindService() {
    bindService(new Intent(this, MyService.class), mConnection, Context.BIND_AUTO_CREATE);
     mIsBound = true;
     textStatus.setText("Binding.");
 }
 void doUnbindService() {
       if (mIsBound) {
          // If we have received the service, and registered with it, then now is the time to unregister.
          if (mService != null) {
             try {
                Message msg = Message.obtain(null, MyService.MSG_UNREGISTER_CLIENT);
                msg.replyTo = mMessenger;
                mService.send(msg);
             } catch (RemoteException e) {// nothing special to do if the service has crashed.
             }
          }
          // Detach our existing connection.
          unbindService(mConnection);
          mIsBound = false;
          textStatus.setText("Unbinding.");
       }
    }
```

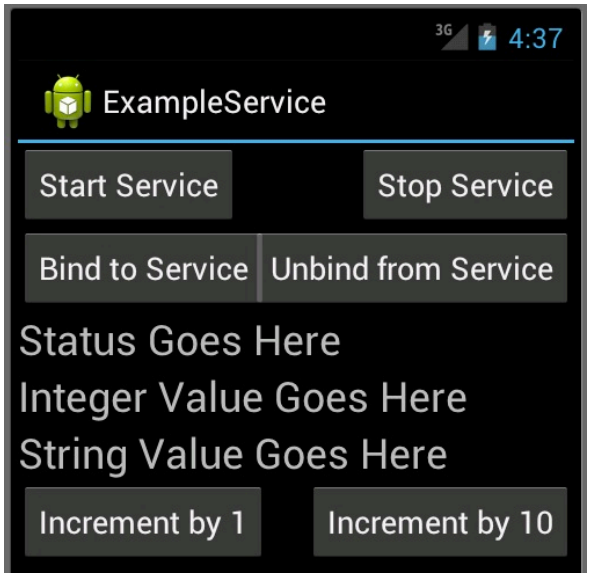
OnCreate

```
public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main);
    btnStart = (Button)findViewById(R.id.btnStart);
    btnStop = (Button)findViewById(R.id.btnStop);
    btnBind = (Button)findViewById(R.id.btnBind);
    btnUnbind = (Button)findViewById(R.id.btnUnbind);
    textStatus = (TextView)findViewById(R.id.textStatus);
    textIntValue = (TextView)findViewById(R.id.textIntValue);
    textStrValue = (TextView)findViewById(R.id.textStrValue);
    btnUpby1 = (Button)findViewById(R.id.btnUpby1);
    btnUpby10 = (Button)findViewById(R.id.btnUpby10);

    btnStart.setOnClickListener(btnStartListener);
    btnStop.setOnClickListener(btnStopListener);
    btnBind.setOnClickListener(btnBindListener);
    btnUnbind.setOnClickListener(btnUnbindListener);
    btnUpby1.setOnClickListener(btnUpby1Listener);
    btnUpby10.setOnClickListener(btnUpby10Listener);

    restoreMe(savedInstanceState);

  //If the service is running when the activity starts, we want to automatically bind to it.
    if (MyService.isRunning()) {
       doBindService() ;
}
```



Listeners

```
private OnClickListener btnStartListener = new OnClickListener() {
    public void onClick(View v){
        startService(new Intent(ServiceFullDemoActivity.this, MyService.class));
    }
};
private OnClickListener btnStopListener = new OnClickListener() {
    public void onClick(View v){
        doUnbindService();
        stopService(new Intent(ServiceFullDemoActivity.this, MyService.class));
    }
};
private OnClickListener btnBindListener = new OnClickListener() {
    public void onClick(View v){
        doBindService();
    }
};
private OnClickListener btnUnbindListener = new OnClickListener() {
    public void onClick(View v){
        doUnbindService();
    }
};
```

Listeners

```
private OnClickListener btnUpby1Listener = new OnClickListener() {
    public void onClick(View v){
      sendMessageToService(1);
    }
  };
  private OnClickListener btnUpby10Listener = new OnClickListener() {
    public void onClick(View v){
      sendMessageToService(10);
    }
  };
  private void sendMessageToService(int intvaluetosend) {
    if (mIsBound) {
      if (mService != null) {
        try {
          Message msg = Message.obtain(null, MyService.MSG_SET_INT_VALUE, intvaluetosend, 0);
          msg.replyTo = mMessenger;
          mService.send(msg);
        } catch (RemoteException e) {
        }
      }
    }
  }
```

onSaveInstanceState

```
@Override
  protected void onSaveInstanceState(Bundle outState) {
     super.onSaveInstanceState(outState);
     outState.putString("textStatus", textStatus.getText().toString());
     outState.putString("textIntValue", textIntValue.getText().toString());
     outState.putString("textStrValue", textStrValue.getText().toString());
  }
  private void restoreMe(Bundle state) {
     if (state!=null) {
        textStatus.setText(state.getString("textStatus"));
        textIntValue.setText(state.getString("textIntValue"));
        textStrValue.setText(state.getString("textStrValue"));
     }
  }
```

Manifest.xml

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.exampleservice"
    android:versionCode="1"
    android:versionName="1.0">
  <application android:icon="@drawable/icon"
          android:label="@string/app_name">
    <activity android:name=".MainActivity"
          android:label="@string/app_name">
      <intent-filter>
        <action android:name="android.intent.action.MAIN" />
        <category android:name="android.intent.category.LAUNCHER" />
      </intent-filter>
    </activity>
  <service android:name=".MyService"></service>
  </application>
  <uses-sdk android:minSdkVersion="13" />
</manifest>
```

Running service in processes

You can specify a process name with a colon in front

```
<service android:name=".MyService"
android:process=:myprocessname ></service>
```

Your service will then run as a different process - thus in a different thread.

You can set this attribute so that each component runs in its own process or so that some components share a process while others do not.

You can also set it so that components of different applications run in the same process—provided that the applications share the same Linux user ID and are signed with the same certificates.

Exercizes

1) Please have a look at the code on StackOverflow, from where this code was extracted and adapted.

It uses also a Notification.

http://stackoverflow.com/questions/4300291/example-communication-between-activity-and-service-using-messaging

2) Try to have run multiple instances of the activity accessing the service:

a) Activating intent associated to the notification
b) Writing yourself a second similar activity that uses the service, and switching between the two activities via the home

Content Providers

Marco Ronchetti
Università degli Studi di Trento

Content Provider

The Content provider design allows applications to share data through a standard set of programming interfaces.

And it's extensible: You can create your own custom content provider to share your data with other packages that works just like the built-in providers.

Default content providers

- Contacts
  - Stores all contacts information. etc
- Call Log Stores
  - call logs, for example: missed calls, answered calls. etc.
- Browser
  - Use by browser for history, favorites. etc.
- Media Store
  - Media files for Gallery, from SD Card. etc.
- Setting
  - Phone device settings. etc.

1. Starting from API Level 5, Contacts Provider is deprecated and superceded by ContactsContract.

Querying a Content Provider

To query a content provider, you provide a query string in the form of a URI, with an optional specifier for a particular row, using the following syntax:

<standard_prefix>://<authority>/<data_path>/<id>

For example, **to retrieve all the bookmarks** stored by your web browsers (in Android), you would use the following content URI:

content://browser/bookmarks

Similarly, **to retrieve all the contacts** stored by the Contacts application, the URI would look like this:

content://contacts/people

To retrieve a particular contact, you can specify the URI with a specific ID:

content://contacts/people/3

```
package it.unitn.science.latemar;

import android.app.Activity;
import android.database.Cursor;
import android.net.Uri;
import android.os.Bundle;
import android.provider.CallLog.Calls;
import android.util.Log;

public class ContentProviderActivity extends Activity {
   /** Called when the activity is first created. */
   @Override
   public void onCreate(Bundle savedInstanceState) {
      super.onCreate(savedInstanceState);
      setContentView(R.layout.main);

      Uri allCalls = Uri.parse("content://call_log/calls");
      Cursor c = managedQuery(allCalls, null, null, null, null);
      if (c.moveToFirst()) {
        do{
          String callType = "";
          switch (Integer.parseInt(c.getString(
            c.getColumnIndex(Calls.TYPE))))
          {
              case 1: callType = "Incoming";
                break;
              case 2: callType = "Outgoing";
                break;
              case 3: callType = "Missed";
          }
          Log.v("Content Providers",
              c.getString(c.getColumnIndex(Calls._ID)) + ", " +
              c.getString(c.getColumnIndex(Calls.NUMBER)) + ", " +
              callType) ;
        } while (c.moveToNext());
      }

   }
}
```

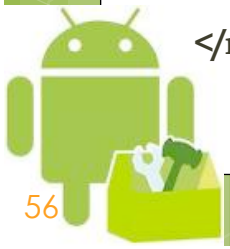
Error!

E/AndroidRuntime(541): java.lang.RuntimeException:
Unable to start activity ComponentInfo{it.unitn.science.latemar/
it.unitn.science.latemar.ContentProviderActivity}:
java.lang.SecurityException: Permission Denial: opening provider
com.android.providers.contacts.CallLogProvider from ProcessRecord{41475a28
541:it.unitn.science.latemar/10041} (pid=541, uid=10041)
requires
android.permission.READ_CONTACTS or
android.permission.WRITE_CONTACTS

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
   package="it.unitn.science.latemar"
   android:versionCode="1"
   android:versionName="1.0" >

   <uses-sdk android:minSdkVersion="13" />

   <application
      android:icon="@drawable/ic_launcher"
      android:label="@string/app_name" >
      <activity
         android:name=".ContentProviderActivity"
         android:label="@string/app_name" >
         <intent-filter>
            <action android:name="android.intent.action.MAIN" />
            <category android:name="android.intent.category.LAUNCHER" />
         </intent-filter>
      </activity>
   </application>
    <uses-permission
     android:name="android.permission.READ_CONTACTS">
    </uses-permission>
</manifest>
```



Screen properties

Marco Ronchetti
Università degli Studi di Trento

Screen related terms and concepts

Resolution The total number of physical pixels on a screen. When adding support for multiple screens, applications do not work directly with resolution; applications should be concerned only with screen size and density, as specified by the generalized size and density groups.

Screen size Actual physical size, measured as the screen's diagonal.

Screen density The quantity of pixels within a physical area of the screen; usually referred to as dpi (dots per inch).

Orientation The orientation of the screen from the user's point of view. This is either landscape or portrait, meaning that the screen's aspect ratio is either wide or tall, respectively. Not only do different devices operate in different orientations by default, but the orientation can change at runtime when the user rotates the device.

Screen Sizes and Densities

Android divides the range of actual screen sizes and densities into:

A set of four generalized **sizes**:
xlarge at least 960dp x 720dp
large at least 640dp x 480dp
normal at least 470dp x 320dp
small at least 426dp x 320dp

A set of four generalized **densities**:
Low density (120 dpi), ldpi
Medium density (160 dpi), mdpi
High density (240 dpi), hdpi
Extra high density (320 dpi), xhdpi

Density-independent pixel (dp)

A virtual pixel unit that you should use when defining UI layout, to express layout dimensions or position in a density-independent way.

The density-independent pixel is equivalent to one physical pixel on a 160 dpi screen, (baseline for a "medium" density screen).

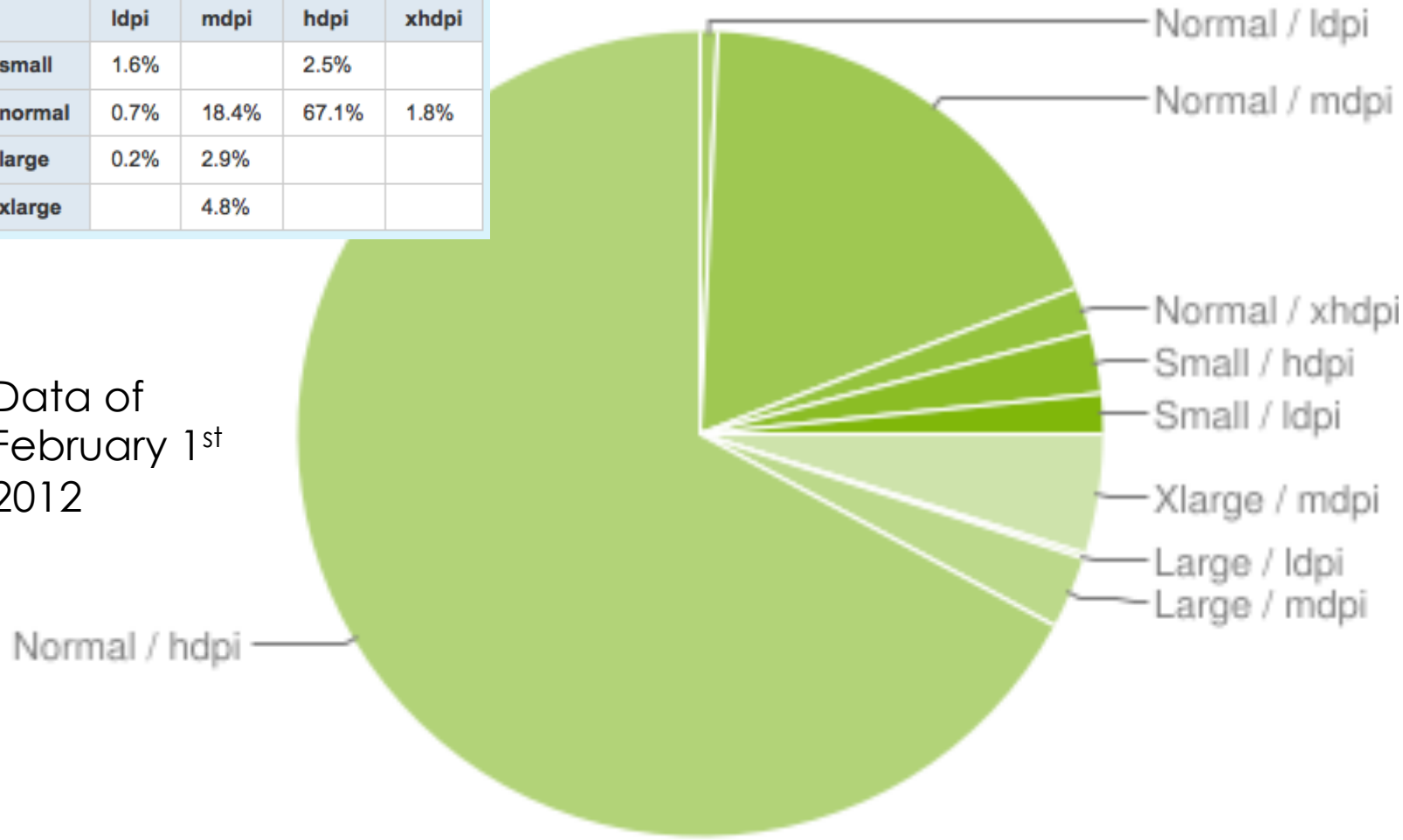
At runtime, the system transparently handles any scaling of the dp units based on the actual density of the screen in use.

px = dp * (dpi / 160). E.g.: on a 240 dpi screen, 1 dp equals 1.5 physical pixels.

You should always use dp units when defining your application's UI, to ensure proper display on screens with different densities.

Screen Sizes and Densities

	ldpi	mdpi	hdpi	xhdpi
small	1.6%		2.5%	
normal	0.7%	18.4%	67.1%	1.8%
large	0.2%	2.9%		
xlarge		4.8%		



Data of February 1st 2012

http://developer.android.com/resources/dashboard/screens.html

Support of multiple versions

Marco Ronchetti
Università degli Studi di Trento

http://android-developers.blogspot.it/2010/07/how-to-have-your-cupcake-and-eat-it-too.html

http://android-developers.blogspot.it/2010/06/future-proofing-your-app.html

Fragments

Marco Ronchetti
Università degli Studi di Trento

Fragments

http://developer.android.com/guide/topics/fundamentals/fragments.html

Adapters: a deeper insight

Marco Ronchetti
Università degli Studi di Trento

http://developer.android.com/resources/tutorials/views/index.html

http://developer.android.com/resources/samples/ApiDemos/src/com/example/android/apis/view/index.html

http://developer.android.com/training/improving-layouts/index.html

http://developer.android.com/guide/topics/ui/declaring-layout.html

Basic UI elements: Hello i18N

Marco Ronchetti
Università degli Studi di Trento

http://developer.android.com/resources/tutorials/localization/index.html

The Zygote

http://coltf.blogspot.com/p/android-os-processes-and-zygote.html

http://www.slideshare.net/RanNachmany/manipulating-android-tasks-and-back-stack

http://www.vogella.de/articles/Android/article.html

http://www.vogella.de/articles/AndroidInternals/article.html

http://benno.id.au/blog/2007/11/13/android-under-the-hood

http://blog.vlad1.com/2009/11/19/android-hacking-part-1-of-probably-many/

http://www.slideshare.net/retomeier/being-epic-best-practices-for-building-android-apps

Fragment

A Fragment represents a behavior or a portion of user interface in an Activity. You can combine multiple fragments in a single activity to build a multi-pane UI and reuse a fragment in multiple activities. You can think of a fragment as a modular section of an activity, which has its own lifecycle, receives its own input events, and which you can add or remove while the activity is running (sort of like a "sub activity" that you can reuse in different activities).

A fragment must always be embedded in an activity and the fragment's lifecycle is directly affected by the host activity's lifecycle. For example, when the activity is paused, so are all fragments in it, and when the activity is destroyed, so are all fragments. However, while an activity is running (it is in the resumed lifecycle state), you can manipulate each fragment independently, such as add or remove them. When you perform such a fragment transaction, you can also add it to a back stack that's managed by the activity—each back stack entry in the activity is a record of the fragment transaction that occurred. The back stack allows the user to reverse a fragment transaction (navigate backwards), by pressing the Back button.

When you add a fragment as a part of your activity layout, it lives in a ViewGroup inside the activity's view hierarchy and the fragment defines its own view layout. You can insert a fragment into your activity layout by declaring the fragment in the activity's layout file, as a <fragment> element, or from your application code by adding it to an existing ViewGroup. However, a fragment is not required to be a part of the activity layout; you may also use a fragment without its own UI as an invisible worker for the activity.

View

the basic building block for user interface components, similar to the Java AWT Component.

A View occupies a rectangular area on the screen and is responsible for drawing and event handling. View is the base class for *widgets*, which are used to create interactive UI components (buttons, text fields, etc.)

Broadcast receiver

A broadcast receiver is a component that responds to system-wide broadcast announcements. Many broadcasts originate from the system—for example, a broadcast announcing that the screen has turned off, the battery is low, or a picture was captured. 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. Although broadcast receivers don't display a user interface, they may create a status bar notification to alert the user when a broadcast event occurs. More commonly, though, a broadcast receiver is just a "gateway" to other components and is intended to do a very minimal amount of work. For instance, it might initiate a service to perform some work based on the event.

A broadcast receiver is implemented as a subclass of BroadcastReceiver and each broadcast is delivered as an Intent object. For more information, see the BroadcastReceiver class.

Content Provider

Content providers manage access to a structured set of data. They encapsulate the data, and provide mechanisms for defining data security. Content providers are the standard interface that connects data in one process with code running in another process.

When you want to access data in a content provider, you use the ContentResolver object in your application's Context to communicate with the provider as a client. The ContentResolver object communicates with the provider object, an instance of a class that implements ContentProvider. The provider object receives data requests from clients, performs the requested action, and returns the results.

You don't need to develop your own provider if you don't intend to share your data with other applications. However, you do need your own provider to provide custom search suggestions in your own application. You also need your own provider if you want to copy and paste complex data or files from your application to other applications.

Android itself includes content providers that manage data such as audio, video, images, and personal contact information. You can see some of them listed in the reference documentation for the android.provider package.

Best practices

Marco Ronchetti
Università degli Studi di Trento

http://developer.android.com/guide/practices/design/performance.html

Design

Marco Ronchetti
Università degli Studi di Trento

Android design

http://developer.android.com/design/index.html