BROADCAST RECEIVER SERVICE

Broadcast receiver

- A broadcast receiver is a dormant component of the Android system.
- Only an *Intent* (for which it is registered) can bring it into action.
- Using a Broadcast Receiver, applications can register for a particular event. Once the event occurs, the system will notify all the registered applications.
  - Examples: Boot completed, Time tick
- The Broadcast Receiver’s job is to activate some sw component, for example to notify the end user something occurred.

Registering a receiver

- There are two ways to register a Broadcast Receiver; one is Static and the other Dynamic.
- **Static:**
  - Use <receiver> tag in your Manifest file. (AndroidManifest.xml)
  - Not all events can be registered statically
  - Some events require permission
- **Dynamic:**
  - Use Context.registerReceiver () method to dynamically register an instance.
  - Note: Unregister when pausing

Broadcast intents

- Broadcast intents are Intent objects that are broadcast via a call to the **sendBroadcast()**, **sendStickyBroadcast()** or **sendOrderedBroadcast()** methods of the Activity class.

- In addition to providing a messaging and event system between application components, broadcast intents are also used by the Android system to notify interested applications about key system events (such as the external power supply or headphones being connected or disconnected).

- When a broadcast intent is created, it must include an action string in addition to optional data and a category string.

-

Broadcast intents

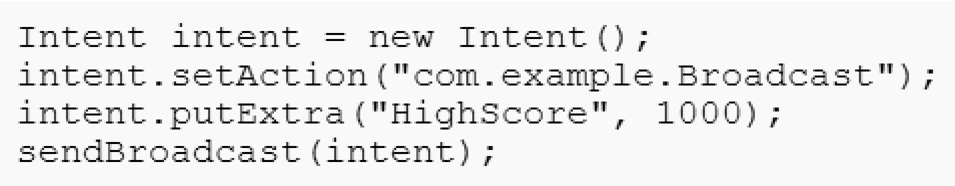
- As with standard intents, data is added to a broadcast intent using key-value pairs in conjunction with the putExtra() method of the intent object.

- The optional *category* string may be assigned to a broadcast intent via a call to the addCategory() method.

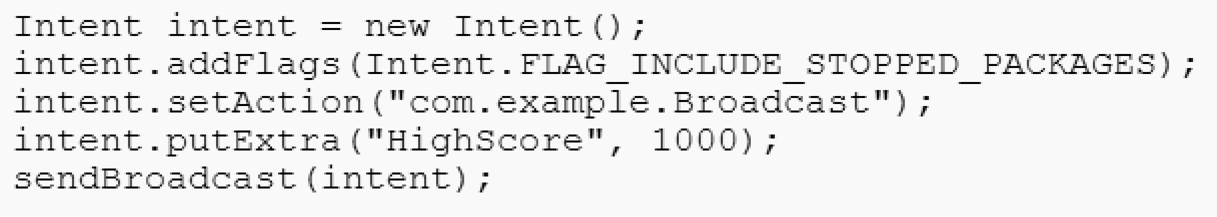
- The action string, which identifies the broadcast event, must be unique and typically uses the application's Java package name syntax. For example, the following code fragment creates and sends a broadcast intent including a unique action string and data:

-

Broadcast intent



```
Intent intent = new Intent();
intent.setAction("com.example.Broadcast");
intent.putExtra("HighScore", 1000);
sendBroadcast(intent);
```



```
Intent intent = new Intent();
intent.addFlags(Intent.FLAG_INCLUDE_STOPPED_PACKAGES);
intent.setAction("com.example.Broadcast");
intent.putExtra("HighScore", 1000);
sendBroadcast(intent);
```

>Android 3.0

Type of broadcasts

- **Ordered Broadcasts:**
  - These broadcasts are synchronous and follows the order specified using **android: priority attribute**.
  - The **receivers with greater priority would receive the broadcast first**.
- **Normal Broadcasts:**
  - Normal broadcasts are **not orderly**.

Broadcast receiver

- An application listens for specific broadcast intents by registering a broadcast receiver.

- Broadcast receivers are implemented by extending the Android *BroadcastReceiver* class and overriding the *onReceive()* method.

- The broadcast receiver may then be registered, either within code (for example within an activity), or within a manifest file.

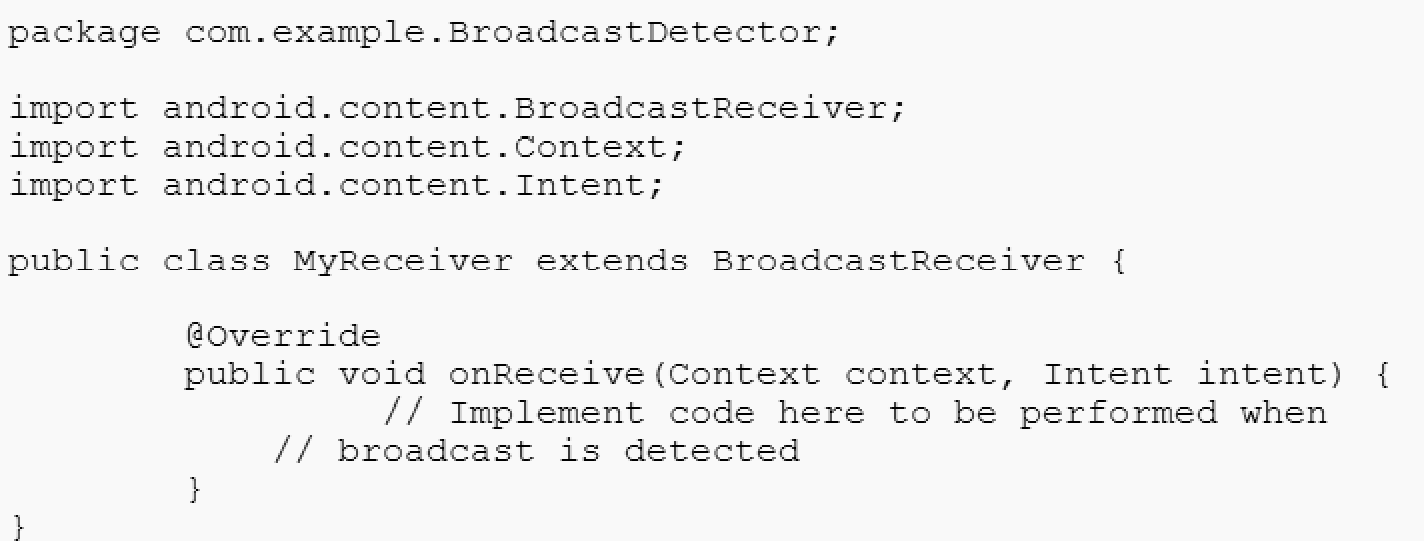
Broadcast receiver

- Part of the registration implementation involves the creation of intent filters to indicate the specific broadcast intents the receiver is required to listen for.

- This is achieved by referencing the action string of the broadcast intent.

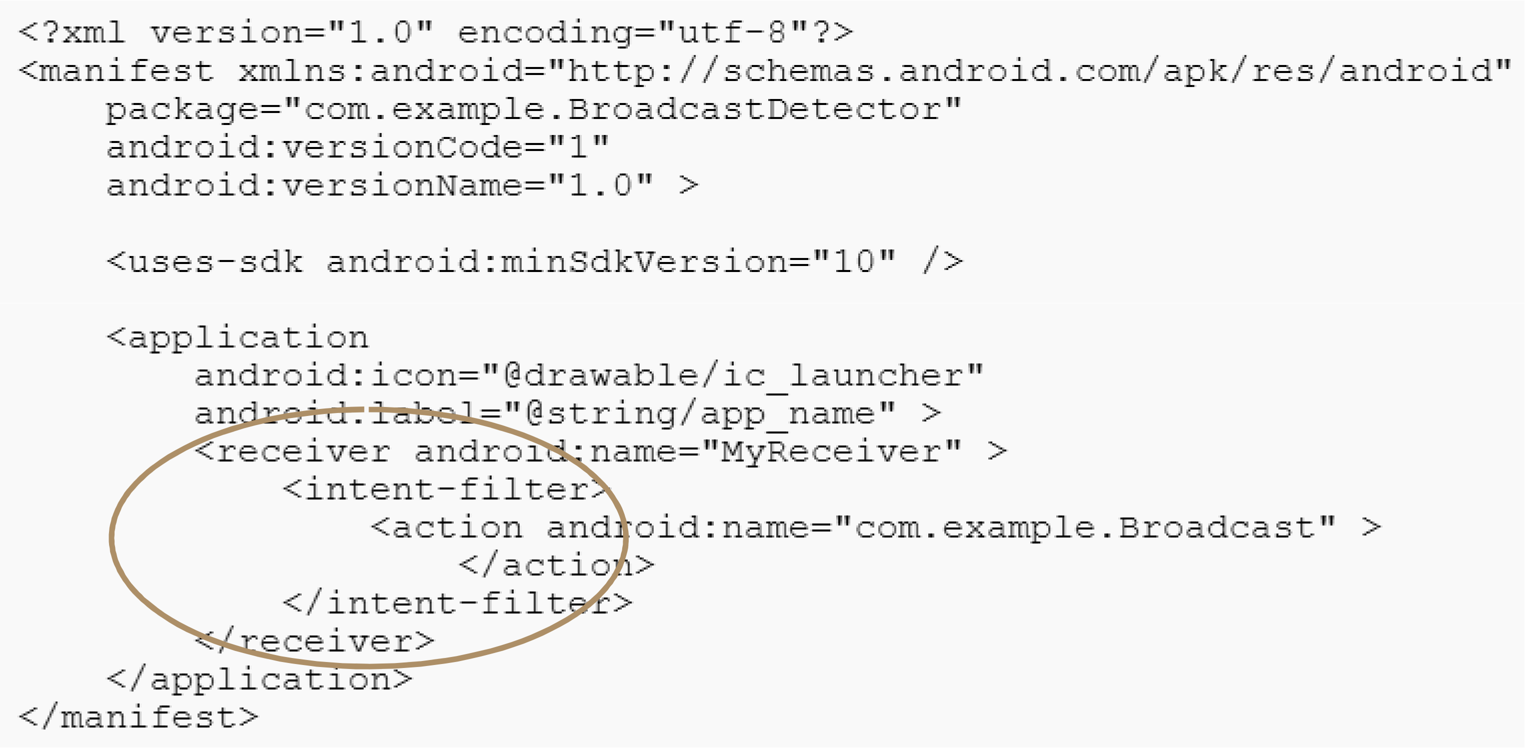
- When a matching broadcast is detected, the onReceive() method of the broadcast receiver is called, at which point the method has 5 seconds within which to perform any necessary tasks before returning.

Broadcast receiver template



```
package com.example.BroadcastDetector;

import android.content.BroadcastReceiver;
import android.content.Context;
import android.content.Intent;

public class MyReceiver extends BroadcastReceiver {

        @Override
        public void onReceive(Context context, Intent intent) {
                 // Implement code here to be performed when
            // broadcast is detected
        }
}
```

Registering a Broadcast receiver



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

    <uses-sdk android:minSdkVersion="10" />

    <application
        android:icon="@drawable/ic_launcher"
        android:label="@string/app_name" >
        <receiver android:name="MyReceiver" >
            <intent-filter>
                <action android:name="com.example.Broadcast" >
                    </action>
            </intent-filter>
        </receiver>
    </application>
</manifest>
```

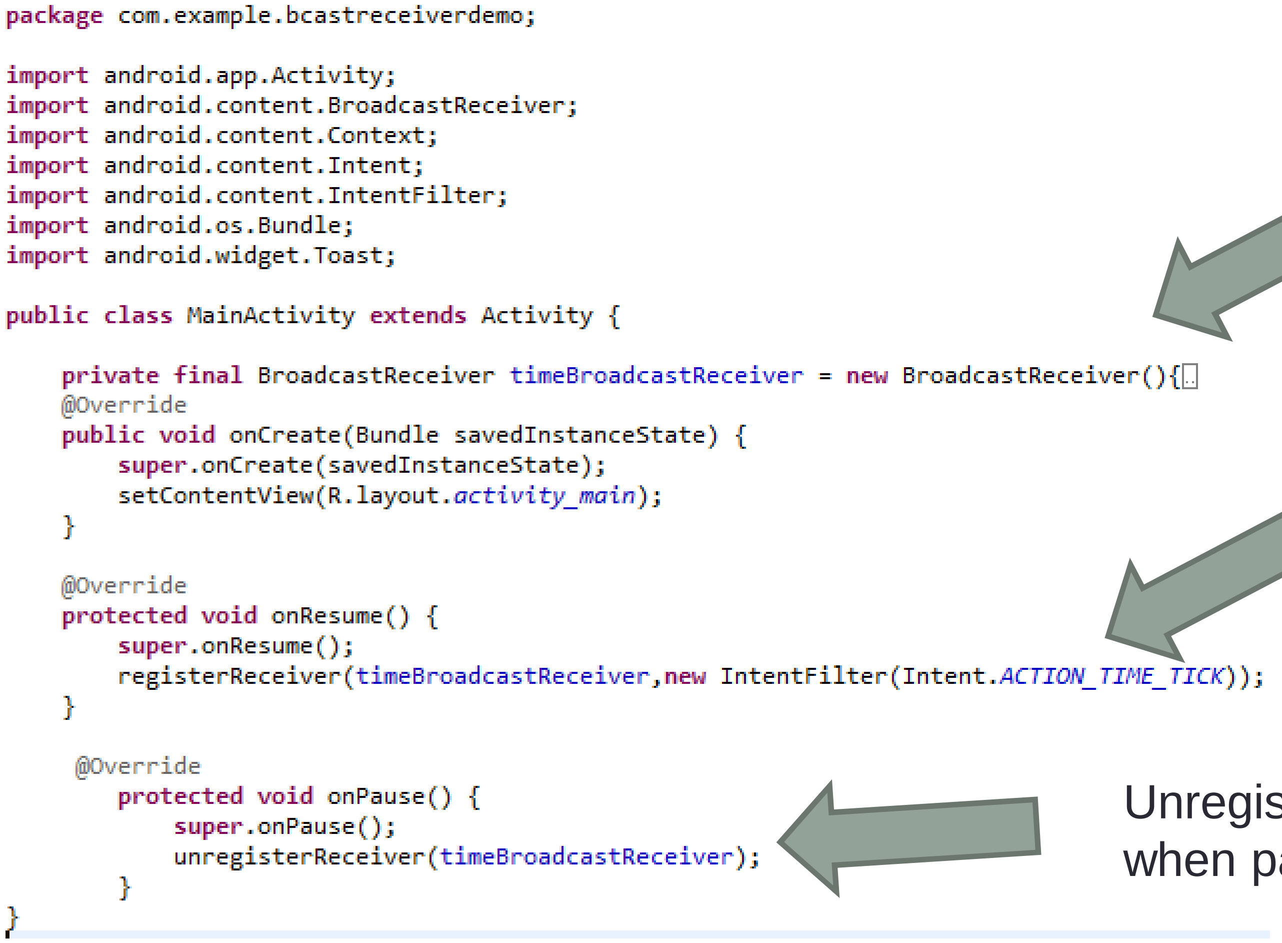
Another example

- An activity creates a broadcast receiver that subscribes *dynamically* for TIME_TICK events (fired every minute)

- The receiver is registered to the event when the activity is started

- The receiver is unregistered when the hosting activity is paused.

Another example



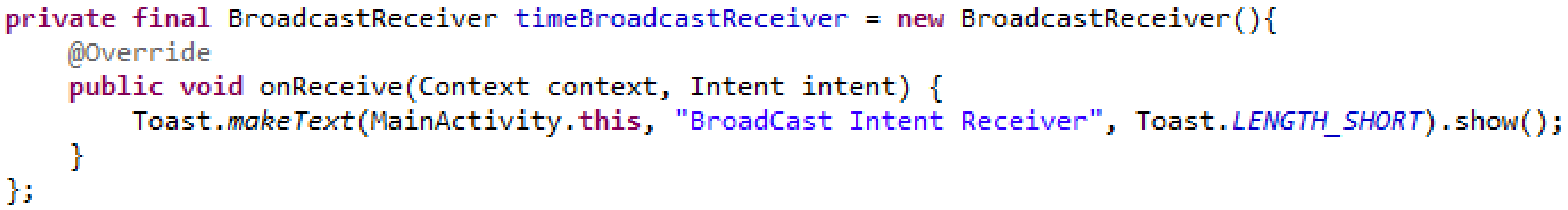
```
package com.example.bcastreceiverdemo;

import android.app.Activity;
import android.content.BroadcastReceiver;
import android.content.Context;
import android.content.Intent;
import android.content.IntentFilter;
import android.os.Bundle;
import android.widget.Toast;

public class MainActivity extends Activity {

    private final BroadcastReceiver timeBroadcastReceiver = new BroadcastReceiver(){□
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
    }

    @Override
    protected void onResume() {
        super.onResume();
        registerReceiver(timeBroadcastReceiver,new IntentFilter(Intent.ACTION_TIME_TICK));
    }

     @Override
        protected void onPause() {
            super.onPause();
            unregisterReceiver(timeBroadcastReceiver);
        }
}
```

Creates the receiver

Register the receiver to receive time ticks…

Unregister the receiver when paused

Another example

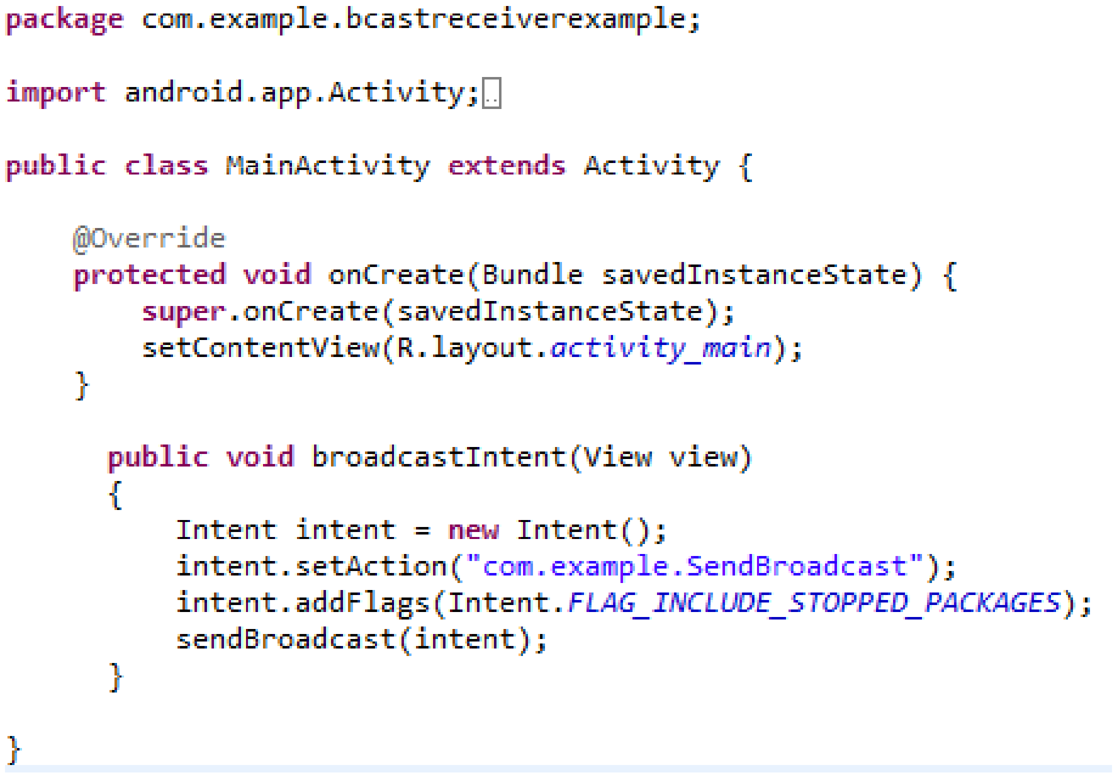


```
private final BroadcastReceiver timeBroadcastReceiver = new BroadcastReceiver(){
    @Override
    public void onReceive(Context context, Intent intent) {
        Toast.makeText(MainActivity.this, "BroadCast Intent Receiver", Toast.LENGTH_SHORT).show();
    }
};
```

Good tutorial:
http://www.grokkingandroid.com/android-tutorial-broadcastreceiver/

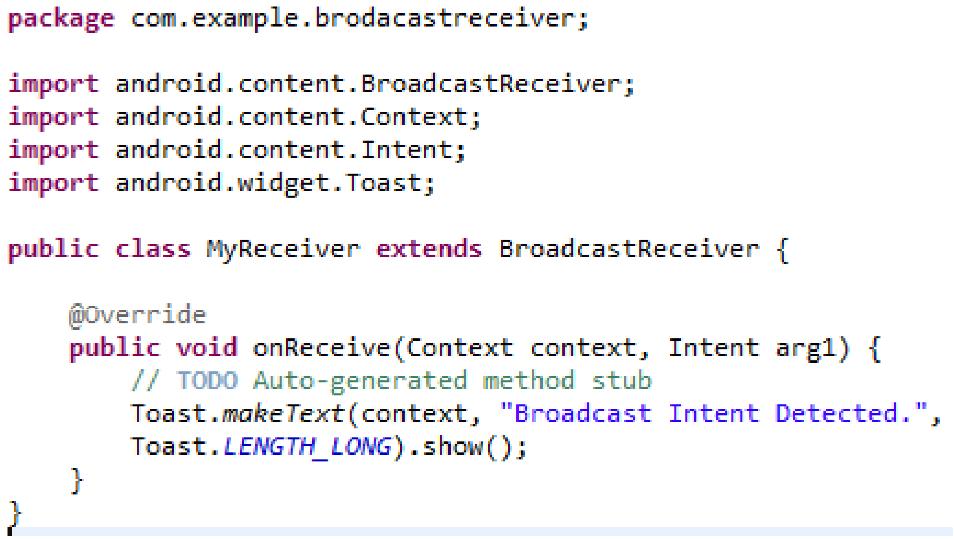
Another example

- An application generates custom bcast intent
- A receiver registers to receive the intent



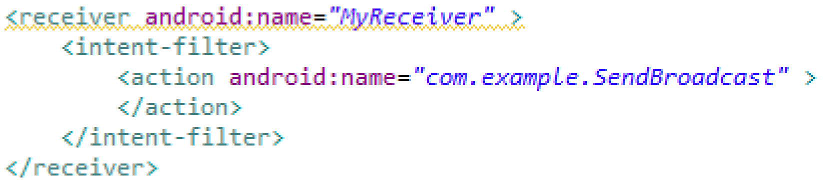
```
package com.example.bcastreceiverexample;

import android.app.Activity;□

public class MainActivity extends Activity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
    }

      public void broadcastIntent(View view)
      {
          Intent intent = new Intent();
          intent.setAction("com.example.SendBroadcast");
          intent.addFlags(Intent.FLAG_INCLUDE_STOPPED_PACKAGES);
          sendBroadcast(intent);
      }

}
```

Another example: the receiver



```java
package com.example.brodacastreceiver;

import android.content.BroadcastReceiver;
import android.content.Context;
import android.content.Intent;
import android.widget.Toast;

public class MyReceiver extends BroadcastReceiver {

    @Override
    public void onReceive(Context context, Intent arg1) {
        // TODO Auto-generated method stub
        Toast.makeText(context, "Broadcast Intent Detected.",
        Toast.LENGTH_LONG).show();
    }
}
```

Add in the manifest file



```xml
<receiver android:name="MyReceiver" >
    <intent-filter>
        <action android:name="com.example.SendBroadcast" >
        </action>
    </intent-filter>
</receiver>
```

Service

- The Android Service class is designed specifically to allow applications to initiate and perform background tasks.

- Unlike broadcast receivers, which are intended to perform a task quickly and then exit, services are designed to perform tasks that take a long time to complete

- …such as downloading a file over an internet connection or streaming music to the user, but do not require a user interface.

Service type

- Intent Service
  - Simplest form of service
  - Created to execute a task in a separate thread and then exit

- Service
  - Started Service
    - Run until explicitly stopped (in the rare case android needs to kill it, the service will be restarted as soon as possible)
    - Started with *startCommand* method
  - Bound Service
    - Allows the exchange data with the interacting software component through an interface (set of methods)
    - Bind to a service interface

Intent service

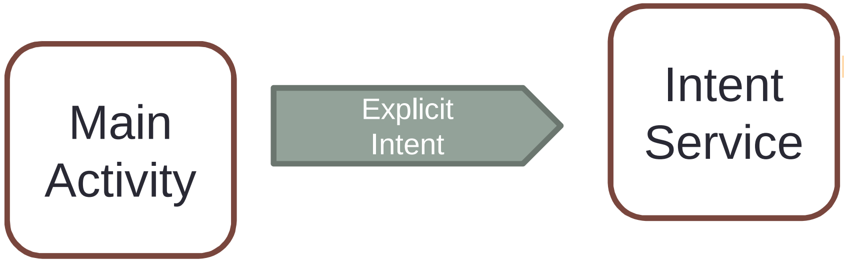
- As previously outlined, services run by default within the same main thread as the component from which they are launched. As such, any CPU intensive tasks that need to be performed by the service should take place within a new thread, thereby avoiding affecting the performance of the calling application.

- The IntentService class is a convenience class (subclassed from the Service class) that sets up a worker thread for handling background tasks and handles each request in an asynchronous manner.

- Once the service has handled all queued requests, it simply exits. All that is required when using the IntentService class is that the onHandleIntent() method be implemented containing the code to be executed for each request.

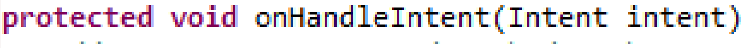
- For services that do not require synchronous processing of requests, IntentService is the recommended option. Services requiring synchronous handling of requests will, however, need to subclass from the Service class and manually implement and manage threading to handle any CPU intensive tasks efficiently.

Intent Service: example


Main Activity

Explicit Intent

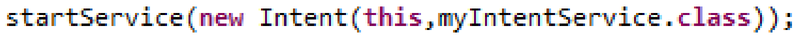
Intent Service



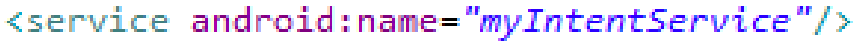
```
protected void onHandleIntent(Intent intent)
```

- The service needs to be registered in the manifest file
- The main activity creates an explicit intent pointing to the service
- The service is started and the *onHandleIntent* method executed
- Intents are queued and served serially

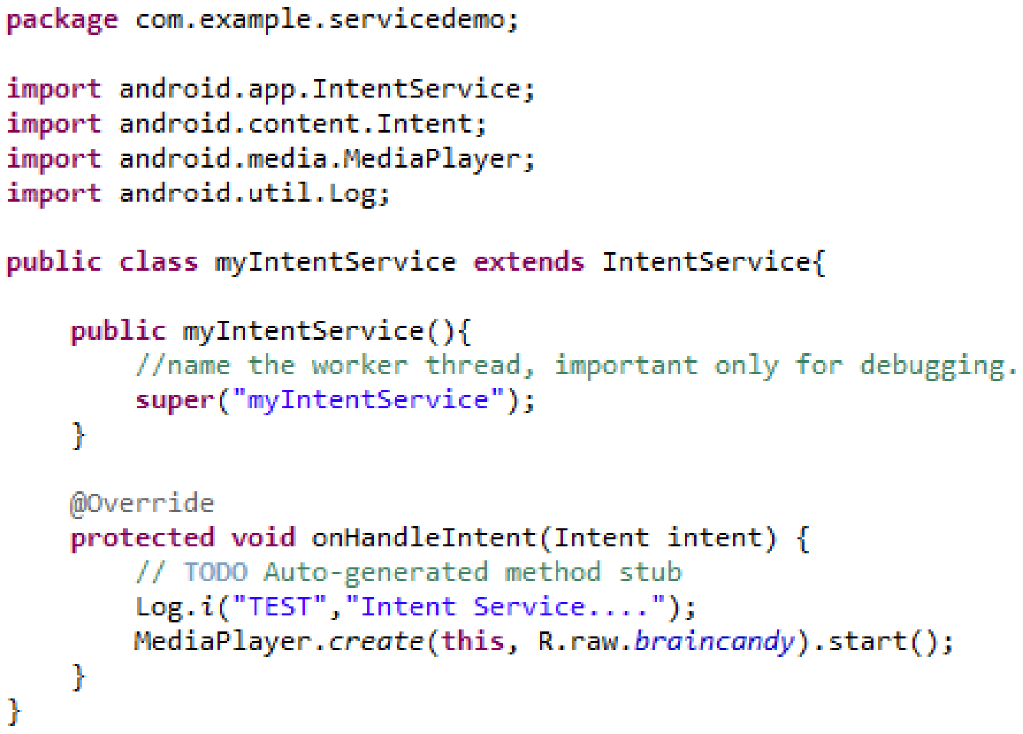
Intent Service: example



```
startService(new Intent(this,myIntentService.class));
```



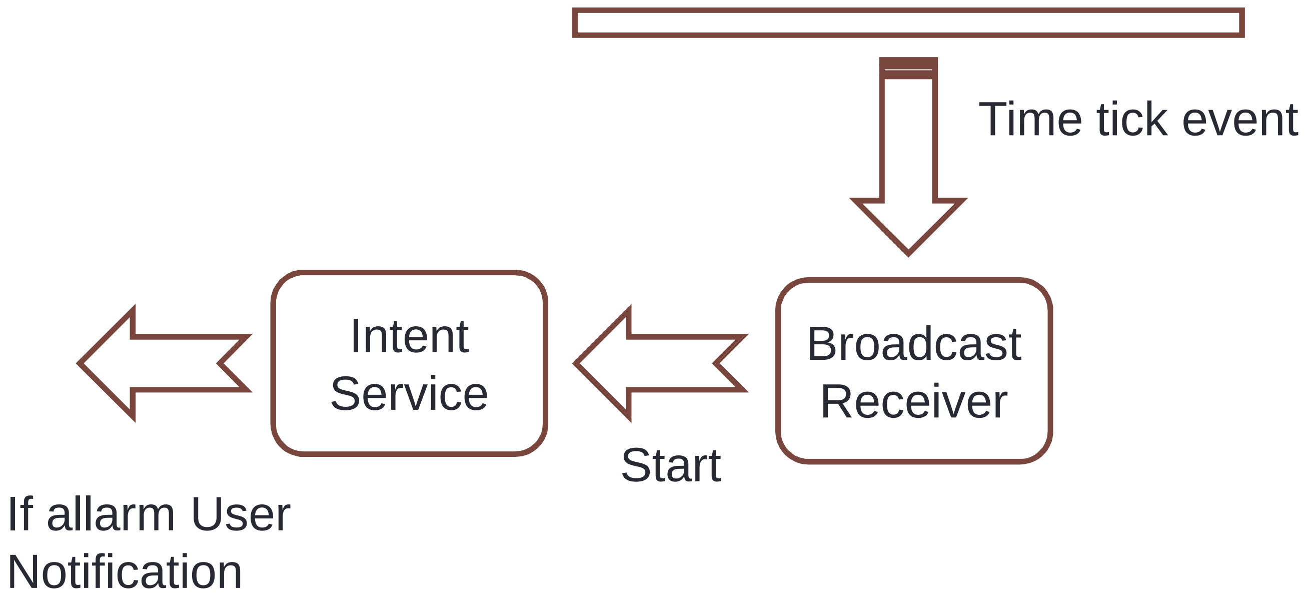
```
<service android:name="myIntentService"/>
```



```
package com.example.servicedemo;

import android.app.IntentService;
import android.content.Intent;
import android.media.MediaPlayer;
import android.util.Log;

public class myIntentService extends IntentService{

    public myIntentService(){
        //name the worker thread, important only for debugging.
        super("myIntentService");
    }

    @Override
    protected void onHandleIntent(Intent intent) {
        // TODO Auto-generated method stub
        Log.i("TEST","Intent Service....");
        MediaPlayer.create(this, R.raw.braincandy).start();
    }
}
```

Example

- Testing the weather condition periodically and send a notification if an alarm occurs



Started service

- Started services are launched by other application components (such as an activity or even a broadcast receiver) and potentially run indefinitely in the background until the service is stopped, or is destroyed by the Android runtime system in order to free up resources.

- A service will continue to run if the application that started it is no longer in the foreground, and even in the event that the component that originally started the service is destroyed.

- By default, a service will run within the same main thread as the application process from which it was launched (referred to as a local service).

- It is important, therefore, that any CPU intensive tasks be performed in a new thread within the service. Instructing a service to run within a separate process (and therefore known as a remote service) requires a configuration change within the manifest file.

Started service

- Unless a service is specifically configured to be private (once again via a setting in the manifest file), that service can be started by other components on the same Android device.

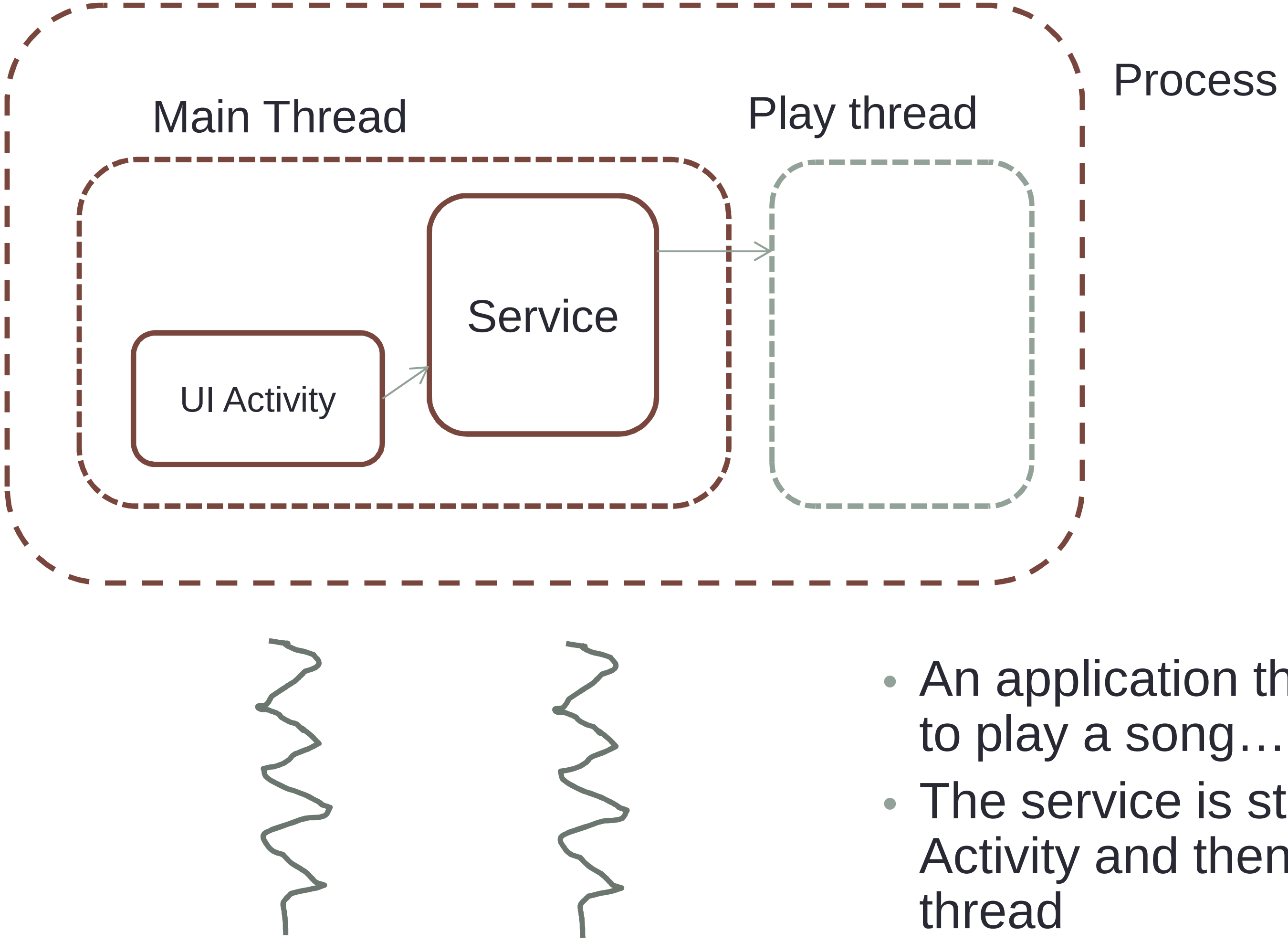
- This is achieved using the Intent mechanism in the same way that one activity can launch another as outlined in preceding slides.

- Started services are launched via a call to the startService() method, passing through as an argument an Intent object identifying the service to be started.

- When a started service has completed its tasks, it should stop itself via a call to stopSelf(). Alternatively, a running service may be stopped by another component via a call to the stopService() method, passing through as an argument the matching Intent for the service to be stopped.

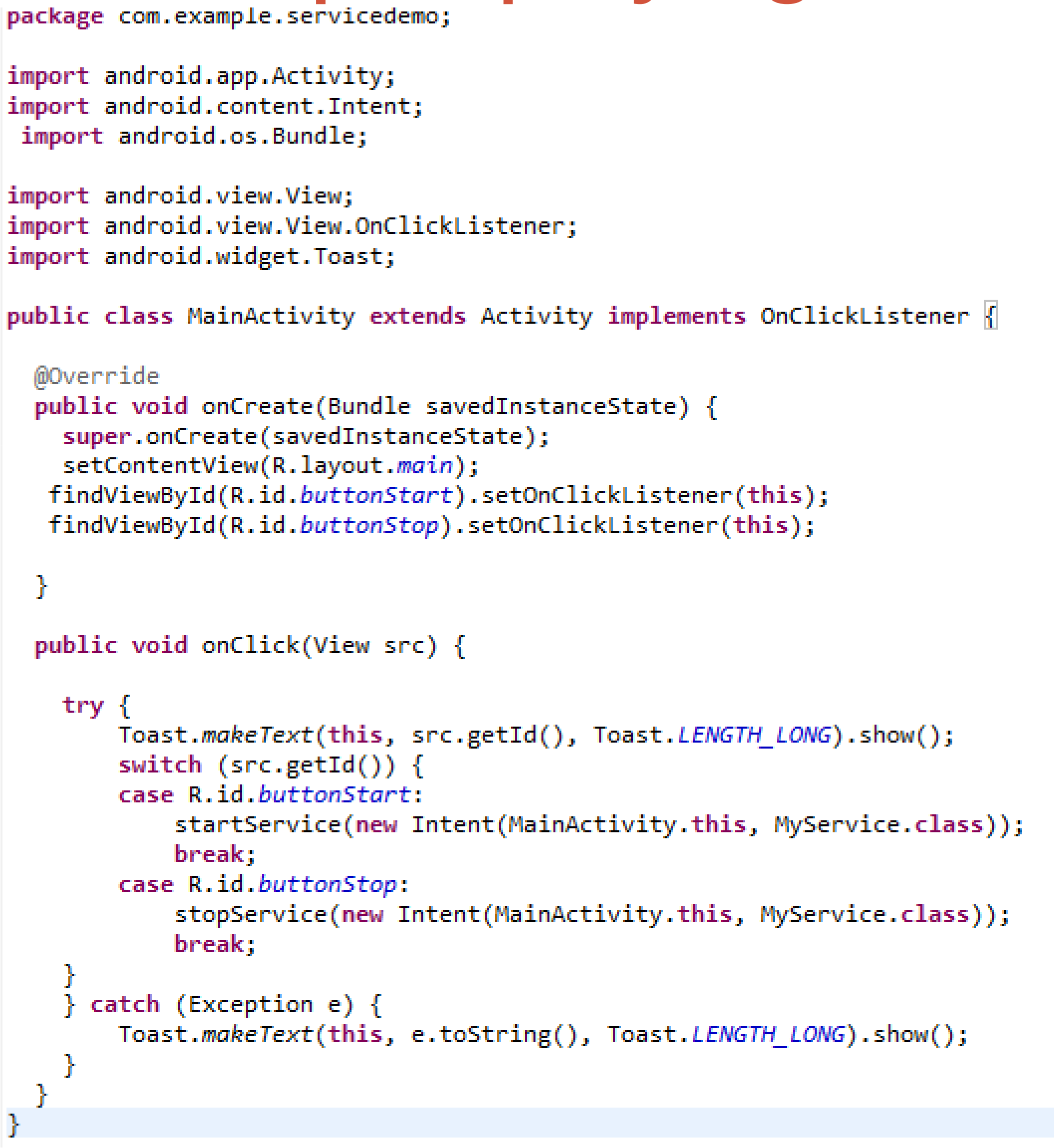
- Services are given a high priority by the Android system and are typically amongst the last to be terminated in order to free up resource

Started service: example, playing music

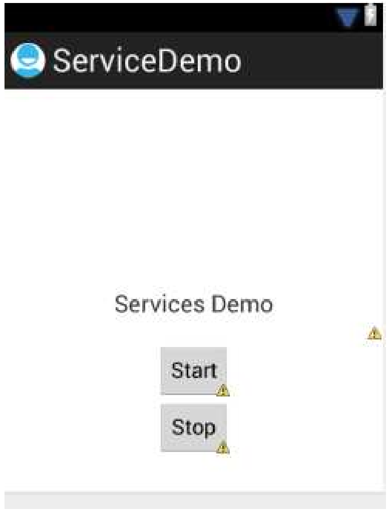


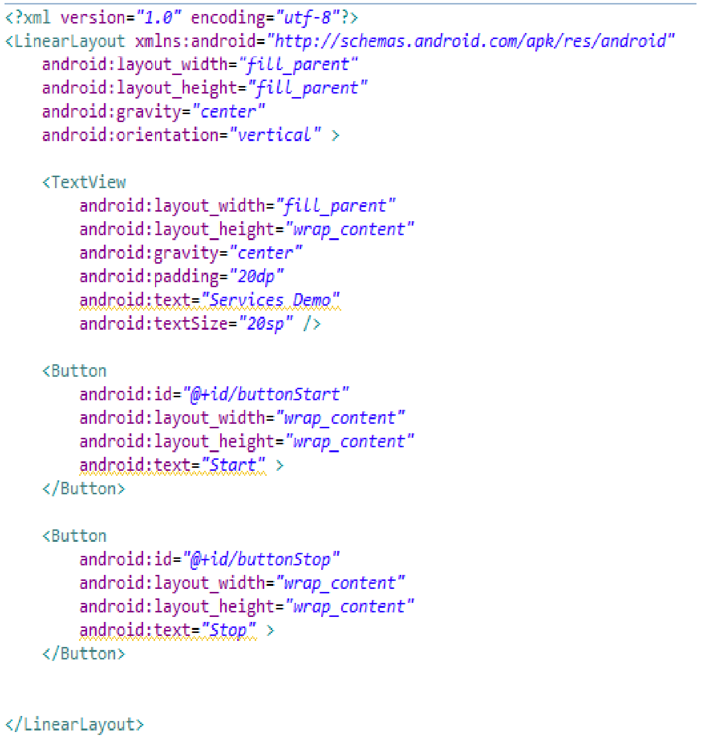
- An application that runs a player to play a song…
- The service is started from the Activity and then it spawns a thread

Example: playing music



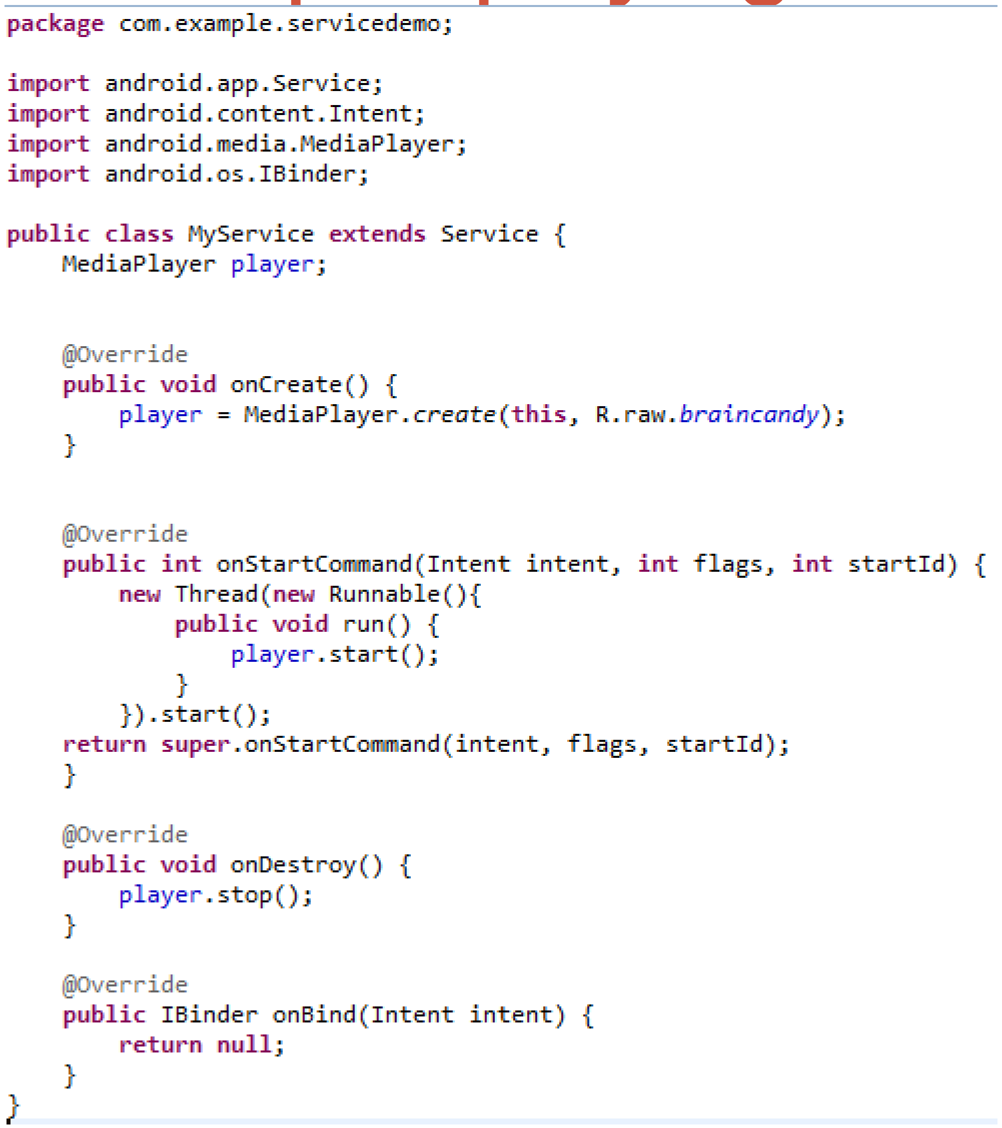
```
package com.example.servicedemo;

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

import android.view.View;
import android.view.View.OnClickListener;
import android.widget.Toast;

public class MainActivity extends Activity implements OnClickListener {

  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main);
   findViewById(R.id.buttonStart).setOnClickListener(this);
   findViewById(R.id.buttonStop).setOnClickListener(this);

  }

  public void onClick(View src) {

    try {
        Toast.makeText(this, src.getId(), Toast.LENGTH_LONG).show();
        switch (src.getId()) {
        case R.id.buttonStart:
            startService(new Intent(MainActivity.this, MyService.class));
            break;
        case R.id.buttonStop:
            stopService(new Intent(MainActivity.this, MyService.class));
            break;
    }
    } catch (Exception e) {
        Toast.makeText(this, e.toString(), Toast.LENGTH_LONG).show();
    }
  }
}
```



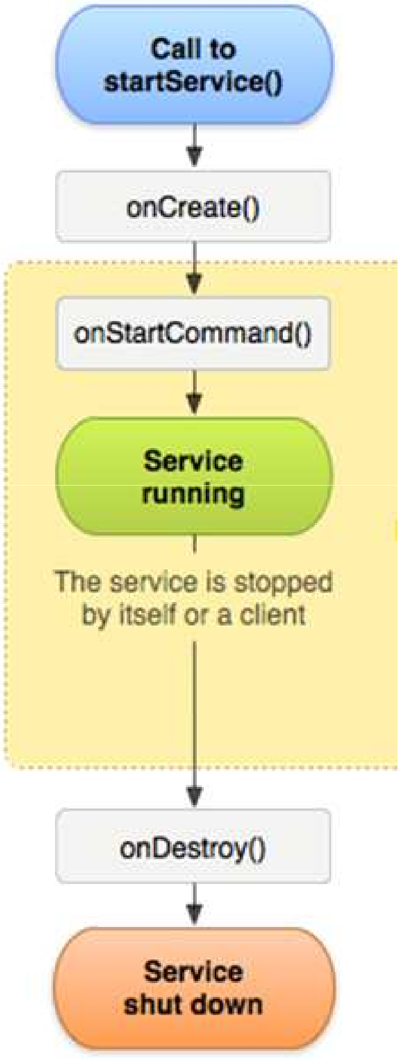


```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:gravity="center"
    android:orientation="vertical" >

    <TextView
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:gravity="center"
        android:padding="20dp"
        android:text="Services Demo"
        android:textSize="20sp" />

    <Button
        android:id="@+id/buttonStart"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Start" >
    </Button>

    <Button
        android:id="@+id/buttonStop"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Stop" >
    </Button>


</LinearLayout>
```

Example: playing music



```java
package com.example.servicedemo;

import android.app.Service;
import android.content.Intent;
import android.media.MediaPlayer;
import android.os.IBinder;

public class MyService extends Service {
    MediaPlayer player;


    @Override
    public void onCreate() {
        player = MediaPlayer.create(this, R.raw.braincandy);
    }


    @Override
    public int onStartCommand(Intent intent, int flags, int startId) {
        new Thread(new Runnable(){
            public void run() {
                player.start();
            }
        }).start();
    return super.onStartCommand(intent, flags, startId);
    }

    @Override
    public void onDestroy() {
        player.stop();
    }

    @Override
    public IBinder onBind(Intent intent) {
        return null;
    }
}
```



Unbounded service

- VEDI SERVICE Demo

Bound Service

- A bound service is similar to a started service with the exception that a started service does not generally return results or permit interaction with the component that launched it.

- A bound service, on the other hand, allows the launching component to interact with, and receive results from, the service. Through the implementation of interprocess communication (IPC), this interaction can also take place across process boundaries.

- An activity might, for example, start a service to handle audio playback. The activity will, in all probability, include a user interface providing controls to the user for the purpose of pausing playback or skipping to the next track.

- Similarly, the service will quite likely need to communicate information to the calling activity to indicate that the current audio track has completed and to provide details of the next track that is about to start playing.

Bound Service

- A component (also referred to in this context as a client) starts and binds to a bound service via a call to the *bindService()* method and multiple components may bind to a service simultaneously.

- When the service binding is no longer required by a client, a call should be made to the *unbindService()* method. When the last bound client unbinds from a service, the service will be terminated by the Android runtime system.

- It is important to keep in mind that a bound service may also be started via call to *startService()*. Once started, components may then bind to it via *bindService()* calls.

- When a bound service is launched via a call to *startService()* it will continue to run even after the last client unbinds from it.

Bound Service

- A bound service must include an implementation of the *onBind()* method which is called both when the service is initially created and when other clients subsequently bind to the running service.

- The purpose of this method is to return to binding clients an object of type IBinder containing the information needed by the client to communicate with the service.

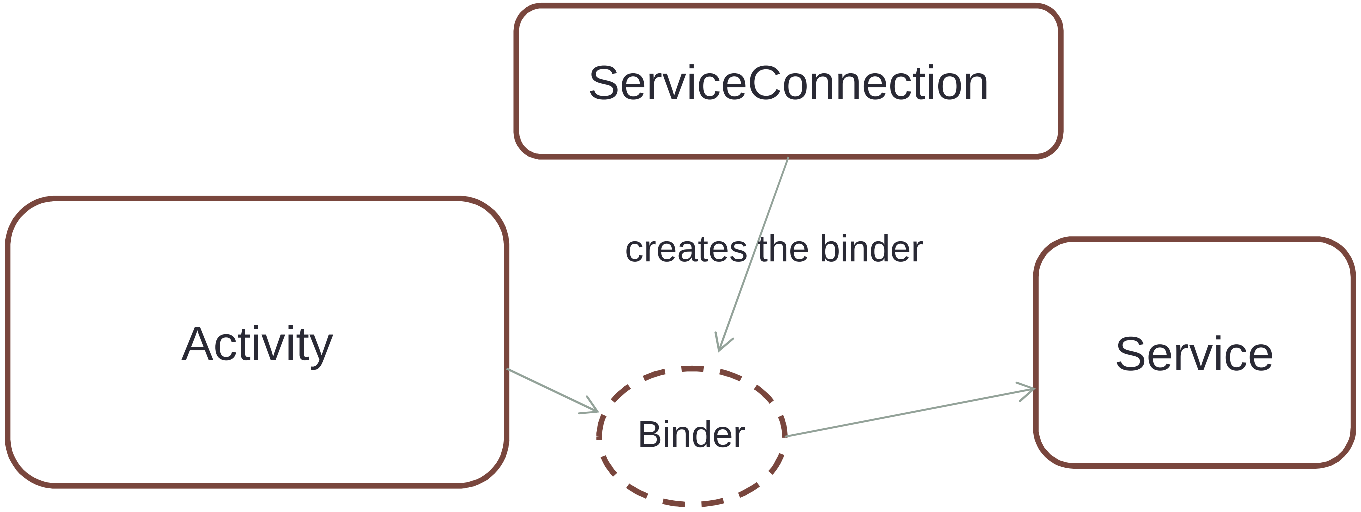
- In terms of implementing the communication between a client and a bound service, the recommended technique depends on whether the client and service reside in the same or different processes and whether or not the service is private to the client.

- Local communication can be achieved by extending the Binder class and returning an instance from the *onBind()* method. Interprocess communication, on the other hand, requires Messenger and Handler implementation.

Bound Service

- A service can be bounded to another SW component, meaning that it can invoke methods implemented by the service through a proxy (Binder) of the Service (which is seen as a remote object)

- Service connection is an interface monitoring connections to a service



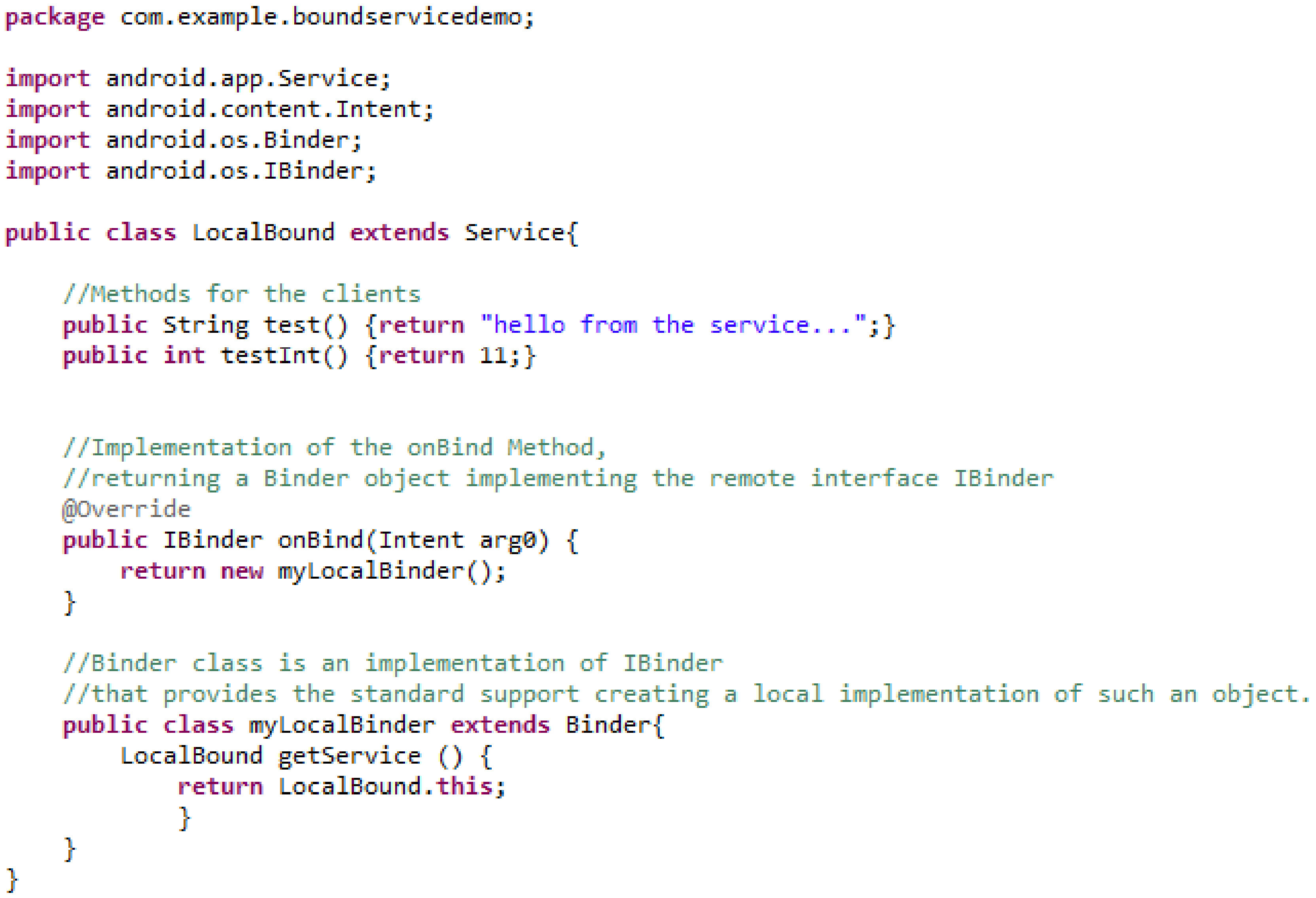
Represents the service

Bound service

- To create a bound service, you must implement the onBind() callback method to return an IBinder that defines the interface for communication with the service.

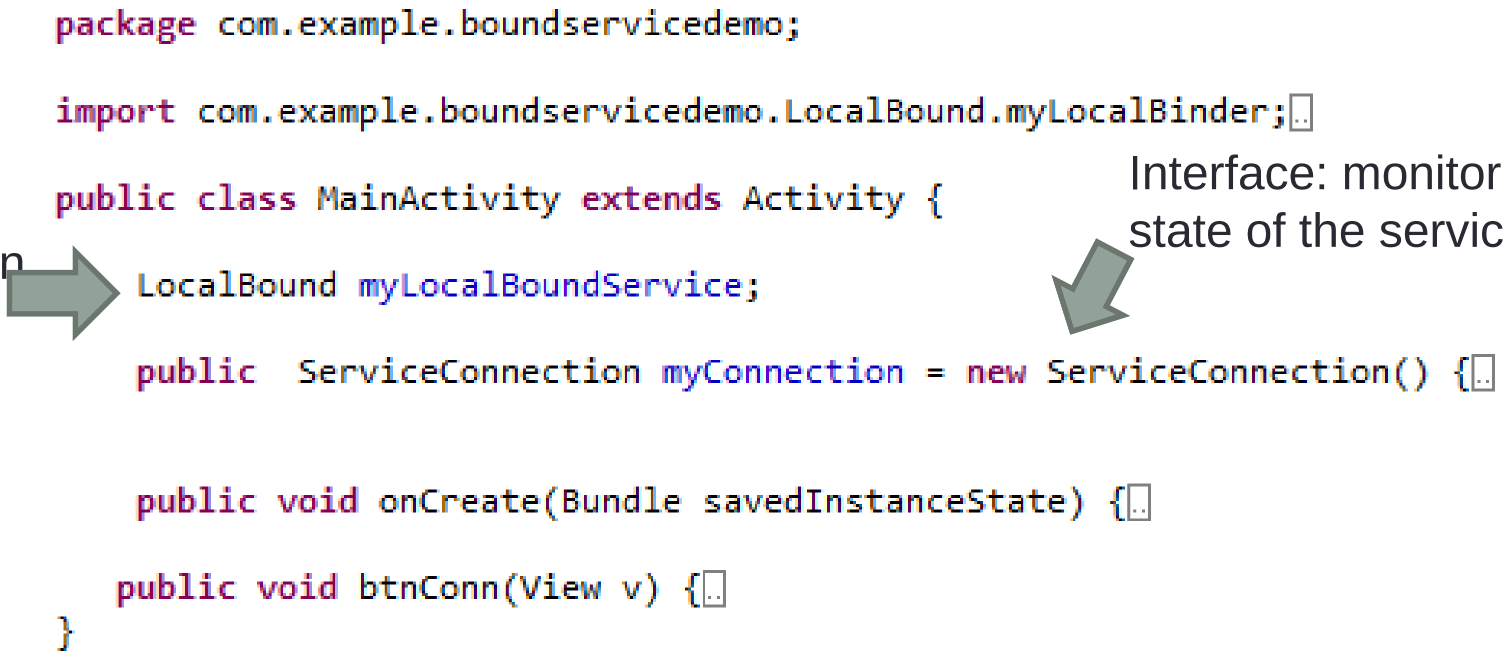
- Other application components can then call bindService() to retrieve the interface and begin calling methods on the service.
    - The client can even call public methods defined in the service (see example)

Example



```
package com.example.boundservicedemo;

import android.app.Service;
import android.content.Intent;
import android.os.Binder;
import android.os.IBinder;

public class LocalBound extends Service{

    //Methods for the clients
    public String test() {return "hello from the service...";}
    public int testInt() {return 11;}


    //Implementation of the onBind Method,
    //returning a Binder object implementing the remote interface IBinder
    @Override
    public IBinder onBind(Intent arg0) {
        return new myLocalBinder();
    }

    //Binder class is an implementation of IBinder
    //that provides the standard support creating a local implementation of such an object.
    public class myLocalBinder extends Binder{
        LocalBound getService () {
            return LocalBound.this;
            }
    }
}
```

Example



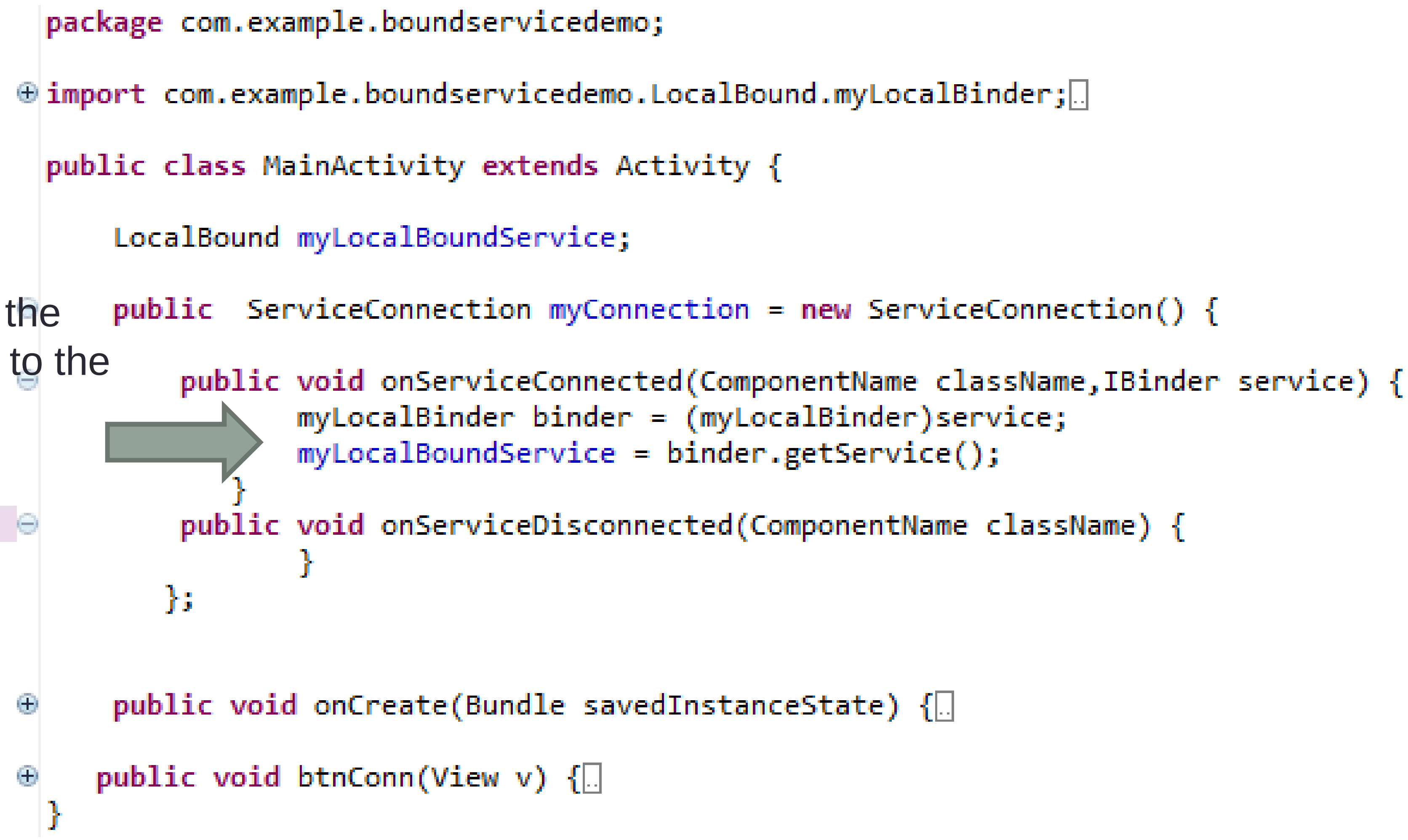
```
package com.example.boundservicedemo;

import com.example.boundservicedemo.LocalBound.myLocalBinder;[..]

public class MainActivity extends Activity {

    LocalBound myLocalBoundService;

    public  ServiceConnection myConnection = new ServiceConnection() {[..]


    public void onCreate(Bundle savedInstanceState) {[..]

   public void btnConn(View v) {[..]
}
```

Local representation of the remote service

Interface: monitor the state of the service

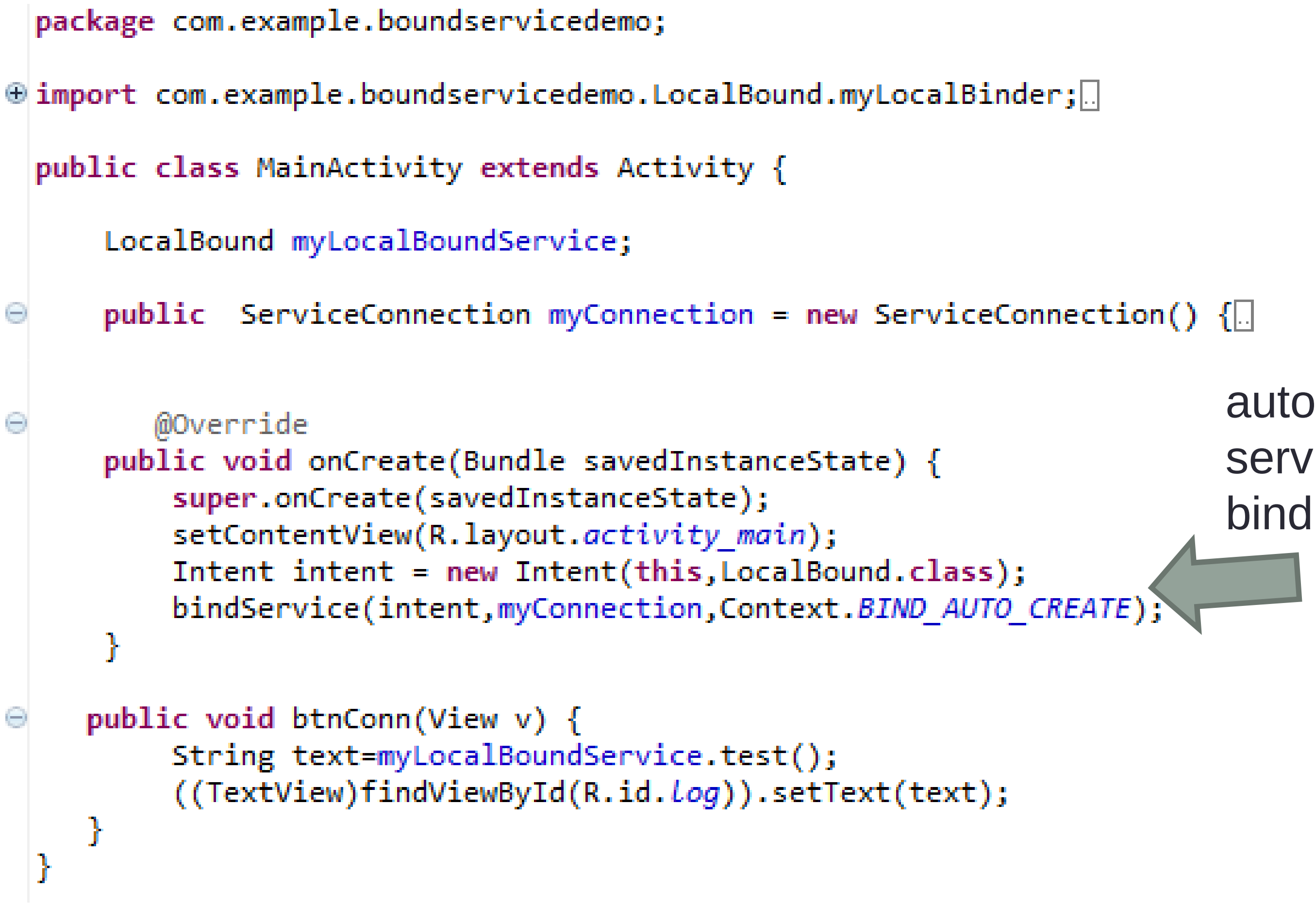
Example

Retrieve the interface to the service



```
package com.example.boundservicedemo;

import com.example.boundservicedemo.LocalBound.myLocalBinder;

public class MainActivity extends Activity {

    LocalBound myLocalBoundService;

    public  ServiceConnection myConnection = new ServiceConnection() {

        public void onServiceConnected(ComponentName className,IBinder service) {
               myLocalBinder binder = (myLocalBinder)service;
               myLocalBoundService = binder.getService();
           }
        public void onServiceDisconnected(ComponentName className) {
               }
       };


    public void onCreate(Bundle savedInstanceState) {

   public void btnConn(View v) {
}
```

Example



```
package com.example.boundservicedemo;

import com.example.boundservicedemo.LocalBound.myLocalBinder;

public class MainActivity extends Activity {

    LocalBound myLocalBoundService;

    public  ServiceConnection myConnection = new ServiceConnection() {


       @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        Intent intent = new Intent(this,LocalBound.class);
        bindService(intent,myConnection,Context.BIND_AUTO_CREATE);
    }

   public void btnConn(View v) {
        String text=myLocalBoundService.test();
        ((TextView)findViewById(R.id.log)).setText(text);
   }
}
```

automatically create the service as long as the binding exists.

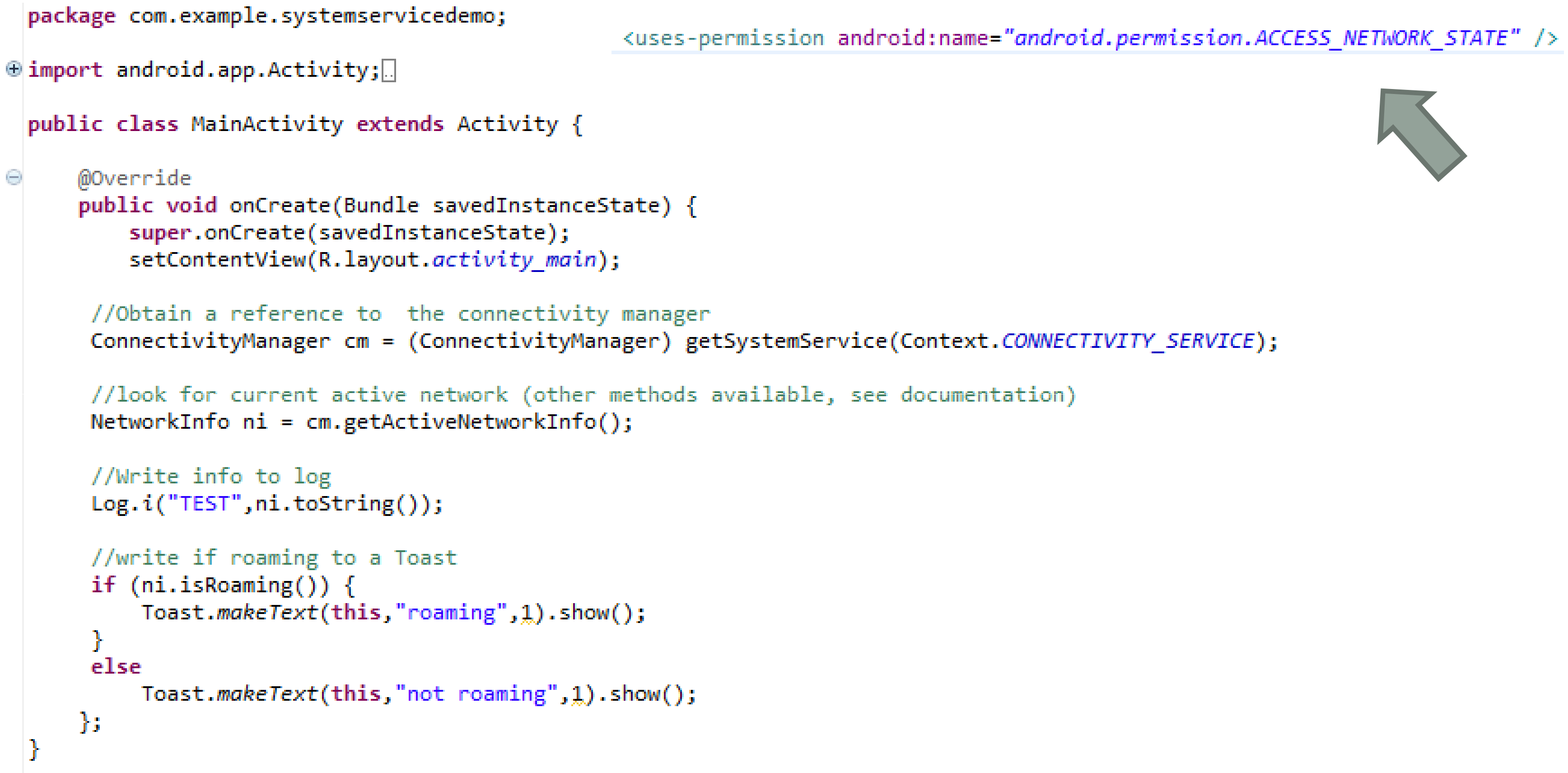
Other example

- LocalBound in Other Projects

System-level services

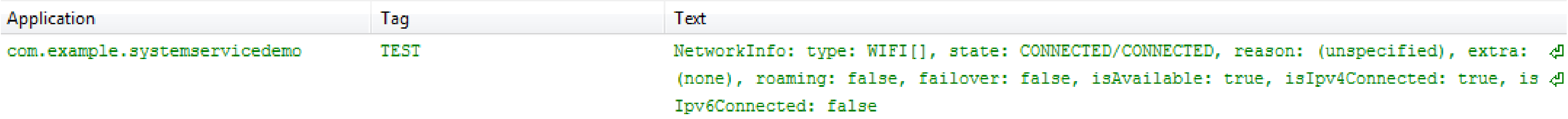
- The Android platform provides a lot of pre-defined services, usually exposed via a Manager class, see:
  - http://developer.android.com/reference/android/content/Context.html
- For example the next applications provides info about the currently connected network….

Example



```
package com.example.systemservicedemo;

import android.app.Activity;

public class MainActivity extends Activity {

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

     //Obtain a reference to  the connectivity manager
     ConnectivityManager cm = (ConnectivityManager) getSystemService(Context.CONNECTIVITY_SERVICE);

     //look for current active network (other methods available, see documentation)
     NetworkInfo ni = cm.getActiveNetworkInfo();

     //Write info to log
     Log.i("TEST",ni.toString());

     //write if roaming to a Toast
     if (ni.isRoaming()) {
         Toast.makeText(this,"roaming",1).show();
     }
     else
         Toast.makeText(this,"not roaming",1).show();
    };
}
```

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



```
Application                   Tag                       Text
com.example.systemservicedemo TEST                      NetworkInfo: type: WIFI[], state: CONNECTED/CONNECTED, reason: (unspecified), extra:
                                                        (none), roaming: false, failover: false, isAvailable: true, isIpv4Connected: true, is
                                                        Ipv6Connected: false
```

Service vs thread

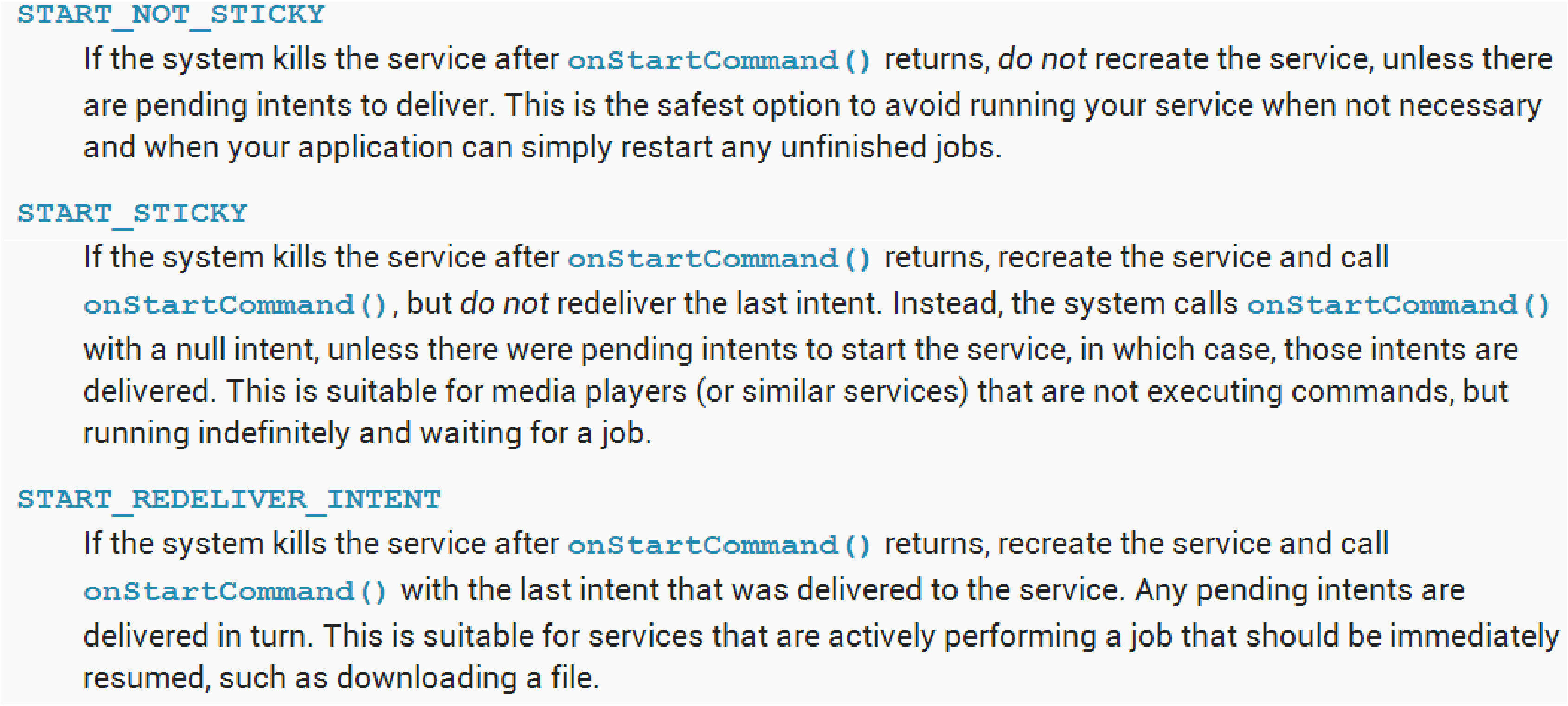
- A service is a component that Android is aware of (it must be declared in the manifest), with its own lifecycle

- A service can be activated from other components (not true for threads)

- A service is destroyed by the system only under very heavy circumstances and re-created according to restart options

- A service is in a sense similar to a Unix’s daemon, e.g, it can be used system-wide and started automatically after the device boot ends

Controlling Destroyed Service restart Options



`START_NOT_STICKY`

If the system kills the service after `onStartCommand()` returns, *do not* recreate the service, unless there are pending intents to deliver. This is the safest option to avoid running your service when not necessary and when your application can simply restart any unfinished jobs.

`START_STICKY`

If the system kills the service after `onStartCommand()` returns, recreate the service and call `onStartCommand()`, but *do not* redeliver the last intent. Instead, the system calls `onStartCommand()` with a null intent, unless there were pending intents to start the service, in which case, those intents are delivered. This is suitable for media players (or similar services) that are not executing commands, but running indefinitely and waiting for a job.

`START_REDELIVER_INTENT`

If the system kills the service after `onStartCommand()` returns, recreate the service and call `onStartCommand()` with the last intent that was delivered to the service. Any pending intents are delivered in turn. This is suitable for services that are actively performing a job that should be immediately resumed, such as downloading a file.

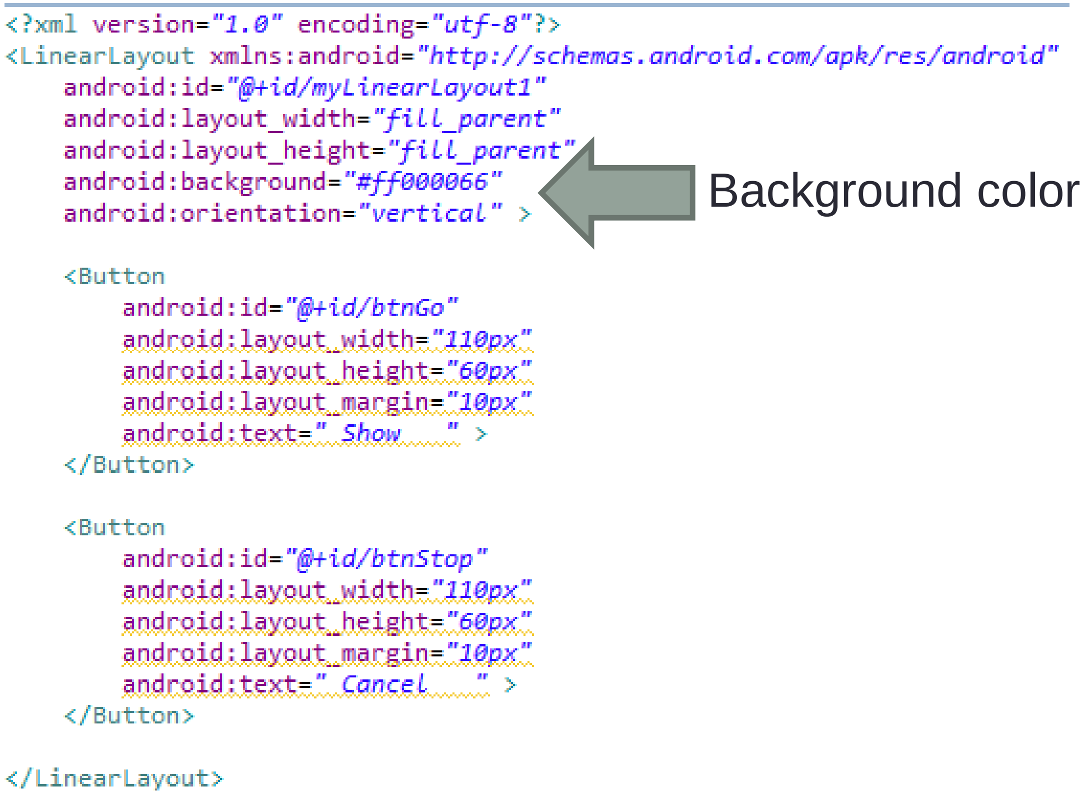
Service priority

- The system kills the process hosting a service if it is under heavy memory pressure.
- However, if this happens, the system will later try to restart the service (and a pending intent can be delivered again)
- A processes hosting service have higher priority than those running an activity

Example: use notification

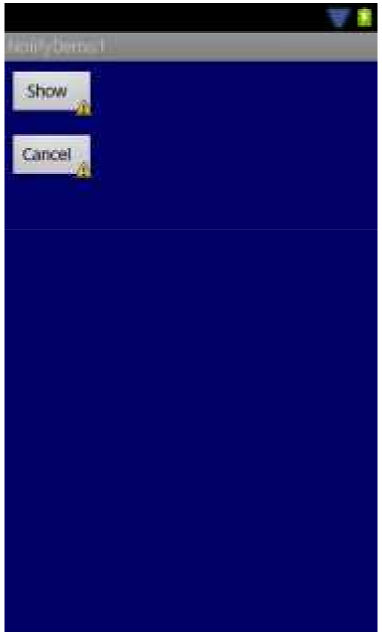
- Send a message, displayed by the status bar
- Read the message associated to the notification

Example: UI



```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:id="@+id/myLinearLayout1"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:background="#ff000066"
    android:orientation="vertical" >

    <Button
        android:id="@+id/btnGo"
        android:layout_width="110px"
        android:layout_height="60px"
        android:layout_margin="10px"
        android:text=" Show     " >
    </Button>

    <Button
        android:id="@+id/btnStop"
        android:layout_width="110px"
        android:layout_height="60px"
        android:layout_margin="10px"
        android:text=" Cancel     " >
    </Button>

</LinearLayout>
```

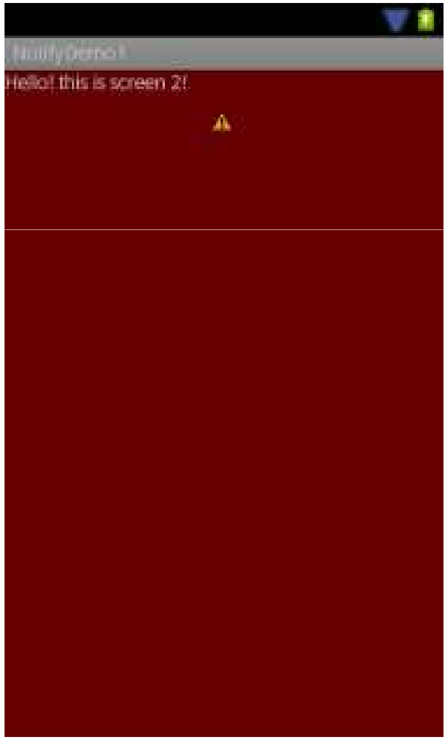
Background color



Adapted from : Victor Matos CS493

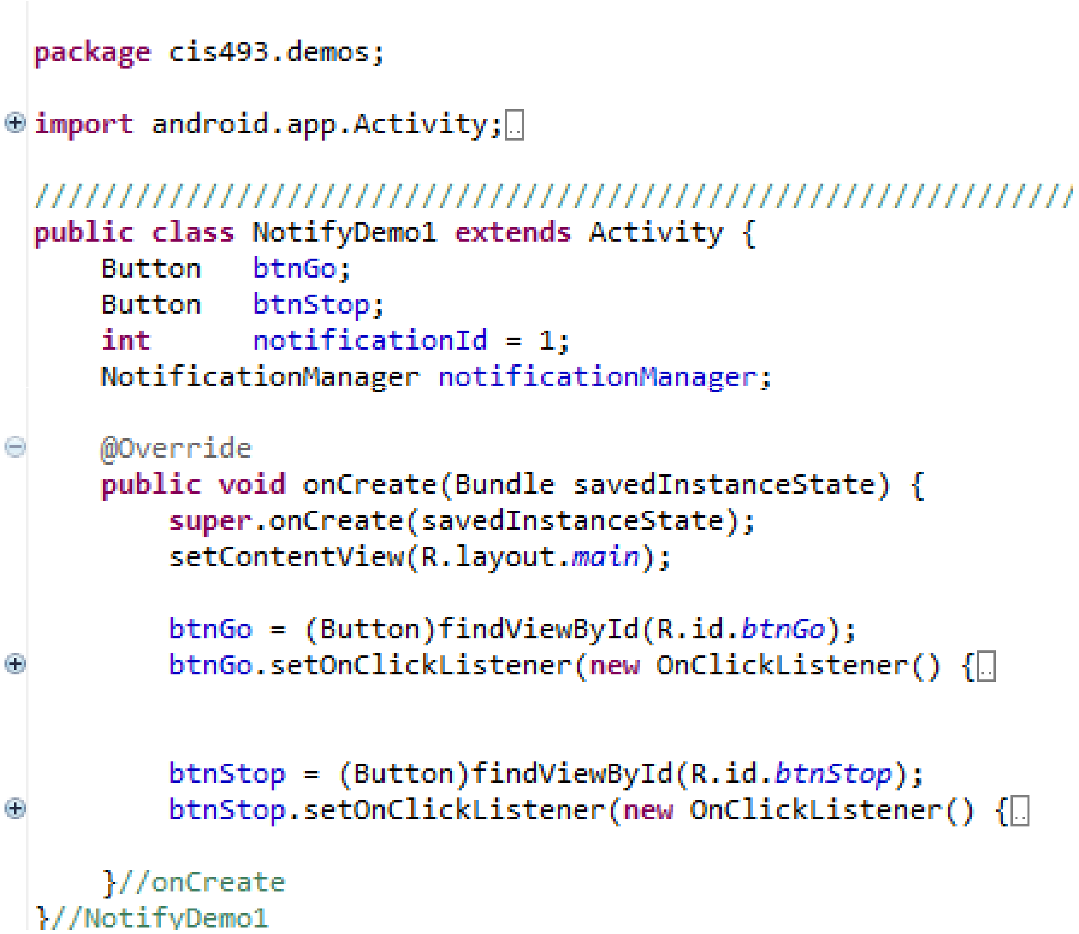
Example: UI

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:id="@+id/main2LinLayout"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:background="#ff660000"
    android:orientation="vertical" >

    <TextView
        android:id="@+id/widget29"
        android:layout_width="251px"
        android:layout_height="69px"
        android:text="Hello! this is screen 2!" >
    </TextView>

</LinearLayout>
```



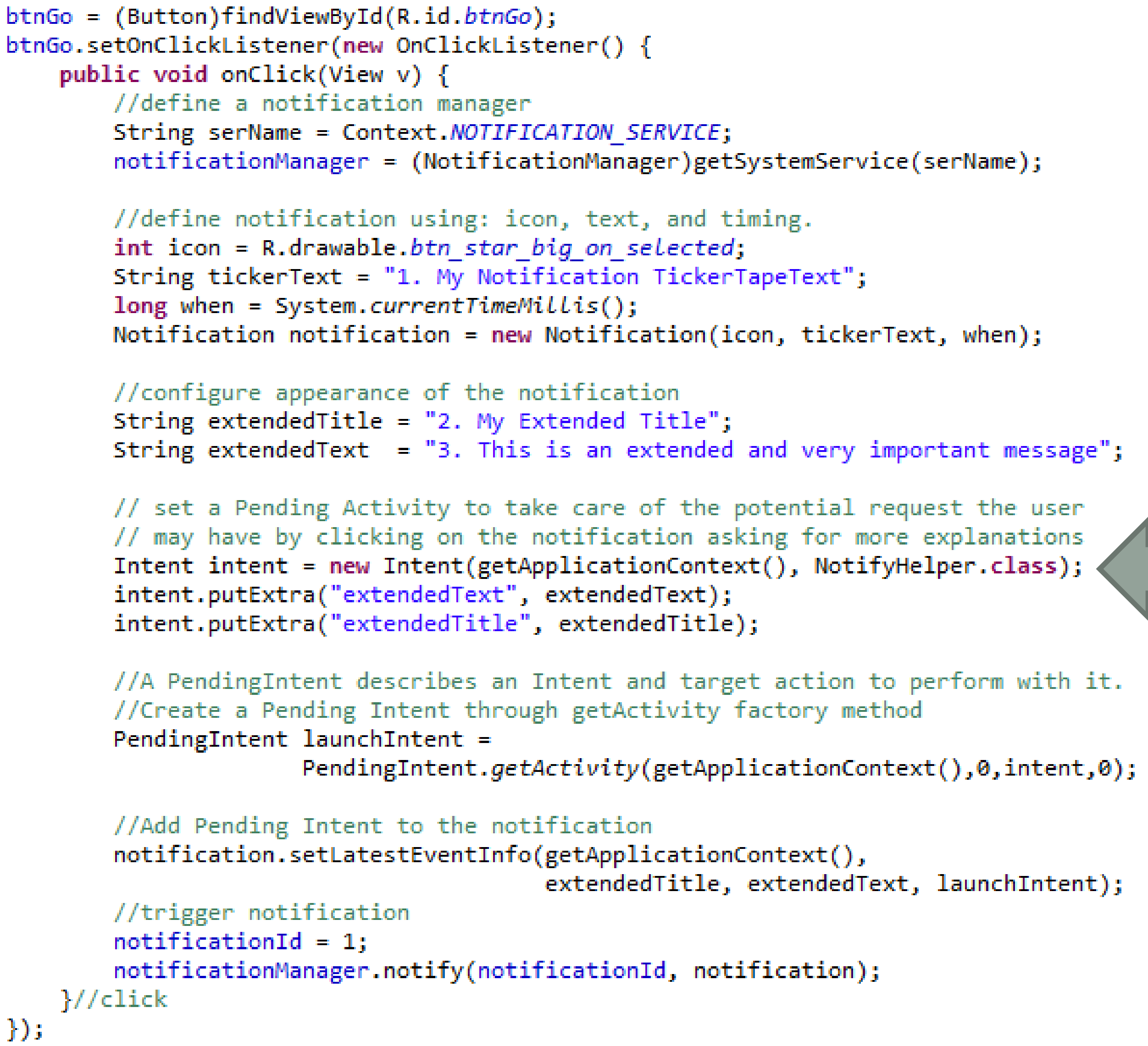
Hello! this is screen 2!

Example



```java
package cis493.demos;

import android.app.Activity;□

//////////////////////////////////////////////////////////////
public class NotifyDemo1 extends Activity {
    Button   btnGo;
    Button   btnStop;
    int      notificationId = 1;
    NotificationManager notificationManager;

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);

        btnGo = (Button)findViewById(R.id.btnGo);
        btnGo.setOnClickListener(new OnClickListener() {□


        btnStop = (Button)findViewById(R.id.btnStop);
        btnStop.setOnClickListener(new OnClickListener() {□

    }//onCreate
}//NotifyDemo1
```

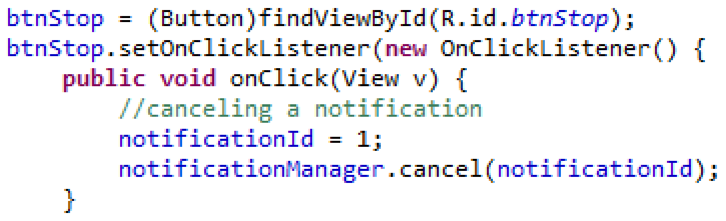
Example: create a notification



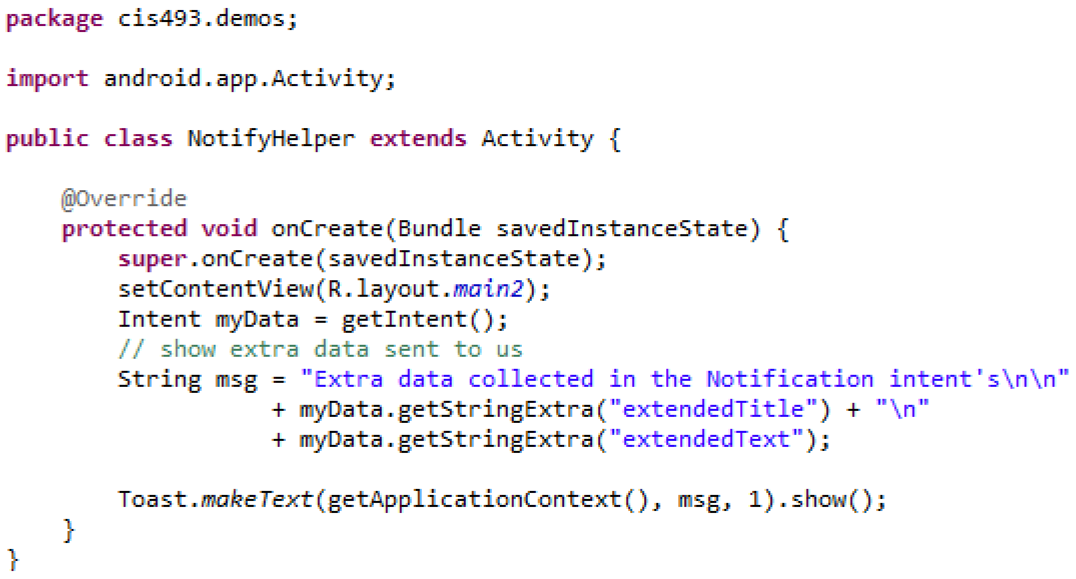
```
btnGo = (Button)findViewById(R.id.btnGo);
btnGo.setOnClickListener(new OnClickListener() {
    public void onClick(View v) {
        //define a notification manager
        String serName = Context.NOTIFICATION_SERVICE;
        notificationManager = (NotificationManager)getSystemService(serName);

        //define notification using: icon, text, and timing.
        int icon = R.drawable.btn_star_big_on_selected;
        String tickerText = "1. My Notification TickerTapeText";
        long when = System.currentTimeMillis();
        Notification notification = new Notification(icon, tickerText, when);

        //configure appearance of the notification
        String extendedTitle = "2. My Extended Title";
        String extendedText  = "3. This is an extended and very important message";

        // set a Pending Activity to take care of the potential request the user
        // may have by clicking on the notification asking for more explanations
        Intent intent = new Intent(getApplicationContext(), NotifyHelper.class);
        intent.putExtra("extendedText", extendedText);
        intent.putExtra("extendedTitle", extendedTitle);

        //A PendingIntent describes an Intent and target action to perform with it.
        //Create a Pending Intent through getActivity factory method
        PendingIntent launchIntent =
                      PendingIntent.getActivity(getApplicationContext(),0,intent,0);

        //Add Pending Intent to the notification
        notification.setLatestEventInfo(getApplicationContext(),
                                        extendedTitle, extendedText, launchIntent);
        //trigger notification
        notificationId = 1;
        notificationManager.notify(notificationId, notification);
    }//click
});
```

See next slide

Example: cancel a notification



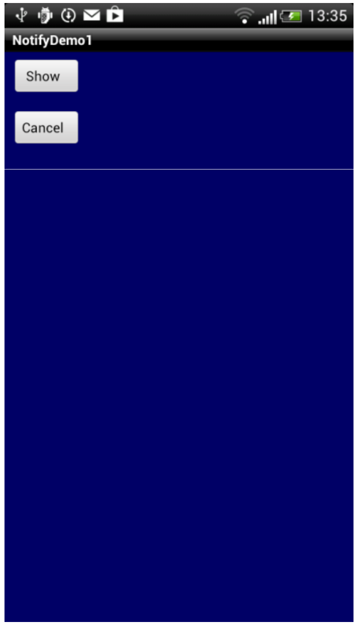
```
btnStop = (Button)findViewById(R.id.btnStop);
btnStop.setOnClickListener(new OnClickListener() {
    public void onClick(View v) {
        //canceling a notification
        notificationId = 1;
        notificationManager.cancel(notificationId);
    }
```



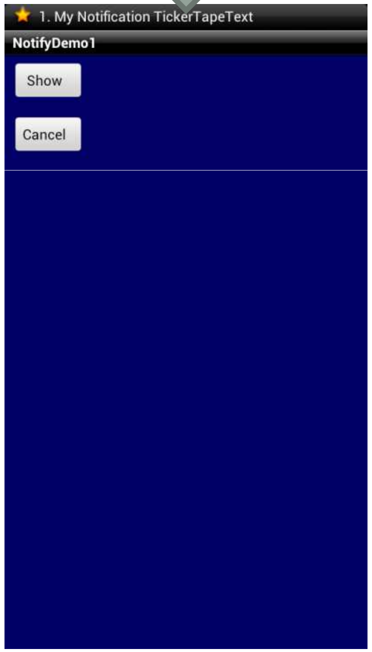
```
package cis493.demos;

import android.app.Activity;

public class NotifyHelper extends Activity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main2);
        Intent myData = getIntent();
        // show extra data sent to us
        String msg = "Extra data collected in the Notification intent's\n\n"
                   + myData.getStringExtra("extendedTitle") + "\n"
                   + myData.getStringExtra("extendedText");

        Toast.makeText(getApplicationContext(), msg, 1).show();
    }
}
```

NotifyHelper

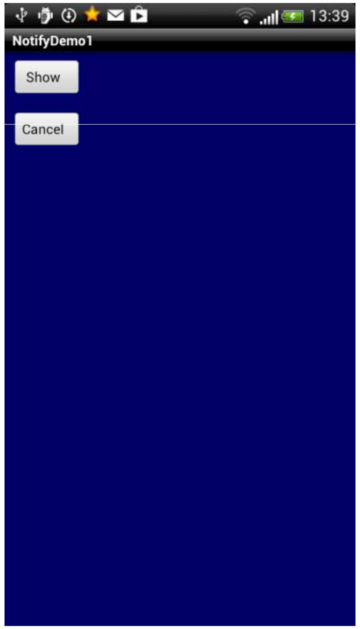
Running the application



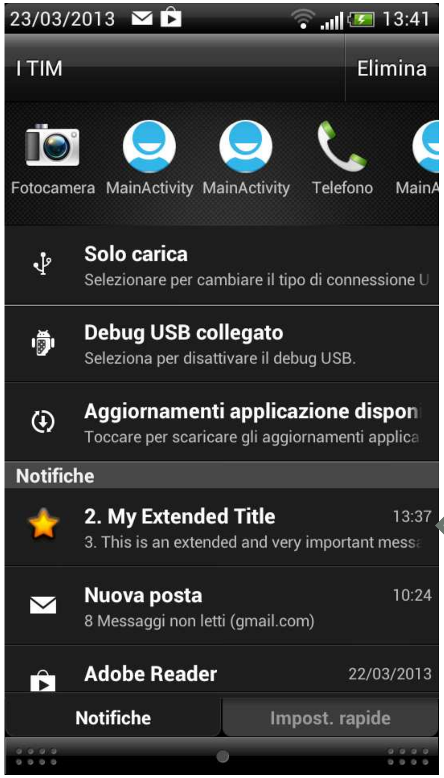
Ticker Tape Text



Icon (star) of the notification



Running the application



Solo carica
Selezionare per cambiare il tipo di connessione U

Debug USB collegato
Seleziona per disattivare il debug USB.

Aggiornamenti applicazione dispon
Toccare per scaricare gli aggiornamenti applica

Notifiche

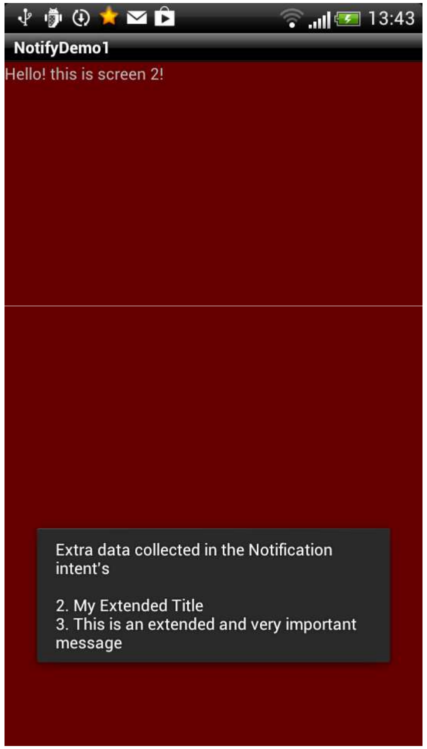
2. My Extended Title 13:37
3. This is an extended and very important messa

Nuova posta 10:24
8 Messaggi non letti (gmail.com)

Adobe Reader 22/03/2013

Activity launched through intent

Extended information



Hello! this is screen 2!

Extra data collected in the Notification intent's

2. My Extended Title
3. This is an extended and very important message