Playing with Intents

Intent-based programming is one of the most distinctive features of the Android platform. The entire application space effectively consists of components (called Activities in Android parlance) and messages among components (called Intents). The Android application model is therefore a simple service-oriented architecture which is indeed an interesting approach.

Intents can be used in many ways to invoke another Activity. The tutorial is not very explicit on the two main kinds of Intents so the I had to discover them myself in the documentation of the Intent class.

- Explicit intent targets a particular class that has been declared intent receiver in the AndroidManifest.xml
- Implicit intent targets an intent receiver with particular characteristics (like a particular action)

Intent-based programming is interesting that's why I decided to play around with it. I had very unpleasant experiences with the Blogger engine rendering preformatted text like program code (in addition, the engine swallowed inexplicably part of an XML sample document that I tried to insert into the post) so I decided to upload the project bundles onto a download page so that I can copy into the post only the relevant code fragments. Note that sdk-folder and android-tools properties in the build.xml files need to be updated so that they reflect the location of your Android SDK installation directories.

I tried out three setups in the simple test program (that actually consists of two Android applications).

- Explicit intent addressing with the invoked activity internal to the application (exp/int).
- Explicit intent addressing with the invoked activity external to the application (IntentSender application invokes IntentReceiver application) (exp/ext).
- Implicit intent addressing with external activity invocation (imp/ext). The invoked activity is again in the IntentReceiver application.

Download the project bundle from here, compile and install both IntentSender and IntentReceiver applications (they are in the intentsender and intentreceiver directories, respectively. Both applications need to be compiled with ant and need to be installed on the emulator as described in an earlier post). You can launch IntentSender and try out the activity invocations by pressing the appropriate buttons.

The implementation of this simple application did have its adventures. :-) The tutorial uses a simple form of explicit intent creation.
Intent i = new Intent(this, NoteEdit.class);

This was not usable for me because in the exp/ext case the target activity class was not located within the application. The solution seemed to be simple and relied on the Intent class' setClassName( pkgName, className ) method. Who could have thought that the right form of parametrization requires full path in className too (after having received the package part in the pkgName parameter)? This took me something like an hour wasted and was resolved by finding out, how the tutorial version of explicit invocation works.

The critical piece of code is the following:

```
Intent i = new Intent();
i.setClassName( "aexp.intentreceiver", "aexp.intentreceiver.IntentReceiver" );
i.putExtra( "i1", new Integer( 4 ) );
i.putExtra( "i2", new Integer( 5 ) );
startSubActivity(i, ACTIVITY_INVOKE);
```

The target activity needs to be declared in the AndroidManifest.xml. IntentReceiver class in IntentReceiver application happens to be the app's main intent receiver therefore no additional declaration is necessary. The internal IntentReceiver in IntentSender needs to be declared in IntentSender's AndroidManifest.xml.

```
[activity class=".IntentReceiver"]
```

(of course, this is XML with <> characters, I just can't get it through the #&@!/ blog engine).

After getting through the explicit addressing's arcane package name/class name convention, I went after the implicit addressing which seemed similarly simple. In this case, there is an extra level of indirection between the invoker and invoked activity. The intent does not carry the target class, it carries a set of information that is used by the system to identify the target activity. In my example, I used only the intent action that I set to an action string I made up myself (aexp.intentsender.add).

Invoking code is simple:

```
Intent i = new Intent();
i.setAction( "aexp.intentsender.add" );
i.putExtra( "i1", new Integer( 5 ) );
i.putExtra( "i2", new Integer( 6 ) );
startSubActivity(i, ACTIVITY_INVOKE);
```

The intent receiver is identified according to the intent filter declaration in IntentReceiver's AndroidManifest.xml.

```
[intent-filter]
[action android:value="aexp.intentsender.add"/
[category android:value="android.intent.category.DEFAULT"/
[/intent-filter]
```

There is one point here to note that costed me again some half an hour wasted. :-) Although I don't use any categories, Intent's constructor creates one category by default, the Intent.DEFAULT_CATEGORY whose string format is android.intent.category.DEFAULT. I did not include the category originally into the AndroidManifest.xml and that's why the intent resolution was not succesful. This problem was rectified using the fantastic adb logcat command.
Guess, what goes into the log if there is an action match whose category does not match?

W/IntentResolver(461): resolveIntent failed: found match, but none with Intent.DEFAULT_CATEGORY

Thanks, emulator developers, this log message was really helpful!

Posted by Gabor Paller at 3:55 PM 🍂

Labels: intent

1 comments:

Anonymous said...

I wish to thank you for this article! I had been trying to invoke an activity across applications and for the life of me couldn't figure out what I was doing wrong! After altering my AndroidManifest.xml file to use the Default category, the cross activity calling worked! The constructor for 'Intent' states that an "empty" Intent is created, but this isn't true based on your findings which I confirmed!

Thanks for the great help!

Intents are broken up into two main categories:

- **Activity Action Intents** Intents used to call Activities outside of your application. Only one Activity can handle the Intent. For example, for a web browser, you need to open the Web Browser Activity to display a page.

- **Broadcast Intents** Intents that are sent out for multiple Activities to handle. An example of a Broadcast Intent would be a message sent out by Android about the current battery level. Any Activity can process this Intent and react accordingly—for example, cancel an Activity if the battery level is below a certain point.
Activity Action Intent Message
ADD_SHORTCUT_ACTION Add a function shortcut to the Android Home Screen
ALL_APPS_ACTION List all the applications available on the device
ANSWER_ACTION Answer an incoming call
CALL_ACTION Place a call to supplied location
DELETE_ACTION Delete the specified data
DIAL_ACTION Open the Dial Activity and dial the specified number
EDIT_ACTION Provide editable access to the supplied data
EMERGENCY_DIAL_ACTION Dial an emergency number
FACTORY_TEST_ACTION Retrieve factory test settings
GET_CONTENT_ACTION Select and return specified data
INSERT_ACTION Insert an empty item
MAIN_ACTION Establish the Activity start point
PICK_ACTION Pick an item and return the selection
PICK_ACTIVITY_ACTION Pick a given Activity (returns a class)
RUN_ACTION Execute the given data
SEARCH_ACTION Launch a search on the system
SEND_ACTION Send data without specifying the recipient
SENDTO_ACTION Send data to the recipient specified
SETTINGS_ACTION Launch System Settings
VIEW_ACTION (DEFAULT_ACTION) Open a View
Table 7-2 lists and describes the current Broadcast Intents that are available. Refer to this list when you need to establish a receiver for a specific Intent.
Broadcast Intent Message
CALL_FORWARDING_STATE_CHANGED_ACTION The phone’s call forwarding state has changed
CAMERA_BUTTON_ACTION The camera button has been pressed
CONFIGURATION_CHANGED_ACTION The device’s configuration has changed
DATA_ACTIVITY_STATE_CHANGED_ACTION The device’s data activity state has changed
DATA_CONNECTION_STATE_CHANGED_ACTION There has been a change in the data connection state
DATE_CHANGED_ACTION The phone’s system date has changed
FOTA_CANCEL_ACTION Cancel pending system update downloads
FOTA_INSTALL_ACTION An update has been downloaded and must be installed immediately (sent by the system)
FOTA_READY_ACTION An update has been downloaded and can be installed—but does not need to be installed immediately (sent by the system)
FOTA_RESTART_ACTION Restart a system update download
FOTA_UPDATE_ACTION Begin the download of a system update
GTALK_SERVICES_CONNECTED_ACTION Sent when a GTALK session has been successfully established
GTALK_SERVICES_DISCONNECTED_ACTION Sent when a GTALK session has been disconnected
MEDIA_BAD_REMOVAL_ACTION Sent when an SD Memory Card was removed but unsuccessfully unmounted from the system
MEDIA_BUTTON_ACTION Sent when the media button has been pressed
**Broadcast Intent Message**

MEDIA_EJECT_ACTION Sent when the eject action has been initiated on an SD Memory Card
MEDIA_MOUNTED_ACTION Sent when an SD Memory Card was successfully mounted to the system
MEDIA_REMOVED_ACTION Sent when an SD memory card was detected as having been removed
MEDIA_SCANNER_FINISHED_ACTION Sent when the scanner has finished
MEDIA_SHARED_STARTED_ACTION Sent when the scanner has begun
MEDIA_UNMOUNTED_ACTION Sent when an SD memory card has been detected but has not been mounted
MESSAGE_WAITING_STATE_CHANGED The “message waiting” state on the phone has changed
NETWORK_TICKLE_RECEIVED_ACTION A new device network notification has been received
PACKAGE_ADDED_ACTION Sent when a new package has been installed on the device
PACKAGE_CHANGE_ACTION Sent when an existing package has been modified
PACKAGE_INSTALL_ACTION A package can be downloaded and installed
PACKAGE_REMOVED_ACTION A package has been removed
PHONE_INTERFACE_ADDED_ACTION The device’s phone interface has been created
PHONE_STATE_CHANGED_ACTION The device’s phone state has changed
PROVIDER_CHANGED_ACTION The device has received a notification from a provider
PROVISIONING_CHECK_ACTION Check for the latest settings from the provisioning service
SCREEN_OFF_ACTION The screen has been shut off (sent by the device)
SCREEN_ON_ACTION The screen has been turned on (sent by the device)
SERVICE_STATE_CHANGED_ACTION The service state has changed
SIGNAL_STRENGTH_CHANGED_ACTION The signal strength has changed

Table 7-2 Broadcast Intents (continued)

**Broadcast Intent Message**

SIM_STATE_CHANGED_ACTION The state of the SIM card has changed
TIME_CHANGED_ACTION The device’s time was set
TIME_TICK_ACTION The current time has changed
TIMEZONE_CHANGED_ACTION The device’s timezone has changed
UMS_CONNECTED_ACTION The device has connected via USB
UMS_DISCONNECTED_ACTION The device has been disconnected from its USB host
WALLPAPER_CHANGED_ACTION The device’s wallpaper has been changed

Table 7-2 Broadcast Intents (continued)

The Intent is only one-third of the picture. An Intent is really just that, an intent to do something; an Intent cannot actually do anything by itself. You need **Intent Filters** and **Intent Receivers** to listen for, and interpret, the Intents.
Can I use this Intent?

Posted by Romain Guy on 05 January 2009 at 6:00 AM

Android offers a very powerful and yet easy to use tool called intents. An intent can be use to turn applications into high-level libraries and make code re-use something even better than before. The Android Home screen and AnyCut use intents extensively to create shortcuts for instance. While it is nice to be able to make use of a loosely coupled API, there is no guarantee that the intent you send will be received by another application. This happens in particular with 3rd party apps, like Panoramio and its RADAR intent.

While working on a new application, I came up with a very simple way to find out whether the system contains any application capable of responding to the intent you want to use. I implemented this technique in my application to gray out the menu item that the user would normally click to trigger the intent. The code is pretty simple and easy to follow:

```java
/**
 * Indicates whether the specified action can be used as an intent. This
 * method queries the package manager for installed packages that can
 * respond to an intent with the specified action. If no suitable package
 * is
 * found, this method returns false.
 * *
 * @param context The application's environment.
 * @param action The Intent action to check for availability.
 * *
 * @return True if an Intent with the specified action can be sent and
 *         responded to, false otherwise.
 */
public static boolean isIntentAvailable(Context context, String action) {
    final PackageManager packageManager = context.getPackageManager();
    final Intent intent = new Intent(action);
    List<ResolveInfo> list =
        packageManager.queryIntentActivities(intent,
PackageManager.MATCH_DEFAULT_ONLY);

    return list.size() > 0;
}

Here is how I use it:

@Override
public boolean onPrepareOptionsMenu(Menu menu) {
    final boolean scanAvailable = isIntentAvailable(this, "com.google.zxing.client.android.SCAN");

    MenuItem item;
    item = menu.findItem(R.id.menu_item_add);
    item.setEnabled(scanAvailable);

    return super.onPrepareOptionsMenu(menu);
}

In this example, the menu is grayed out if the Barcode Scanner application is not installed. Another, simpler, way to do this is to catch the ActivityNotFoundException when calling startActivity() but it only lets you react to the problem, you cannot predict it and update the UI accordingly to prevent the user from doing something that won't work. The technique described here can also be used at startup time to ask the user whether he'd like to install the missing package, you can then simply redirect him to the Android Market by using the appropriate URI.
Intent Broadcast & Receiver

In this post we will see about the IntentReceivers.

Android Intent receivers are part of the event handling mechanism. The intent receiver can handle the broadcast intents and thus can be used as event handler.

Creating Intent Receivers:

android has defined a ‘BroadcastReceiver’ class. All the intent receivers have to inherit from this class. Following is the example of the intent receiver

package com.wissen.testApp.receiver;

public class MyIntentReceiver extends BroadcastReceiver {

/**
 * @see android.content.BroadcastReceiver#onReceive(android.content.Context, android.content.Intent)
 */

@Override
public void onReceive(Context context, Intent intent) {
...
}
}

The intent receiver has to override the method onReceive as shown above. The onReceive method gets called when intent receiver’s intent is broadcasted, and thus onReceive() method is the entry point for the intent receiver.

The intent receiver entry has to be done in the android Manifest.xml file. The intent receiver can be defined in manifest xml as follows:

<intent-filter>

android:name="com.wissen.testApp.MY_INTENT_RECEIVER" />

</intent-filter>
The intent receiver can also be registered dynamically with the help of Context.registerReceiver() method as follows:

```
MyIntentReceiver intentReceiver = new MyIntentReceiver();
IntentFilter intentFilter = new IntentFilter("com.wissen.testApp.MY_INTENT_RECEIVER");
registerReceiver(intentReceiver, intentFilter);
```

**Broadcasting Intent:**

The intent can be broadcasted with the help of sendBroadcast() and sendOrderedBroadcast() method of the Context class. The sendBroadcast method will send the broadcast to all the registered Intent Receivers, on the other hand sendOrderedBroadcast() method calls the intent receivers one at a time. This gives the intent receivers ability to exchange result of previous intent receiver or to abort the broadcast. The order of execution is controls by the android :priority attribute defined in the tag of AndroidManifest.xml file.

**Receiver Lifecycle:**

There is only one lifecycle method for the intent receiver as onReceive(). The method gets called when intent is broadcasted. The receiver object is only valid for the duration of the onReceive() method, after the method ends the object is considered to be non-active and can be garbage collected. Because of this the intent receiver onReceive() method should not handle any asynchronous operation.

To receiver an intent by the intent receiver the application does not need to be running. When the intent is broadcasted the system will start the application to call the intent receiver.

**Permissions:**

Sometimes for defining intent receivers for some intent, permissions need to be specified by the intent receiver. The intent receiver can specify the required permission in the android Manifest.xml file with the help of tag. While registering the intent receiver dynamically, the permission can be specified as the parameter of the registerReceiver method.

To enforce a permission while broadcasting an intent, a non-null permission argument need to be specified for the sendBroadcast() or sendOrderedBroadcast() method.
This is all on the intent receivers. In next post we will see android services and content providers.

**Simple Class for Holding Data**

**public methods for access**

class StationData {
  public String title = "";
  public String url = "";

  public String toString() {
    return title;
  }
}