IntentService in Android

In this tutorial I will explain briefly what an **IntentService** is and how it should be used in an Android app.
To begin with, let's start with a **FAQ**...

What is a Service?
An Android app is made of **components**. The most important components are the following: Activity, **Service**, BroadcastReceiver and ContentProvider.
An IntentService is a special kind of Service and therefore a component of an Android app.

What are Services used for?
Services in Android are generally used for operations that are **not directly involved with the user interface (UI)**. For example: network operations, writing and reading data to/from the local storage (a database, file, etc.), playing music, and other tasks like these.
It is generally better to use **one Service for each different task**: one Service for network operations, one different Service for managing the database and so on.
Some ot these tasks can also be managed within a normal Activity, using a dedicated thread (AsyncTask or Handler, as I explained in a different tutorial), especially if the data involved **comes from the user interface** (for example: writing to a local database the data entered by the user).
But there are other tasks (like network operations or playing music) that are bettere managed by Services.
Generally speaking Services allow you **to efficiently separate** the logic of background tasks from the code that handles the user interface.

Are Services running in a different thread?
No, the code in a Service runs **in the main thread** (the thread of the GUI). So it is very important **to create a different thread within the Service** (using an AsyncTask or Handler) to manage long running or slow operations, otherwise they could slow down or block the user interface.

So, what is an IntentService in the end?
An IntentService is a special utility class which represents a Service whose operations run in a different thread than the thread of the GUI.
This is **managed automatically** by the IntentService class, so you don't have to implement an AsynckTask or a Handler.

Let's see an example.
The following IntentService is used to upload a text or a photo to a remote server in a different thread.

```
public class MyIntentService extends IntentService {

    //constants
    private static final String CLASSNAME = "MyIntentService";

    //the following 2 actions have to be added in the intent filter for the
    //IntentService in the manifest file
    public static final String ACTION_UPLOAD_TEXT = "com.tutorial.UPLOADTEXT";
    public static final String ACTION_UPLOAD_PHOTO = "com.tutorial.UPLOADPHOTO";

    //extras to put in the Intent that starts this IntentService
    public static final String TEXT = "text";
    public static final String PHOTO= "photo";

    public MyIntentService() {
        super(CLASSNAME);

        //if the system shuts down the service, the Intent is not redelivered
        //and therefore the Service won't start again
        setIntentRedelivery(false);
    }

    //this method runs in a different thread (not the main thread)
    @Override
    protected void onHandleIntent(Intent intent) {
        String action = intent.getAction();

        if(action.equals(ACTION_UPLOAD_TEXT)) {
            String text = intent.getStringExtra(TEXT);
            uploadText(text);
        }
        else if(action.equals(ACTION_UPLOAD_PHOTO)) {
            Bitmap photo = itent.getParcelableExtra(PHOTO);
            uploadPhoto(photo);
        }
    }

    private void uploadText(String text) {
        //in this method you upload the text to the server: omitted for brevity
    }

    private void uploadPhoto(Bitmap photo) {
        //in this method you upload the photo to the server: omitted for brevity
    }
}
```

If you want to use this IntentService from your Activity to upload a text or a photo to the remote server in a dedicated thread, **you just have to use the command Context.startService()** with the Itent carrying the specific action and extras.

There is only **one disadvantage** in this approach: if you have multiple calls to the IntentService, these will be put in a queue and processed **one at a time** (there is no parallel execution). If you need parallel execution, to make sure that each background operation is executed as soon as possibile, you have to use a different approach, but this will be the object of a different tutorial.

Tejas Lagvankar wrote a nice post about this subject. Below are some key differences between Service and IntentService.

When to use?

- The *Service* can be used in tasks with no UI, but shouldn't be too long. If you need to perform long tasks, you must use threads within Service.
- The *IntentService* can be used in long tasks usually with no communication to Main Thread. If communication is required, can use Main Thread handler or broadcast intents. Another case of use is when callbacks are needed (Intent triggered tasks).

How to trigger?

- The *Service* is triggered by calling method `startService()`.
- The *IntentService* is triggered using an Intent, it spawns a new worker thread and the method `onHandleIntent()` is called on this thread.

Triggered From

- The *Service* and *IntentService* may be triggered from any thread, activity or other application component.

Runs On

- The *Service* runs in background but it runs on the Main Thread of the application.
- The *IntentService* runs on a separate worker thread.

Limitations / Drawbacks

- The *Service* may block the Main Thread of the application.
- The *IntentService* cannot run tasks in parallel. Hence all the consecutive intents will go into the message queue for the worker thread and will execute sequentially.

When to stop?

- If you implement a *Service*, it is your responsibility to stop the service when its work is done, by calling `stopSelf()` or `stopService()`. (If you only want to provide binding, you don't need to implement this method).
- The *IntentService* stops the service after all start requests have been handled, so you never have to call `stopSelf()`.

`IntentService` does not start in separate process by default. It is a regular `Service` with the addition of logic that delegates execution of work to a single background thread and stops the service once all incoming `Intents` had been handled.

You want to do the following:

1. Perform data update from the web server
2. Notify once update completed
3. Get a reference to updated data from `Activity`

Your options are (from the top of my head):

1. Use standard android `Service` which can be both started and bound. Start this `Service` when update required, send notification when update completes and bind the `Service` from `Activity` in order to get the data.
2. Implement some in-memory cache which is neither tied to `IntentService`, nor to `Activity`. For example: implement `DataManager` class and instantiate it in `Application`. Both the `IntentService` and `Activity` can get a reference to `Application`, therefore they will be able to get a reference to `DataManager` object. `IntentService` will write the data into it, and `Activity` will read this data later
3. Implement data cache SQLite. This approach is very similar to #2 above, but data is not stored in memory, but written into SQLite database.

The easiest amongs the three options above is #2. Furthermore, if you go with #2 then it will be relatively easy to migrate to #3 if such a need arises (which is the best for "data heavy" apps).

Thursday, March 31, 2011

Android Thread Constructs(Part 4): Comparisons

In this series of posts we have seen the following thread constructs:
1. Basic threads and communication between them [see article]
2. Understanding the Main thread or the UI thread [see article]
3. IntentService [see article]
4. AsyncTask [see article]

NOTE: These are Android specific constructs. Android also includes the java.util.concurrent package which can be leveraged for concurrent tasks. That is out of the scope of my discussion.]

So, when to use what ? What is the exact purpose of each ? What are the drawbacks of using one over the other ? Which one is easier to program ?

I came across all these questions when trying to understand each of these. In due course of time, I have come up with some guidelines which I discuss in this post.

NOTE: Any of the recommendations made in this post and the table below are not comprehensive and final. There may be better and alternate ways of doing things. I would love to know any different views and get an insight into a different thought process. The views expressed in this post are my personal views.

Service
Before proceeding further, let me touch on the Service class. Traditionally speaking, the notion of service reminds us of task running in the background while we can work and interact with the UI. This causes confusion for newbies. Because in Android, even if the Service runs in the background, it runs on the Main Thread of the application. So, if at the same time if you have an activity displayed, the running service will take the main thread and the activity will seem slow. It is important to note that a Service is just a way of telling Android that something needs to run without a user interface in the background while the user may not interacting with your application. So, if you expect the user to be interacting with the application while the service is running and you have a long task to perform in a service, you need to create a worker thread in the Service to carry out the task.
So, even if Service is not a threading construct, its a way of executing a task at hand. Hence I have included it in the comparison.

The table below tries to summarize various aspects of four task execution mechanisms : Service, Thread, IntentService, AsyncTask.

Most of the points in the table are self-explanatory. However, some points that need an explanation are numbered and explained below the table. Also, this table is just to summarize about the concepts discussed in the previous posts. So, if anything is still unclear, I recommend to go through each individual posts in this series.

	Service	**Thread**	**IntentService**	**AsyncTask**
When to use ?	Task with no UI, but shouldn't be too long. Use threads within service for long tasks.	- Long task in general. - For tasks in parallel use Multiple threads (traditional mechanisms)	- Long task **usually** with no communication to main thread. **(Update)**- If communication is required, can use main thread handler or broadcast intents[3] - When callbacks are needed (Intent triggered tasks).	- Relatively long task (UI thread blocking) with a need to communicate with main thread.[3] - For tasks in parallel use multiple instances OR Executor [1]
Trigger	Call to method onStartService()	Thread start() method	Intent	Call to method execute()
Triggered From (thread)	Any thread	Any Thread	Main Thread (Intent is received on main thread and then worker thread is spawed)	Main Thread
Runs On (thread)	Main Thread	Its own thread	Separate worker thread	Worker thread. However, Main thread methods may be invoked in between to publish progress.
Limitations / Drawbacks	May block main thread	- Manual thread management - Code may become difficult to read	- Cannot run tasks in parallel. - Multiple intents are queued on the same worker thread.	- one instance can only be executed once (hence cannot run in a loop) [2] - Must be created and executed from the Main thread

[1] API Level 11 (Android 3.0) introduces the executeOnExecutor() method, that runs multiple tasks on a thread pool managed by AsyncTask. Below API Level 11, we need to create multiple instances of AsyncTask and call execute() on them in order to start parallel execution of multiple tasks.

[2] Once you create an object of an AsyncTask and call execute, you cannot call execute on that object again. Hence, trying to run an AsyncTask inside a loop will require you to each time create a new object in the loop before calling the execute on it.

[3] Recently I was brought to notice that AsyncTask is not recommended for long running tasks. While this is technically possible, I agree with the commenter. The idea here is any task which would potentially block the UI thread - I referred to this as a long task. May be I should have been clear. Thank you for pointing it out.

To be very precise, you cannot do something like :

```
TestAsyncTask myATask = new TestAsyncTask();
for (int i = 0; i < count; i++) {
    myATask.execute("one", "two", "three", "four");
}
```

But you can do :

```
for (int i = 0; i < count; i++) {
    TestAsyncTask myATask = new TestAsyncTask();
    myATask.execute("one", "two", "three", "four");
}
```

[3] Thanks to comment posted by Mark Murphy (@commonsguy) : "*there's nothing stopping an IntentService from communicating back to activities. A broadcast Intent is good for that, particularly with setPackage() to keep it constrained to your app (available on Android 2.1, IIRC). An ordered broadcast can be particularly useful, as you can rig it up such that if none of your activities are on-screen, that some manifest-registered BroadcastReceiver picks up the broadcast and raises a Notification.*"

Conclusion:
With a pretty understanding of the AsyncTask and the IntentService classes provided by Android, it would be a good idea to leverage these for most of the tasks as against trying to manage threads manually or implement your class as a Service unless there is really a need to do so.

Tricky Things:
There are some tricky scenarios that need some special handling. After all, we are dealing with a mobile environment and things are pretty unpredictable. For instance, the user opens your activity in which you start an AsyncTask, but before completion of the task, the user decides to exit the application (say by pressing the back button). Or your activity may be terminated (read kicked off) due to say an incoming call, when you are running a long task in the background. In such cases, where your application is not shutdown, but any foreground tasks have been closed or changed, all the background tasks need to know of this and try to exit gracefully.

Android Fundamentals: IntentService Basics

by Shane Conder & Lauren Darcey 14 Jun 2011

Difficulty: Intermediate Length: Medium Languages: English

Android SDK Mobile Development

This post is part of a series called Android Fundamentals.

Android Fundamentals: Database Dates and Sorting

Android Fundamentals: Picking App Components

Today we'd like to take a quick tour of one of the lesser known, but highly useful types of services you might want to use: the IntentService.

IntentService (android.app.IntentService) is a simple type of service that can be used to handle asynchronous work off the main thread by way of Intent requests. Each intent is added to the IntentService's queue and handled sequentially.

IntentService is one of the simplest ways to offload "chunks" of processing off the UI thread of your application and into a remote work queue. There's no need to launch an AsyncTask and manage it each and every time you have more processing. Instead, you simply define your own service, package up an intent with the appropriate data you want to send for processing, and start the service. You can send data back to the application by simply broadcasting the result as an Intent object, and using a broadcast receiver to catch the result and use it within the app.

Step 0: Getting Started

We have provided a sample application which illustrates the difference between trying to perform processing on the main UI thread (a no-no for application responsiveness) and offloading that same processing to an IntentService. The code can be downloaded via the code download link at the top of this tutorial.

Step 1: Defining the Processing of Messages

First, it's important to understand when and why to use services in general. One good reason to use an IntentService is when you have work that needs to occur off the main thread to keep the application responsive and efficient. Another reason is when you may have multiple processing requests, and they need to be queued up and handled on the fly.

So let's say we have an app that needs to do some "processing." We wanted something simple, so we basically will define processing in this case as taking in a string parameter, doing "stuff" to it, and returning the string result. To keep this tutorial simple, the "stuff" we will do is sleep for 30 seconds, pretending to do something useful. In reality, the "stuff" would likely be image processing, connecting to a network, or some other blocking operation.

Therefore, if the entire "processing" were to occur within the main application, you might have an EditText control for taking message input, and a TextView control for splitting out the result. You could place this code in a Button handler to trigger the processing. The code in the Button click handler within the app Activity class would look something like this:

```
EditText input = (EditText) findViewById(R.id.txt_input);
String strInputMsg = input.getText().toString();

SystemClock.sleep(30000); // 30 seconds, pretend to do work

TextView result = (TextView) findViewById(R.id.txt_result);
result.setText(strInputMsg + " " + DateFormat.format("MM/dd/yy h:mmaa", System.currentTimeMillis()));
```

The results are not ideal in any way. As soon as the user clicks the button, the entire application becomes unresponsive. The screen is frozen, the user cannot continue with their business, add any new messages. The user basically has to wait around for the processing to finish before they can do a thing.

Step 2: Implementing an IntentService

We would much rather have our processing not interfere with the application. We would also like to be able to add multiple message process requests easily. This is the perfect time to use an IntentService! So let's implement one.

Create another class file in your project and add stubs for the methods you need to implement. You should add a constructor with the name of your new service. You will need to implement just one other method called onHandleIntent(). This method is where your processing occurs. Any data necessary for each processing request can be packaged in the intent extras, like so (imports, comments, exception handling removed for code clarity, see the full source code for details):

```
public class SimpleIntentService extends IntentService {
    public static final String PARAM_IN_MSG = "imsg";
    public static final String PARAM_OUT_MSG = "omsg";

    public SimpleIntentService() {
        super("SimpleIntentService");
    }

    @Override
    protected void onHandleIntent(Intent intent) {

        String msg = intent.getStringExtra(PARAM_IN_MSG);
        SystemClock.sleep(30000); // 30 seconds
        String resultTxt = msg + " "
            + DateFormat.format("MM/dd/yy h:mmaa", System.currentTimeMillis());
    }
}
```

Note we also define the intent extra parameters for the incoming and outgoing intent data. We use the PARAM_IN_MSG extra for the incoming message data. We will soon use the PARAM_OUT_MSG extra to send results back to the main application.

Step 3: Launch the Service from your Application Activity

Next, you need to start the service from your application activity. Add a second Button control that, instead of doing the processing on the main thread, delegates the processing to your new service instead. The new button handler looks like this:

```
EditText input = (EditText) findViewById(R.id.txt_input);
String strInputMsg = input.getText().toString();
Intent msgIntent = new Intent(this, SimpleIntentService.class);
msgIntent.putExtra(SimpleIntentService.PARAM_IN_MSG, strInputMsg);
startService(msgIntent);
```

The code for the service method of processing is pretty straightforward. First, an Intent instance is generated and any data (like the message text) is packaged in the intent using the extras. Finally, the IntentService is started with a call to startService().

The service takes over from here, catching each intent request, processing it, and shutting itself down when it's all done. The main user interface remains responsive throughout the processing, allowing the user to continue to interact with the application. The user can input multiple messages, hit the button again and again, and each request is added to the service's work queue and handled. All in all, a better solution.

But we're not quite done yet.

Step 4: Define the Broadcast Receiver

Your IntentService can now do its job of processing, but we need it to inform the main application's activity when it's processed each request so that the UI can be updated. This only really matters when the activity is running, so we'll use a simple broadcast sender/receiver model.

Let's begin by defining a BroadcastReceiver subclass within the main activity.

```
public class ResponseReceiver extends BroadcastReceiver {
   public static final String ACTION_RESP =
      "com.mamlambo.intent.action.MESSAGE_PROCESSED";

   @Override
    public void onReceive(Context context, Intent intent) {
       TextView result = (TextView) findViewById(R.id.txt_result);
       String text = intent.getStringExtra(SimpleIntentService.PARAM_OUT_MSG);
       result.setText(text);
    }
}
```

The onReceive() method does all the work. It updates the activity's TextView control based upon the intent extra data packaged in the incoming result.

Note: If you were updating a database or shared preferences, there are alternate ways for the user interface to receive these changes. In these cases, the overhead of a broadcast receiver would not be necessary.

Step 5: Broadcast the Result

Next, you need to send a broadcast from the onHandleIntent() method of the IntentService class after the processing is complete and a result is available, like this:

```
// processing done here….
Intent broadcastIntent = new Intent();
broadcastIntent.setAction(ResponseReceiver.ACTION_RESP);
broadcastIntent.addCategory(Intent.CATEGORY_DEFAULT);
broadcastIntent.putExtra(PARAM_OUT_MSG, resultTxt);
sendBroadcast(broadcastIntent);
```

To send the result back to the main application, we package up another intent, stick the result data in as an extra, and blast it back using the sendBroadcast() method.

Step 7: Register the Broadcast Receiver

Finally, you must instantiate and register the receiver you've defined in your activity. Register the receiver in the onCreate() method with the appropriate intent filter to catch the specific result intent being sent from the IntentService.

```
public class IntentServiceBasicsActivity extends Activity {
    private ResponseReceiver receiver;

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

        IntentFilter filter = new IntentFilter(ResponseReceiver.ACTION_RESP);
        filter.addCategory(Intent.CATEGORY_DEFAULT);
        receiver = new ResponseReceiver();
        registerReceiver(receiver, filter);
    }
}
```

The result: each time the IntentService finishes processing a request, it fires off a broadcast with the result. The main activity can listen for the broadcast, and the onReceive() method of the broadcast receiver does the rest updating the UI accordingly. Don't forget to unregister the receiver in the proper time as well (onPause() is recommended).

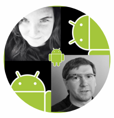
Conclusion

Offloading work from the main UI thread of an application to a work queue in an IntentService is an easy and efficient way to process multiple requests. Doing so keeps your main application responsive and your users happy. For many purposes, using an IntentService may be easier and more desirable than an AsyncTask, which is limited to a single execution, or a Thread, which has more coding overhead.

About the Authors

Mobile developers Lauren Darcey and Shane Conder have coauthored several books on Android development: an in-depth programming book entitled *Android Wireless Application Development, Second Edition* and *Sams Teach Yourself Android Application Development in 24 Hours, Second Edition*. When not writing, they spend their time developing mobile software at their company and providing consulting services. They can be reached at via email to androidwirelessdev+mt@gmail.com, via their blog at androidbook.blogspot.com, and on Twitter @androidwireless.

Need More Help Writing Android Apps? Check out our Latest Books and Resources!



Shane Conder & Lauren Darcey

Mobile developers Lauren Darcey and Shane Conder have coauthored numerous books on Android development. Our latest books include Sams Teach Yourself Android Application Development in 24 Hours (3rd Edition), Introduction to Android Application Development: Android Essentials (4th Edition), and Advanced Android Application Development (4th Edition). Lauren and Shane run a boutique consulting firm specializing in the development of commercial-grade Android applications for smartphones, tablets, wearables (i.e. Google Glass), and more. They can be reached on Google+, their blog at androidbook.blogspot.com and on Twitter @androidwireless.