Module 2

App components

App components are the essential building blocks of an Android app. Each component is an entry point through which the system or a user can enter your app. Some components depend on others.
There are four different types of app components:

- Activities
- Services
- Broadcast receivers
- Content providers

Each type serves a distinct purpose and has a distinct lifecycle that defines how the component is created and destroyed.

Activities

An *activity* is the entry point for interacting with the user. It represents a single screen with a user interface. For example, an email app might have one activity that shows a list of new emails, another activity to compose an email, and another activity for reading emails. Although the activities work together to form a cohesive user experience in the email app, each one is independent of the others. As such, a different app can start any one of these activities if the email app allows it. For example, a camera app can start the activity in the email app that composes new mail to allow the user to share a picture. An activity facilitates the following key interactions between system and app:

- Keeping track of what the user currently cares about (what is on screen) to ensure that the system keeps running the process that is hosting the activity.
- Knowing that previously used processes contain things the user may return to (stopped activities), and thus more highly prioritize keeping those processes around.
- Helping the app handle having its process killed so the user can return to activities with their previous state restored.
- Providing a way for apps to implement user flows between each other, and for the system to coordinate these flows. (The most classic example here being share.)

Services

A *service* is a general-purpose entry point for keeping an app running in the background for all kinds of reasons. It is a component that runs in the background to perform long-running operations or to perform work for remote processes. A service does not provide a user interface.

For example, a service might play music in the background while the user is in a different app, or it might fetch data over the network without blocking user interaction with an activity. Another component, such as an activity, can start the service and let it run or bind to it in order to interact with it.

A service can take two forms:
· **Started (Unbounded) :** After a service starts, it can run indefinitely and usually performs single operation. No result is returned to user. For example, uploading a file. After the task is completed, it should terminate itself.
· **Bound:** In this case, a component is bound to a service so that a particular task can be completed. This type of service provides a client-server like interface. Requests can be send, receive requests, and return result to the user. Inter process communication is achieved through this service. App component can bind to a service. Multiple components can be bound to this type of service. After the destruction of component, service terminates.

A service is implemented as a subclass of Service.

```
public class ServiceName extends Service {
}
```

Broadcast receivers

A *broadcast receiver* is a component that enables the system to deliver events to the app outside of a regular user flow, allowing the app to respond to system-wide broadcast announcements. Because broadcast receivers are another well-defined entry into the app, the system can deliver broadcasts even to apps that aren't currently running. So, for example, an app can schedule an alarm to post a notification to tell the user about an upcoming event... and by delivering that alarm to a BroadcastReceiver of the app, there is no need for the app to remain running until the alarm goes off. 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. Apps can also initiate broadcasts—for example, to let other apps 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. A broadcast receiver is implemented as a subclass of BroadcastReceiver and each broadcast is delivered as an Intent object.

Content providers

A *content provider* manages a shared set of app data that you can store in the file system, in a SQLite database, on the web, or on any other persistent storage location that your app can access. Through the content provider, other apps can query or modify the data if the content

provider allows it. For example, the Android system provides a content provider that manages the user's contact information. As such, any app with the proper permissions can query the content provider, such as ContactsContract.Data, to read and write information about a particular person.
Content providers are also useful for reading and writing data that is private to your app and not shared. For example, the Note Pad sample app uses a content provider to save notes.
A content provider is implemented as a subclass of ContentProvider and must implement a standard set of APIs that enable other apps to perform transactions.

Activating components

Three of the four component types—activities, services, and broadcast receivers—are activated by an asynchronous message called an *intent*. Intents bind individual components to each other at runtime. You can think of them as the messengers that request an action from other components, whether the component belongs to your app or another.

Activity

An activity represents a single screen in your app with an interface the user can interact with. An application can have zero or more activities. Typically, applications have one or more activities, and the main aim of an activity is to interact with the user. For example, an email app might have one activity that shows a list of new emails, another activity to compose an email, and another activity for reading individual messages. Your app is a collection of activities that you either create yourself, or that you reuse from other apps.

Although the activities in your app work together to form a cohesive user experience in your app, each one is independent of the others. This enables your app to start activities in other apps, and other apps can start your activities (if your app allows it). For example, a messaging app you write could start an activity in a camera app to take a picture, and then start the activity in an email app to let the user share that picture in email.

Typically, one activity in an app is specified as the "main" activity, which is presented to the user when launching the application for the first time. Each activity can then start other activities in order to perform different actions.

Each time a new activity starts, the previous activity is stopped, but the system preserves the activity in a stack (the "back stack"). When the user is done with the current activity and presses the Back button, it is popped from the stack (and destroyed) and the previous activity resumes.

To implement an activity in your app, do the following:

- Create an activity Java class.
- Implement a user interface for that activity.
- Declare that new activity in the app manifest.

To create an activity, you create a Java class that extends the Activity base class:

```
packagenet.learn2develop.Activities;
import android.app.Activity;
import android.os.Bundle;
public class MainActivity extends Activity {
        /** Called when the activity is first created. */
        @Override
        public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);
        }
}
```

Activity class would then load its UI component using the XML file defined in res/layout folder. In this example, you would load the UI from the main.xml file.

```
        setContentView(R.layout.main);
```

Each activity in your app must be declared in the Android app manifest with the <activity> element, inside <application>. When you create a new project or add a new activity to your project in Android Studio, your manifest is created or updated to include skeleton activity declarations for each activity. Here's the declaration for the main activity.

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

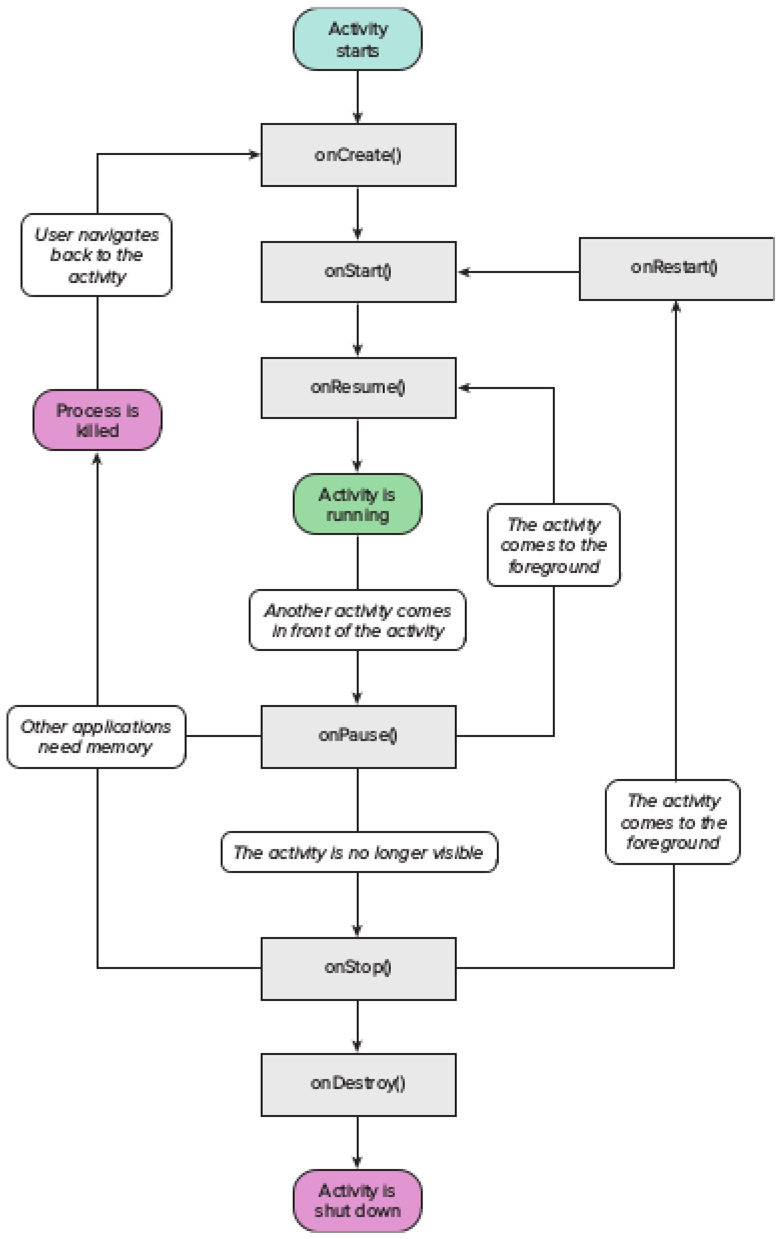
The <activity> element includes a number of attributes to define properties of the activity such as its label, icon, or theme. The only required attribute is android:name, which specifies the class name for the activity (such as "MainActivity"). The <activity> element can also include declarations for intent filters. The intent filters specify the kind of intents your activity will accept.

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

Intent filters must include at least one <action> element, and can also include a <category> and optional <data>. The main activity for your app needs an intent filter that defines the "main" action and the "launcher" category so that the system can launch your app. The <action> element specifies that this is the "main" entry point to the application. The <category> element specifies that this activity should be listed in the system's application launcher.Other activities in your app can also declare intent filters, but only your main activity should include the "main" action

Activity Lifecycle

From the moment an activity appears on the screen to the moment it is hidden, it goes through a number of stages, known as an activity's life cycle. Understanding the life cycle of an activity is vital to ensuring that your application works correctly. The Activity base class defines a series of events that governs the life cycle of an activity. The Activity class defines the following events:
➢ onCreate() — Called when the activity is first created
➢ onStart() — Called when the activity becomes visible to the user
➢ onResume() — Called when the activity starts interacting with the user
➢ onPause() — Called when the current activity is being paused and the previous activity is being resumed
➢ onStop() — Called when the activity is no longer visible to the user
➢ onDestroy() — Called before the activity is destroyed by the system (either manually or by the system to conserve memory)
➢ onRestart() — Called when the activity has been stopped and is restarting again
By default, the activity created for you contains the onCreate() event. Within this event handler is the code that helps to display the UI elements of your screen.



Activity starts
onCreate()
User navigates back to the activity
onStart()
onRestart()
onResume()
Process is killed
Activity is running
The activity comes to the foreground
Another activity comes in front of the activity
Other applications need memory
onPause()
The activity comes to the foreground
The activity is no longer visible
onStop()
onDestroy()
Activity is shut down

onCreate()

We must implement this callback, which fires when the system creates activity. In this implementation should initialize the essential components of activity: For example, app should create views and bind data to lists here. Most importantly, this is where you must call setContentView() to define the layout for the activity's user interface. When onCreate() finishes, the next callback is always onStart().

onStart()

As onCreate() exits, the activity enters the Started state, and the activity becomes visible to the user. This callback contains what amounts to the activity’s final preparations for coming to the foreground and becoming interactive.

onResume()

The system invokes this callback just before the activity starts interacting with the user. At this point, the activity is at the top of the activity stack, and captures all user input. Most of an app’s core functionality is implemented in the onResume() method.
The onPause() callback always follows onResume().

onPause()

The system calls onPause() when the activity loses focus and enters a Paused state. This state occurs when, for example, the user taps the Back or Recents button. When the system calls onPause() for your activity, it technically means your activity is still partially visible, but most often is an indication that the user is leaving the activity, and the activity will soon enter the Stopped or Resumed state.
Once onPause() finishes executing, the next callback is either onStop() or onResume(), depending on what happens after the activity enters the Paused state.

onStop()

The system calls onStop() when the activity is no longer visible to the user. This may happen because the activity is being destroyed, a new activity is starting, or an existing activity is entering a Resumed state and is covering the stopped activity. In all of these cases, the stopped activity is no longer visible at all.
The next callback that the system calls is either onRestart(), if the activity is coming back to interact with the user, or by onDestroy() if this activity is completely terminating.

onRestart()

The system invokes this callback when an activity in the Stopped state is about to restart. onRestart() restores the state of the activity from the time that it was stopped.
This callback is always followed by onStart().

onDestroy()

The system invokes this callback before an activity is destroyed.
This callback is the final one that the activity receives. onDestroy() is usually implemented to ensure that all of an activity’s resources are released when the activity, or the process containing it, is destroyed.

About intents

All Android activities are started or activated with an *intent*. Intents are message objects that make a request to the Android runtime to start an activity or other app component in your app or in some other app. An intent is basically the “glue” that enables different activities from different applications to work together seamlessly, ensuring that tasks can be performed as though they all belong to one single application.
When your app is first started from the device home screen, the Android runtime sends an intent to your app to start your app's main activity (the one defined with the MAIN action and the LAUNCHER category in the Android Manifest). To start other activities in your app, or request that actions be performed by some other activity available on the device, you build your own intents with the Intent class and call the startActivity() method to send that intent.
In addition to starting activities, intents are also used to pass data between activities. When you create an intent to start a new activity, you can include information about the data you want that new activity to operate on. So, for example, an email activity that displays a list of messages can send an intent to the activity that displays that message.

Intent types

There are two types of intents in Android:

- *Explicit intents* specify the receiving activity (or other component) by that activity's fully-qualified class name. Use an explicit intent to start a component in your own app because you already know the package and class name of that component.
- *Implicit intents* do not specify a specific activity or other component to receive the intent. Instead you declare a general action to perform in the intent. The Android system matches your request to an activity or other component that can handle your requested action.

Intent objects and fields

An Intent object is an instance of the Intent class. For explicit intents, the key fields of an intent include the following:

- The activity *class* (for explicit intents). This is the class name of the activity or other component that should receive the intent
- The intent *data*. The intent data field contains a reference to the data you want the receiving activity to operate on, as a Uri object.
- Intent *extras*. These are key-value pairs that carry information the receiving activity requires to accomplish the requested action.

- Intent *flags*. These are additional bits of metadata, defined by the Intent class. The flags may instruct the Android system how to launch an activity or how to treat it after it's launched.

Starting an activity with an explicit intent

To start a specific activity from another activity, use an explicit intent and the startActivity() method. Explicit intents include the fully-qualified class name for the activity or other component in the Intent object. All the other intent fields are optional, and null by default.

For example, if you wanted to start the ShowMessageActivity to show a specific message in an email app, use code like this.

```
Intent messageIntent = new Intent(this, ShowMessageActivity.class);
startActivity(messageIntent);
```

The Intent constructor takes two arguments for an explicit intent.

- An application context. In this example, the activity class provides the content (here, this).
- The specific component to start (ShowMessageActivity.class).

Use the startActivity() method with the new intent object as the only argument. The startActivity() method sends the intent to the Android system, which launches the ShowMessageActivity class on behalf of your app. The new activity appears on the screen, and the originating activity is paused.

The started activity remains on the screen until the user taps the back button on the device, at which time that activity closes and is reclaimed by the system, and the originating activity is resumed. You can also manually close the started activity in response to a user action (such as a button click) with the finish() method:

```
public void closeActivity (View view) {
   finish();
}
```

Passing data between activities with intents

In addition to simply starting one activity from another, you also use intents to pass information between activities. The intent object you use to start an activity can include intent *data* (the URI of an object to act on), or intent *extras*, which are bits of additional data the activity might need.

In the first (sending) activity, you:

1. Create the Intent object.

2. Put data or extras into that intent.
3. Start the new activity with startActivity().

In the second (receiving) activity, you:

1. Get the intent object the activity was started with.
2. Retrieve the data or extras from the Intent object.

When to use intent data or intent extras

You can use either intent data and intent extras to pass data between the activities. The intent *data* can hold only one piece of information. A URI representing the location of the data you want to operate on. That URI could be a web page URL (http://), a telephone number (tel://), a geographic location (geo://) or any other custom URI you define.

Use the intent data field:

- When you only have one piece of information you need to send to the started activity.
- When that information is a data location that can be represented by a URI.

Intent *extras* are for any other arbitrary data you want to pass to the started activity. Intent extras are stored in a Bundle object as key and value pairs. Bundles are a map, optimized for Android, where the keys are strings, and the values can be any primitive or object type. When we use objects it must implement the Parcelable interface. To put data into the intent extras you can use any of the Intent class's putExtra() methods, or create your own bundle and put it into the intent with putExtras().
Use the intent extras:

- If you want to pass more than one piece of information to the started activity.
- If any of the information you want to pass is not expressible by a URI.

To add data to an explicit intent from the originating activity, create the intent object as you did before:
Intent messageIntent = new Intent(this, ShowMessageActivity.class);

Use the setData() method with a Uri object to add that URI to the intent. Some examples of using setData() with URIs:
// A web page URL
messageIntent.setData(Uri.parse("http://www.google.com"));
// a Sample file URI
messageIntent.setData(Uri.fromFile(new File("/sdcard/sample.jpg")));
if you call setData() multiple times only the last value is used. Use intent extras to include additional information.
After you've added the data, you can start the activity with the intent as usual.

startActivity(messageIntent);
To add intent extras to an explicit intent from the originating activity:

1. Determine the keys to use for the information you want to put into the extras, or define your own. Each piece of information needs its own unique key.
2. Use the putExtra() methods to add your key/value pairs to the intent extras. Optionally you can create a Bundle object, add your data to the bundle, and then add the bundle to the intent.

Create an intent object
Intent messageIntent = new Intent(this, ShowMessageActivity.class);

Use a putExtra() method with a key to put data into the intent extras. The Intent class defines many putExtra() methods for different kinds of data:
messageIntent.putExtra(“msg”, "this is my message");
messageIntent.putExtra(“x” 100);
messageIntent.putExtra(“y”, 500);

Alternately, you can create a new bundle and populate that bundle with your intent extras. Bundle defines many "put" methods for different kinds of primitive data as well as objects that implement Android's Parcelable interface.
Bundle extras = new Bundle();
extras.putString(“msg”, "this is my message");
extras.putInt(“x”, 100);
extras.putInt(“y”, 500);

After you've populated the bundle, add it to the intent with the putExtras() method.
messageIntent.putExtras(extras);
Start the activity with the intent as usual:
startActivity(messageIntent);

Retrieve the data from the intent in the started activity

When you start an activity with an intent, the started activity has access to the intent and the data it contains.
To retrieve the intent the activity (or other component) was started with, use the getIntent() method:
Intent intent = getIntent();

Use getData() to get the URI from that intent:
Uri locationUri = getData();

To get the extras out of the intent, you'll need to know the keys for the key/value pairs. You can use the standard Intent extras if you used those, or you can use the keys you defined in the originating activity (if they were defined as public.)
Use one of the getExtra() methods to extract extra data out of the intent object:

```
String message = intent.getStringExtra(“msg”);
int positionX = intent.getIntExtra(“x”);
int positionY = intent.getIntExtra(“y”);
```

Or you can get the entire extras bundle from the intent and extract the values with the various Bundle methods:

```
Bundle extras = intent.getExtras();
String message = extras.getString(“msg”);
```

Getting data back from an activity

When you start an activity with an intent, the originating activity is paused, and the new activity remains on the screen until the user clicks the back button, or you call the finish() method in a click handler or other function that ends the user's involvement with this activity.
Sometimes when you send data to an activity with an intent, you would like to also get data back from that intent. For example, you might start a photo gallery activity that lets the user pick a photo. In this case your original activity needs to receive information about the photo the user chose back from the launched activity.
To launch a new activity and get a result back, do the following steps in your originating activity:

1. Instead of launching the activity with startActivity(), call startActivityForResult() with the intent and a request code.
2. Create a new intent in the launched activity and add the return data to that intent.
3. Implement onActivityResult() in the originating activity to process the returned data.

To get data back from a launched activity, start that activity with the startActivityForResult() method instead of startActivity().

```
startActivityForResult(messageIntent, TEXT_REQUEST);
```

The startActivityForResult() method, like startActivity(), takes an intent argument that contains information about the activity to be launched and any data to send to that activity. The startActivityForResult() method, however, also needs a request code.
The request code is an integer that identifies the request and can be used to differentiate between results when you process the return data. For example, if you launch one activity to take a photo and another to pick a photo from a gallery, you'll need different request codes to identify which request the returned data belongs to.
Conventionally you define request codes as static integer variables with names that include REQUEST. Use a different integer for each code. For example:

```
public static final int PHOTO_REQUEST = 1;
public static final int PHOTO_PICK_REQUEST = 2;
public static final int TEXT_REQUEST = 3;
```

Return a response from the launched activity

The response data from the launched activity back to the originating activity is sent in an intent. You construct this return intent and put the data into it in much the same way you do for the sending intent. Typically your launched activity will have an onClick or other user input callback method in which you process the user's action and close the activity. This is also where you construct the response.
To return data from the launched activity, create a new empty intent object.

```
Intent returnIntent = new Intent();
```

Return result intents do not need a class or component name to end up in the right place. The Android system directs the response back to the originating activity for you.
Add data or extras to the intent the same way you did with the original intent. You may need to define keys for the return intent extras at the start of your class.

```
public final static String EXTRA_RETURN_MESSAGE =
   "com.example.mysampleapp.RETURN_MESSAGE";
```

Then put your return data into the intent as usual. Here the return message is an intent extra with the key EXTRA_RETURN_MESSAGE.

```
messageIntent.putExtra(EXTRA_RETURN_MESSAGE, mMessage);
```

Use the setResult() method with a response code and the intent with the response data:

```
setResult(RESULT_OK,replyIntent);
```

The response codes are defined by the Activity class, and can be

- RESULT_OK. the request was successful.
- RESULT_CANCELED: the user cancelled the operation.
- RESULT_FIRST_USER. for defining your own result codes.

You'll use the result code in the originating activity.
Finally, call finish() to close the activity and resume the originating activity:

```
finish();
```

Read response data in onActivityResult()

Now that the launched activity has sent data back to the originating activity with an intent, that first activity must handle that data. To handle returned data in the originating activity, implement the onActivityResult() callback method. Here is a simple example.

```
public void onActivityResult(int requestCode, int resultCode,  Intent data) {
   super.onActivityResult(requestCode, resultCode, data);
```

```
    if (requestCode == TEXT_REQUEST) {
        if (resultCode == RESULT_OK) {
            String reply =
                data.getStringExtra(SecondActivity.EXTRA_RETURN_MESSAGE);
                // process data
        }
    }
}
```

The three arguments to the onActivityResult() contain all the information you need to handle the return data.

- **Request code.** The request code you set when you launched the activity with startActivityForResult(). If you launch different activities to accomplish different operations, use this code to identify the specific data you're getting back.
- **Result code:** the result code set in the launched activity, usually one of RESULT_OK or RESULT_CANCELED.
- **Intent data**. the intent that contains the data returned from the launch activity.

The example method shown above shows the typical logic for handling the request and response codes. The first test is for the TEXT_REQUEST request, and that the result was successful. Inside the body of those tests you extract the return information out of the intent. Use getData() to get the intent data, or getExtra() to retrieve values out of the intent extras with a specific key.

About implicit intents

A more flexible use of intents is the *implicit intent*. With implicit intents you do not specify the exact activity (or other component) to run—instead, you include just enough information in the intent about the task you want to perform. The Android system matches the information in your request intent with activities available on the device that can perform that task. If there's only one activity that matches, that activity is launched. If there are multiple matching activities, the user is presented with an app chooser that enables them to pick which app they would like to perform the task.
For example, you have an app that lists available snippets of video. If the user touches an item in the list, you want to play that video snippet. Rather than implementing an entire video player in your own app, you can launch an intent that specifies the task as "play an object of type video." The Android system then matches your request with an activity that has registered itself to play objects of type video. Activities register themselves with the system as being able to handle implicit intents with intent *filters*, declared in the Android manifest.

Intent actions, categories, and data

Implicit intents, like explicit intents, are instances of the Intent class.

- The intent *action*, which is the generic action the receiving activity should perform. The available intent actions are defined as constants in the Intent class and begin with the word ACTION_. A common intent action is ACTION_VIEW, which you use when you have some information that an activity can show to the user, such as a photo to view in a gallery app, or an address to view in a map app. You can specify the action for an intent in the intent constructor, or with the setAction() method.
- An intent *category*, which provides additional information about the category of component that should handle the intent. Intent categories are optional, and you can add more than one category to an intent. Intent categories are also defined as constants in the Intent class and begin with the word CATEGORY_. You can add categories to the intent with the addCategory() method.
- The data *type*, which indicates the MIME type of data the activity should operate on. Usually, this is inferred from the URI in the intent data field, but you can also explicitly define the data type with the setType() method.

Sending implicit intents

Starting activities with implicit intents, and passing data between those activities, works much the same way as it does for explicit intents:

1. In the sending activity, create a new Intent object.
2. Add information about the request to the Intent object, such as data or extras.
3. Send the intent with startActivity() or startActivityforResult()

Once you have an Intent object you can add other information (category, data, extras) with the various Intent methods. For example, this code creates an implicit Intent object, sets the intent action to ACTION_SEND, defines an intent extra to hold the text, and sets the type of the data to the MIME type "text/plain".

```
Intent sendIntent = new Intent();
sendIntent.setAction(Intent.ACTION_SEND);
sendIntent.putExtra(Intent.EXTRA_TEXT, textMessage);
sendIntent.setType("text/plain");
```

Intent filters

Define intent filters with one or more <intent-filter> elements in the app's manifest file, nested in the corresponding <activity> element. Inside <intent-filter>, specify the type of intents your activity can handle. The Android system matches an implicit intent with an activity or other app component only if the fields in the Intent object match the intent filters for that component.

An intent filter may contain these elements, which correspond to the fields in the Intent object described above:

- <action>: The intent action.
- <data>: The type of data accepted, including the MIME type or other attributes of the data URI (such as scheme, host, port, path, and so no).
- <category>: The intent category.

example for an implicit intent to share a bit of text. This intent filter matches the implicit intent example from the previous section:

```
<activity android:name="ShareActivity">
   <intent-filter>
      <action android:name="android.intent.action.SEND"/>
      <category android:name="android.intent.category.DEFAULT"/>
      <data android:mimeType="text/plain"/>
   </intent-filter>
</activity>
```

Actions

An intent filter can declare zero or more <action> elements for the intent action. The action is defined in the name attribute, and consists of the string "android.intent.action." plus the name of the intent action, minus the ACTION_ prefix.

For example, this intent filter matches either ACTION_EDIT and ACTION_VIEW:

```
<intent-filter>
   <action android:name="android.intent.action.EDIT" />
   <action android:name="android.intent.action.VIEW" />
   ...
</intent-filter>
```

You must include at least one intent action for an incoming implicit intent to match.

Categories

An intent filter can declare zero or more <category> elements for intent categories. The category is defined in the name attribute, and consists of the string "android.intent.category." plus the name of the intent category, minus the CATEGORY prefix.

For example, this intent filter matches either CATEGORY_DEFAULT and CATEGORY_BROWSABLE:

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

Note that all activities that you want to accept implicit intents must include the android.intent.category.DEFAULT intent-filter. This category is applied to all implicit Intent objects by the Android system.

Data

An intent filter can declare zero or more <data> elements for the URI contained in the intent data. As the intent data consists of a URI and (optionally) a MIME type, you can create an intent filter for various aspects of that data, including:

- URI Scheme
- URI Host
- URI Path
- Mime type

For example, this intent filter matches data intents with a URI scheme of http and a MIME type of either "video/mpeg" or "audio/mpeg".

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

Intent flags

Intent flags are options that specify how the activity (or other app component) that receives the intent should handle that intent. Intent flags are defined as constants in the Intent class and begin with the word FLAG_. You add intent flags to an Intent object with setFlag() or addFlag().

- FLAG_ACTIVITY_NEW_TASK: start the activity in a new task. This is the same behavior as the singleTask launch mode.
- FLAG_ACTIVITY_SINGLE_TOP: if the activity to be launched is at the top of the back stack, route the intent to that existing activity instance. Otherwise create a new activity instance. This is the same behavior as the singleTop launch mode.
- FLAG_ACTIVITY_CLEAR_TOP: If an instance of the activity to be launched already exists in the back stack, destroy any other activities on top of it and route the intent to that existing instance. When used in conjunction with FLAG_ACTIVITY_NEW_TASK, this flag locates any existing instances of the activity in any task and brings it to the foreground.