SESSION #3

GUI (II)

OUTLINE

- Activities
- Activities Lifecycle
- Intents. Managing multiple activities
- HelloIntents

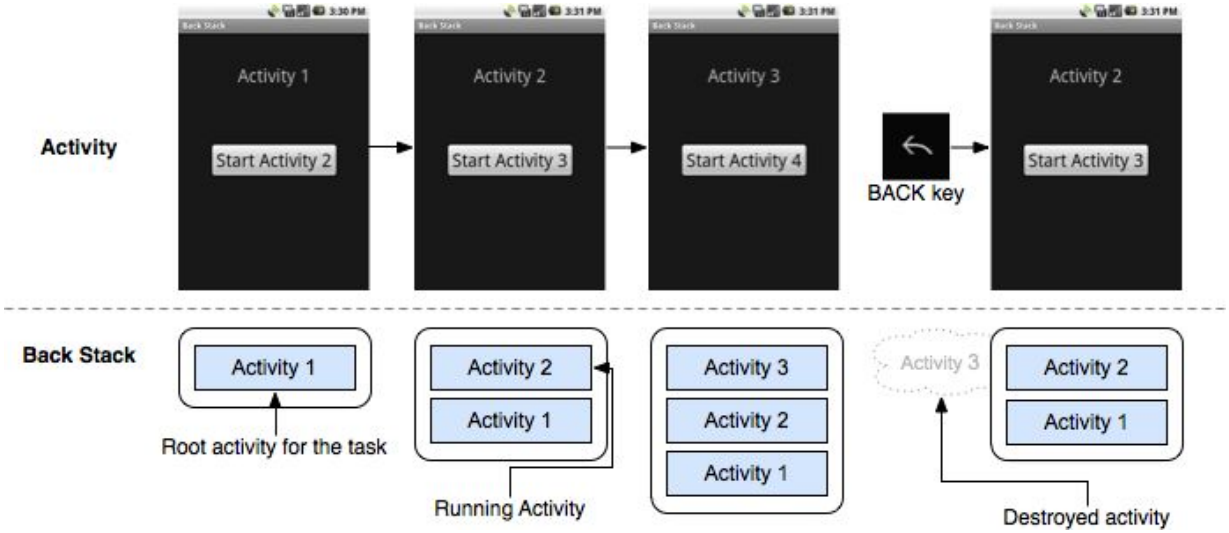
ACTIVITY

- An Activity is an application component that provides a screen with which users can interact in order to do something.
  - Each activity is given a window in which to draw its user interface.

- An application might consist of just one activity or it main contain multiple activities.
  - One activity in an application is specified as the "main" activity
  - But activities can start another, however, each activity is independent of the others

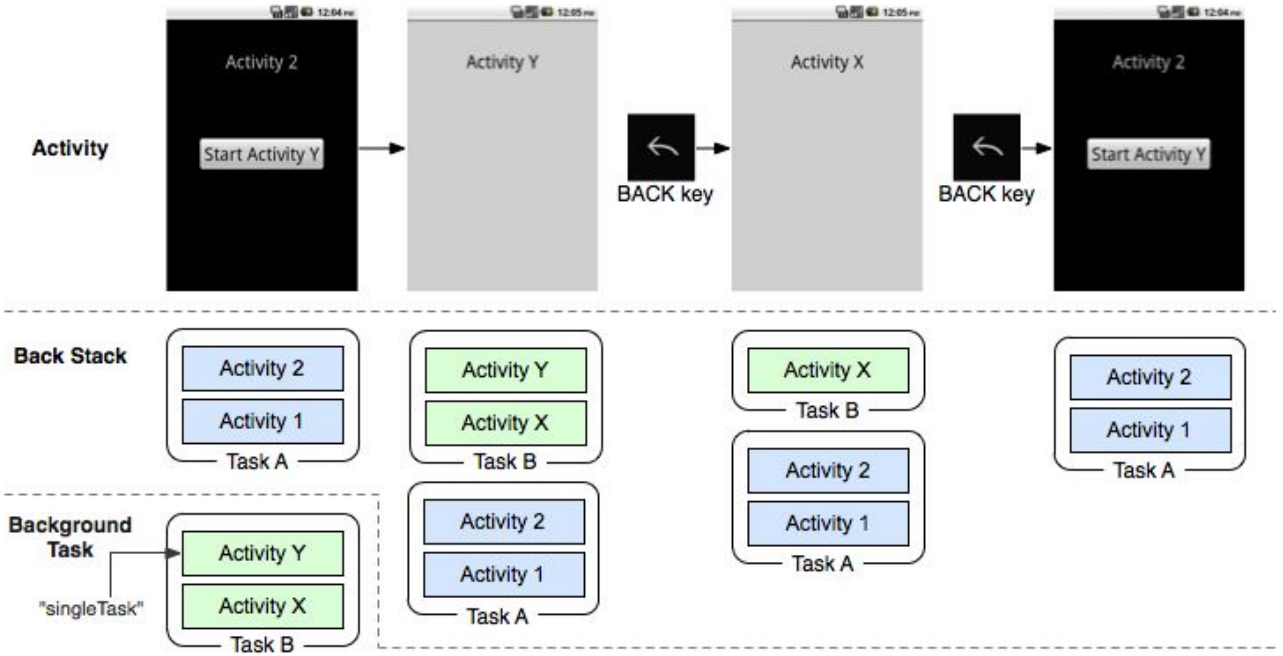
BACK STACK

- The activities are arranged in a stack (the "back stack"), in the order in which each activity is opened.
  - Each time a new activity starts, the previous activity is stopped, but the system preserves the activity in the "back stack“.
- The back stack abides to the basic "last in, first out" queue mechanism:
  - When a new activity starts, it is pushed onto the back stack and takes user focus.
  - When the current activity is done , it is popped from the stack (and destroyed) and the previous activity resumes



TASKS

- A task is a collection of activities that users interact with when performing a certain job.
- When a task go to the background, all the activities in the task are stopped, but the back stack for the task remains intact.
  - The system might destroy background activities in order to recover memory.



ACTIVITY

LIFECYCLE

An activity can exist in essentially three states:

- *Running/Resumed.*

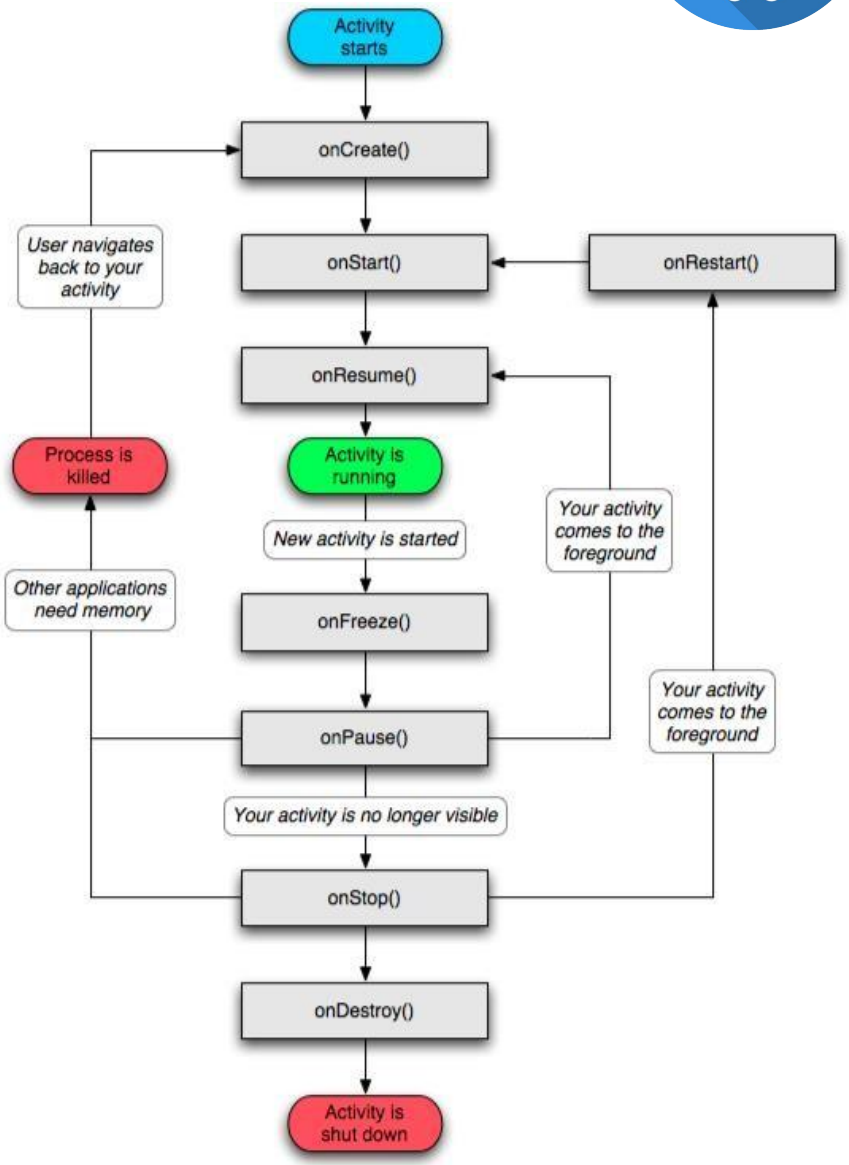
  The activity is in the foreground of the screen and has user focus.

- *Paused* .

  Another activity is in the foreground and has focus, but this one is still visible.

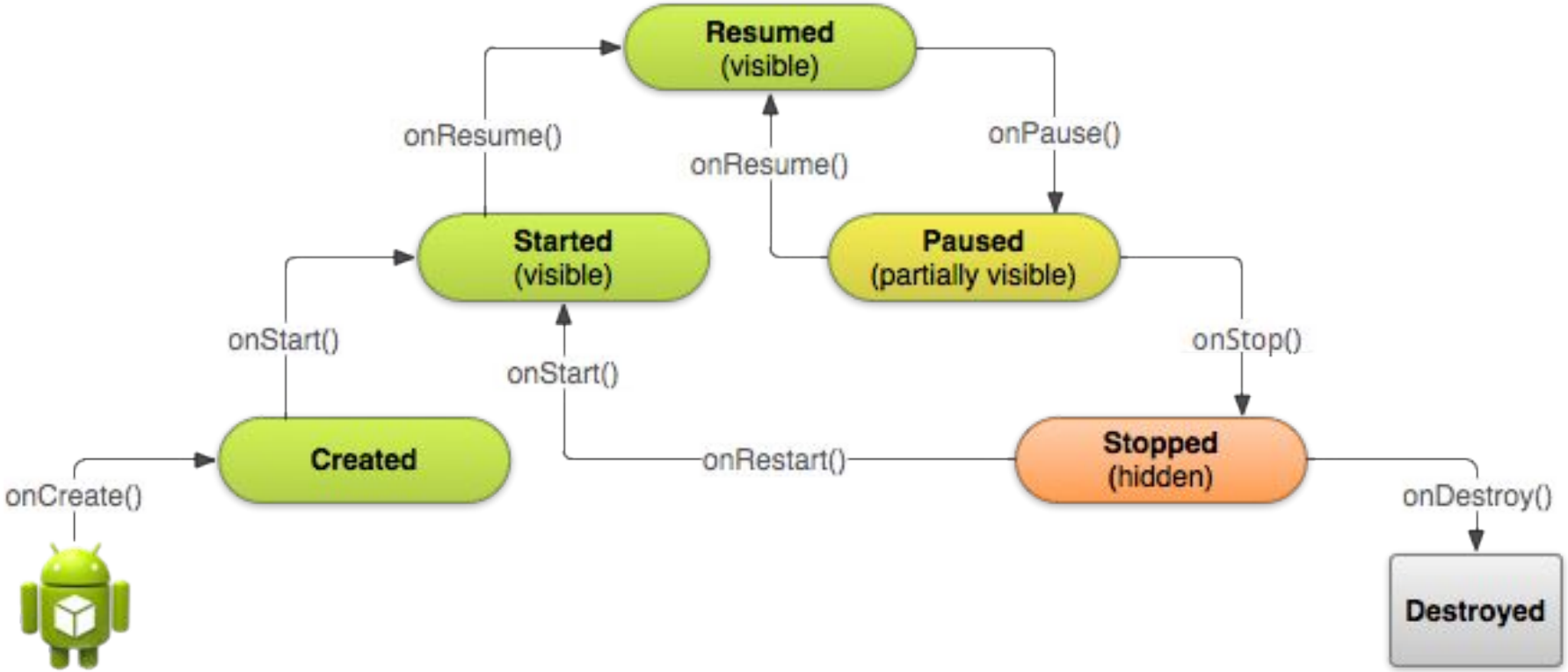
- *Stopped* .

  The activity is completely obscured by another activity (the activity is now in the "background").



ACTIVITY

LIFECYCLE



LIFECYCLE

✧ *Static* states: the activity can exist in one of only three states for an extended period of time:

- **Resumed:** the activity is in the foreground and the user can interact with it. (Also called as the "running" state.)
- **Paused:** the activity is partially obscured by another activity. The paused activity does not receive user input and cannot execute any code.
- **Stopped:** the activity is completely hidden and not visible to the user; it is considered to be in the background. While stopped, the activity instance and all its state information such as member variables is retained, but it cannot execute any code.

✧ *Transient* states: the system quickly moves from them to the next state:

- **Created:** the activity is first created, it is doing the static set up
- **Started:** the activity is created and it is going to begin its execution.

STARTING (I)

✧ The Android system initiates code in an Activity by invoking specific callback methods (*onCreate*, *onStart*, *onResume*).

✧ When the user selects your app icon from the Home screen, the system calls the *onCreate()* method for the Activity in your app that you've declared to be the "**launcher**" (or "main") activity.

- This is the activity that serves as the main entry point to your app's user interface.
- The main activity for your app must be declared in the manifest with an <intent-filter> that includes the MAIN action and LAUNCHER category:

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

- If either the MAIN action or LAUNCHER category are not declared for one of your activities, then your app icon will not appear in the Home screen's list of apps.

STARTING (II)

- The system creates every new instance of an Activity calling *onCreate()*
  - You must implement the *onCreate()* method to perform basic application startup logic that should happen only once for the entire life of the activity:
    - Declare the user interface (defined in an XML layout file) and configuring some of the UI.
    - Defining member variables.

```
TextView mTextView; // Member variable for text view in the layout

@Override
public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);

    // Set the user interface layout for this Activity
    // The layout file is defined in the project res/layout/main_activity.xml file
    setContentView(R.layout.main_activity);

    // Initialize member TextView so we can manipulate it later
    mTextView = (TextView) findViewById(R.id.text_message);
}
```

- Once the *onCreate()* finishes execution, the system calls the *onStart()* and *onResume()* methods in quick succession.

DESTROYING

The system calls *onDestroy()* method on your activity as the final signal that your activity instance is being completely removed from the system memory.

```
@Override
public void onDestroy() {
    super.onDestroy();  // Always call the superclass

    // Stop method tracing that the activity started during onCreate()
    android.os.Debug.stopMethodTracing();
}
```

PAUSING

- When an activity lost the user focus but it is still visible, the system pauses de activity.
- As your activity enters the paused state, the system calls the *onPause()* method, which allows you to stop ongoing actions that should not continue while paused:
  - Stop animations or other ongoing actions that could consume CPU.
  - Commit unsaved changes, but only if users expect such changes to be permanently saved when they leave.
  - Release system resources or any resources that may affect battery.

```
@Override
public void onPause() {
    super.onPause();  // Always call the superclass method first

    // Release the Camera because we don't need it when paused
    // and other activities might need to use it.
    if (mCamera != null) {
        mCamera.release()
        mCamera = null;
    }
}
```

RESUMING

✧ If the user returns to your activity from the paused state, the system resumes it and calls the *onResume()* method.

```
@Override
public void onResume() {
    super.onResume();  // Always call the superclass method first

    // Get the Camera instance as the activity achieves full user focus
    if (mCamera == null) {
        initializeCamera(); // Local method to handle camera init
    }
}
```

- Be aware that the system calls this method every time your activity comes into the foreground, including when it's created for the first time.

STOPPING

- When your activity stops it's no longer visible and should release almost all resources that aren't needed while the user is not using it.
  - Once your activity is stopped, the system might destroy the instance if it needs to recover system memory.
  - In extreme cases, the system might simply kill your app process without calling the activity's final *onDestroy()* callback.

- Previous to stop an Activity, it receives a call to the *onStop()* method.
  - You should use *onStop()* to perform larger, more CPU intensive shut-down operations.
- You can stop one activity invoking the method *finish()*.

```
@Override
protected void onStop() {
    super.onStop();  // Always call the superclass method first

    // Save the note's current draft, because the activity is stopping
    // and we want to be sure the current note progress isn't lost.
    ContentValues values = new ContentValues();
    values.put(NotePad.Notes.COLUMN_NAME_NOTE, getCurrentNoteText());
    values.put(NotePad.Notes.COLUMN_NAME_TITLE, getCurrentNoteTitle());

    getContentResolver().update(
            mUri,    // The URI for the note to update.
            values,  // The map of column names and new values to apply to them.
            null,    // No SELECT criteria are used.
            null     // No WHERE columns are used.
            );
}
```

START/RESTART

- When your activity comes back to the foreground from the stopped state, it receives a call to *onRestart()* method.
  - You can use it to perform special restoration work that might be necessary only if the activity was previously stopped, but not destroyed.

- The system also calls the *onStart()* method, which happens every time your activity becomes visible.
  - You should usually use the *onStart()* callback method as the counterpart to the *onStop()* method.
  - It can be used to verify that required system features are enabled

```
@Override
protected void onStart() {
    super.onStart();  // Always call the superclass method first

    // The activity is either being restarted or started for the first time
    // so this is where we should make sure that GPS is enabled
    LocationManager locationManager =
            (LocationManager) getSystemService(Context.LOCATION_SERVICE);
    boolean gpsEnabled = locationManager.isProviderEnabled(LocationManager.GPS_PROVIDER

    if (!gpsEnabled) {
        // Create a dialog here that requests the user to enable GPS, and use an intent
        // with the android.provider.Settings.ACTION_LOCATION_SOURCE_SETTINGS action
        // to take the user to the Settings screen to enable GPS when they click "OK"
    }
}

@Override
protected void onRestart() {
    super.onRestart();  // Always call the superclass method first

    // Activity being restarted from stopped state
}
```

RECREATING

- The system can destroy an activity although the user was not finish it:
  - The system must shut down background processes to recover memory.
  - The activity can be destroyed and recreated each time the user rotates the screen.
- The system remembers that it existed and if the user navigates back to it, the system creates a new instance of the activity using a set of saved data that describes the state of the activity when it was destroyed.
  - The saved data is called the "instance state" and is a collection of key-value pairs stored in a ***Bundle*** object.
  - The system saves in the *bundle* information about each View object in your activity layout. So, the state of the layout is automatically restored to its previous state.

- You can add additional data to the saved instance state for your activity:
  - The method *onSaveInstanceState()* is called when the user leaves the activity, passing it the bundle object so you can add information to it.
  - It passes the same Bundle object to your activity's *onRestoreInstanceState()* and *onCreate()* methods to restore your state.

SAVING ACTIVITY STATE

- When an activity stops, the system calls *onSaveInstanceState()* so you can save state information with a collection of key-value pairs.
  - The default implementation of this method saves information about the state of the activity's view hierarchy.
  - You must always call the superclass implementation of *onSaveInstanceState()*
- You can save additional state information for your activity implementing *onSaveInstanceState()* and add key-value pairs to the **Bundle** object:

```
static final String STATE_SCORE = "playerScore";
static final String STATE_LEVEL = "playerLevel";
...

@Override
public void onSaveInstanceState(Bundle savedInstanceState) {
    // Save the user's current game state
    savedInstanceState.putInt(STATE_SCORE, mCurrentScore);
    savedInstanceState.putInt(STATE_LEVEL, mCurrentLevel);

    // Always call the superclass so it can save the view hierarchy state
    super.onSaveInstanceState(savedInstanceState);
}
```

RESTORING ACTIVITY STATE

When your activity is recreated, you can recover your saved state from the **Bundle** that the system passes your activity.

- Both the *onCreate()* and *onRestoreInstanceState()* callback methods receive the same Bundle that contains the instance state information.
- In the *onCreate()* method you must check whether the state Bundle is null before you attempt to read it. If it is null, then the system is creating a new instance of the activity, instead of restoring a previous one that was destroyed.

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState); // Always call the superclass first

    // Check whether we're recreating a previously destroyed instance
    if (savedInstanceState != null) {
        // Restore value of members from saved state
        mCurrentScore = savedInstanceState.getInt(STATE_SCORE);
        mCurrentLevel = savedInstanceState.getInt(STATE_LEVEL);
    } else {
        // Probably initialize members with default values for a new instance
    }
    ...
}
```

```
public void onRestoreInstanceState(Bundle savedInstanceState) {
    // Always call the superclass so it can restore the view hierarchy
    super.onRestoreInstanceState(savedInstanceState);

    // Restore state members from saved instance
    mCurrentScore = savedInstanceState.getInt(STATE_SCORE);
    mCurrentLevel = savedInstanceState.getInt(STATE_LEVEL);
}
```

CALLBACKS (I)

ACTIVITY

METHOD0	DESCRIPTION	KILLABLE AFTER?	NEXT
onCreate()	Called when the activity is first created. This is where you should do all of your normal static set up (create views, bind data to lists, and so on)	No	onStart()
onRestart()	Called after the activity has been stopped, just prior to it being started again.	No	onStart()
onStart()	Called just before the activity becomes visible to the user.	No	onResume() or onStop()
onResume()	Called just before the activity starts interacting with the user. At this point the activity is at the top of the activity stack, with user input going to it.	No	onPause()

CALLBACKS (II)

ACTIVITY

METHOD	DESCRIPTION	KILLABLE AFTER?	NEXT
onPause()	Called when the system is about to start resuming another activity. This method is typically used to commit unsaved changes to persistent data, stop animations and other things that may be consuming CPU, and so on.	**Yes**	onResume() or onStop()
onStop()	Called when the activity is no longer visible to the user. This may happen because it is being destroyed, or because another activity has been resumed and is covering it.	**Yes**	onRestart() or onDestroy()
onDestroy()	Called before the activity is destroyed. This is the final call that the activity will receive. It could be called either because the activity is finishing or because the system is temporarily destroying this instance of the activity to save space.	**Yes**	*nothing*

DEFINITION

ACTIVITY 

- ✧ An activity is a single, focused thing that the user can do.
- ✧ Almost all activities interact with the user, so the Activity class takes care of creating a window for you in which you can place your UI with setContentView(View).
- ✧ While activities are often presented to the user as full-screen windows, they can also be used in other ways:
  - as floating windows (via a theme with windowIsFloating set)
  - or embedded inside of another activity (using ActivityGroup).

START

✧ To start another activity call *startActivity()*, passing it an **Intent** that describes the activity you want to start.

- The intent specifies either the exact activity you want to start or describes the type of action you want to perform (and the system selects the appropriate activity for you, which can even be from a different application).
- An intent can also carry small amounts of data to be used by the activity that is started.

✧ To launch a known activity from your own app, you only need to create an intent that explicitly defines the activity you want to start, using the class name.

- For example, here's how one activity starts another activity named *SecondActivity*:

```
Intent intent = new Intent(this,
SecondActivity.class); startActivity(intent);
```

PARAMETERS

- ✧ You can pass information between activities using the intent's extra bundle.
- The data can be passed to other activity using intent *putExtra()* method.
- Data is passed as extras and are key/value pairs. The key is always a String. As value you can use the primitive data types int, float, chars, etc.
-

```
Intent intentActivity2 = new Intent (this, SecondActivity.class);
intentActivity2.putExtra ("MyData", "Data from Activity1");
intentActivity2.putExtra ("MyInt",10)
StartActivity (intentActivity2);
```

✧ You can retrieve the information using *getData()* methods on the Intent object. The Intent object can be retrieved via the *getIntent()* method.

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    ......
    Intent intent = getIntent();
    if (null != intent) {
        String stringData = intent.getStringExtra("MyData");
        int numberData = intent.getIntExtra("MyInt", defaultValue);
    }
}
```

START FOR A RESULT

- To receive a result from the activity that you start, you should start the activity by calling *startActivityForResult()* (instead of *startActivity()*):

```
static final int REQUEST_CODE = 1; // The request code

Intent i = new Intent(this, ActivityTwo.class);
// set the request code to any code you like, you can identify the callback via this code
startActivityForResult(i, REQUEST_CODE);
```

- When the sub-activity finishes, it can send results back to its caller via an **Intent** on your own *finish()* method, using the *setResult(int resultCode, Intent data)* method.

```
public void finish() {
    Intent return_data = new Intent(); // Prepare return result intent
    return_data.putExtra("returnKey1", "Swinging on a star. ");
    setResult(RESULT_OK, return_data); // Activity finished ok, return the data
    super.finish();
}
```

✧ To receive the result from the subsequent activity, implement the *onActivityResult()* callback method, it returns a result in an **Intent**.

```
@Override
protected void onActivityResult(int requestCode, int resultCode, Intent data) {
    if (resultCode == RESULT_OK && requestCode == REQUEST_CODE) {
        if (data.hasExtra("returnKey1")) {
            Toast.makeText(this, data.getExtras().getString("returnKey1"),
                    Toast.LENGTH_SHORT).show();
        }
    }
}
```

SHUT DOWN

- You can shut down an activity by calling its *finish()* method.
- You can also shut down a separate activity that you previously started by calling *finishActivity()*.
- However, in most cases, you **should not** explicitly **finish** an activity using these methods.
- The Android system manages the life of an activity for you.

INTENT

- An Intent defines an **asynchronous message** within the same application or between different applications.
- Intents allow to **send** or **receive data** from and to other activities or services.
  - An Intent object can contain information for the receiving component.
  - Also contain information for the Android system so that the system can determine which component should handle the request.
- Intents are a powerful concept as they allow the creation of **loosely coupled applications**:
  - Intents bind individual components to each other at runtime: activities, services, and broadcast receivers.
  - Allow to activate either a specific component or a specific type of component

INTENT OBJECT

INTENT

An Intent object contains information of interest to the component that receives the intent plus information of interest to the Android system

Intents are used to launch or activate an **activity**, to initiate a **service** or deliver new instructions to an ongoing one and to deliver it to all interested **broadcast receivers**.

Android supports explicit intents and implicit intents. **Explicit intent** names the component, which should be called. **Implicit intents** asked the system to perform a service without telling who should do this service.

The Android system will determine suitable applications for an implicit intent based on **intent filters**. Intent filters are typically defined via the Manifest file.

INTENT OBJECT COMPONENTS (I)

- **Component name:** The name of the component that should handle the intent.
  - The component name is optional.
    - If it is set, the Intent object is delivered to an instance of the designated class.
    - If it is not set, Android uses other information in the Intent object to locate a suitable target
  - The component name is set by *setComponent()*, *setClass()*, or *setClassName()* and read by *getComponent()*.
- **Action:** It is a string naming the action to be performed or, in the case of broadcast intents, the action that took place and is being reported.
  - The action determines how the rest of the intent is structured (the data and extras fields).
  - You can also define your own action strings for activating the components in your application. Those you invent should include the application package as a prefix.
    - For example: "**com.example.project.SHOW_COLOR**".

INTENT ACTIONS (I)

CONSTANT	TARGET COMPONENT	ACTION
ACTION_MAIN	activity	Start up as the initial activity of a task, with no data input and no returned output.
ACTION_VIEW	activity	Display the data to the user[1].
ACTION_EDIT	activity	Display data for the user to edit[1].
ACTION_PICK	activity	Pick/Catch an item from the data[1], returning what was selected
ACTION_INSERT	activity	Insert an empty item into the given container.
ACTION_DELETE	activity	Delete the given data from its container.
ACTION_CALL	activity	Perform a call to someone specified by the data[1].
ACTION_DIAL	activity	Dial a number as specified by the data[1].
ACTION_CHOOSER	activity	Display an activity chooser, allowing the user to pick what they want to before proceeding.
ACTION_SYNC	activity	Synchronize data on a server with data on the mobile device.

(1) You can use *getData()* method to access de URI from which to retrieve data.

INTENT ACTIONS (II)

CONSTANT	TARGET COMPONENT	ACTION
ACTION_BATTERY_LOW	broadcast receiver	A warning that the battery is low[2].
ACTION_HEADSET_PLUG	broadcast receiver	A headset has been plugged into the device, or unplugged from it.
ACTION_SCREEN_ON	broadcast receiver	The screen has been turned on[2].
ACTION_TIMEZONE_CHANGED	broadcast receiver	The setting for the time zone has changed.
ACTION_TIME_TICK	broadcast receiver	The current time has changed[2]. Sent every minute.
ACTION_PACKAGE_ADDED	broadcast receiver	A new application package has been installed on the device. The data contains the name of the package
ACTION_SHUTDOWN	broadcast receiver	Device is shutting down[2]

(2) This is a protected intent that can only be sent by the system.

INTENT OBJECT COMPONENTS (II)

- **Data:** The **URI** (Uniform Resource Identifier) of the data to be acted on and the **MIME type** of that data.
  - Different actions are paired with different kinds of data specifications.
  - When matching an intent to a component that is capable of handling the data, it's often important to know the type of data (its MIME type) in addition to its URI
    - The *setData()* method specifies data only as a URI, *setType()* specifies it only as a MIME type, and *setDataAndType()* specifies it as both a URI and a MIME type.
    - The URI is read by *getData()* and the type by *getType()*.
- **Category:** A string containing additional information about the kind of component that should handle the intent.
  - Any number of category descriptions can be placed in an Intent object.
    - The *addCategory()* method places a category in an Intent object, *removeCategory()* deletes a category previously added, and *getCategories()* gets the set of all categories currently in the object.

INTENT CATEGORIES

CONSTANT	MEANING
CATEGORY_DEFAULT	Set if the activity should be an option for the default action to perform on a piece of data.
CATEGORY_BROWSABLE	The target activity can be safely invoked by the browser to display data referenced by a link, an image or an e-mail message.
CATEGORY_GADGET	The activity can be embedded inside of another activity that hosts gadgets.
CATEGORY_HOME	The activity displays the home screen, the first screen the user sees when the device is turned on or when the HOME key is pressed.
CATEGORY_LAUNCHER	The activity can be the initial activity of a task and is listed in the top-level application launcher.
CATEGORY_PREFERENCE	The target activity is a preference panel.
CATEGORY_APP_MARKET	This activity allows the user to browse and download new applications.

INTENT OBJECT COMPONENTS (III)

- **Extras:** Key-value pairs for additional information that should be delivered to the component handling the intent.
  - Just as some actions are paired with particular kinds of data URIs, some are paired with particular extras.
    - The Intent object has a series of *put...()* methods for inserting various types of extra data and a similar set of *get...()* methods for reading the data. These methods parallel those for Bundle objects.
    - In fact, the extras can be installed and read as a Bundle using the *putExtras()* and *getExtras()* methods.
- **Flags:** Instruct the Android system how to launch an activity (for example, which task the activity should belong to) and how to treat it after it's launched (for example, whether it belongs in the list of recent activities).

INTENT

- **Explicit intents**: Android delivers an explicit intent to an instance of the designated target class.
  - Nothing in the Intent object other than the component name matters for determining which component should get the intent.
- **Implicit intents**: Android system must find the best component (or components) to handle the intent.
  - It does so by comparing the contents of the Intent object to ***intent filters***, structures associated with components that can potentially receive intents.
  - Filters advertise the capabilities of a component and delimit the intents it can handle.
  - Only three aspects of an Intent object are consulted when the object is tested against an intent filter: action, data (both URI and data type) and category.

INTENT

INTENT FILTERS

- To inform the system which implicit intents they can handle, activities, services, and broadcast receivers can have one or more intent filters.
  - Each filter describes a capability of the component, a set of intents that the component is willing to receive.
  - A component (activity, service) has separate filters for each job it can do, each face it can present to the user.
- To react to a certain implicit intent an application component must register itself via an IntentFilter in the "AndroidManifest.xml" to this event.
  - If a component does not define intent filters it can only be called by explicit intents.
  - A filter has fields that test in parallel the action, data, and category fields
    of an Intent object.
    - An implicit intent is tested against the filter in all three areas.
    - To be delivered to the component that owns the filter, it must pass all three tests.

INTENT FILTERS: ACTION TEST

```
<intent-filter . . . >
    <action android:name="com.example.project.SHOW_CURRENT" />
    <action android:name="com.example.project.SHOW_RECENT" />
    <action android:name="com.example.project.SHOW_PENDING" />
    . . .
</intent-filter>
```

- A filter may list more than one action. However, the list cannot be empty or it will block all intents.
- The action specified in the Intent object must match one of the actions listed in the filter.
- If the object or the filter does not specify an action, the results are as follows:
  - If the filter fails to list any actions, there is nothing for an intent to match,
    so all intents fail. No intents can get through the filter.
  - On the other hand, an Intent object that doesn't specify an action automatically passes.

INTENT FILTERS: CATEGORY TEST

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

- For an intent to pass the category test, every category in the Intent object must match a category in the filter.
  - The filter can list additional categories, but it cannot omit any that are in the intent.
- An Intent object with no categories should always pass this test, regardless of what's in the filter.
  - There are one important exception: Android treats all implicit intents passed to startActivity() as if they contained at least one category: "**android.intent.category.DEFAULT**" (the **CATEGORY_DEFAULT** constant).
  - Therefore, activities that are willing to receive implicit intents must include "**android.intent.category.DEFAULT**" in their intent filters.

INTENT FILTERS: DATA TEST

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

✧ Each <data> element can specify a URI and a data type (MIME media type).

- There are separate attributes (scheme, host, port, and path) for each part of the URI:
  - **scheme**://*host*:port/path
- Example: **content**://*com.example.project*:200/folder/subfolder/etc

✧ A URI in an Intent object is compared only to the parts of the URI actually mentioned in the filter.

✧ The type attribute of a <data> element specifies the MIME type.

- Both the Intent object and the filter can use a "*" wildcard for the subtype field. For example, "text/*" or "audio/*“, indicating any subtype matches.

INTENT FILTERS: EXAMPLES

✧ This filter declares the main entry point for an application.

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

The standard MAIN action is an entry point that does not require any other information in the Intent, and the LAUNCHER category says that this entry point should be listed in the application launcher.

✧ Th ry of files.

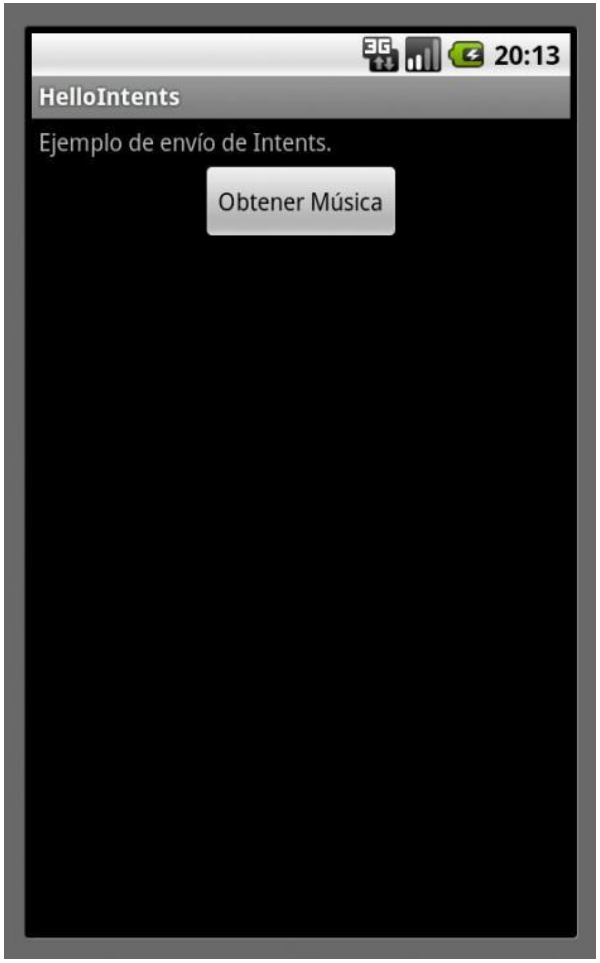
```
<intent-filter>
    <action android:name="android.intent.action.VIEW" />
    <action android:name="android.intent.action.EDIT" />
    <action android:name="android.intent.action.PICK" />
    <category android:name="android.intent.category.DEFAULT" />
    <data android:mimeType="vnd.android.cursor.dir/vnd.google.note" />
</intent-filter>
```

The mimeType attribute of the <data> element specifies the kind of data that these actions operate on (Google notes).

- It indicates that the activity can get a Cursor over zero or more items (vnd.android.cursor.dir) from a content provider that holds Note Pad data (vnd.google.note).
- The Intent object that launches the activity would include a content: URI specifying the exact data of this type that the activity should open.

HELLOINTENTS

- Create an application that use intents to access the functionality of other components:
  - Open an activity that allows me to select audio content type



HELLOINTENTS

CREATING THE PROJECT

✧ Start a new project and Activity called "HelloIntents“

1. Open the *res/layout/main.xml* file and replace it with the right code:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:padding="4dip"
    android:gravity="center_horizontal"
    android:layout_width="match_parent" android:layout_height="match_parent">
    <TextView
        android:layout_width="match_parent" android:layout_height="wrap_content"
        android:layout_weight="0"
        android:paddingBottom="4dip"
        android:text="@string/intents"/>
    <Button android:id="@+id/get_music"
        android:layout_width="wrap_content" android:layout_height="wrap_content"
        android:text="@string/get_music">
        <requestFocus />
    </Button>
</LinearLayout>
```

2. Insert the following strings on strings.xml file:

```
<string name="intents">Example of launching various Intents.</string>
<string name="get_music">Get Music</string>
```

HELLOINTENT

USING INTENTS

3. Create the onClick listener for get_music button, to create a new intent to show a Chooser to manage audio files dialog:

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

        // Watch for button clicks.
        Button button = (Button)findViewById(R.id.get_music);
        button.setOnClickListener(mGetMusicListener);
    }

    private OnClickListener mGetMusicListener = new OnClickListener() {
        public void onClick(View v) {
            Intent intent = new Intent(Intent.ACTION_GET_CONTENT);
            intent.setType("audio/*");
            startActivity(Intent.createChooser(intent, "Select music"));
        }
    };
}
```

HELLOINTENT

CREATING THE SECOND ACTIVITY

- Create new Activity called “ResultActivity.java “
  1. Open the *res/layout/activity_result.xml* file and replace it with the right code:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical" >
    <TextView
        android:id="@+id/displayintentextra
        "
        android:layout_width="wrap_content"
        android:layout_height="wrap_content
        " android:text="Input" />
    <EditText
        android:id="@+id/returnValue"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        >
        <requestFocus />
    </EditText>
</LinearLayout>
```

HELLOINTENT

MANAGING DATA

2. Create the onClick listener for get_music button, to create a new intent to show the ResultActivity new activity:

```
buttonStartIntent.setOnClickListener(new Button.OnClickListener(){
        public void onClick(View view)
        {
            String string = textEdit.getText().toString();
            Intent i = new Intent(MainActivity.this, ResultActivity.class);
            i.putExtra("yourkey", string);
            startActivityForResult(i, REQUEST_CODE);
        }
});
```

3. Receive data in the activity:

```
public class ResultActivity extends Activity
{
    @Override
    public void onCreate(Bundle bundle) {
        super.onCreate(bundle);
        setContentView(R.layout.activity_result); Bundle
        extras = getIntent().getExtras();
        String inputString = extras.getString("yourkey");
        TextView view = (TextView) findViewById(R.iddisplayintentextra);
        view.setText(inputString);
    }
```

HELLOINTENT

SENDING RESULTS

4. We Override finish method of ResultActivity.java to return the result to the calling activity as an intent:

```
public class ResultActivity extends Activity
{
    @Override
    public void finish() {
        Intent intent = new Intent();
        EditText editText= (EditText) findViewById(R.idreturnValue);
        String string = editText.getText().toString();
        intent.putExtra("returnkey", string);
        setResult(RESULT_OK, intent);
        super.finish();
    }
```

HELLOINTENT

GET THE RESULT

5. In the calling activy, we can **get the results** of other activity implementing the onActivityResult callback:

```
@Override
protected void onActivityResult(int requestCode, int resultCode, Intent data)
{
    switch(requestCode){
        case PICKFILE_RESULT_CODE:
            if(resultCode==RESULT_OK){
                String FilePath =
                data.getData().getPath();
                textFile.setText(FilePath);
            }
            break;
        case REQUEST_CODE:
            if (resultCode == RESULT_OK) {
                if (data.hasExtra("returnkey")) {
                      String result = data.getExtras().getString("returnkey");
                     if (result != null && result.length() > 0) {
                           Toast.makeText(this, result,
                           Toast.LENGTH_SHORT).show();
                     }
                }
            }
    }
```