Android

Desenvolvimento de Software e Sistemas Móveis (DSSMV)
Licenciatura em Engenharia de Telecomunicações e Informática
LETI/ISEP

2025/26

Paulo Baltarejo Sousa
pbs@isep.ipp.pt

Disclaimer

Material and Slides

Some of the material/slides are adapted from various:

- Presentations found on the internet;
- Books;
- Web sites;
- ...

Outline

1 Intents

2 Shared Preferences

3 Android Adapter

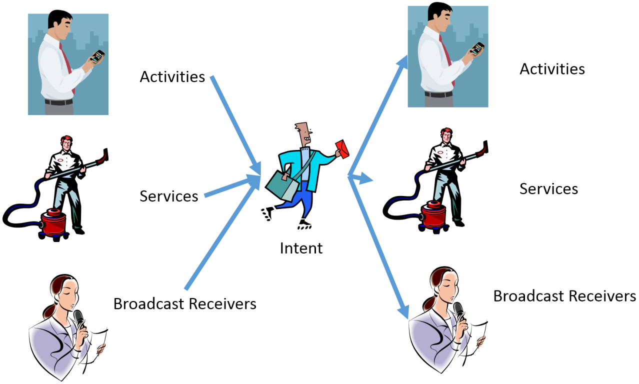
4 Threads

5 Bibliography

Intents

What is an Intent

- An `Intent` is a messaging object you can use to request an action from another app component.
- Intents **facilitate communication between components** in several ways.

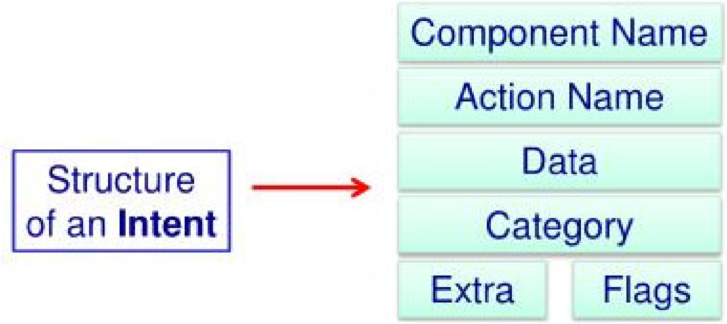


Intents are used for

- To start an activity:
  - The `Intent` describes the activity to start and carries any necessary data and is passed to `startActivity` method.
  - If you want to receive a result from the activity when it finishes, call `startActivityForResult`. Your activity receives the result as a separate Intent object in your activity's `onActivityResult` callback.
- To start a service:
  - A `Service` is a component that performs operations in the background without a user interface. You can start a service by passing an `Intent` to `startService`. The `Intent` describes the service to start and carries any necessary data.
- To deliver a broadcast:
  - A broadcast is a message that any app can receive. The system delivers various broadcasts for system events, such as when the system boots up or the device starts charging. You can deliver a broadcast to other apps by passing an Intent to `sendBroadcast`.

What is in an Intent (I)

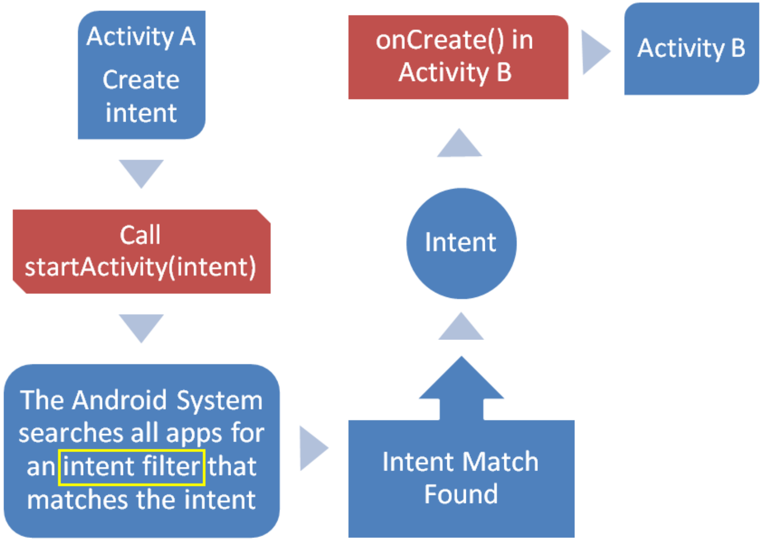
- An Intent object **carries information that the Android system uses to determine which component to start**, plus information that the recipient component uses in order to properly perform the action.



What is in an Intent (II)

- The primary information contained in an `Intent` is the following:
    - **Component name**: The name of the component to start.
    - **Action**: A string that specifies the generic action to perform (such as view or pick).
    - **Data**: The URI (a Uri object) that references the data to be acted on and/or the MIME type of that data.
    - **Category**: A string containing additional information about the kind of component that should handle the intent.
    - **Extras**: Key-value pairs that carry additional information required to accomplish the requested action.
    - **Flags**: Flags defined in the Intent class that function as metadata for the intent.

How it works?

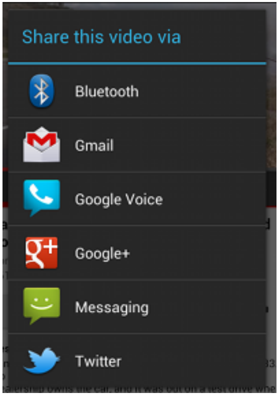

Activity A
Create intent
Call startActivity(intent)
The Android System searches all apps for an intent filter that matches the intent
Intent Match Found
Intent
onCreate() in Activity B
Activity B

Intent Types

- **Implicit intents**
  - Do not name a specific component, but instead declare a general action to perform, which allows a component from another app to handle it.
  - For example, if you want to show the user a location on a map, you can use an implicit intent to request that another capable app show a specified location on a map.
- **Explicit intents**
  - Specify the component to start by name (the fully-qualified class name).
  - Used to start a component in your own app, because you know the class name of the activity or service you want to start.

Implicit Intents

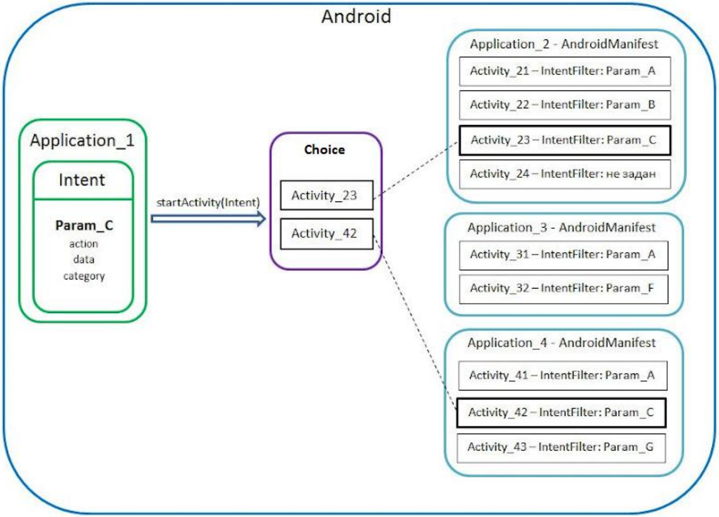
- The **Android system finds the appropriate component** to start by comparing the contents of the intent to the **intent filters declared in the manifest file** of other apps on the device.
    - If the intent matches an intent filter, the system starts that component and delivers it the `Intent` object.
    - If multiple intent filters are compatible, the system displays a dialog so the user can pick which app to use.



Intent-Filter (I)

```
<activity android:name="MainActivity">
<!-- This activity is the main entry, should appear in app launcher -->
<intent-filter>
<action android:name="android.intent.action.MAIN" />
<category android:name="android.intent.category.LAUNCHER" />
</intent-filter>
</activity>

<activity android:name="ShareActivity">
<!-- This activity handles "SEND" actions with text data -->
<intent-filter>
<action android:name="android.intent.action.SEND"/>
<category android:name="android.intent.category.DEFAULT"/>
<data android:mimeType="text/plain"/>
</intent-filter>
<!-- This activity also handles "SEND" and "SEND_MULTIPLE" with media data -->
<intent-filter>
<action android:name="android.intent.action.SEND"/>
<action android:name="android.intent.action.SEND_MULTIPLE"/>
<category android:name="android.intent.category.DEFAULT"/>
<data android:mimeType="application/vnd.google.panorama360+jpg"/>
<data android:mimeType="image/*"/>
<data android:mimeType="video/*"/>
</intent-filter>
</activity>
```

Intent-Filter (II)


Android
Application_1
Intent
Param_C
action
data
category
startActivity(Intent)
Choice
Activity_23
Activity_42
Application_2 - AndroidManifest
Activity_21 – IntentFilter: Param_A
Activity_22 – IntentFilter: Param_B
Activity_23 – IntentFilter: Param_C
Activity_24 – IntentFilter: не задан
Application_3 - AndroidManifest
Activity_31 – IntentFilter: Param_A
Activity_32 – IntentFilter: Param_F
Application_4 - AndroidManifest
Activity_41 – IntentFilter: Param_A
Activity_42 – IntentFilter: Param_C
Activity_43 – IntentFilter: Param_G

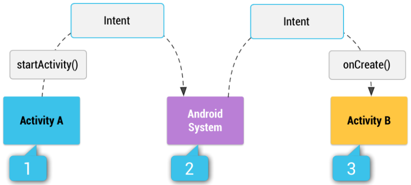
Implicit Intent: Purpose

- Using an **implicit intent is useful when your app cannot perform the action, but other apps probably can** and you would like the user to pick which app to use.
  - For example, if you have content you want the user to share with other people, create an intent with the `ACTION_SEND` action and add extras that specify the content to share. When you call `startActivity` with that intent, the user can pick an app through which to share the content.

```
// Create the text message with a string
Intent sendIntent = new Intent();
sendIntent.setAction(Intent.ACTION_SEND);
sendIntent.putExtra(Intent.EXTRA_TEXT, textMessage);
sendIntent.setType("text/plain");

// Verify that the intent will resolve to an activity
if (sendIntent.resolveActivity(getPackageManager()) != null) {
  startActivity(sendIntent);
}
```

Explicit intents

- The system starts the app component specified in the Intent object.



- Illustration of how an implicit intent is delivered through the system to start another activity:
  1. `Activity` A creates an `Intent` with an action description and passes it to `startActivity()` method.
  2. The Android System searches all apps for an intent filter that matches the `Intent`.
  3. When a match is found the system starts the matching activity (`Activity` B) by invoking its `onCreate()` method and passing it the `Intent`.

Explicit Intent

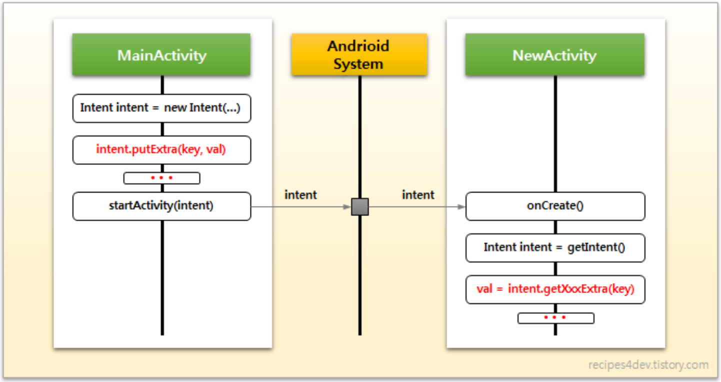
- An **explicit intent** is one that you use to launch a specific app component, such as a particular activity or service in your app. To create an explicit intent, define the component name for the Intent object – all other intent properties are optional.

```
    intent = new Intent(MainActivity.this, SearchActivity.class);
    intent.putStringArrayListExtra("LIST",list);
    startActivity(intent);
```

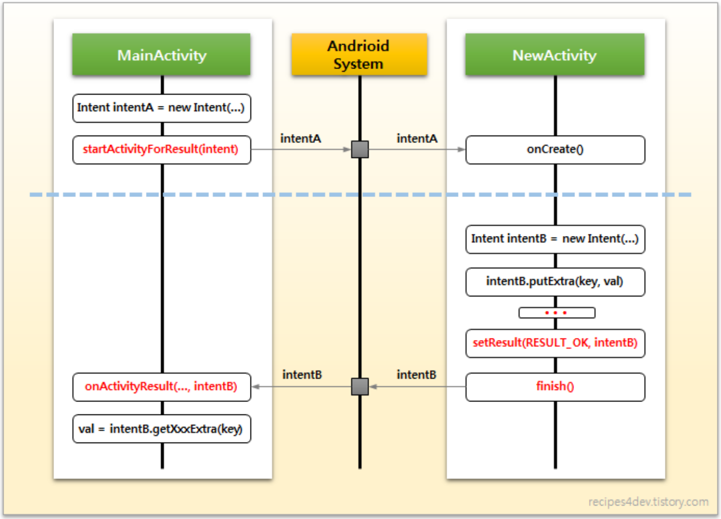
- The `Intent(Context, Class)` constructor supplies the app `Context` and the component a `Class` object.

Intent: data transfer

- `putExtra (Key, Value)`
  - Add extended data to the intent.
- `GetXxxxExtra (Key)`
  - Retrieve data from the intent.

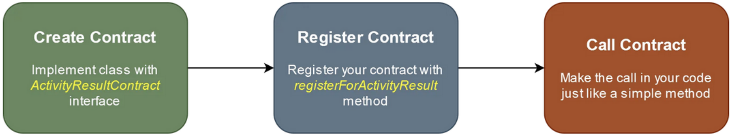


Intent: data returning (I)


MainActivity
Andrioid System
NewActivity
Intent intentA = new Intent(...)
startActivityForResult(intent)
intentA
intentA
onCreate()
Intent intentB = new Intent(...)
intentB.putExtra(key, val)
• • •
setResult(RESULT_OK, intentB)
intentB
intentB
onActivityResult(..., intentB)
finish()
val = intentB.getXxxExtra(key)
recipes4dev.tistory.com

Intent: data returning (II)

- `startActivityForResult` is deprecated
- **Solution**: Activity Result API
  - Steps:


Create Contract
Implement class with *ActivityResultContract* interface
Register Contract
Register your contract with *registerForActivityResult* method
Call Contract
Make the call in your code just like a simple method

`startActivityForResult` deprecated

- Solution: Activity Results API

```
ActivityResultLauncher<Intent> mainActivity4Launcher = registerForActivityResult(
new ActivityResultContracts.StartActivityForResult(),
new ActivityResultCallback<ActivityResult>() {
 @Override
 public void onActivityResult(ActivityResult result) {
  if (result.getResultCode() == Activity.RESULT_OK) {
   // There are no request codes
   Intent data = result.getData();
   String msg = data.getStringExtra("EXTRA_MESSAGE");
   tv.setText("The message received from MainActivity4:\n\t" + msg);
  }
 }
});

Button button3 = (Button)findViewById(R.id.button3);
button3.setOnClickListener(new View.OnClickListener() {
 public void onClick(View arg0) {
  Intent i = new Intent(MainActivity.this, MainActivity4.class);
mainActivity4Launcher.launch(i);
 }
});
```

Activity Result API (I)

- `registerForActivityResult` - For registering the result callback
- `ActivityResultContract` - A contract specifying that an activity can be called with an input of type I and produce an output of type O.
- `ActivityResultCallback` - A type-safe callback to be called when an activity result is available.
  - It is a single method interface with an `onActivityResult` method that takes an object of the output type defined in the `ActivityResultContract`
- `ActivityResultLauncher` - A launcher for a previously-prepared call to start the process of executing an `ActivityResultContract`.

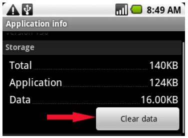
Activity Result API (II)

- `registerForActivityResult` takes an `ActivityResultContract` and an `ActivityResultCallback` and returns an `ActivityResultLauncher` which you?ll use to launch the other activity.
- `launch` starts the process of producing the result.

Shared Preferences

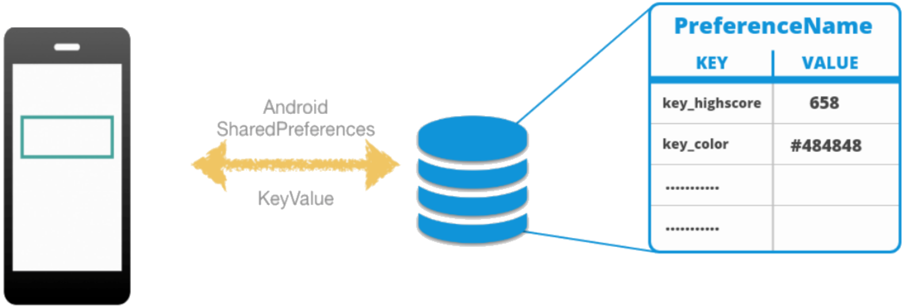
What are `SharedPreferences`? (I)

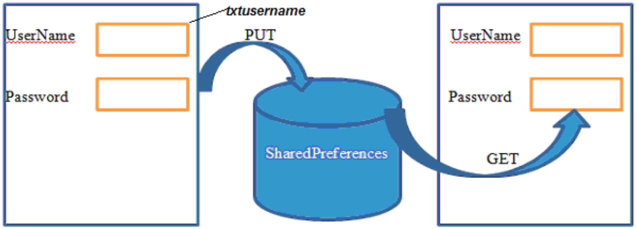
- Shared Preferences is the way in which one can store and retrieve small amounts of primitive data as key/value pairs to a file on the device storage such as String, int, float, Boolean that make up your preferences in an XML file inside the app on the device storage.
- A `SharedPreferences` object points to a file containing `key,value` pairs and provides simple methods to read and write them.
- Each `SharedPreferences` file is managed by the framework and can be private or shared.
- **They are accessible from anywhere within the app to either put data into the file or take data out of the file.**
- They can be removed or cleared from: `Settings -> Applications -> Manage applications -> (choose your app) -> Clear data` or `Uninstall`



Total	140KB
Application	124KB
Data	16.00KB

What are `SharedPreferences`? (II)





Where `SharedPreferences` should be used?

- Typically for small collection of data.
- **In all situations where apps needs to save data about the application state so that users progress is not lost**.
- Example:
  - In a game for saving user's name, high score, and state of the game when they logged off.
  - Then the next time they log in, their score and game level would be retrieved from the preference file and they would continue the game from where they ended it when they logged off.

Get a Handle to a `SharedPreferences`

- You can create a new shared preference file or access an existing one by calling one of two methods:
  - `getSharedPreferences`: Use this if you need multiple shared preference files identified by name
  - `getPreferences`: Use this from an Activity if you need to use only one shared preference file for the activity.
    - Because this retrieves a default shared preference file that belongs to the activity, you don’t need to supply a name.

Get Shared Preferences

- If you need multiple files

```
Context context = getActivity();
SharedPreferences sharedPref = context.getSharedPreferences("MyPrefs",
     Context.MODE_PRIVATE);
```

  - The first parameter is the key and the second parameter is the MODE.
    - `MODE_APPEND`: This will append the new preferences with the already existing preferences
    - `MODE_PRIVATE`: By setting this mode , the file can only be accessed using calling application
    - ...: some modes allow others apps to read or to write the preferences.
- If you need just one shared preference file for your activity

```
SharedPreferences sharedPref = getActivity().getPreferences(Context.
     MODE_PRIVATE);
```

Write to Shared Preferences

- Create a `SharedPreferences.Editor` by calling `edit` on your `SharedPreferences`.
- Then call `commit` to save the changes.

```
/******* Create SharedPreferences *******/
SharedPreferences sharedPref = context.getSharedPreferences("MyPrefs", Context.
     MODE_PRIVATE);
SharedPreferences.Editor editor = sharedPref.edit();

/**************** Storing data as KEY/VALUE pair ******************/

editor.putBoolean("key_name1", true); // Saving boolean - true/false
editor.putInt("key_name2", "int value"); // Saving integer
editor.putFloat("key_name3", "float value"); // Saving float
editor.putLong("key_name4", "long value"); // Saving long
editor.putString("key_name5", "string value"); // Saving string

// Save the changes in SharedPreferences
editor.commit(); // commit changes
```

Read from Shared Preferences

- To retrieve values from a shared preferences file, call methods such as `getInt` and `getString`, providing the key for the value you want, and optionally a default value to return if the key is not present.
- Checking if a particular preference exists (is set) or not using the `contains` method on the `SharedPreferences`

```
/******* Create SharedPreferences *******/
SharedPreferences sharedPref = context.getSharedPreferences("MyPrefs", Context.
     MODE_PRIVATE);
/**************** Get SharedPreferences data ******************/
// If value for key not exist then return second param value - In this case null
sharedPref.getBoolean("key_name1", null); // getting boolean
sharedPref.getInt("key_name2", null); // getting Integer
sharedPref.getFloat("key_name3", null); // getting Float
sharedPref.getLong("key_name4", null); // getting Long
sharedPref.getString("key_name5", null); // getting String
/**************** Checking SharedPreferences data ******************/
boolean exists = sharedPref.contains("username");
```

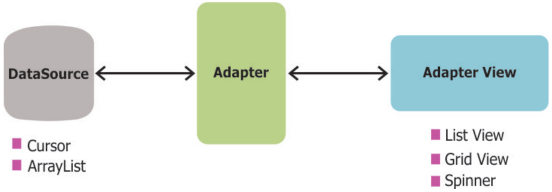
Commit changes

- In order to commit changes can be used two methods:
  - `apply`
    - This saves your data into memory immediately and saves the data to disk on a separate thread. So there is no chance of blocking the main thread (your app won't hang).
    - It is the preferred technique but has only been available since Gingerbread (API 9, Android 2.3).
  - `commit`
    - Calling this will save the data to the file however, the process is carried out in the thread that called it, stopping everything else until the save is complete. It returns true on successful completion, false on failure.
    - Use `commit` if you need confirmation of the success of saving your data or if you are developing for pre-Gingerbread devices.
    - It has been available since API 1

Android Adapter

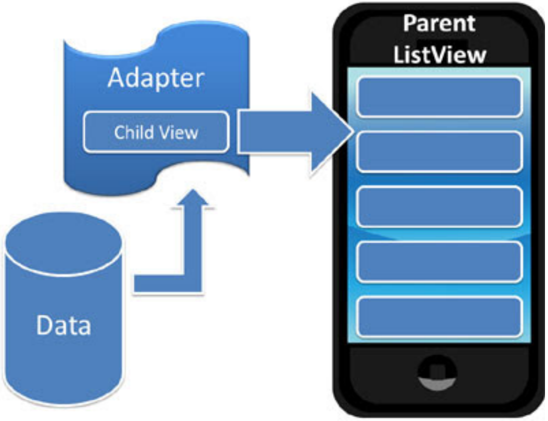
Adapters (I)

- Android's Adapter is described in the API documentation, as **a bridge** between an `AdapterView` and the underlying data for that `View`.
    - An `AdapterView` is a group of `View` components in Android that include the `ListView`, `Spinner`, and `GridView`.



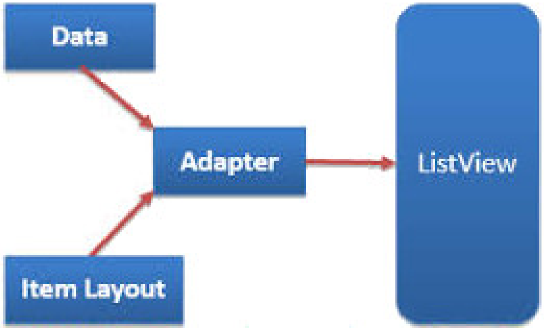
Adapters (II)

- Adapters get the data and pass it, along with a child view, to the parent `AdapterView` which displays the child view and the data



Custom Adapter(I)

- Create a Java class to model Data.
- Create a `Listview` item layout
- Create a subclass of the `BaseAdapter`
    - Override `BaseAdapter` methods



Custom Adapter(II)

```
public class CustomAdapter extends BaseAdapter{
  Context context;
  int rowLayout;
  ArrayList<Data> data;
  public CustomAdapter (Context context, int rowLayout, ArrayList<Data> data){
    this.context = context;
    this.rowLayout = rowLayout;
    this.data = data;
  }
  @Override
  public int getCount() {
    return data.size();
  }

  @Override
  public Object getItem(int position) {
    return data.get(position);
  }

  @Override
  public long getItemId(int position) {
    return 0;
  }

...
```

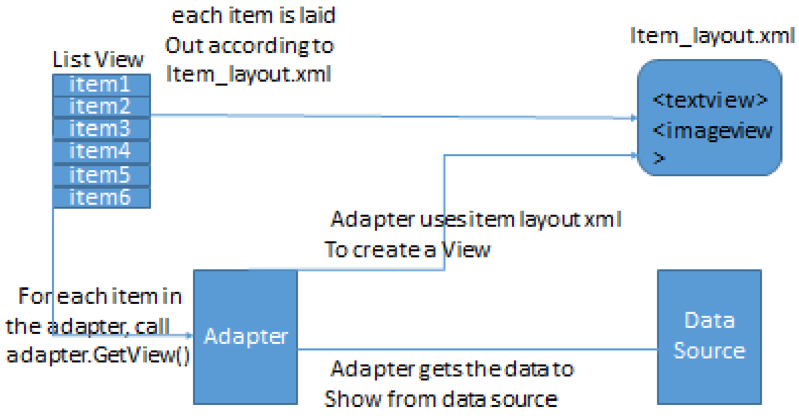
Custom Adapter(III)

```
public class CustomAdapter extends BaseAdapter{
...
 @Override
 public View getView(int position, View convertView, ViewGroup parent) {
   View rowView;
   LayoutInflater inflater = (LayoutInflater)context.getSystemService(Context.
        LAYOUT_INFLATER_SERVICE);
   rowView = inflater.inflate(this.rowLayout, null);
   TextView name = (TextView) rowView.findViewById(R.id.version_name);
   TextView date = (TextView) rowView.findViewById(R.id.version_date);
   TextView api = (TextView) rowView.findViewById(R.id.version_api);
   ImageView icon = (ImageView) rowView.findViewById(R.id.version_icon);
   Data row = (Data)getItem(position);
   name.setText(row.getName());
   date.setText(row.getDate());
   api.setText("API: "+row.getApi());
   Drawable drawable = ContextCompat.getDrawable(this.context, row.getIcon());
   if(drawable == null) {
   drawable = ContextCompat.getDrawable(this.context, R.drawable.noimage);
   }
   icon.setImageDrawable(drawable);
   return rowView;
 }
}
```

Custom Adapter(IV)

```
public class MainActivity extends AppCompatActivity {
 ArrayList<Data> data;
 ListView lv;
 @Override
 protected void onCreate(Bundle savedInstanceState) {
  super.onCreate(savedInstanceState);
  setContentView(R.layout.activity_main);
  lv = (ListView) findViewById(R.id.listView);
  data = getData();
  CustomAdapter adapter = new CustomAdapter(this, R.layout.list_item, data);
  lv.setAdapter(adapter);
 }
...
}
```

Custom Adapter(V)

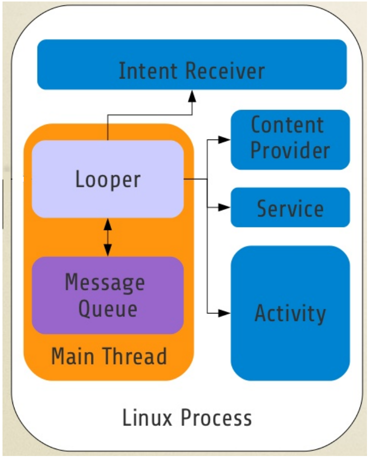


- Whenever there is a change on the Data Source, we must call Adapter `notifyDataSetChanged` method.

Threads

Android Process

- When an application component starts the **Android system starts a new Linux process for the application with a single thread of execution**.
- By default, **all components of the same application run in the same process and thread** (called the **Main Thread** or **User Interface (UI) Thread**).



Main thread (UI Thread)

- This thread is very important because **it is in charge of dispatching events**, specially, UI events,
  - When your app performs intensive work in response to user interaction, this single thread model can yield poor performance.
    - Specifically, if everything is happening in the UI thread, performing long operations such as network access or database queries will block the whole UI.
    - **When the thread is blocked, no events can be dispatched**.
- **Andoid UI toolkit is not thread-safe**.
- **UI has to be manipulated by main thread**.
  - Thus, there are simply two rules to Android's single thread model:
    - **Do not block the Main thread**;
    - **Do not access the Android UI toolkit from outside the Main thread**.

Background or Worker threads

- Because of the single thread model, it’s vital to the responsiveness of an app’s UI that do not block the Main thread.
- If there are operations to perform that are not instantaneous, they should be performed in separate threads, **background** or **worker** threads.
- However, note that **you cannot update the UI from any thread other than the UI thread**.
- To fix this problem, Android offers several ways to access the UI thread from other threads:
  - `Activity.runOnUiThread(Runnable)`
  - `View.post(Runnable)`
  - `Handler`

Activity.runOnUiThread(Runnable)

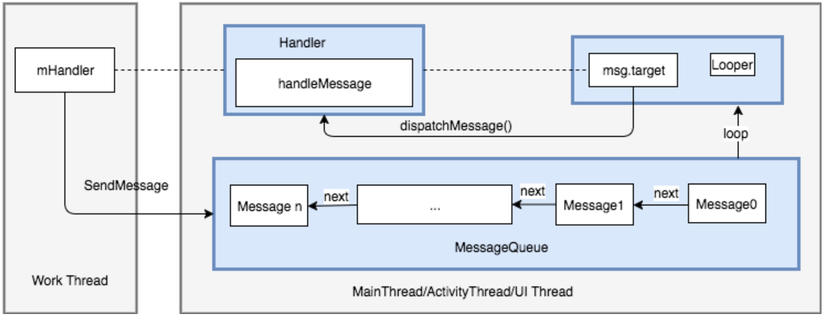
```
Button button = (Button) findViewById(R.id.button);
button.setOnClickListener(new View.OnClickListener() {
 @Override
 public void onClick(View v) {
  new Thread() {
   public void run() {
    while (i++ < 1000) {
     try {
      runOnUiThread(new Runnable() {
       @Override
       public void run() {
        textView.setText("#" + i);
       }
      });
      Thread.sleep(300);
     } catch (InterruptedException e) {
      e.printStackTrace();
     }
    }
   }
  }.start();
 }
});
```

View.post(Runnable)

```
Button button2 = (Button) findViewById(R.id.button2);
button2.setOnClickListener(new View.OnClickListener() {
  @Override
  public void onClick(View v) {
    new Thread() {
      public void run() {
        while (i++ < 1000) {
          try {
            textView.post(new Runnable() {
              @Override
              public void run() {
                textView.setText("#" + i);
              }
            });
            Thread.sleep(300);
          } catch (InterruptedException e) {
            e.printStackTrace();
          }
        }
      }
    }.start();

  }
});
```

`Handler` (I)

- A Handler allows you to send and process Message and Runnable objects associated with a thread's MessageQueue.
- Each Handler instance is associated with a single thread and that thread's message queue



Handler (II)

```
public class MainActivity extends AppCompatActivity {
 TextView textView;
 int i = 0;

 Handler handler = new Handler(Looper.getMainLooper()){
   @Override
   public void handleMessage(Message msg) {
     Bundle bundle = msg.getData();
     String string = bundle.getString("MESSAGE_KEY");
     textView.setText(string);
   }
 };
  ...
}
```

Handler (III)

```
Button button = (Button) findViewById(R.id.button);
button.setOnClickListener(new View.OnClickListener() {
 @Override
 public void onClick(View v) {
   new Thread() {
     public void run() {
       while (i++ < 1000) {
         try {
           Message message = handler.obtainMessage();
           Bundle bundle = new Bundle();
           bundle.putString("MESSAGE_KEY", "#" + i);
           message.setData(bundle);
           handler.sendMessage(message);
           Thread.sleep(300);
         } catch (InterruptedException e) {
           e.printStackTrace();
         }
       }
     }
   }.start();
 }
});
```

Asynctask

- `Asynctask` allows to **perform asynchronous work**.
  - It performs the blocking operations in a worker thread and then publishes the results on the Main thread.

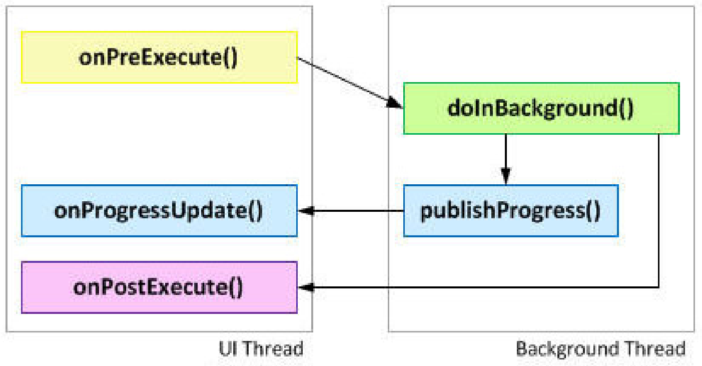
DEPRECATED

Asynctask (I)

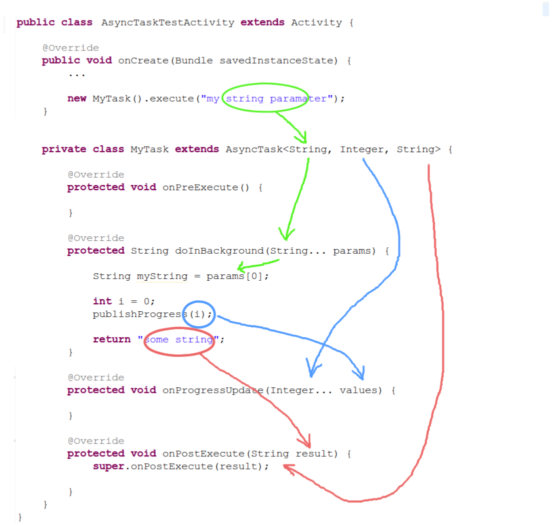
- `Asynctask` allows to **perform asynchronous work**.
  - It performs the blocking operations in a worker thread and then publishes the results on the Main thread.
  - **It is deprecated**

```
public void onClick(View v) {
  new DownloadImageTask().execute("http://example.com/image.png");
}

private class DownloadImageTask extends AsyncTask<String, Void, Bitmap> {
  /** The system calls this to perform work in a worker thread and
  * delivers it the parameters given to AsyncTask.execute() */
  protected Bitmap doInBackground(String... urls) {
    return loadImageFromNetwork(urls[0]);
  }

  /** The system calls this to perform work in the UI thread and delivers
  * the result from doInBackground() */
  protected void onPostExecute(Bitmap result) {
    mImageView.setImageBitmap(result);
  }
}
```

Asynctask (II)



Asynctask (III)



```
public class AsyncTaskTestActivity extends Activity {

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

        new MyTask().execute("my string parameter");
    }

    private class MyTask extends AsyncTask<String, Integer, String> {

        @Override
        protected void onPreExecute() {

        }

        @Override
        protected String doInBackground(String... params) {

            String myString = params[0];

            int i = 0;
            publishProgress(i);

            return "some string";
        }

        @Override
        protected void onProgressUpdate(Integer... values) {

        }

        @Override
        protected void onPostExecute(String result) {
            super.onPostExecute(result);

        }
    }
}
```

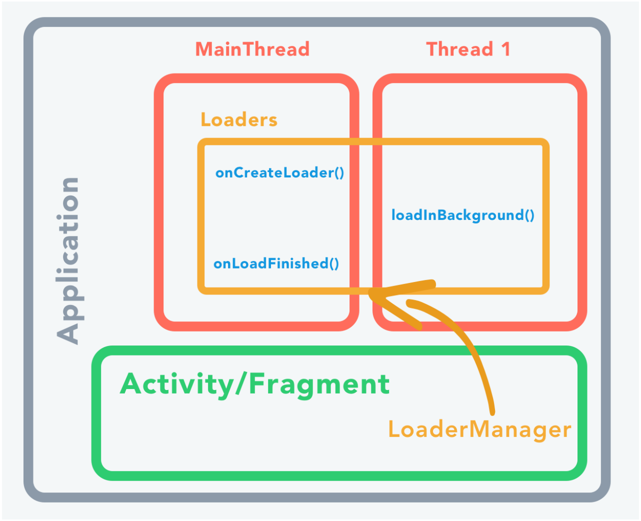
`AsyncTaskLoader` (I)

- Consider a user launching an app that loads up a collection of data from a network call handled on a separate thread created by an `Asynctask`.
- If while the connection is still on, the user decides to rotate the phone
  - The initial activity is destroyed, and a new activity is created.
  - But because the thread doing the network call was created by an `Asynctask`, it does not get informed that the activity from where it was launched has been killed, it still returns the data it collected from the network call to the dead activity.
  - This implies that the AsyncTask thread keeps in memory a zombie activity.

`AsyncTaskLoader` (II)

- `AsyncTaskLoader` provides a thread that is tied to the application lifecycle- knowing the state of the activity that created it.
- If the activity is dead, and the new one tries to create another thread to do the same thing, it does not create a new one.
- Instead it continues to use the old thread, and returns the data to the new activity when ready.

AsyncTaskLoader (III)



Implementing an `AsyncTaskLoader` (I)

- Create a `AsyncTaskLoader` subclass

```
public class StringLoader extends AsyncTaskLoader<String> {
  private String mString;
  public StringLoader(@NonNull Context context, String mString) {
    super(context);
    this.mString = mString;
    this.mString +="StringLoader\n";
  }
  @Override
  protected void onStartLoading() {
    super.onStartLoading();
    this.mString +="onStartLoading\n";
    forceLoad();// Starts the loadInBackground method
  }

  @Nullable
  @Override
  public String loadInBackground() {
    this.mString +="loadInBackground\n";
    return this.mString;
  }
}
```

Implementing an `AsyncTaskLoader` (II)

- Create a `LoaderManager.LoaderCallbacks` variable

```
public class MainActivity extends AppCompatActivity {
  TextView textView;

  LoaderManager.LoaderCallbacks<String> loader = new LoaderManager.
       LoaderCallbacks<String>() {
    @NonNull
    @Override
    public Loader<String> onCreateLoader(int id, @Nullable Bundle args) {
      return new StringLoader(MainActivity.this, args.getString("MESSAGE"));
    }

    @Override
    public void onLoadFinished(@NonNull Loader<String> loader, String data)
          {
      textView.setText(data+ "onLoadFinished\n");
    }

    @Override
    public void onLoaderReset(@NonNull Loader<String> loader) {
    }
  };
  ...
}
```

Implementing an `AsyncTaskLoader` (III)

- Start `AsyncTaskLoader`

```
    ...
    private static final int LOADER_ID = 20;
    ...
    @Override
    protected void onCreate(Bundle savedInstanceState) {
      super.onCreate(savedInstanceState);
      setContentView(R.layout.activity_main);
      textView = (TextView) findViewById(R.id.textview);

      Button button = (Button) findViewById(R.id.button);
      button.setOnClickListener(new View.OnClickListener(){
        @Override
        public void onClick(View view) {
          Bundle queryBundle = new Bundle();
          queryBundle.putString("MESSAGE", "onClick\n");
          LoaderManager.getInstance(MainActivity.this).restartLoader(LOADER_ID,
                 queryBundle, loader);
        }
      });
      if(LoaderManager.getInstance(this).getLoader(LOADER_ID) !=null){
        LoaderManager.getInstance(this).initLoader(LOADER_ID,null,loader);
      }
    }
  }
```

Implementing an `AsyncTaskLoader` (IV)

- `LoadManager`: Interface associated with an `Activity` or `Fragment` for managing one or more `Loader` instances associated with it.
- `Loader`: A class that performs asynchronous loading of data. While Loaders are active they should monitor the source of their data and deliver new results when the contents change.
- `getInstance(T owner)`:Gets a `LoaderManager` associated with the given owner, such as an `Activity` or `Fragment`.
- `getLoader(int id)`:Return the `Loader` with the given `id` or `null` if no matching Loader is found.
- `initLoader (int id, Bundle args, LoaderCallbacks<D> callback)`: Ensures a loader is initialized and active. If the loader doesn't already exist, one is created and (if the activity/fragment is currently started) starts the loader. Otherwise the last created loader is re-used.

Implementing an `AsyncTaskLoader` (V)

- `LoaderCallbacks<D> callback`: Callback interface for a client to interact with the manager.
- `onCreateLoader (int id, Bundle args)`: Instantiate and return a new `Loader` for the given ID.
- `onLoadFinished(Loader<D> loader, D data)`: Called when a previously created loader has finished its load.
- `loadInBackground()`:Called on a worker thread to perform the actual load and to return the result of the load operation.
- `forceLoad()`: Force an asynchronous load.
- `startLoading()`:
  - This function will normally be called for you automatically by `LoaderManager` when the associated fragment/activity is being started.
  - When using a `Loader` with `LoaderManager`, you must not call this method yourself, or you will conflict with its management of the `Loader`.

Bibliography

Resources

- "Mastering Android Application Development", by Antonio Pachon Rui, 2015
- `https://developer.android.com/index.html`
- `http://simple.sourceforge.net/home.php` `http://simple.sourceforge.net/download/stream/doc/tutorial/tutorial.php`