Android - Persistent Storage

Activity Lifecycle

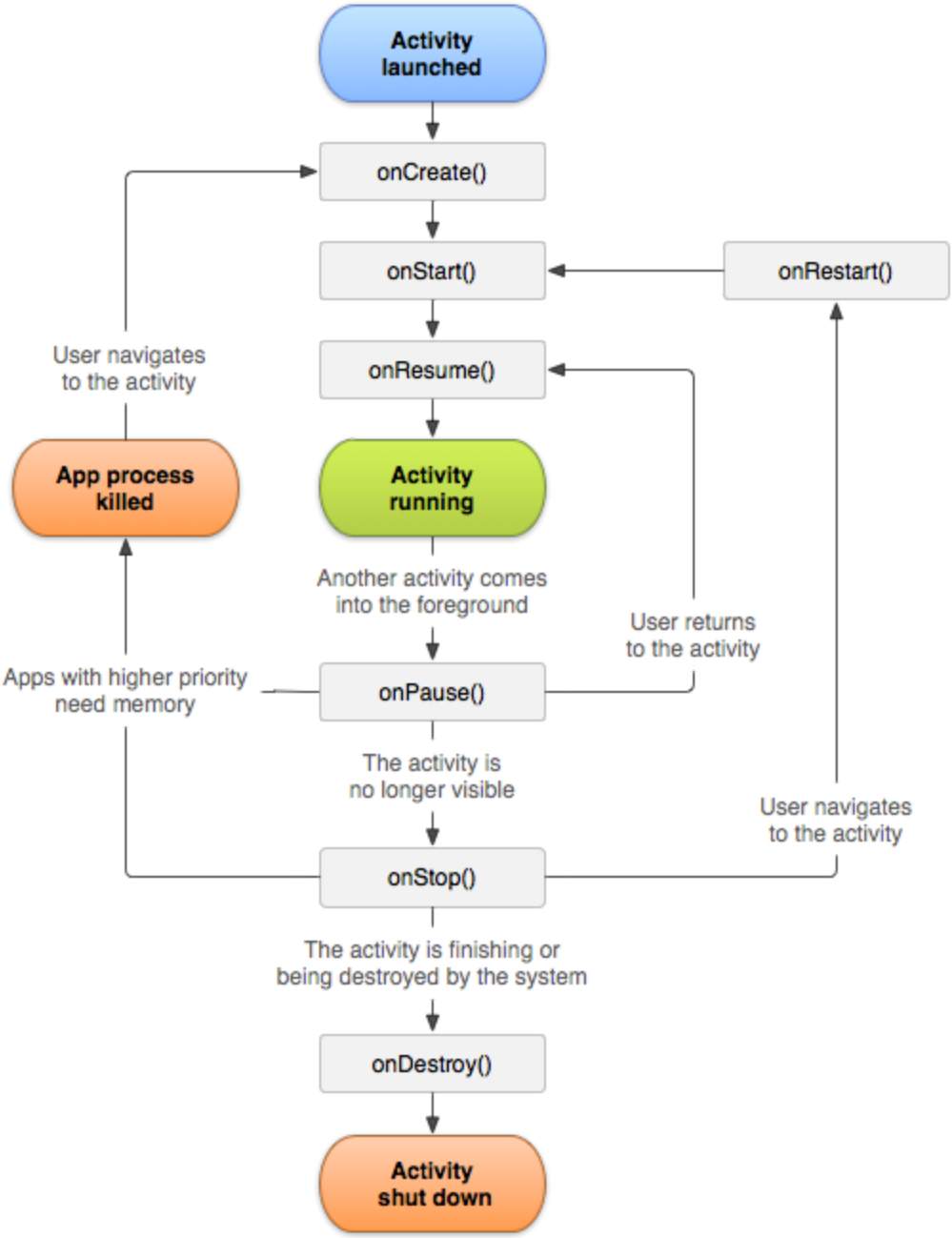
Activities in the system are managed as an *activity stack*. When a new activity is started, it is placed on the top of the stack and becomes the running activity -- the previous activity always remains below it in the stack, and will not come to the foreground again until the new activity exits.

An activity has essentially four states:

- If an activity in the foreground of the screen (at the top of the stack), it is *active* or ***running***.
- If an activity has lost focus but is still visible (that is, a new non-full-sized or transparent activity has focus on top of your activity), it is ***paused***. A paused activity is completely alive (it maintains all state and member information and remains attached to the window manager), but can be killed by the system in extreme low memory situations.
- If an activity is completely obscured by another activity, it is ***stopped***. It still retains all state and member information, however, it is no longer visible to the user so its window is hidden and it will often be killed by the system when memory is needed elsewhere.
- If an activity is paused or stopped, **the system can drop the activity from memory** by either asking it to finish, or simply killing its process. When it is displayed again to the user, it must be completely restarted and restored to its previous state.

The following diagram shows the important state paths of an Activity. The square rectangles represent callback methods you can implement to perform operations when the Activity moves between states. The colored ovals are major states the Activity can be in.

Written By: Eran Katsav - All Rights Reserved ©



There are three key loops you may be interested in monitoring within your activity:

- The **entire lifetime** of an activity happens between the first call to `onCreate(Bundle)` through to a single final call to `onDestroy()`.
- The **visible lifetime** of an activity happens between a call to `onStart()` until a corresponding call to `onStop()`. During this time the user can see the activity on-screen,

Written By: Eran Katsav - All Rights Reserved ©

though it may not be in the foreground and interacting with the user. Between these two methods you can maintain resources that are needed to show the activity to the user.

- The **foreground lifetime** of an activity happens between a call to `onResume()` until a corresponding call to `onPause()`. During this time the activity is in front of all other activities and interacting with the user. An activity can frequently go between the resumed and paused states. For example when a dialog indicating a new message arrived, a call received, or any other window that is in the foreground even if it is not fully hides out activity. **In other words: When another window hide even a part of our activity the function onPause is called and when our activity receive the full from onResume will called - This functions are the best places to save user info to persistent storage. Note : When overriding each function it is very important to call super first.**

Persistent storage - options

As we have seen, when the android system called the onDestroy() function all the app memory is deallocated and its resources are freed. So if we need to save some information across the user session we need to use the persistent storage option. Android provides several options for you to save persistent application data. The solution you choose depends on your specific needs:

- Shared Preferences - Store private primitive data in key-value pairs.
- Internal Storage - Store private data on the device memory, this storage is designed to be the app “sandbox”, it’s private to your app and will be deleted when the user uninstall your app.
- External Storage - Store public data on the shared external storage this information can be shared with other apps and can be saved even after your app is deleted.

shared preference

The `SharedPreferences` class provides you the easiest way of saving data to the device memory, the data will be saved while you app is installed on the device. We save all the information with the key-value bundle(hash Table) we have seen before. the Data saved will allow us to save information

Written By: Eran Katsav - All Rights Reserved ©

needed for the app even if onDestroy() is called the the app is no longer “lives” in the device memory. You can use SharedPreferences to save any primitive data: booleans, floats, ints, longs, and strings. **This data will persist across user sessions (even if your application is killed)**.

To get a SharedPreferences object for your application, use one of two methods:

- PreferenceManager.getDefaultSharedPreferences(context) - this option will return a SharedPreference instance same to all of your app. The filename is fixed and compose of the package name with _prefrence ending. This is a very easy option to use but lacks flexibility and return the same file all the time. Use this option when retrieving data saved with the PrefrenceFragment or PreferenceActivity.
- getSharedPreferences() - Use this if you need multiple preferences files identified by name, which you specify with the first parameter (use a constant). Use this method to get a preference file to be used across all activities, meaning a file that is accessed from anywhere in your app. for example:
  sp = getSharedPreferences(“fileName”,Activity.MODE_PRIVATE);
- getPreferences() - Use this if you need only one preferences file for your **Activity**. Because this will be the only preferences file for your Activity, you don't supply a name. For example: SharedPreferences sp = getPreferences(Activity.MODE_PRIVATE);

To write values:

1. Call edit() to get a SharedPreferences.Editor.
2. Add values with methods such as putBoolean() and putString().
3. Commit the new values with commit()

To read values, use SharedPreferences methods such as getBoolean() and getString().

example for using the “.edit” function:

```
SharedPreferences.Editor editor = sp.edit();
editor.putString(“keyname”,anyString);
editor.commit();
```

To load this information you must use the function “.get” with the key and a default value of your choosing

Written By: Eran Katsav - All Rights Reserved ©

example to get the information (assuming sp is already referencing the shared preference):

String name = sp.getString(“keyname”,null);

Creating files in the internal storage of device

You can save files directly on the device's internal storage. By default, files saved to the internal storage are private to your application and other applications cannot access them (nor can the user). When the user uninstalls your application, these files are removed.

To create and write a private file to the internal storage:

1. Call `openFileOutput()` with the name of the file and the operating mode. This returns a `FileOutputStream`.
2. Write to the file with `write()`.
3. Close the stream with `close()`.

First you must create a “FileOutputStream” with a static function the allows you to immediately input the directory of the file and the file will open, this is called “openFileOutput()” which needs to receive the files name and the mode that the file will open and manipulate the file. Example: fileOutputStream = openFileOutput(“details.txt”, MODE_APPEND);

Note The “MODE_APPEND” tells that whenever something is written inside the file it doesn’t overwrite what has already been written(don’t delete the file content with the new content but append it). other modes can be `MODE_PRIVATE` will create the file (or replace a file of the same name) and make it private to your application or `MODE_WORLD_READABLE`, and `MODE_WORLD_WRITEABLE` that will make the file visible to other apps.

After creating the “FileOutputStream” use the “.write” function to input information into the file and don’t forget to close() the file after doing that.

Written By: Eran Katsav - All Rights Reserved ©

write & getBytes

If you want to save a string to the file you can use the “str.getBytes()” for the string to be changed into an array of bytes. If you don’t use this you need to use the “File Output stream writer” to receive the string properly. After the “getBytes” function but still inside the “.write” function you must add from where you want the string to be written inside the byte array and until where. Example: fileOutputStream.write(str.getBytes(),0,str.getBytes().length);

Reading from files previously created

you must create the “FileInputStream” and use the static function “openFileInput” with the name of the file that you would like to access. Then you must create variables of the information that you would like to use from the file (mostly will be a byte array that will also have a predefined amount buffer), after that use the “.read()” function and implement the previously created variable.

To read a file from internal storage:

1. Call `openFileInput()` and pass it the name of the file to read. This returns a `FileInputStream`.
2. Read bytes from the file with `read()`.
3. Then close the stream with `close()`.

how to save custom objects into a file

To save an object it must implement the class “Serializable”. inside the a class that you would like to save the file you can also want create two function inside the class “writeToFile” and “readFromFile” these functions will define how the object will be saved and read. These functions are not mandatory at all but rather nice object oriented programing design.

writeToFile

the “writeToFile” will get a context and a string that will be the file name.

Written By: Eran Katsav - All Rights Reserved ©

inside the function use the “FileOutputStream” and “ObjectOutputStream”. First get an fileOutputStream with the static methods we have seen before, after that open an objectOutputStream chained to the fileOutputStream instance and then use the writeObject function of the objectOutputStream instance.

readFromFile

resembles the “writeToFile” just instead of the output streams we use the input streams.

Note: When saving an object he and **all of it’s class members must** implements the Serializable interface. by default String class implements this interface. If from any reason part of his members don’t implements this interface you must use the **transient** keyword to exclude this members from being saved to the file.
When doing so Java has a special mechanism to manually saving and reading custom objects this is done by creating the functions writeObject and readObject which will be called by the JVM when saving and reading the object and it pases the objectInputStream or objectOutputStream accordingly.
There you can perform the custom fields (the transient fields) spacial save or read with the and later on call defaultWriteObject or defaultReadObject to let the system handle the rest of the class members saving. This is not an overriding mechanism but a special mechanism designed for that particular problem solving.

how to save image bitmap into file

When using a Bitmap object there are some very handy and easy-to-use methods designed to save these object into the internal storage.
compress function - each Bitmap instance has a compress method that requires the result output format, the compression ratio and the fileOutputStream instance. for example
compress(Bitmap.CompressFormat.*JPEG*,100,fos);

to access the image save you need to use the function bitmapFactory

Written By: Eran Katsav - All Rights Reserved ©

bitmapFactory

is a class for decoding the encoded image file you are transferring. All you need to input after this function is the ”.decodeStream()” function and inside that function input the filename that possess the image. for example: BitmapFactory.*decodeStream*(fis) where fis is the fileInputStream instance.

External Storage Vs Internal Storage

All Android devices have two file storage areas: "internal" and "external" storage. These names come from the early days of Android, when most devices offered built-in non-volatile memory (internal storage), plus a removable storage medium such as a micro SD card (external storage). Today the external storage can still be stored on a non removable SD card and still be called “external” the difference between them is in only in the API and not on the hardware. You should choose one of them according to your needs:

Internal storage:

- It's always available.
- Files saved here are accessible by only your app by default.
- When the user uninstalls your app, the system removes all your app's files from internal storage.

Internal storage is best when you want to be sure that neither the user nor other apps can access your files.

External storage:

- It's not always available, because the user can mount the external storage as USB storage and in some cases remove it from the device.
- It's world-readable, so files saved here may be read outside of your control.

Written By: Eran Katsav - All Rights Reserved ©

- When the user uninstalls your app, the system removes your app's files from here only if you save them in the directory from `getExternalFilesDir()`.

External storage is the best place for files that don't require access restrictions and for files that you want to share with other apps or allow the user to access with a computer.

To read and write from external storage you should request the necessary permissions in the manifest file:

```
<uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE" />

<uses-permission android:name="android.permission.READ_EXTERNAL_STORAGE" />
```

Note: there is no need to request the read permission if you already requested the write permission, it includes it.(Also in Android 4.4 and later there is no need to ask for those permissions if you are writing in the external storage private folders).

External storage - Public files

Files that should be freely available to other apps and to the user. When the user uninstalls your app, these files should remain available to the user.
For example, photos captured by your app or other downloaded files.

External storage - Private files

Files that rightfully belong to your app and should be deleted when the user uninstalls your app. When the user uninstalls your app, the system deletes all files in your app's external private directory.
For example, additional resources downloaded by your app or temporary media files.

If you want to save public files on the external storage, use the `getExternalStoragePublicDirectory()`method to get a `File` representing the appropriate directory on the external storage. The method takes an argument specifying the type of file you want to save so that they can be logically organized with other public files, such as `DIRECTORY_MUSIC` or `DIRECTORY_PICTURES`.

Written By: Eran Katsav - All Rights Reserved ©

For example: `File file = new File(Environment.getExternalStoragePublicDirectory (Environment.DIRECTORY_PICTURES), albumName);`

If you want to save files that are private to your app, you can acquire the appropriate directory by calling `getExternalFilesDir()` and passing it a name indicating the type of directory you'd like. Each directory created this way is added to a parent directory that encapsulates all your app's external storage files, which the system deletes when the user uninstalls your app.
For example, here's a method you can use to create a directory for an individual photo album:

```
File file = new File(context.getExternalFilesDir(Environment.DIRECTORY_PICTURES), albumName);
```

Regardless of whether you use `getExternalStoragePublicDirectory()` for files that are shared or `getExternalFilesDir()` for files that are private to your app, it's important that you use directory names provided by API constants like DIRECTORY_PICTURES. These directory names ensure that the files are treated properly by the system. For instance, files saved in DIRECTORY_RINGTONES are categorized by the system media scanner as ringtones instead of music.

The external storage general path can be obtained by :

Environment.getExternalStorageDirectory()

References

Android developer guide: Activity and Saving files by Google ©

Written By: Eran Katsav - All Rights Reserved ©