How to declare global variables in Android?

Asked 15 years, 8 months ago Modified 7 years, 11 months ago Viewed 312k times

Part of Mobile Development Collective

608

I am creating an application which requires login. I created the main and the login activity.

In the main activity `onCreate` method I added the following condition:

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

    ...

    loadSettings();
    if(strSessionString == null)
    {
        login();
    }
    ...
}
```

The `onActivityResult` method which is executed when the login form terminates looks like this:

```
@Override
public void onActivityResult(int requestCode,
                             int resultCode,
                             Intent data)
{
    super.onActivityResult(requestCode, resultCode, data);
    switch(requestCode)
    {
        case(SHOW_SUBACTICITY_LOGIN):
        {
            if(resultCode == Activity.RESULT_OK)
            {

                strSessionString = data.getStringExtra(Login.SESSIONSTRING);
                connectionAvailable = true;
                strUsername = data.getStringExtra(Login.USERNAME);
            }
        }
    }
```

The problem is the login form sometimes appears twice (the `login()` method is called twice) and also when the phone keyboard slides the login form appears again and I guess the problem is the variable `strSessionString`.

Does anyone know how to set the variable global in order to avoid login form appearing after the user already successfully authenticates?

MD android singleton global-variables state

Share

Improve this question

Follow

edited Jan 3, 2017 at 13:55

Willi Mentzel

29.7k ● 21 ● 118 ● 126

asked Apr 2, 2009 at 1:54

Niko Gamulin

66.5k ● 96 ● 228 ● 293

a good tutorial on how to handle an activity state using saved instance state bundle quicktips.in/… – Deepak Swami Aug 25, 2015 at 2:45

17 Answers

Sorted by: Highest score (default)

968

I wrote this answer back in '09 when Android was relatively new, and there were many not well established areas in Android development. I have added a long addendum at the bottom of this post, addressing some criticism, and detailing a philosophical disagreement I have with the use of Singletons rather than subclassing Application. Read it at your own risk.

ORIGINAL ANSWER:

The more general problem you are encountering is how to save state across several Activities and all parts of your application. A static variable (for instance, a singleton) is a common Java way of achieving this. I have found however, that a more elegant way in Android is to associate your state with the Application context.

As you know, each Activity is also a Context, which is information about its execution environment in the broadest sense. Your application also has a context, and Android guarantees that it will exist as a single instance across your application.

The way to do this is to create your own subclass of android.app.Application, and then specify that class in the application tag in your manifest. Now Android will automatically create an instance of that class and make it available for your entire application. You can access it from any `context` using the `Context.getApplicationContext()` method (`Activity` also provides a method `getApplication()` which has the exact same effect). Following is an extremely simplified example, with caveats to follow:

```
class MyApp extends Application {

  private String myState;
```

```
  public String getState(){
    return myState;
  }
  public void setState(String s){
    myState = s;
  }
}

class Blah extends Activity {

  @Override
  public void onCreate(Bundle b){
    ...
    MyApp appState = ((MyApp)getApplicationContext());
    String state = appState.getState();
    ...
  }
}
```

This has essentially the same effect as using a static variable or singleton, but integrates quite well into the existing Android framework. Note that this will not work across processes (should your app be one of the rare ones that has multiple processes).

Something to note from the example above; suppose we had instead done something like:

```
class MyApp extends Application {

  private String myState = /* complicated and slow initialization */;

  public String getState(){
    return myState;
  }
}
```

Now this slow initialization (such as hitting disk, hitting network, anything blocking, etc) will be performed every time Application is instantiated! You may think, well, this is only once for the process and I'll have to pay the cost anyways, right? For instance, as Dianne Hackborn mentions below, it is entirely possible for your process to be instantiated -just- to handle a background broadcast event. If your broadcast processing has no need for this state you have potentially just done a whole series of complicated and slow operations for nothing. Lazy instantiation is the name of the game here. The following is a slightly more complicated way of using Application which makes more sense for anything but the simplest of uses:

```
class MyApp extends Application {

  private MyStateManager myStateManager = new MyStateManager();

  public MyStateManager getStateManager(){
```

```
    return myStateManager ;
  }
}

class MyStateManager {

  MyStateManager() {
    /* this should be fast */
  }

  String getState() {
    /* if necessary, perform blocking calls here */
    /* make sure to deal with any multithreading/synchronicity issues */

    ...

    return state;
  }
}

class Blah extends Activity {

  @Override
  public void onCreate(Bundle b){
    ...
    MyStateManager stateManager =
((MyApp)getApplicationContext()).getStateManager();
    String state = stateManager.getState();
    ...
  }
}
```

While I prefer Application subclassing to using singletons here as the more elegant solution, I would rather developers use singletons if really necessary over not thinking at all through the performance and multithreading implications of associating state with the Application subclass.

NOTE 1: Also as anticafe commented, in order to correctly tie your Application override to your application a tag is necessary in the manifest file. Again, see the Android docs for more info. An example:

```
<application
     android:name="my.application.MyApp"
     android:icon="..."
     android:label="...">
</application>
```

NOTE 2: user608578 asks below how this works with managing native object lifecycles. I am not up to speed on using native code with Android in the slightest, and I am not qualified to answer how that would interact with my solution. If someone does have an answer to this, I am willing to credit them and put the information in this post for maximum visibility.

ADDENDUM:

As some people have noted, this is **not** a solution for **persistent** state, something I perhaps should have emphasized more in the original answer. I.e. this is not meant to be a solution for saving user or other information that is meant to be persisted across application lifetimes. Thus, I consider most criticism below related to Applications being killed at any time, etc..., moot, as anything that ever needed to be persisted to disk should not be stored through an Application subclass. It is meant to be a solution for storing temporary, easily re-creatable application state (whether a user is logged in for example) and components which are single instance (application network manager for example) (**NOT** singleton!) in nature.

Dayerman has been kind enough to point out an interesting conversation with Reto Meier and Dianne Hackborn in which use of Application subclasses is discouraged in favor of Singleton patterns. Somatik also pointed out something of this nature earlier, although I didn't see it at the time. Because of Reto and Dianne's roles in maintaining the Android platform, I cannot in good faith recommend ignoring their advice. What they say, goes. I do wish to disagree with the opinions, expressed with regards to preferring Singleton over Application subclasses. In my disagreement I will be making use of concepts best explained in this StackExchange explanation of the Singleton design pattern, so that I do not have to define terms in this answer. I highly encourage skimming the link before continuing. Point by point:

Dianne states, "There is no reason to subclass from Application. It is no different than making a singleton..." This first claim is incorrect. There are two main reasons for this. 1) The Application class provides a better lifetime guarantee for an application developer; it is guaranteed to have the lifetime of the application. A singleton is not EXPLICITLY tied to the lifetime of the application (although it is effectively). This may be a non-issue for your average application developer, but I would argue this is exactly the type of contract the Android API should be offering, and it provides much more flexibility to the Android system as well, by minimizing the lifetime of associated data. 2) The Application class provides the application developer with a single instance holder for state, which is very different from a Singleton holder of state. For a list of the differences, see the Singleton explanation link above.

Dianne continues, "...just likely to be something you regret in the future as you find your Application object becoming this big tangled mess of what should be independent application logic." This is certainly not incorrect, but this is not a reason for choosing Singleton over Application subclass. None of Diane's arguments provide a reason that using a Singleton is better than an Application subclass, all she attempts to establish is that using a Singleton is no worse than an Application subclass, which I believe is false.

She continues, "And this leads more naturally to how you should be managing these things -- initializing them on demand." This ignores the fact that there is no reason you cannot initialize on demand using an Application subclass as well. Again there is no difference.

Dianne ends with "The framework itself has tons and tons of singletons for all the little shared data it maintains for the app, such as caches of loaded resources, pools of objects, etc. It works great." I am not arguing that using Singletons cannot work fine or are not a legitimate alternative. I am arguing that Singletons do not provide as strong a contract with the Android system as using an Application subclass, and further that using Singletons generally points to inflexible design, which is not easily modified, and leads to many problems down the road. IMHO, the strong contract the Android API offers to developer applications is one of the most appealing and pleasing aspects of programming with Android, and helped lead to early developer adoption which drove the Android platform to the success it has today. Suggesting using Singletons is implicitly moving away from a strong API contract, and in my opinion, weakens the Android framework.

Dianne has commented below as well, mentioning an additional downside to using Application subclasses, they may encourage or make it easier to write less performance code. This is very true, and I have edited this answer to emphasize the importance of considering perf here, and taking the correct approach if you're using Application subclassing. As Dianne states, it is important to remember that your Application class will be instantiated every time your process is loaded (could be multiple times at once if your application runs in multiple processes!) even if the process is only being loaded for a background broadcast event. It is therefore important to use the Application class more as a repository for pointers to shared components of your application rather than as a place to do any processing!

I leave you with the following list of downsides to Singletons, as stolen from the earlier StackExchange link:

- Inability to use abstract or interface classes;
- Inability to subclass;
- High coupling across the application (difficult to modify);
- Difficult to test (can't fake/mock in unit tests);
- Difficult to parallelize in the case of mutable state (requires extensive locking);

and add my own:

- Unclear and unmanageable lifetime contract unsuited for Android (or most other) development;

Share

Improve this answer

Follow

edited Apr 12, 2017 at 7:31

Community Bot

1 • 1

answered Apr 2, 2009 at 4:34

sooniln

14.6k • 4 • 31 • 35

5 For anyone wondering how to "specify that class in the application tag in your manifest", there are, as of this writing, two other answers for this question that describe how to do it (use android:name), one by ebuprofen and one by Mike Brown. – Tyler Collier Nov 11, 2010 at 17:09

9 Soonil, your answer is right, but could you notice that we should add <application android:name=".MyApp" ... /> into Android Manifest file? – anticafe Feb 19, 2011 at 23:40

13 Let me repeat one more time, you should not be using Application for globals. It is of no use, gives no benefits over singletons, and can be actively harmful, such as harming the performance of launching your process. At the time Application is being created, you have no idea what your process is being created for. By lazily initializing singletons as needed, you only need to do work that is necessary. For example, if your process is being launched to handle a broadcast about some background event, there is no reason to initialize whatever global state is needed by your UI. – hackbod Apr 8, 2014 at 15:18

5 On top of that, if your application is using multiple processes, an Application object means you need to do all of that global initialization (time and memory consumption) in all of them. Ouch. And there are certain situations where your Application object will not be created, in particular during a restore, that can trip you up. – hackbod Apr 8, 2014 at 15:20

15 Also, let's be really clear here -- all of your arguments against singletons are perfectly valid, when we are talking about situations where you are actually choosing between a singleton and another approach that isn't a global; singletons are globals, with all the caveats about globals that apply. However, *Application is also a singleton*. You aren't escaping those problems by switching to subclassing Application, an Application is exactly the same as a singleton (but worse), it is just letting you to trick yourself that you are doing something more clean. But you aren't. – hackbod Apr 8, 2014 at 15:31

155

Create this subclass

```
public class MyApp extends Application {
  String foo;
}
```

In the AndroidManifest.xml add android:name

Example

```
<application android:name=".MyApp"
       android:icon="@drawable/icon"
       android:label="@string/app_name">
```

Share

Improve this answer

Follow

edited Jun 22, 2012 at 10:06

Nikhil

16.2k ● 20 ● 65 ● 81

answered Jul 26, 2010 at 20:31

Guillaume

1,670 ● 1 ● 10 ● 8

3 For it to work for me I had to remove the "." within ".MyApp" – Someone Somewhere May 5, 2011 at 0:07

3 just declare it *after* the main activity, otherwise it may not install/deploy – sami Aug 29, 2011 at 10:36

11 just wanna say, this goes in the MAIN application tag that is already there... this isnt a second one :) had to learn the hard way. – bwoogie Oct 14, 2011 at 1:04

`java.lang.IllegalAccessException: access to class is not allowed` – Raptor Jan 10, 2014 at 10:22

142

The suggested by Soonil way of keeping a state for the application is good, however it has one weak point - there are cases when OS kills the entire application process. Here is the documentation on this - Processes and lifecycles.

Consider a case - your app goes into the background because somebody is calling you (Phone app is in the foreground now). In this case && under some other conditions (check the above link for what they could be) the OS may kill your application process, including the `Application` subclass instance. As a result the state is lost. When you later return to the application, then the OS will restore its activity stack and `Application` subclass instance, however the `myState` field will be `null`.

AFAIK, the only way to guarantee state safety is to use any sort of persisting the state, e.g. using a private for the application file or `SharedPrefernces` (it eventually uses a private for the application file in the internal filesystem).

Share

Improve this answer

Follow

edited Oct 25, 2011 at 19:42

dvd

1,470 ● 1 ● 16 ● 24

answered Jan 9, 2011 at 21:49

Vit Khudenko

28.4k ● 10 ● 65 ● 93

10 +1 for persisting with `SharedPreferences`; this is how I've seen it done. I do find it strange to abuse the preference system for saved state, but it works so well that the issue becomes just a question of terminology. – Cheezmeister Feb 22, 2011 at 17:07

1 could you please post the code (or provide a link to an explanation) as to how SharedPreferences is used to solve the problem that Arhimed describes – Someone Somewhere May 4, 2011 at 23:47

2 Preferences, database, file serialization, etc. Each activity can maintain state if they use the onSaveInstanceState but it won't help if the user backs out of the activity and which removes

it from the history stack, force closes, or turns off their device. – Darren Hinderer Jun 23, 2011 at 16:10

1 This behaviour is very annoying - it wouldn't be so bad if the onTerminate() method of your application was called so you could deal with the situation elegantly. – Dean Wild Jun 28, 2012 at 14:30

2 This is the correct answer in my opinion. It's a bug to rely on the same application instance existing across activities. In my experience it's quite common for Android to completely tear down and recreate your entire process while you are backgrounded. Being backgrounded could just mean launching a camera intent, browser intent or receiving a phone call. – Jared Kells Apr 19, 2013 at 3:51

25

Just a note ..

add:

```
android:name=".Globals"
```

or whatever you named your subclass to the **existing** `<application>` tag. I kept trying to add another `<application>` tag to the manifest and would get an exception.

Share Improve this answer Follow

answered Apr 13, 2011 at 21:29

Gimbl
1,540 ● 1 ● 13 ● 23

Hi, Gimbl. I had the same problem. I also had my own <application> tag and, when I try to add another <application> tag I had the same problem as you (exception message). But I did what you mentioned, and it didn't work. I add android:name=".GlobalClass" to my <application> tag but it doesn't work. Can you fully explain how you solved it?? – Sonhja Sep 23, 2011 at 10:35

3 **Good** <manifest> <application android:name=".GlobalData"> </application></manifest>. **Bad** <manifest><application></application> <application android:name=".GlobalData"> </application> </manifest> – Gimbl Sep 26, 2011 at 15:16

13

What about ensuring the collection of native memory with such global structures?

Activities have an `onPause/onDestroy()` method that's called upon destruction, but the Application class has no equivalents. What mechanism are recommended to ensure that global structures (especially those containing references to native memory) are garbage collected appropriately when the application is either killed or the task stack is put in the background?

Share edited Jun 22, 2012 at 10:06 answered Feb 25, 2011 at 18:32

Improve this answer

Follow

Nikhil
16.2k ● 20 ● 65 ● 81

user608578
231 ● 2 ● 5

1 The obvious solution is to implement the Closeable interface for your objects responsible for native resources and ensure they are managed by a try-with-resources statement or something else. Worst case you can always use an object finalizer. – sooniln May 14, 2014 at 3:30

12

I couldn't find how to specify the application tag either, but after a lot of Googling, it became obvious from the manifest file docs: use android:name, in addition to the default icon and label in the application stanza.

android:name The fully qualified name of an Application subclass implemented for the application. When the application process is started, this class is instantiated before any of the application's components.

The subclass is optional; most applications won't need one. In the absence of a subclass, Android uses an instance of the base Application class.

Share Improve this answer Follow

answered Jan 14, 2010 at 5:26

Mike Brown
121 ● 1 ● 2

5

Just you need to define an application name like below which will work:

```
<application
  android:name="ApplicationName" android:icon="@drawable/icon">
</application>
```

Share

Improve this answer

Follow

edited Aug 24, 2015 at 5:53

kenju
5,924 ● 1 ● 43 ● 41

answered Jan 7, 2011 at 5:57

Anand
1,325 ● 1 ● 12 ● 18

4

Like there was discussed above OS could kill the APPLICATION without any notification (there is no onDestroy event) so there is no way to save these global variables.

SharedPreferences could be a solution EXCEPT you have COMPLEX STRUCTURED variables (in my case I had integer array to store the IDs that the user has already handled). The problem with the SharedPreferences is that it is hard to store and retrieve these structures each time the values needed.

In my case I had a background SERVICE so I could move this variables to there and because the service has onDestroy event, I could save those values easily.

Share Improve this answer Follow

answered Nov 2, 2012 at 8:53

Adorjan Princz
11.8k ● 3 ● 36 ● 25

onDestroy() is not guaranteed to be called even for a service. – Learn OpenGL ES Jul 25, 2013 at 5:07

Yes, this could happen but only in case of critical situations. – Adorjan Princz Jul 26, 2013 at 9:51

4

If some variables are stored in sqlite and you must use them in most activities in your app. then Application maybe the best way to achieve it. Query the variables from database when application started and store them in a field. Then you can use these variables in your activities.

So find the right way, and there is no best way.

Share

Improve this answer

Follow

edited Aug 24, 2015 at 6:02

kenju
5,924 ● 1 ● 43 ● 41

answered Apr 20, 2011 at 7:32

user716653
41 ● 2

3

You can have a static field to store this kind of state. Or put it to the resource Bundle and restore from there on onCreate(Bundle savedInstanceState). Just make sure you entirely understand Android app managed lifecycle (e.g. why login() gets called on keyboard orientation change).

Share Improve this answer Follow

answered Apr 2, 2009 at 2:19

yanchenko
57.1k ● 33 ● 149 ● 165

2

DO N'T Use another `<application>` tag in manifest file.Just do one change in existing `<application>` tag , add this line `android:name=".ApplicationName"` where, `ApplicationName` will be name of your subclass(use to store global) that, you is about to create.

so, finally your **ONE AND ONLY** `<application>` tag in manifest file should look like this :-

```
<application
        android:allowBackup="true"
        android:icon="@mipmap/ic_launcher"
        android:label="@string/app_name"
        android:theme="@style/Theme.AppCompat.NoActionBar"
        android:name=".ApplicationName"
        >
```

Share Improve this answer Follow

answered Jun 22, 2015 at 11:20

kumar kundan

2,057 ● 1 ● 29 ● 42

1

you can use Intents , Sqlite , or Shared Preferences . When it comes to the media storage, like documents , photos , and videos, you may create the new files instead.

Share Improve this answer Follow

answered Jul 16, 2013 at 1:35

Jeff Bootsholz

3,019 ● 15 ● 73 ● 147

1

You can do this using two approaches:

1. Using Application class
2. Using Shared Preferences
3. Using Application class

Example:

```
class SessionManager extends Application{

  String sessionKey;

  setSessionKey(String key){
    this.sessionKey=key;
```

```
  }

  String getSessisonKey(){
    return this.sessionKey;
  }
}
```

You can use above class to implement login in your MainActivity as below. Code will look something like this:

```
@override
public void onCreate (Bundle savedInstanceState){
  // you will this key when first time login is successful.
  SessionManager session= (SessionManager)getApplicationContext();
  String key=getSessisonKey.getKey();
  //Use this key to identify whether session is alive or not.
}
```

This method will work for temporary storage. You really do not any idea when operating system is gonna kill the application, because of low memory. When your application is in background and user is navigating through other application which demands more memory to run, then your application will be killed since operating system given more priority to foreground processes than background. Hence your application object will be null before user logs out. Hence for this I recommend to use second method Specified above.

2. Using shared preferences.

```
String MYPREF="com.your.application.session"

SharedPreferences pref= context.getSharedPreferences(MyPREF,MODE_PRIVATE);

//Insert key as below:

Editot editor= pref.edit();

editor.putString("key","value");

editor.commit();

//Get key as below.

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

String key= getResources().getString("key");
```

Share

Improve this answer

Follow

edited May 18, 2016 at 8:15

bg17aw

2,988 ● 1 ● 22 ● 27

answered Dec 19, 2015 at 13:03

Krishna Satwaji Khandagale

2,260 ● 17 ● 25

0

On activity result is called before on resume. So move you login check to on resume and your second login can be blocked once the secomd activity has returned a positive result. On resume is called every time so there is not worries of it not being called the first time.

Share Improve this answer Follow

answered Oct 11, 2014 at 7:36

user3044482
420 ● 4 ● 12

0

The approach of subclassing has also been used by the BARACUS framework. From my point of view **subclassing** Application was intended to work with the lifecycles of Android; this is what **any** Application Container does. Instead of having globals then, I register beans to this context an let them beeing injected into any class manageable by the context. Every injected bean instance actually is a singleton.

See this example for details

Why do manual work if you can have so much more?

Share Improve this answer Follow

answered Nov 11, 2014 at 13:01

gorefest
884 ● 8 ● 22

0

```
class GlobaleVariableDemo extends Application {

    private String myGlobalState;

    public String getGlobalState(){
     return myGlobalState;
    }
    public void setGlobalState(String s){
     myGlobalState = s;
    }
}

class Demo extends Activity {

@Override
public void onCreate(Bundle b){
    ...
    GlobaleVariableDemo appState =
((GlobaleVariableDemo)getApplicationContext());
    String state = appState.getGlobalState();
    ...
    }
}
```

Share

Improve this answer

Follow

edited Aug 24, 2015 at 5:35

kenju

5,924 ● 1 ● 43 ● 41

answered Jul 17, 2014 at 10:31

Vaishali Sutariya

5,111 ● 31 ● 32

0

You could create a class that extends `Application` class and then declare your variable as a field of that class and providing getter method for it.

```
public class MyApplication extends Application {
    private String str = "My String";

    synchronized public String getMyString {
        return str;
    }
}
```

And then to access that variable in your Activity, use this:

```
MyApplication application = (MyApplication) getApplication();
String myVar = application.getMyString();
```

Share Improve this answer Follow

answered Nov 28, 2015 at 13:05

Amit Tiwari

3,692 ● 7 ● 36 ● 75

Highly active question. Earn 10 reputation (not counting the association bonus) in order to answer this question. The reputation requirement helps protect this question from spam and non-answer activity.