Persist data when the screen is flipped

November 30, 2017

Contents

1 Introduction

How to persista data when you flip the screen of your phone. In this example we will do it with a TextView.

2 Create the methods

1. Create a key String it will be a constant to identify your bundle.
2. Override onSaveInstanceState.
3. Call super.onSaveInstanceState.
4. Put the text from the TextView in the outState bundle with bundle.putString(key, value).
5. Check in your onCreate() if savedInstanceState is not null and contains your contant, set that text on our TextView.

```
1  public class MainActivity extends AppCompatActivity {
2
3    private static final String TAG =
         MainActivity.class.getSimpleName();
4
5    // COMPLETED (1) Create a key String called
         LIFECYCLE_CALLBACKS_TEXT_KEY
6    /*
7    * This constant String will be used to store the content of the
         TextView used to display the
8    * list of callbacks. The reason we are storing the contents of
         the TextView is so that you can
9    * see the entire set of callbacks as they are called.
10   */
11   private static final String LIFECYCLE_CALLBACKS_TEXT_KEY =
         "callbacks";
12
```

```
13  /* Constant values for the names of each respective lifecycle
        callback */
14  private static final String ON_CREATE = "onCreate";
15  private static final String ON_START = "onStart";
16  private static final String ON_RESUME = "onResume";
17  private static final String ON_PAUSE = "onPause";
18  private static final String ON_STOP = "onStop";
19  private static final String ON_RESTART = "onRestart";
20  private static final String ON_DESTROY = "onDestroy";
21  private static final String ON_SAVE_INSTANCE_STATE =
        "onSaveInstanceState";
22
23  /*
24  * This TextView will contain a running log of every lifecycle
        callback method called from this
25  * Activity. This TextView can be reset to its default state by
        clicking the Button labeled
26  * "Reset Log"
27  */
28  private TextView mLifecycleDisplay;
29
30  /**
31  * Called when the activity is first created. This is where you
        should do all of your normal
32  * static set up: create views, bind data to lists, etc.
33  *
34  * Always followed by onStart().
35  *
36  * @param savedInstanceState The Activity's previously frozen
        state, if there was one.
37  */
38  @Override
39  protected void onCreate(Bundle savedInstanceState) {
40     super.onCreate(savedInstanceState);
41     setContentView(R.layout.activity_main);
42
```

```
43      mLifecycleDisplay = (TextView)
            findViewById(R.id.tv_lifecycle_events_display);
44
45      /*
46      * If savedInstanceState is not null, that means our Activity
            is not being started for the
47      * first time. Even if the savedInstanceState is not null, it
            is smart to check if the
48      * bundle contains the key we are looking for. In our case, the
            key we are looking for maps
49      * to the contents of the TextView that displays our list of
            callbacks. If the bundle
50      * contains that key, we set the contents of the TextView
            accordingly.
51      */
52      if (savedInstanceState != null) {
53          if
                (savedInstanceState.containsKey(LIFECYCLE_CALLBACKS_TEXT_KEY))
                {
54              String allPreviousLifecycleCallbacks = savedInstanceState
55              .getString(LIFECYCLE_CALLBACKS_TEXT_KEY);
56              mLifecycleDisplay.setText(allPreviousLifecycleCallbacks);
57          }
58      }
59
60      logAndAppend(ON_CREATE);
61  }
62
63  /**
64  * Called when the activity is becoming visible to the user.
65  *
66  * Followed by onResume() if the activity comes to the foreground,
        or onStop() if it becomes
67  * hidden.
68  */
69  @Override
```

```
70    protected void onStart() {
71       super.onStart();
72
73       logAndAppend(ON_START);
74    }
75
76    /**
77    * Called when the activity will start interacting with the user.
          At this point your activity
78    * is at the top of the activity stack, with user input going to
          it.
79    *
80    * Always followed by onPause().
81    */
82    @Override
83    protected void onResume() {
84       super.onResume();
85
86       logAndAppend(ON_RESUME);
87    }
88
89    /**
90    * Called when the system is about to start resuming a previous
          activity. This is typically
91    * used to commit unsaved changes to persistent data, stop
          animations and other things that may
92    * be consuming CPU, etc. Implementations of this method must be
          very quick because the next
93    * activity will not be resumed until this method returns.
94    *
95    * Followed by either onResume() if the activity returns back to
          the front, or onStop() if it
96    * becomes invisible to the user.
97    */
98    @Override
99    protected void onPause() {
```

```
100        super.onPause();
101
102        logAndAppend(ON_PAUSE);
103    }
104
105    /**
106    * Called when the activity is no longer visible to the user,
           because another activity has been
107    * resumed and is covering this one. This may happen either
           because a new activity is being
108    * started, an existing one is being brought in front of this one,
           or this one is being
109    * destroyed.
110    *
111    * Followed by either onRestart() if this activity is coming back
           to interact with the user, or
112    * onDestroy() if this activity is going away.
113    */
114    @Override
115    protected void onStop() {
116        super.onStop();
117
118        logAndAppend(ON_STOP);
119    }
120
121    /**
122    * Called after your activity has been stopped, prior to it being
           started again.
123    *
124    * Always followed by onStart()
125    */
126    @Override
127    protected void onRestart() {
128        super.onRestart();
129
130        logAndAppend(ON_RESTART);
```

```
131     }
132
133     /**
134     * The final call you receive before your activity is destroyed.
            This can happen either because
135     * the activity is finishing (someone called finish() on it, or
            because the system is
136     * temporarily destroying this instance of the activity to save
            space. You can distinguish
137     * between these two scenarios with the isFinishing() method.
138     */
139     @Override
140     protected void onDestroy() {
141        super.onDestroy();
142
143        logAndAppend(ON_DESTROY);
144     }
145
146     @Override
147     protected void onSaveInstanceState(Bundle outState) {
148        super.onSaveInstanceState(outState);
149        logAndAppend(ON_SAVE_INSTANCE_STATE);
150        String lifecycleDisplayTextViewContents =
               mLifecycleDisplay.getText().toString();
151        outState.putString(LIFECYCLE_CALLBACKS_TEXT_KEY,
               lifecycleDisplayTextViewContents);
152     }
153
154     /**
155     * Logs to the console and appends the lifecycle method name to
            the TextView so that you can
156     * view the series of method callbacks that are called both from
            the app and from within
157     * Android Studio's Logcat.
158     *
159     * @param lifecycleEvent The name of the event to be logged.
```

```
160     */
161     private void logAndAppend(String lifecycleEvent) {
162         Log.d(TAG, "Lifecycle Event: " + lifecycleEvent);
163 
164         mLifecycleDisplay.append(lifecycleEvent + "\n");
165     }
166 
167     /**
168     * This method resets the contents of the TextView to its default
            text of "Lifecycle callbacks"
169     *
170     * @param view The View that was clicked. In this case, it is the
            Button from our layout.
171     */
172     public void resetLifecycleDisplay(View view) {
173         mLifecycleDisplay.setText("Lifecycle callbacks:\n");
174     }
175 }
```

Android LifeCycle Resource