ANDROID
ACTIVITY LIFECYCLE AND STATES

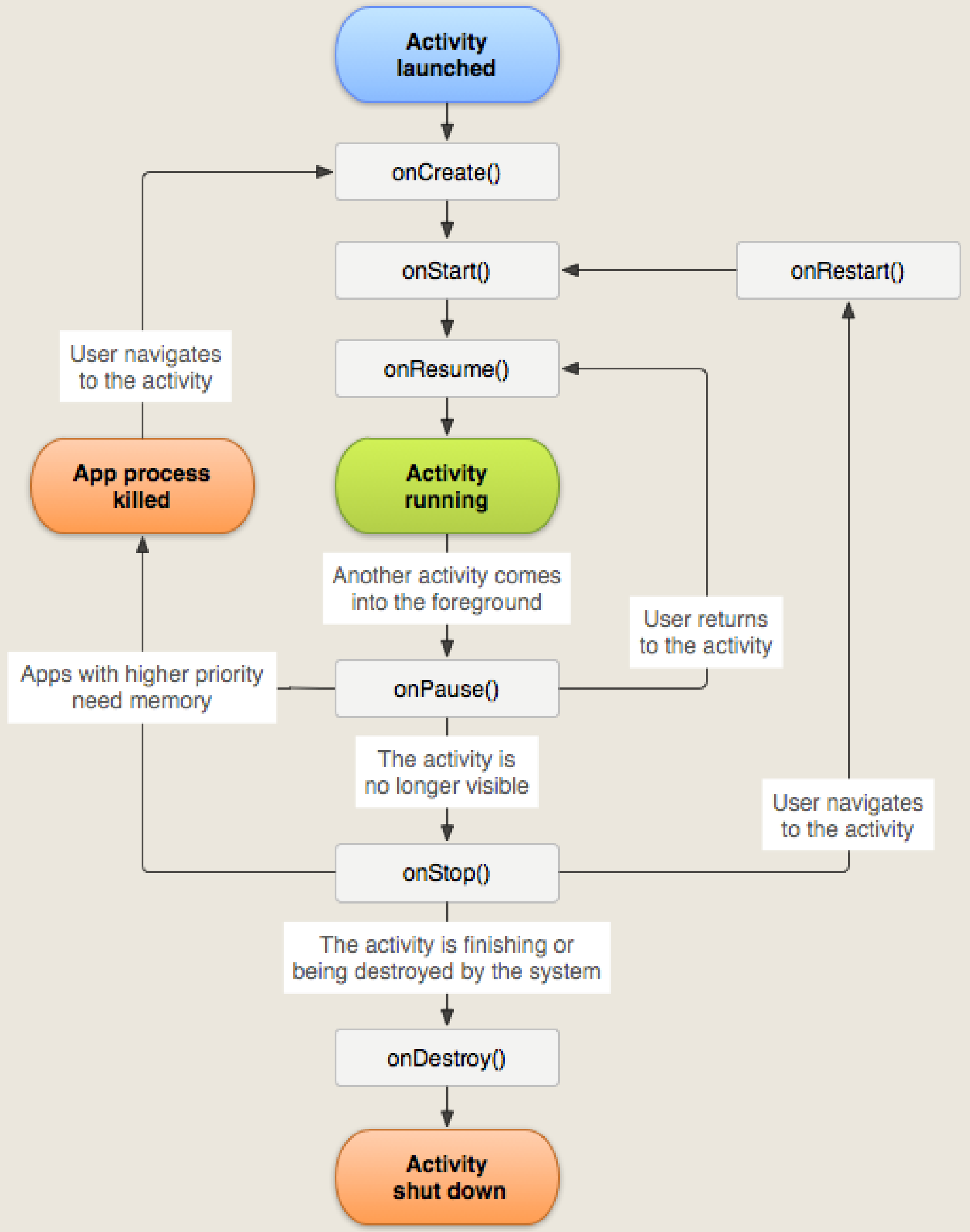
Efthimios Alepis

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
- When a new activity is started, 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

Activity basic states

- If an activity is 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*
- If an activity is completely obscured by another activity, it is *stopped*
- 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


Activity launched
onCreate()
onStart()
onRestart()
User navigates to the activity
onResume()
App process killed
Activity running
Another activity comes into the foreground
User returns to the activity
Apps with higher priority need memory
onPause()
The activity is no longer visible
User navigates to the activity
onStop()
The activity is finishing or being destroyed by the system
onDestroy()
Activity shut down

Analyzing Activity Lifetime

- The **entire lifetime** of an activity happens between the first call to onCreate(Bundle) through to a single final call to onDestroy(). An activity will do all setup of "global" state in onCreate(), and release all remaining resources in 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, 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

Activity Lifetime methods

```
public class Activity extends ApplicationContext {
     protected void onCreate(Bundle savedInstanceState);

     protected void onStart();

     protected void onRestart();

     protected void onResume();

     protected void onPause();

     protected void onStop();

     protected void onDestroy();
 }
```

Important!

- All activities require onCreate() method
- Always call up to your superclass when implementing these methods

onCreate() method

- 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, etc. This method also provides you with a Bundle containing the activity's previously frozen state, if there was one
- Always followed by onStart()

onRestart() method

- Called after your activity has been stopped, prior to it being started again
- Always followed by onStart()

onStart() method

- Called when the activity is becoming visible to the user
- Followed by onResume() if the activity comes to the foreground, or onStop() if it becomes hidden

onResume() method

- Called when the activity will start interacting with the user. At this point your activity is at the top of the activity stack, with user input going to it
- Always followed by onPause()

onPause() method

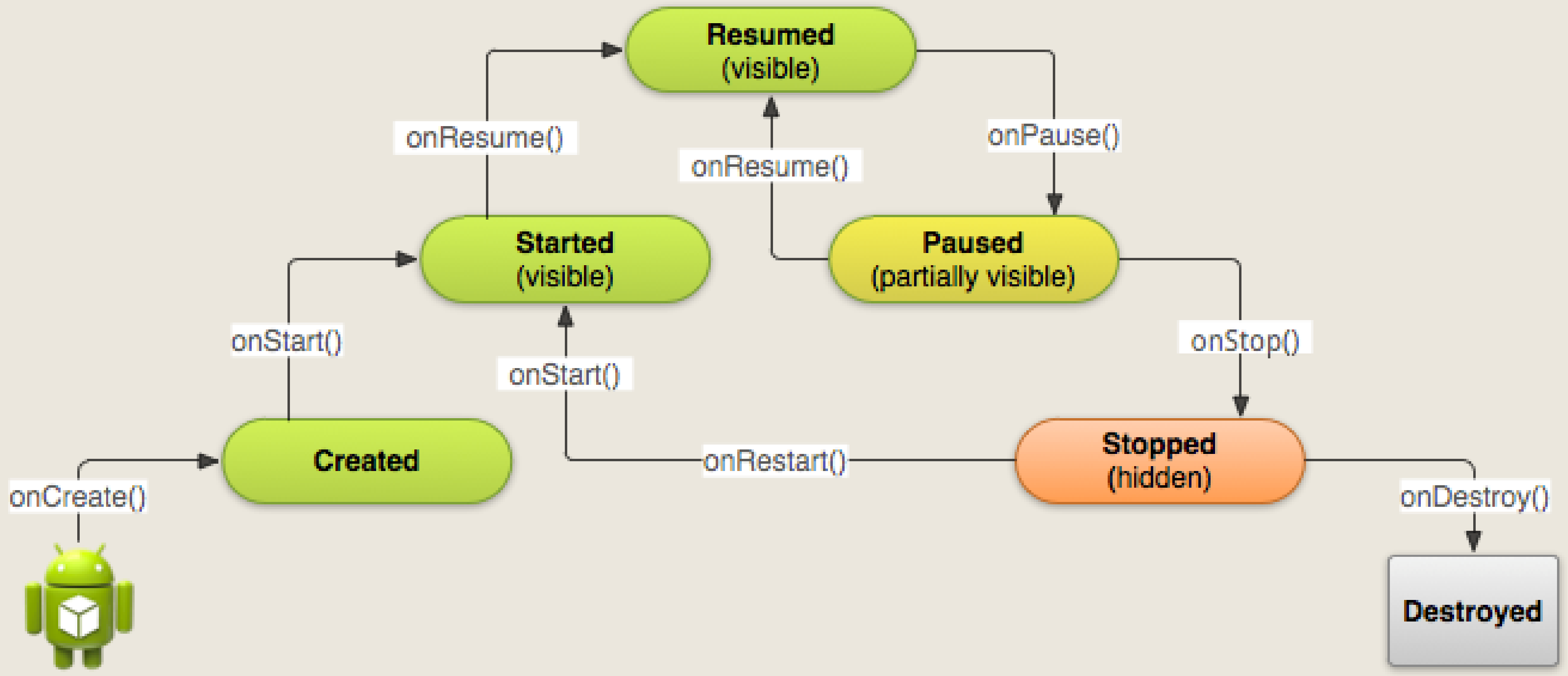
- Called when the system is about to start resuming a previous activity. This is typically used to commit unsaved changes to persistent data, stop animations and other things that may be consuming CPU, etc. Implementations of this method must be very quick because the next activity will not be resumed until this method returns
- Followed by either onResume() if the activity returns back to the front, or onStop() if it becomes invisible to the user

onStop() method

- Called when the activity is no longer visible to the user, because another activity has been resumed and is covering this one. This may happen either because a new activity is being started, an existing one is being brought in front of this one, or this one is being destroyed
- Followed by either onRestart() if this activity is coming back to interact with the user, or onDestroy() if this activity is going away

onDestroy() method

- The final call you receive before your activity is destroyed. This can happen either because the activity is finishing (someone called finish() on it, or because the system is temporarily destroying this instance of the activity to save space. You can distinguish between these two scenarios with the isFinishing() method


Resumed (visible)
onResume()
onResume()
onPause()
Started (visible)
Paused (partially visible)
onStart()
onStart()
onStop()
Created
onRestart()
Stopped (hidden)
onCreate()
onDestroy()
Destroyed

Pausing and Resuming an Activity

- During normal app use, the app sometimes loses focus, causing the activity to pause
- E.g. when a semi-transparent activity opens (such as one in the style of a dialog), the previous activity pauses
- As long as the activity is still partially visible but currently not the activity in focus, it remains paused
- However, once the activity is fully-obstructed and not visible, it *stops*

Stopping and Restarting an Activity

- Key scenarios:
  - *The user opens the Recent Apps window and switches from your app to another app. The activity in your app that's currently in the foreground is stopped. If the user returns to your app from the Home screen launcher icon or the Recent Apps window, the activity restarts*
  - *The user performs an action in your app that starts a new activity. The current activity is stopped when the second activity is created. If the user then presses the Back button, the first activity is restarted*
  - The user receives a phone call while using your app on his or her phone
- Unlike the paused state, which identifies a partial UI obstruction, the stopped state guarantees that the UI is no longer visible and the user's focus is in a separate activity

Coordination between Activities

- Order of operations that occur when Activity A starts Activity B:
  - *Activity A's onPause() method executes.*
  - *Activity B's onCreate(), onStart(), and onResume() methods execute in sequence. (Activity B now has user focus.)*
  - *Then, if Activity A is no longer visible on screen, its onStop() method executes.*
- Question: What should a developer do, if s/he must write to a database when the first activity stops so that the second activity can read it?

Recreating an Activity

- There are a few scenarios in which your activity is destroyed due to normal app behavior:
  - *When the user presses the Back button*
  - *Your activity signals its own destruction by calling finish()*
  - *The system may also destroy your activity if it's currently stopped and hasn't been used in a long time or the system needs to free up resources and must shut down cached processes to recover memory*
  - *Your activity will be destroyed and recreated each time the device configuration changes (such as screen orientation, keyboard availability, and language)*

When is Activity Stated saved?

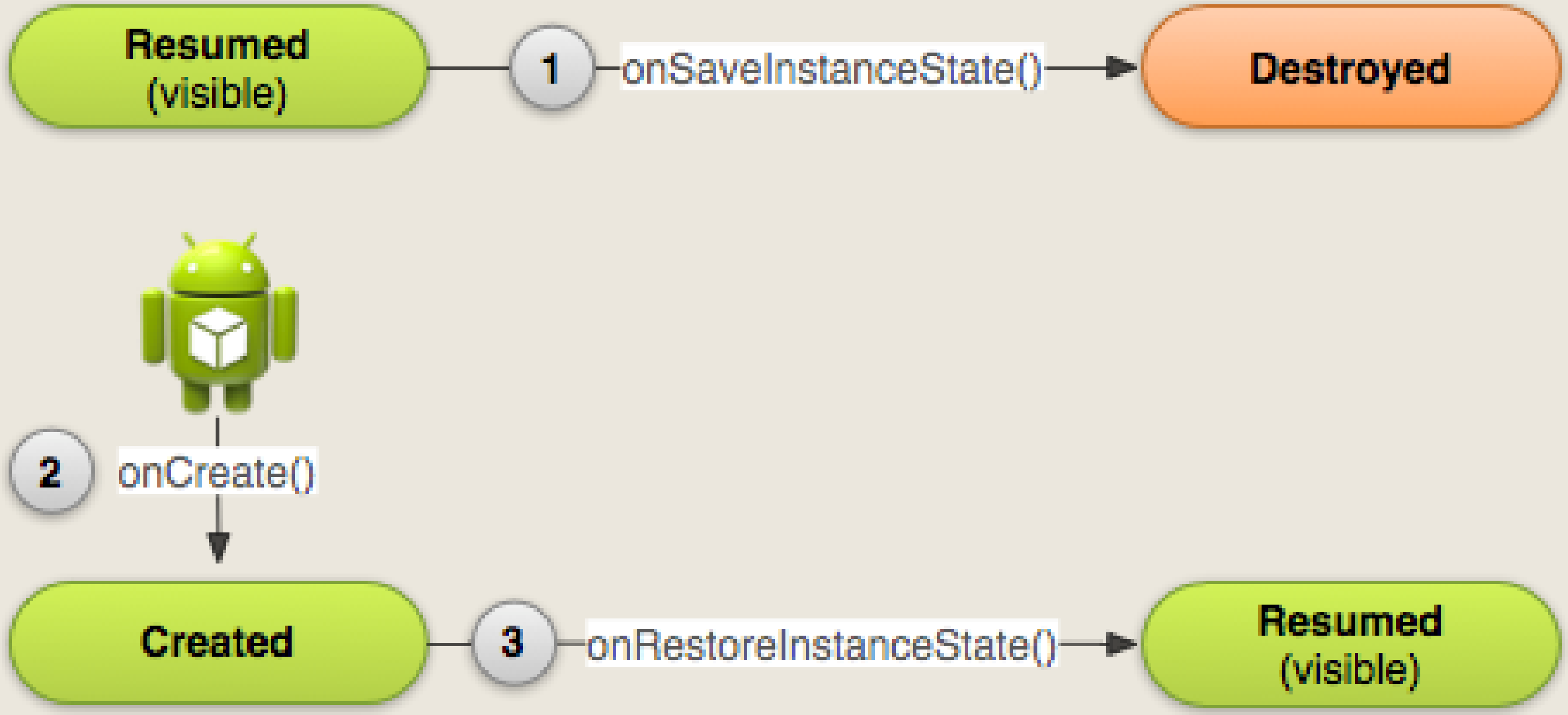
- When your activity is destroyed because the user presses Back or the activity finishes itself, all traces of the Activity instance is gone forever (normal app behavior)
- However, if the system destroys the activity due to system constraints (rather than normal app behavior), then although the actual Activity instance is gone, the system remembers that it existed such that 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 same happens in configuration changes
- Activity state is also saved when another activity appears. However, it is not restored automatically…!
- The saved data that the system uses to restore the previous state is called the "instance state" and is a collection of key-value pairs stored in a Bundle object

Saved Instance States

- By default, the system uses the Bundle instance state to save information about each View object in your activity layout (such as the text value entered into an EditText object)
- If your activity instance is destroyed and recreated, the state of the layout is restored to its previous state with no code required by you
- However, your activity might have more transient state information that you'd like to restore, such as member variables that track the user's progress in the activity
- Important Notice! In order for the Android system to restore the state of the views in your activity, each view must have a unique ID, supplied by the android:id attribute

Saving Additional Data

- To save additional data about the activity state, you must override the onSaveInstanceState() method
- The system calls this method when the user is leaving your activity and passes it the Bundle object that will be saved in the event that your activity is destroyed
- If the system must recreate the activity instance later, it passes the same Bundle object to both the onRestoreInstanceState() and onCreate() methods

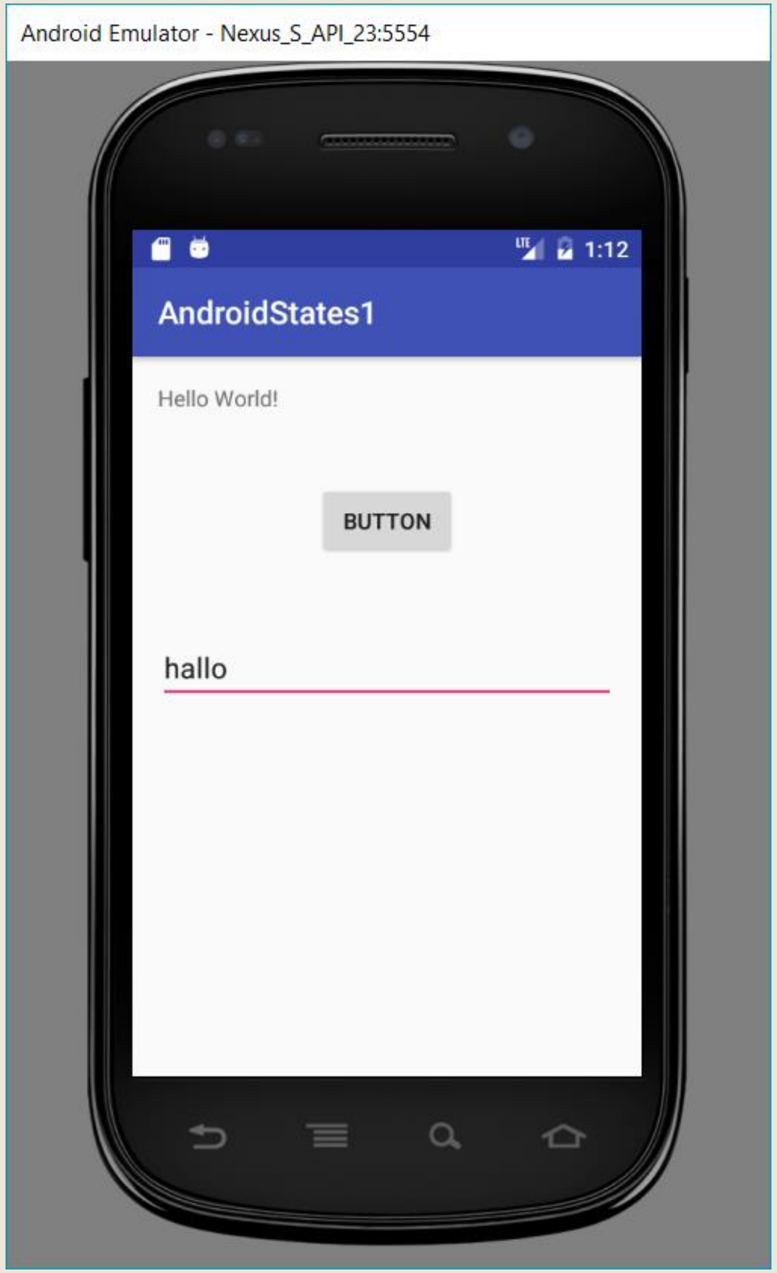

Resumed (visible)
1
onSaveInstanceState()
Destroyed
2
onCreate()
Created
3
onRestoreInstanceState()
Resumed (visible)

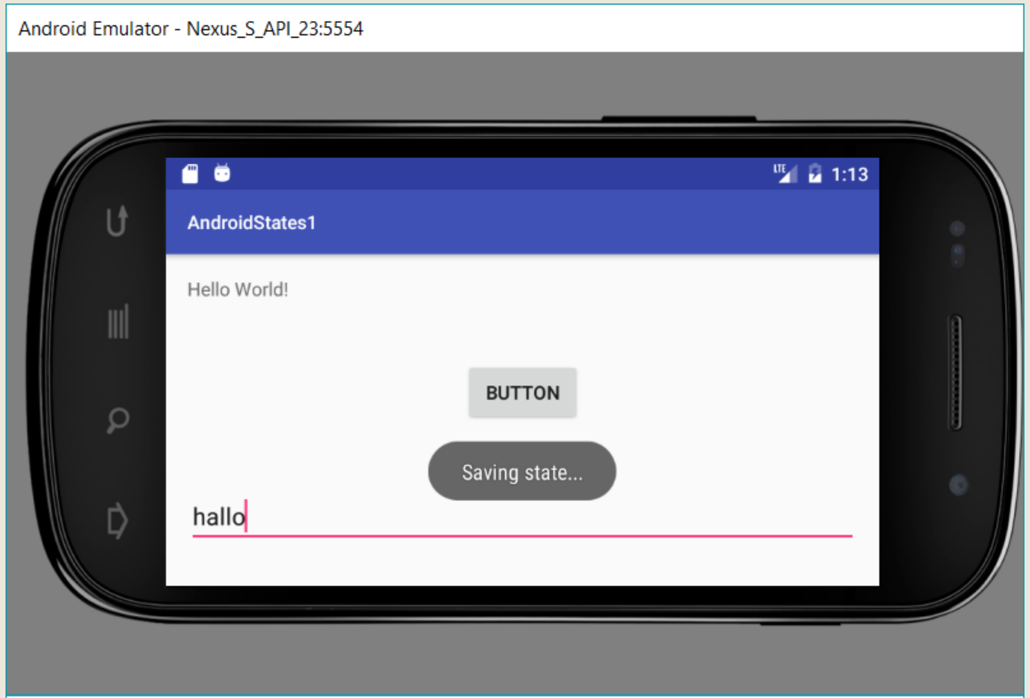
onSaveInstanceState() method

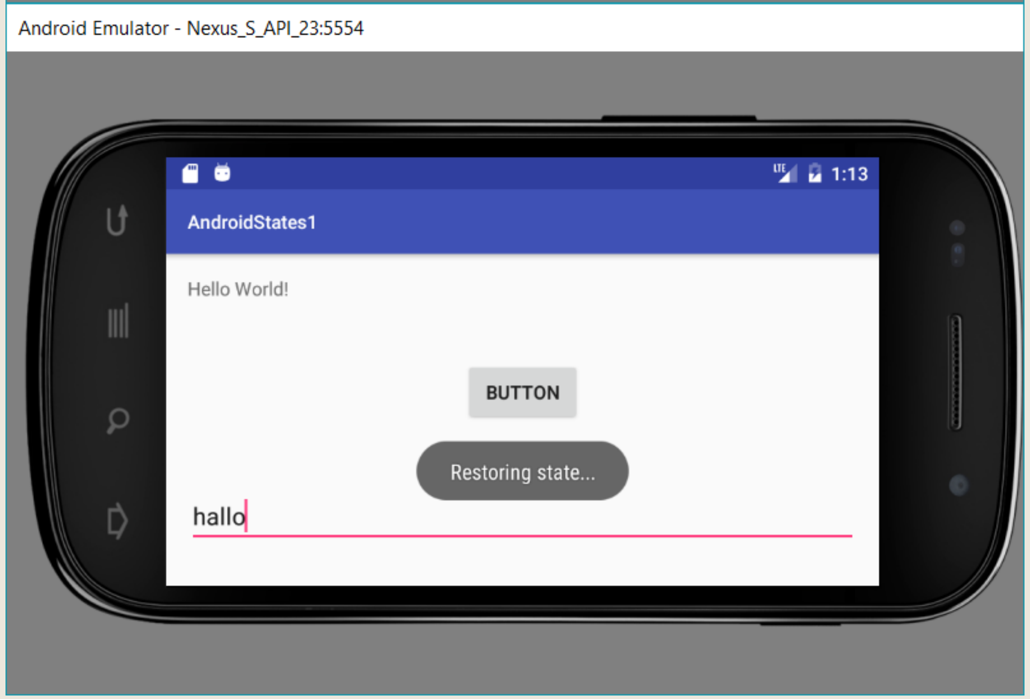
- As your activity begins to stop, the system calls onSaveInstanceState() so your activity can save state information with a collection of key-value pairs
- The default implementation of this method saves information about the state of the activity's view hierarchy, such as the text in an EditText widget or the scroll position of a ListView
- To save additional state information for your activity, you must implement onSaveInstanceState() and add key-value pairs to the Bundle object
- Caution! Always call the superclass implementation of onSaveInstanceState() so the default implementation can save the state of the view hierarchy

onRestoreInstanceState() method

- When your activity is recreated after it was previously destroyed, 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
- Because the onCreate() method is called whether the system is creating a new instance of your activity or recreating a previous one, 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
- Caution! Always call the superclass implementation of onRestoreInstanceState() so the default implementation can restore the state of the view hierarchy


Android Emulator - Nexus_S_API_23:5554


Android Emulator - Nexus_S_API_23:5554


Android Emulator - Nexus_S_API_23:5554

Retaining Objects During Configuration Changes

- Sometimes, restarting an activity requires recovering of large sets of data, re-establishing network connection, or performing other intensive operations
- In such cases, a full restart due to a configuration change might result on a slow user experience
- It might also not be possible to completely restore an activity state with the Bundle that the system saves with the onSaveInstanceState() callback—it is not designed to carry large objects (such as bitmaps)
- In such situations, a solution is retaining a Fragment when your activity is restarted due to a configuration change
- When the Android system shuts down your activity due to a configuration change, the fragments of your activity that you have marked to retain are not destroyed

Retaining stateful objects in a fragment

1. Extend the Fragment class and declare references to your stateful objects
2. Call setRetainInstance(boolean) when the fragment is created
3. Add the fragment to your activity
4. Use FragmentManager to retrieve the fragment when the activity is restarted

Retained Fragment example

```java
public class RetainedFragment extends Fragment {

    // data object we want to retain
    private MyDataObject data;

    // this method is only called once for this fragment
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        // retain this fragment
        setRetainInstance(true);
    }

    public void setData(MyDataObject data) {
        this.data = data;
    }

    public MyDataObject getData() {
        return data;
    }
}
```

```
public class MyActivity extends Activity {

    private RetainedFragment dataFragment;

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);
        // find the retained fragment on activity restarts
        FragmentManager fm = getFragmentManager();
        dataFragment = (DataFragment) fm.findFragmentByTag(“data”);
        // create the fragment and data the first time
        if (dataFragment == null) {
            // add the fragment
            dataFragment = new DataFragment();
            fm.beginTransaction().add(dataFragment, “data”).commit();
            // load the data from the web
            dataFragment.setData(loadMyData());
        }
        // the data is available in dataFragment.getData()
        ...
    }
    @Override
    public void onDestroy() {
        super.onDestroy();
        // store the data in the fragment
        dataFragment.setData(collectMyLoadedData());
    }
}
```

Handling the Configuration Change Yourself

- If your application doesn't need to update resources during a specific configuration change *and* you have a performance limitation that requires you to avoid the activity restart, then you can declare that your activity handles the configuration change itself
- Accordingly, this prevents the system from restarting your activity
- To declare that your activity handles a configuration change, edit the appropriate <activity> element in your manifest file to include the android:configChanges attribute with a value that represents the configuration you want to handle
- Caution! This technique should be considered a last resort when you must avoid restarts due to a configuration change