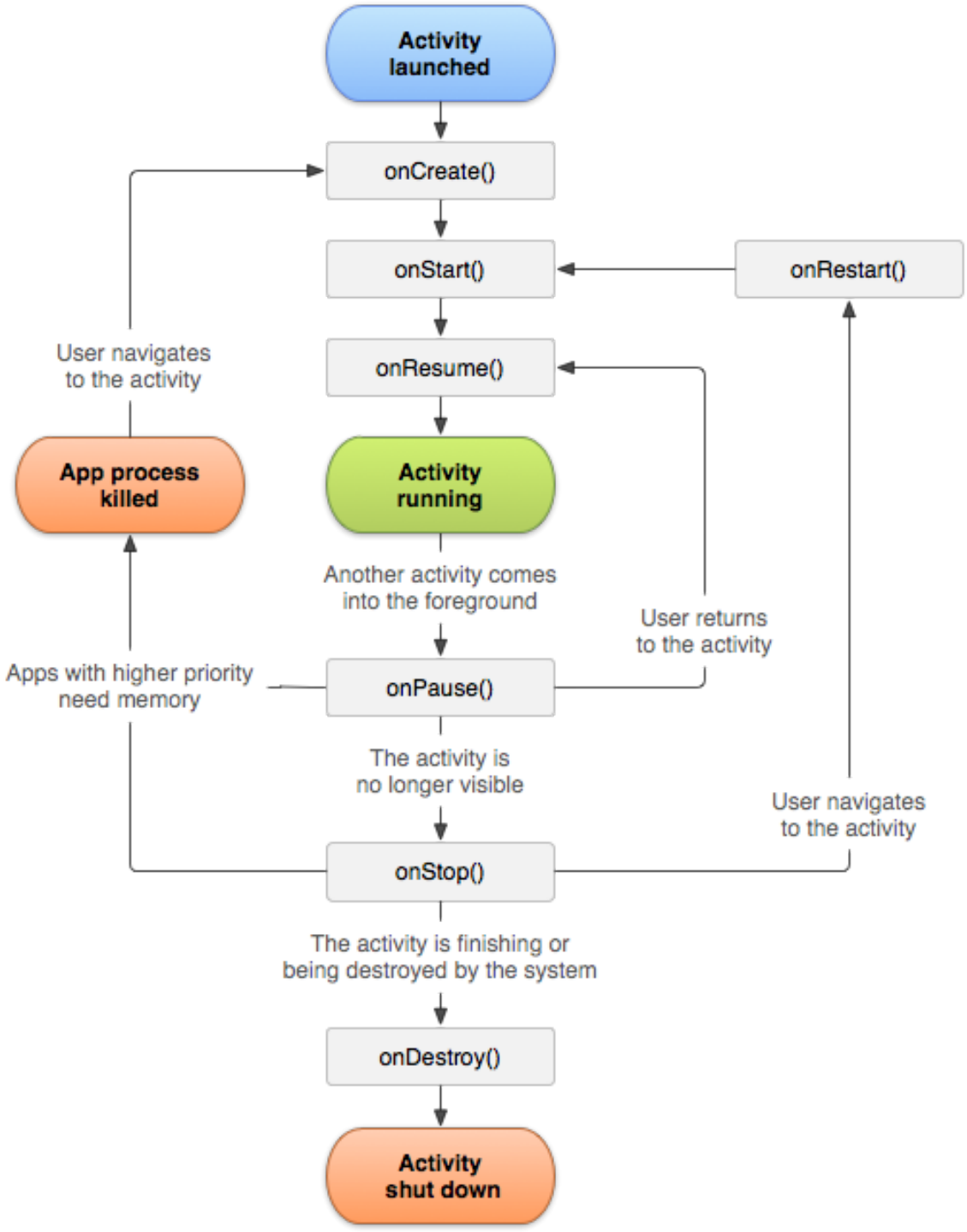
1. Activity's Life Cycle

The life cycle of an activity is managed via the following *call-back* methods, defined in the android.app.Activity base class:



- onCreate(), onDestroy(): Called back when the Activity is created/destroyed.
- onStart(), onStop(), onRestart(): Called back when the Activity is starting, stopping and re-starting.
- onPause(), onResume(): Called when the Activity is leaving the foreground and back to the foreground, can be used to release resources or initialize states.

Refer to Android API on Activity (http://developer.android.com/reference/android/app/Activity.html) for more details.

1.1 Example 8: Activity's Life Cycle

To illustrate the activity's life cycle, create a new Android project with application name of "Life Cycle", and package name "com.example.lifecycle". Create a activity called "MainActivity" with layout "activity_main".

Modify the "MainActitivity.java" as follows:

```
package ......;

import ......;
import android.util.Log;

public class MainActivity extends ActionBarActivity {
  private static final String TAG = "MainActivity: ";

  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);
    Log.d(TAG, "in onCreate()");
  }

  @Override
  public void onStart() {
    super.onStart();
    Log.d(TAG, "in onStart()");
  }

  @Override
  public void onRestart() {
    super.onRestart();
    Log.d(TAG, "in onReStart()");
  }

  @Override
  public void onResume() {
    super.onResume();
    Log.d(TAG, "in onResume()");
  }

  @Override
  public void onPause() {
    super.onPause();
    Log.d(TAG, "in onPause()");
  }

  @Override
  public void onStop() {
    super.onStop();
    Log.d(TAG, "in onStop()");
  }

  @Override
  public void onDestroy() {
    super.onDestroy();
    Log.d(TAG, "in onDestroy()");
  }
```

```
    ......
    ......
}
```

"LogCat" & DDMS

In the MainActivity, we use static method Log.d() to write a "debug" message to the "LogCat" - this is the standard way of writing log messages in Android. System.out.println() does not work under ADT.

LogCat is the Android Logging Utility that allows system and applications to output logging information at various logging levels. Each log entry consists of a time stamp, a logging level (V - verbose, D - debug, I - informational, W - warning , E - error), the process ID (pid) from which the log came, a tag defined by the logging application itself, and the actual logging message.

You could use the various static methods in the java.Android.Log class to log a message at various logging levels:

```
public static void v(String tag, String message);  // Verbose
public static void d(String tag, String message);  // Debug
public static void i(String tag, String message);  // Informational
public static void w(String tag, String message);  // Warning
public static void e(String tag, String message);  // Error
  // where tag identifies the application
```

To show the LogCat panel, press "Alt+6" to show the "Android DDMS" panel. Logcat messages are displayed under "Devices | Logcat" tab.

From the LogCat panel, you can select a particular logging level, or filter based on pid, tag and log level.

Run the App

Run the application, and observe the message on the "LogCat" panel.

```
07-14 03:45:48.698: D/MainActivity:(752): in onCreate()
07-14 03:45:48.698: D/MainActivity:(752): in onStart()
07-14 03:45:48.708: D/MainActivity:(752): in onResume()
```

Click the "BACK" button:

```
07-14 03:46:29.559: D/MainActivity:(752): in onPause()
07-14 03:46:31.849: D/MainActivity:(752): in onStop()
07-14 03:46:31.899: D/MainActivity:(752): in onDestroy()
```

Run the application again from the "App Menu":

```
07-14 03:47:24.749: D/MainActivity:(752): in onCreate()
07-14 03:47:24.749: D/MainActivity:(752): in onStart()
07-14 03:47:24.759: D/MainActivity:(752): in onResume()
```

Click the "HOME" button to send the activity to the background, then run the app from the "App Menu" again. Notice that onDestroy() is not called in this case.

```
07-14 03:48:06.369: D/MainActivity:(752): in onPause()
07-14 03:48:08.258: D/MainActivity:(752): in onStop()
```

```
07-14 03:49:14.650: D/MainActivity:(752): in onReStart()
07-14 03:49:14.650: D/MainActivity:(752): in onStart()
07-14 03:49:14.659: D/MainActivity:(752): in onResume()
```

Click the "PHONE" button, and then "BACK":

```
07-14 03:51:31.829: D/MainActivity:(752): in onPause()
07-14 03:51:36.700: D/MainActivity:(752): in onStop()
07-14 03:51:38.308: D/MainActivity:(752): in onReStart()

07-14 03:51:38.308: D/MainActivity:(752): in onStart()
07-14 03:51:38.329: D/MainActivity:(752): in onResume()
```

Take note that clicking the "BACK" button destroys the activity, but "HOME" and "PHONE" button does not.

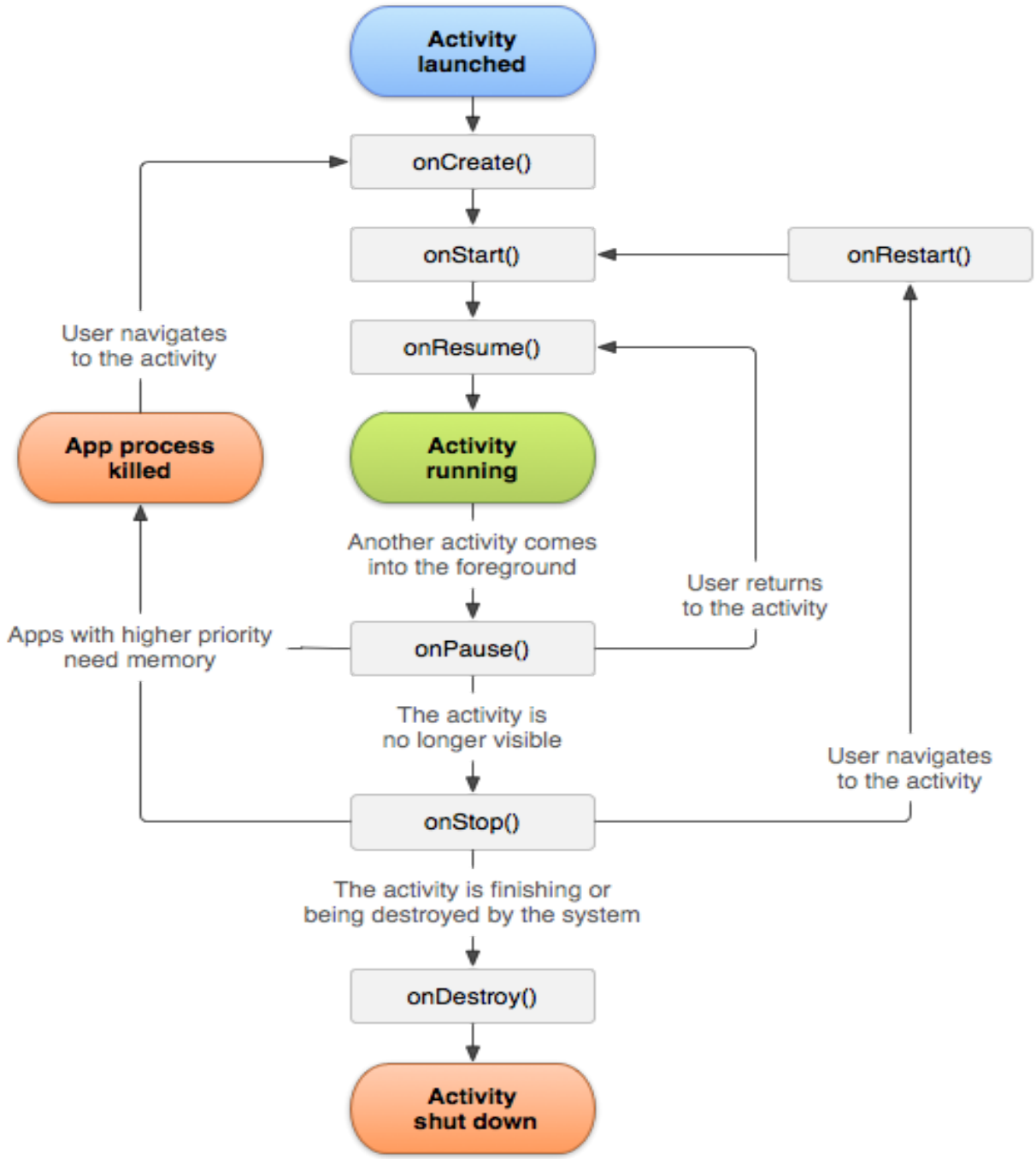
Activity Lifecycle

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 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*. 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.



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(). An activity will do all setup of "global" state in onCreate(), and release all remaining resources in onDestroy(). For example, if it has a thread running in the background to download data from the network, it may create that thread in onCreate() and then stop the thread 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. For example, you can register a BroadcastReceiver in onStart() to monitor for changes that impact your UI, and unregister it in onStop() when the user no longer sees what you are displaying. The onStart() and onStop() methods can be called multiple times, as the activity becomes visible and hidden 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 the device goes to sleep, when an activity result is delivered, when a new intent is delivered -- so the code in these methods should be fairly lightweight.

The entire lifecycle of an activity is defined by the following Activity methods. All of these are hooks that you can override to do appropriate work when the activity changes state. All activities will implement onCreate(Bundle) to do their initial setup; many will also implement onPause() to commit changes to data and otherwise prepare to stop interacting with the user. You should always call up to your superclass when implementing these 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();
}
```

In general the movement through an activity's lifecycle looks like this:

Method	Description	Killable?	Next
onCreate()	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().	No	onStart()
onRestart()	Called after your activity has been stopped, prior to it being started again.	No	onStart()

	Always followed by onStart()		
onStart()	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.	No	onResume() or onStop()
onResume()	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().	No	onPause()
onPause()	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.	**Pre-**HONEYCOMB	onResume() or onStop()
onStop()	Called when the activity is no longer visible to the user, because another activity has been resumed and is covering	**Yes**	onRestart() or onDestroy()

	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()	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.	**Yes**	*nothing*

Note the "Killable" column in the above table -- for those methods that are marked as being killable, after that method returns the process hosting the activity may be killed by the system *at any time* without another line of its code being executed. Because of this, you should use the onPause() method to write any persistent data (such as user edits) to storage. In addition, the method onSaveInstanceState(Bundle) is called before placing the activity in such a background state, allowing you to save away any dynamic instance state in your activity into the given Bundle, to be later received in onCreate(Bundle) if the activity needs to be re-created. See the Process Lifecycle section for more information on how the lifecycle of a process is tied to the activities it is hosting. Note that it is important to save persistent data in onPause() instead of onSaveInstanceState(Bundle) because the latter is not part of the lifecycle callbacks, so will not be called in every situation as described in its documentation.

Be aware that these semantics will change slightly between applications targeting platforms starting with HONEYCOMB vs. those targeting prior platforms. Starting with Honeycomb, an application is not in the killable state until its onStop() has returned. This impacts when onSaveInstanceState(Bundle) may be called (it may be safely called after onPause() and allows and application to safely wait until onStop() to save persistent state.

For those methods that are not marked as being killable, the activity's process will not be

killed by the system starting from the time the method is called and continuing after it returns. Thus an activity is in the killable state, for example, between after onPause() to the start of onResume().