Application Components

Prepared by KJK; some content from Chapter 4

Overview

- Solidify your understanding of the Activity lifecycle
- Understand the purpose of all lifecycle callbacks
- First look at Android fragments
- Manage Activity transitions and organize navigation with intents
- Understand the purpose of services
- Investigate other uses for intents

Lifecycle of an Android Activity

- Android allows multiple apps to run concurrently (provided memory and processing power are available).
- Applications can have background behavior.
- Applications can be interrupted and paused when events such as phone calls occur.
- There can be only one active application visible to the user at a time — specifically, a single application Activity is in the foreground at any given time.

Lifecycle of an Android Activity

- Android keeps track of all Activity objects running by placing them on an Activity stack.
- As mentioned before, Activity stack is referred to as the “back stack.”
- When a new Activity starts, the Activity on the top of the stack (the current foreground Activity) pauses, and the new Activity pushes onto the top of the stack.
- When that Activity finishes, it is removed from the Activity stack, and the Activity below on the stack resumes.

Lifecycle Callback Methods

```
public class Activity extends ...    {
    protected void onCreate(Bundle savedInstanceState);
    protected void onStart();
    protected void onRestart();
    protected void onResume();
    protected void onPause();
    protected void onStop();
    protected void onDestroy();
}
```

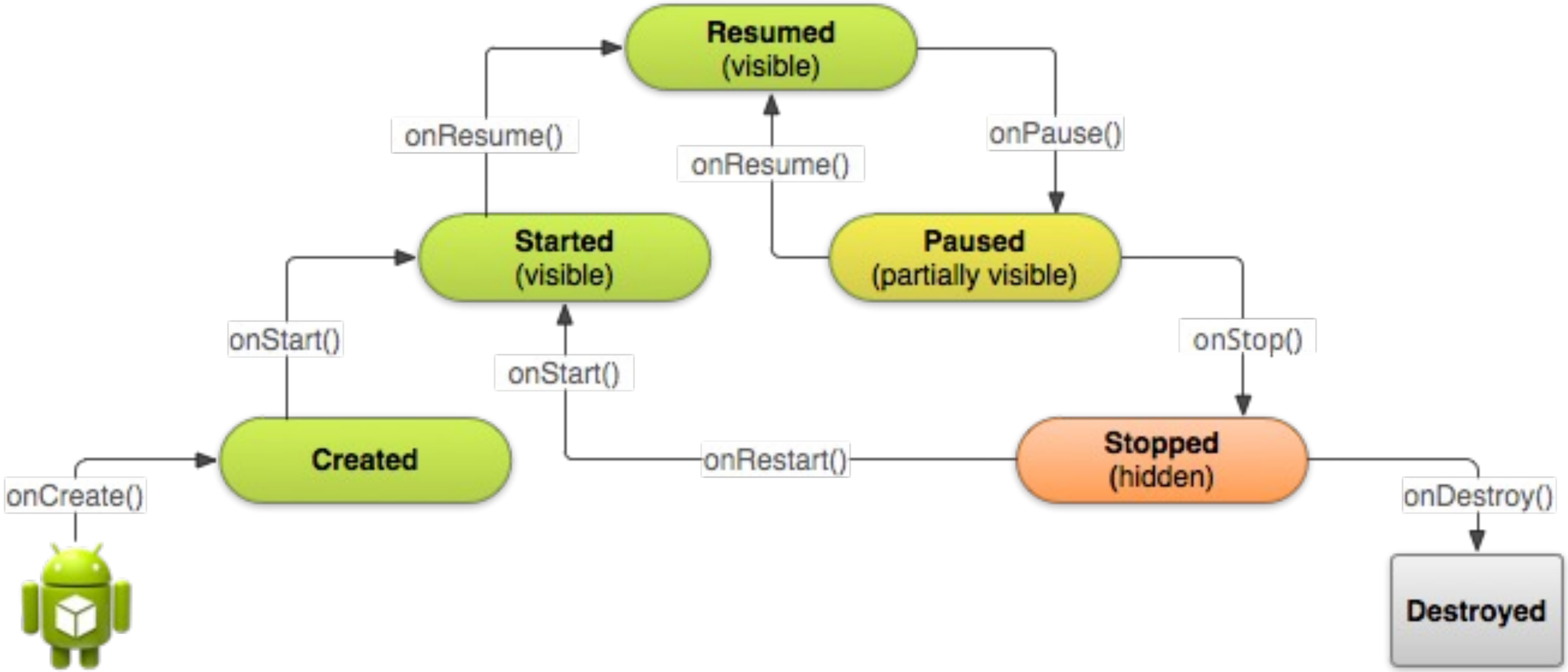
- Application’s activity class extends `Activity` and can override any of the above methods.
- At least, `onCreate` should be overridden.

Lifecycle Callback Methods

- Not all lifecycle methods must be implemented
- However, the application should behave well:
  - Should not crash if the user switches to a different app or receives a phone call
  - Should not crash if the user changes the orientation of the device, e.g., portrait to landscape
  - Should not lose user’s data and app’s progress when the user switches to a different app or receives a phone call
  - Should not hold resources when not in active use

Lifecycle of an Android Activity

Activity state changes

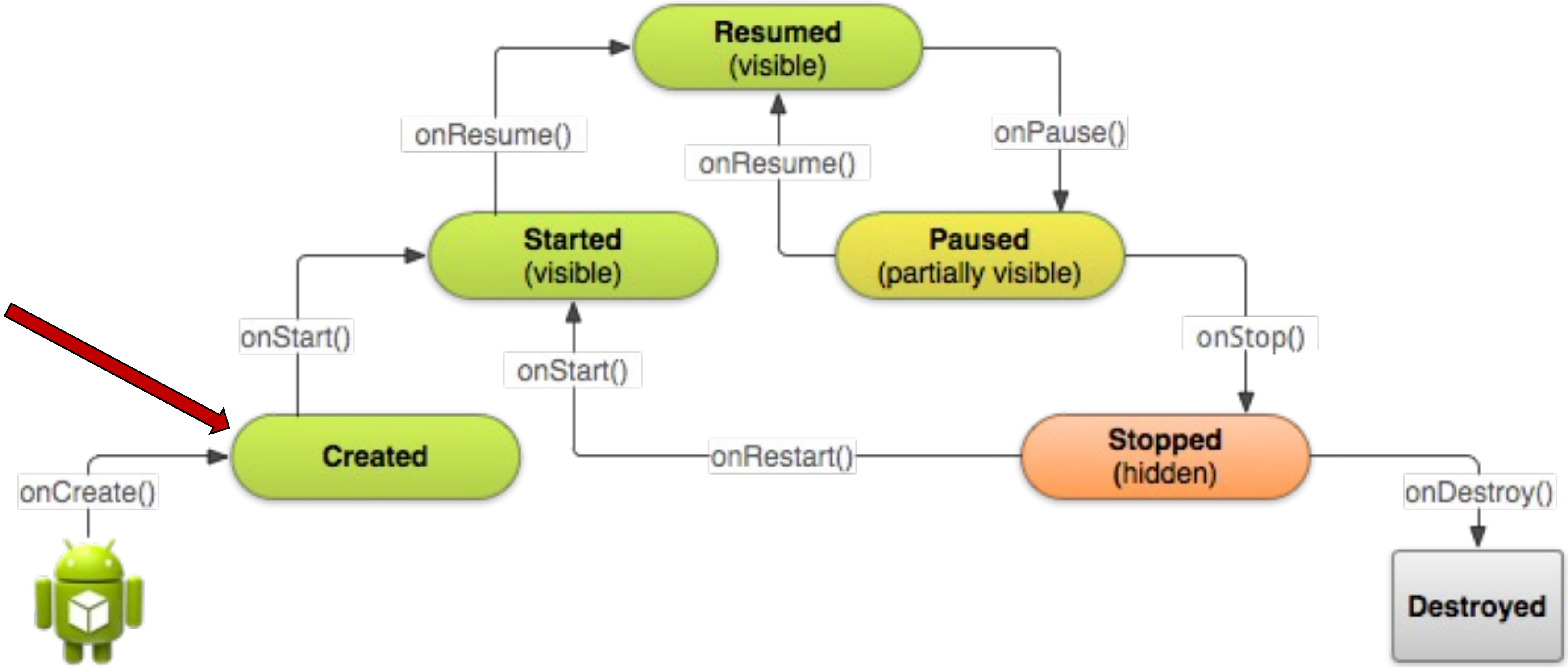


From: Android Developer Fundamentals Course,
Google Developer Training

Starting an Activity

- When an Activity first starts, Android creates an instance of the Activity class
- Launcher activity is specified in the Manifest
- Android then calls `onCreate()`
- `onCreate()` must create the user interface for the activity and perform any necessary initialization of the interface
  - `setContentView()`
- The activity then enters the *Created* state

Starting an Activity



From: Android Developer Fundamentals Course,
Google Developer Training

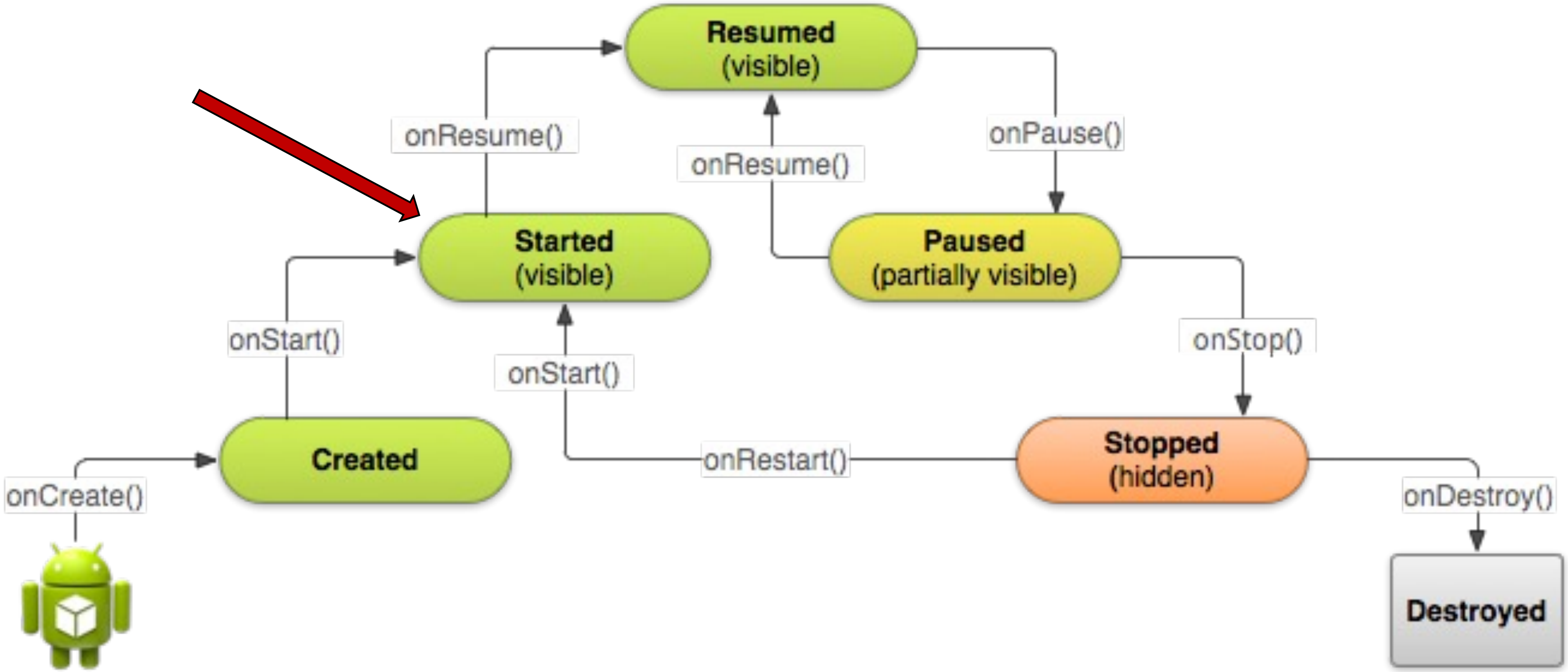
Starting an Activity

- Note: `onCreate()` has a single parameter, a Bundle
- If the activity is newly created, that parameter is `null`.
- If this Activity is a restarted Activity, the Bundle parameter contains the previous state data, so that the activity can reinitiate.

Starting an Activity

- Android then calls `onStart()`
- At this time, the UI screen of the Activity becomes visible to the user, but it doesn’t respond to user input, yet
- Typically, the app may access some important listeners, e.g., a GPS sensor
- The activity then enters the *Started* state (also referred to as *Visible*)

Starting an Activity

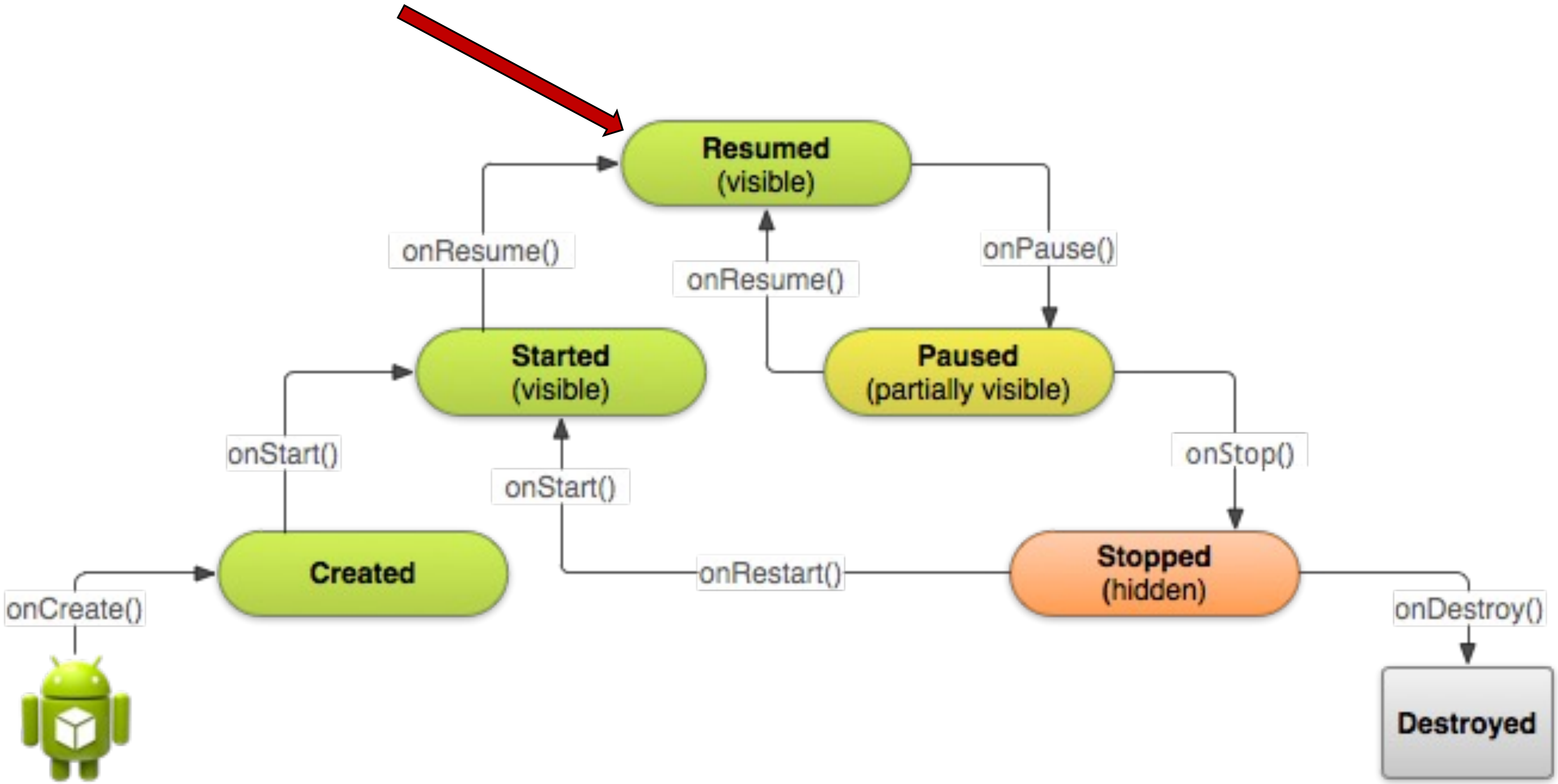


From: Android Developer Fundamentals Course,
Google Developer Training

Starting an Activity

- Android then calls `onResume()`
- The activity becomes the foreground activity and is placed on top of the activity backstack
- The activity then enters the *Resumed* state
- At this time, the UI screen of the Activity is visible to the user, and it does respond to user input
- NOTE: an activity never *remains* in either Created or Started state, only in Resumed

Starting an Activity



From: Android Developer Fundamentals Course,
Google Developer Training

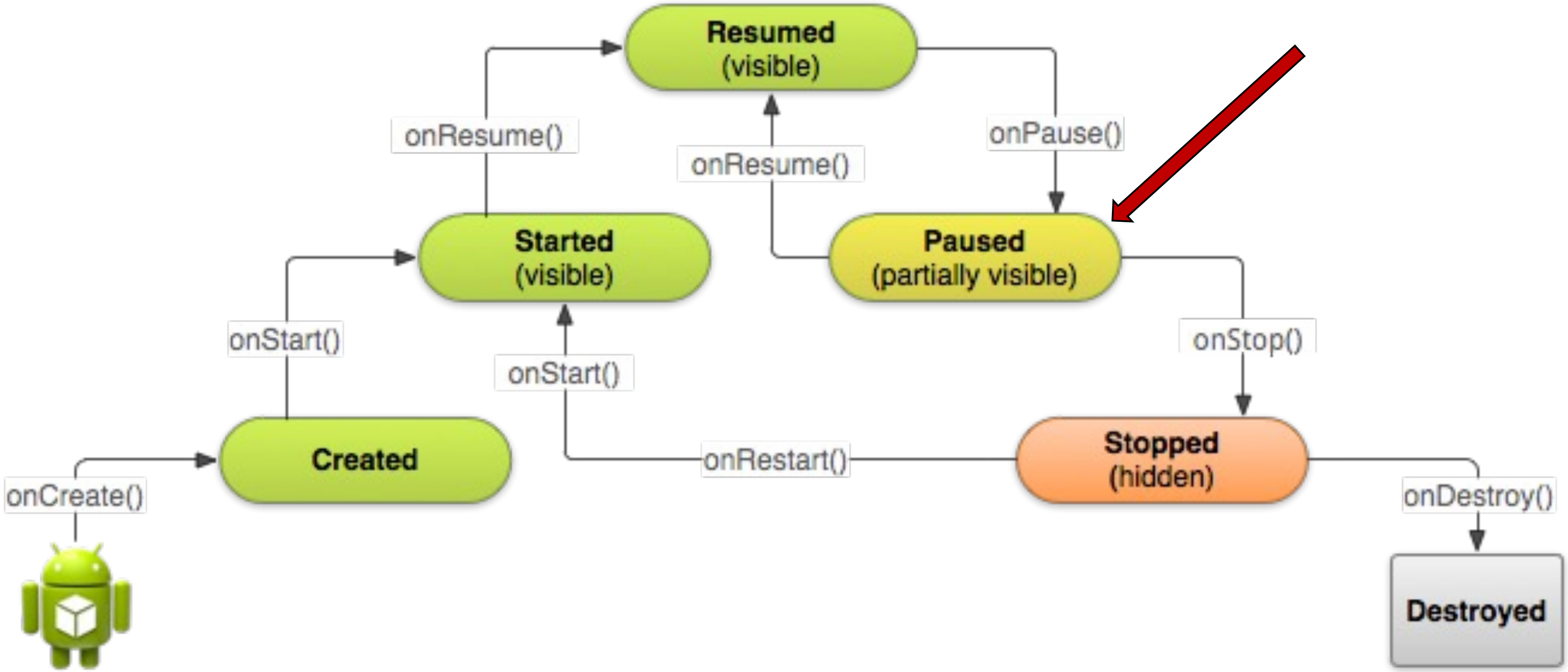
Pausing and Resuming an Activity

- It may happen that a Resumed activity becomes partially obscured, e.g., by a dialog box, or a semi-transparent activity
- Android then calls `onPause()`
- The activity enters the *Paused* state
- At this time, the UI screen of the Activity is only *partially visible* to the user, and it does not respond to user input

Pausing and Resuming an Activity

- While in the `onPause()` callback, the Activity should prepare to stop. For example, it should:
  - save any important application data
  - stop animations and/or movie playback
  - release any Android resources it may have, e.g., camera or the GPS sensor

Pausing and Resuming an Activity

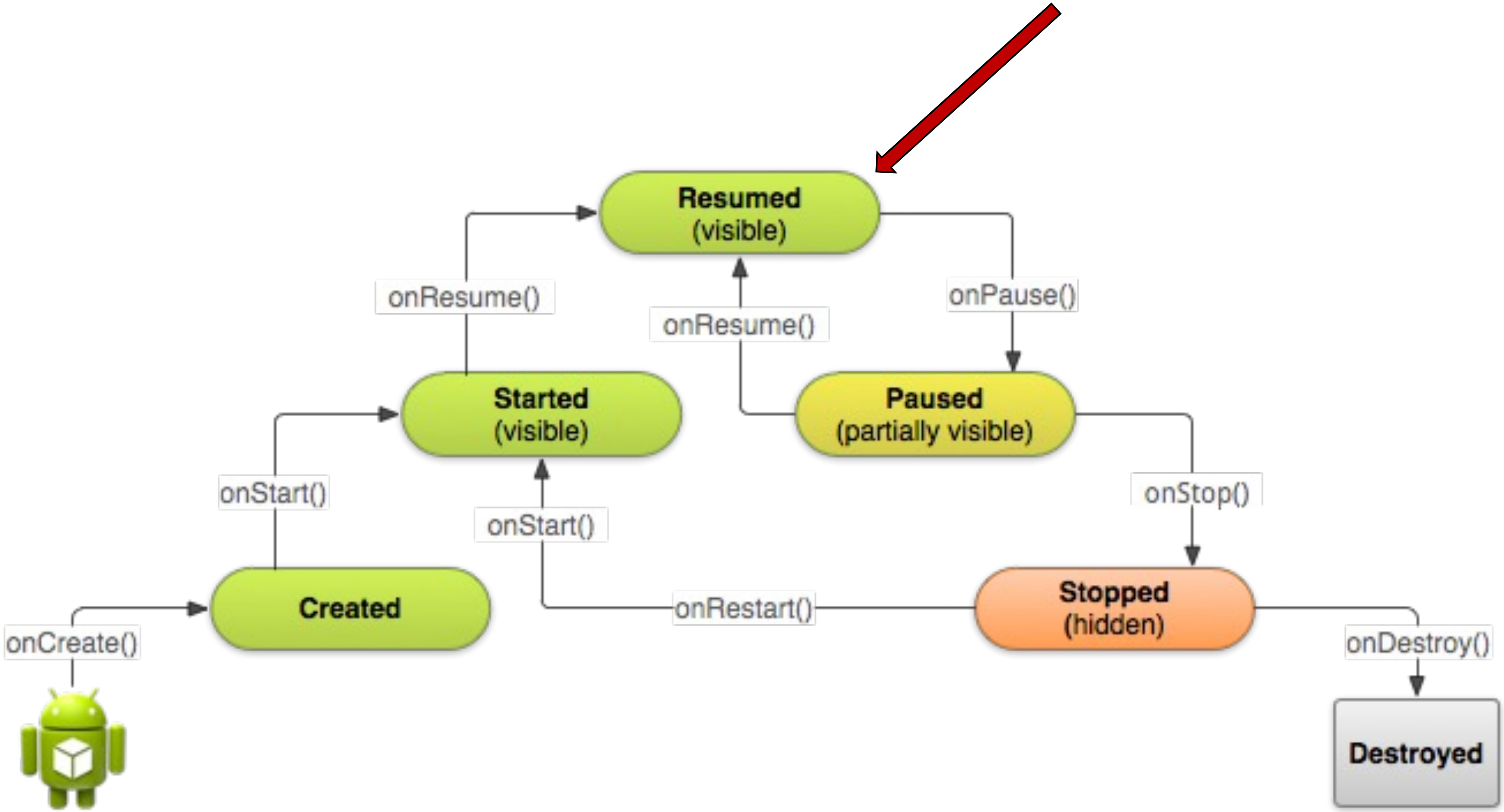


From: Android Developer Fundamentals Course,
Google Developer Training

Pausing and Resuming an Activity

- If the user dismisses a dialog (or other screen) partially obscuring the Activity screen, the Activity is ready to resume
- Android calls the `onResume()` callback, where the Activity should prepare to enter the Resumed state. For example, it should:
  - restore any important application data
  - resume animations and/or movie playback
  - regain access to necessary Android resources, e.g., camera or the GPS sensor

Pausing and Resuming an Activity



From: Android Developer Fundamentals Course,
Google Developer Training

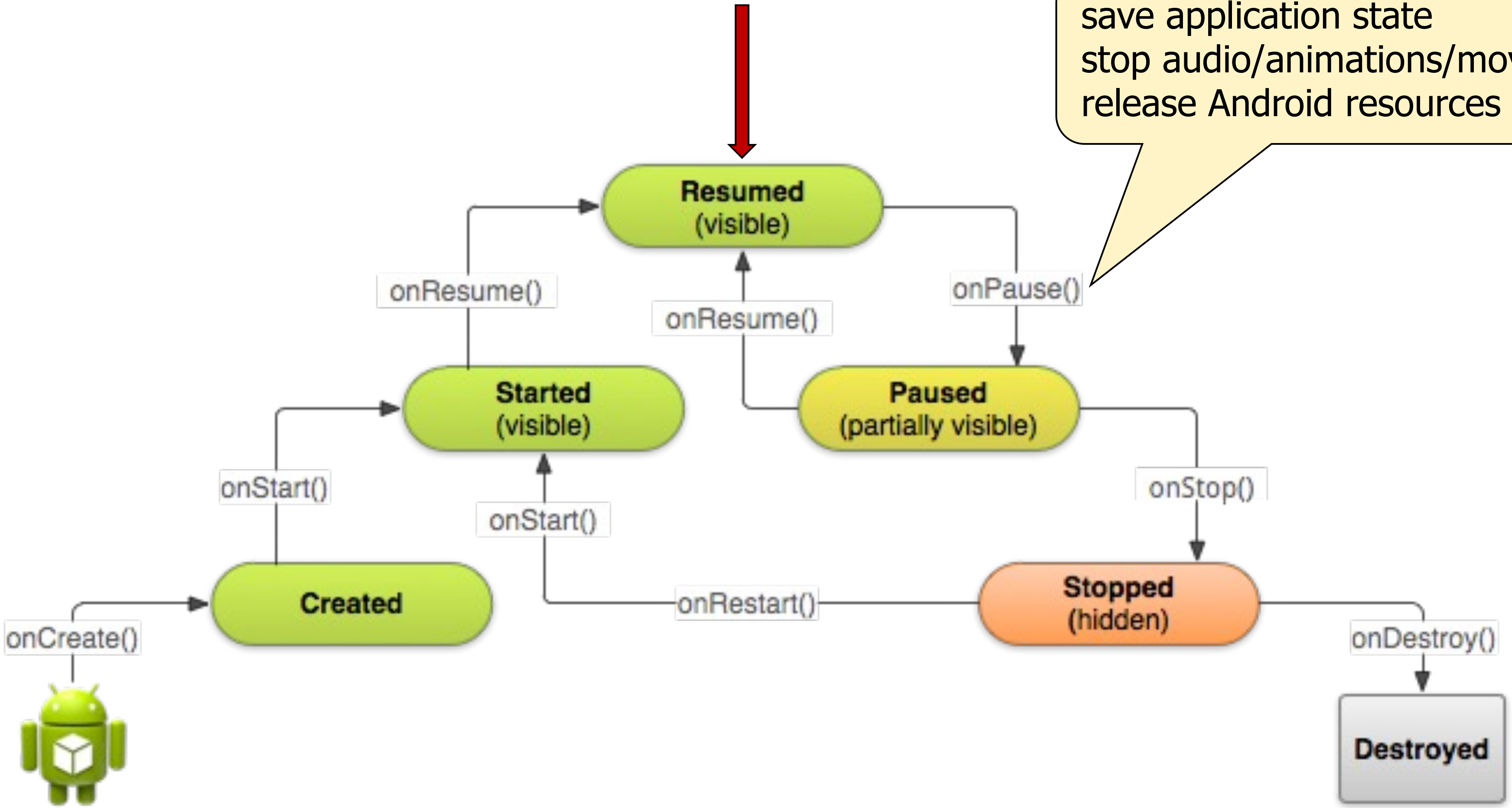
Activity Data Setup in onResume()

- `onResume()` is the most appropriate place to retrieve any instances of resources that the `Activity` needs to run, even if it is just being created for the first time
- In general, these resources are the most CPU intensive, so we keep them around only while the `Activity` is in the foreground

Activity Data Release in onPause()

- Also, `onPause()` is the most appropriate place to stop audio/video/animations started in `onResume()`
- Any uncommitted data should be saved here, as well
- Before Honeycomb (Android 3.0), `onPause()` was *killable*, that is, Android could kill the app to reclaim resources *without* calling `onStop()`.
- It is not the case after Honeycomb and Android guarantees to call `onStop()` before killing the app
- Perform anything in `onPause()` in a timely fashion.

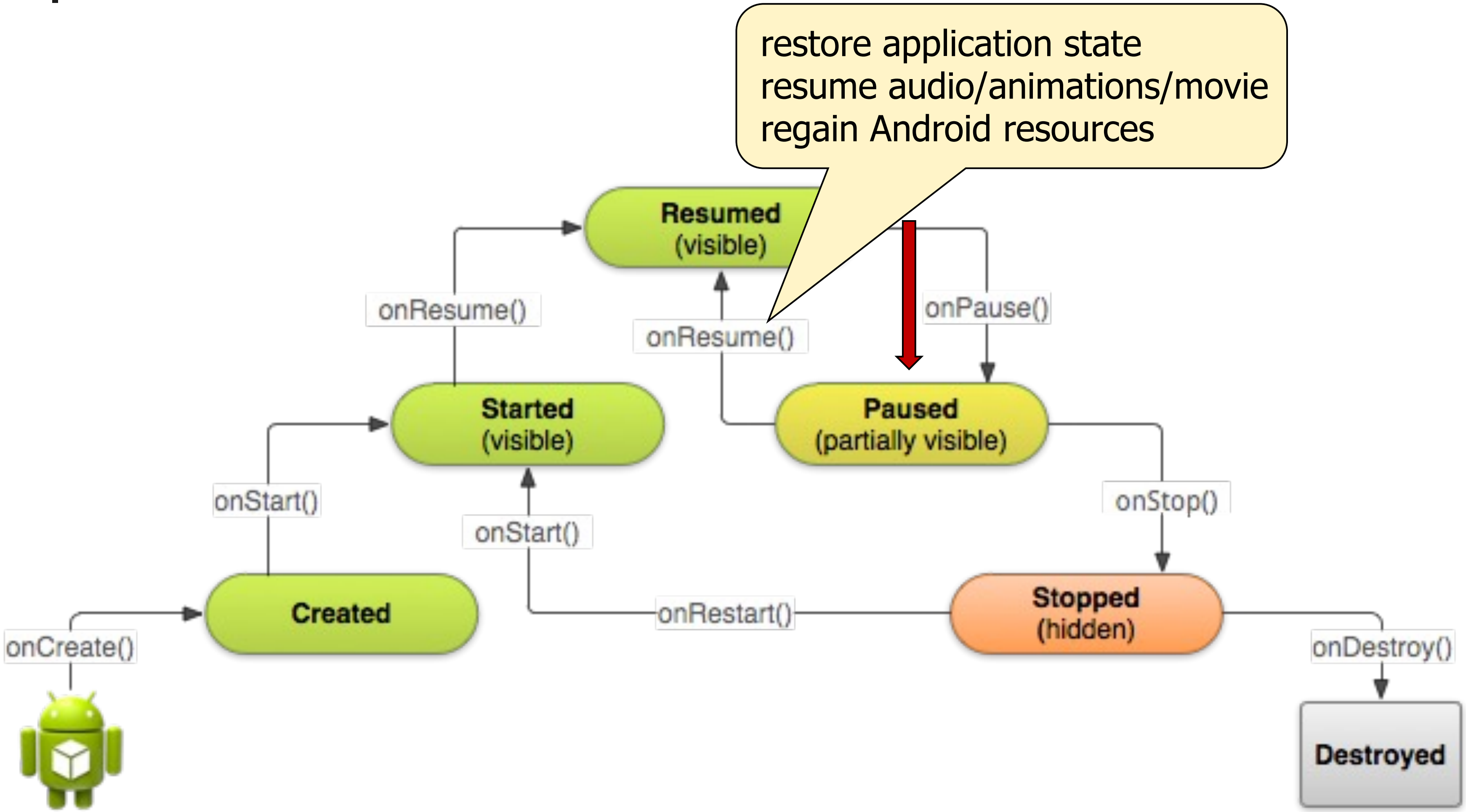
Pausing and Resuming an Activity



save application state
stop audio/animations/movie
release Android resources

From: Android Developer Fundamentals Course,
Google Developer Training

Pausing and Resuming an Activity



restore application state
resume audio/animations/movie
regain Android resources

From: Android Developer Fundamentals Course,
Google Developer Training

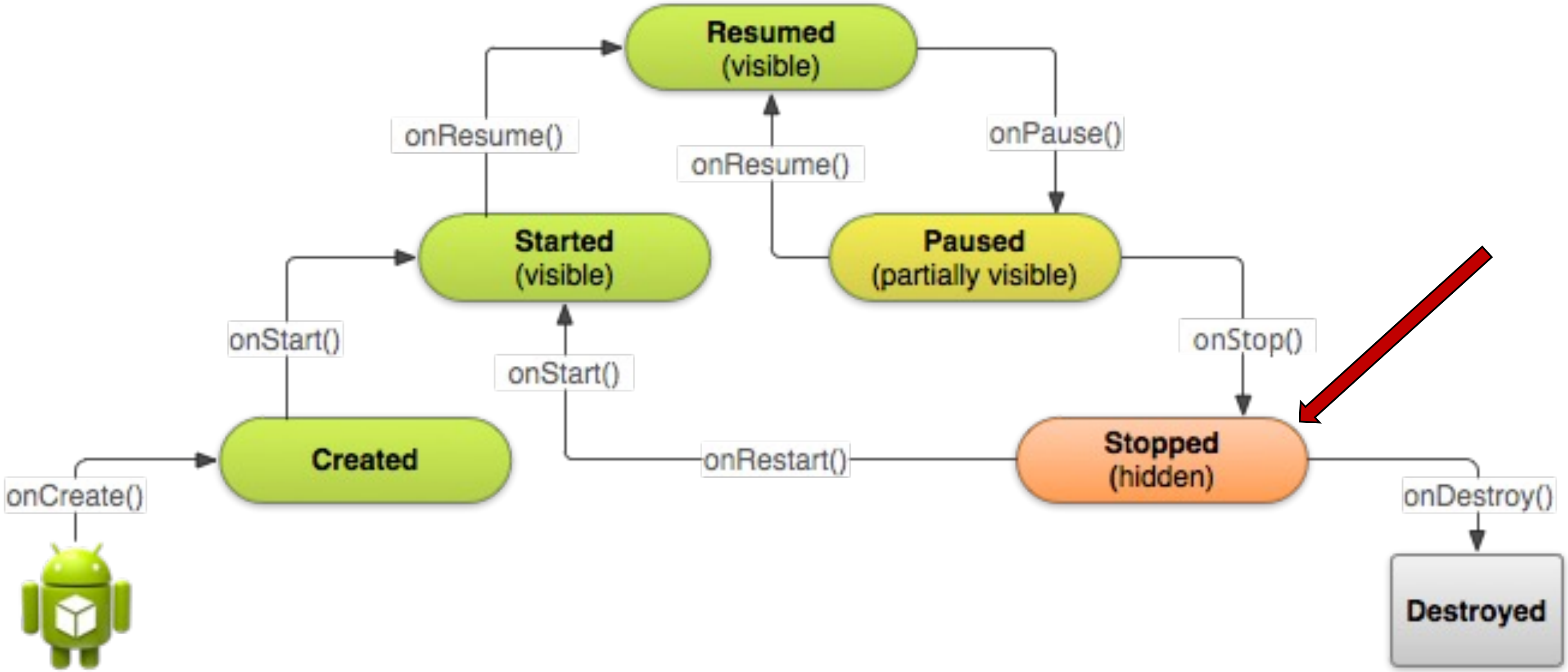
Stopping and Restarting an Activity

- A *resumed* (foreground) activity is *stopped*, if the user:
  - selects a different app from a list of recent apps
  - causes an action in the current activity to start a different activity (transitions to a different activity)
  - receives a phone call on the device
- Activity enters the *Stopped* state
- The activity is pushed to back stack (it is no longer the foreground activity)

Stopping and Restarting an Activity

- More precisely, Android calls `onPause()` and the activity enters the *Paused* state, but only briefly
- Android then immediately calls `onStop()` and the Activity enters the *Stopped* state
- At this time, the UI screen of the Activity is *not visible* to the user
- The user can no longer interact with the activity

Pausing and Resuming an Activity



From: Android Developer Fundamentals Course,
Google Developer Training

Stopping and Restarting an Activity

- An Activity may be killed by Android after `onStop` (due to low resources), and the `onDestroy()` method may not be called.
- The more resources are released by an Activity in the `onPause()` and `onStop()` methods, the less likely the Activity is to be killed while in the background without further state methods being called.

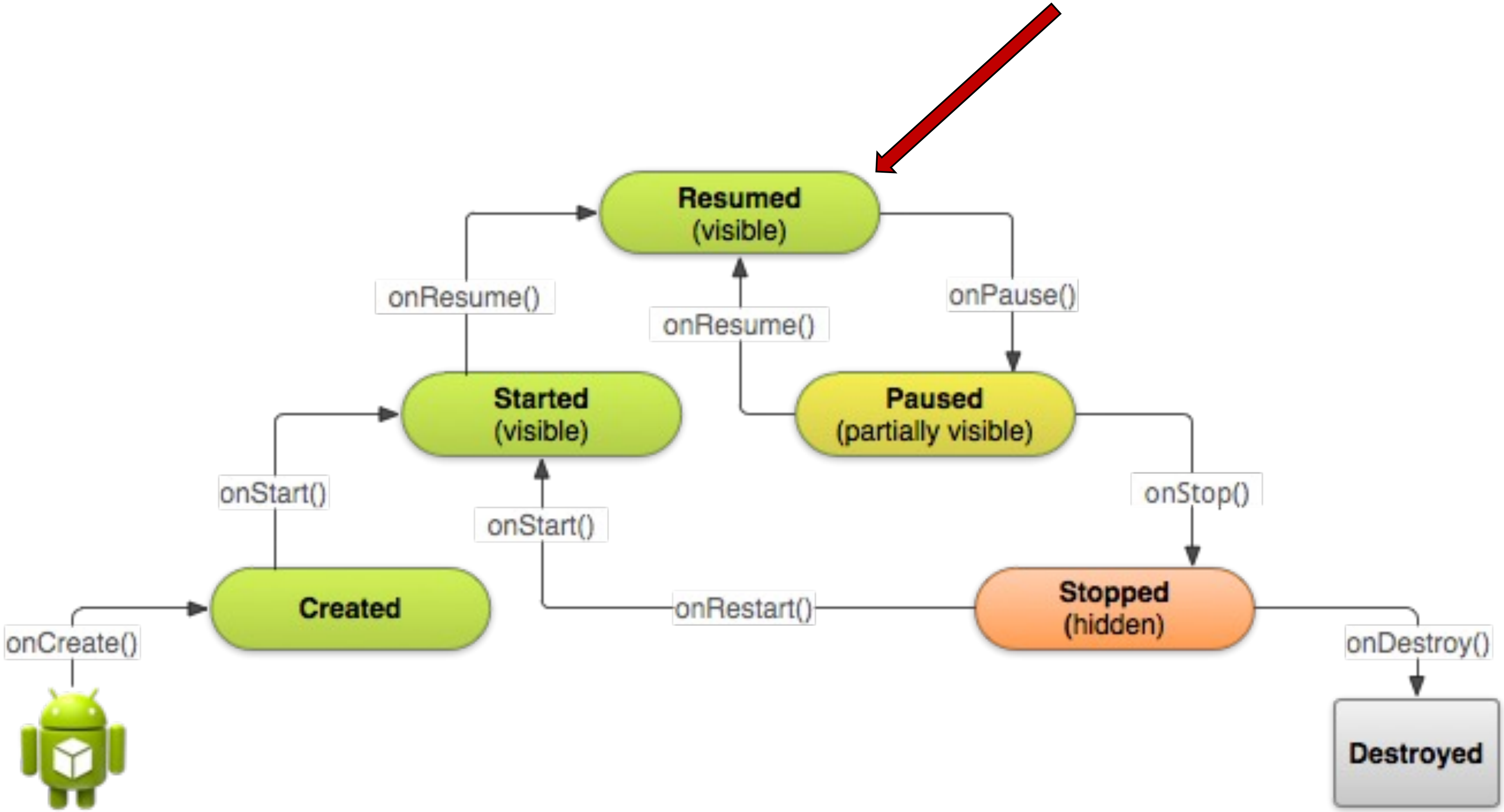
Stopping and Restarting an Activity

- A *stopped* activity is *restarted*, if the user:
  - selects the activity from a list of recent apps
  - selects the activity's app from the all-apps screen
  - presses the back button and the activity is immediately below the top on the back stack
  - terminates the phone call on the device
- The activity is restarted and becomes the foreground activity
- The screen becomes visible, and the user can interact with it again

Stopping and Restarting an Activity

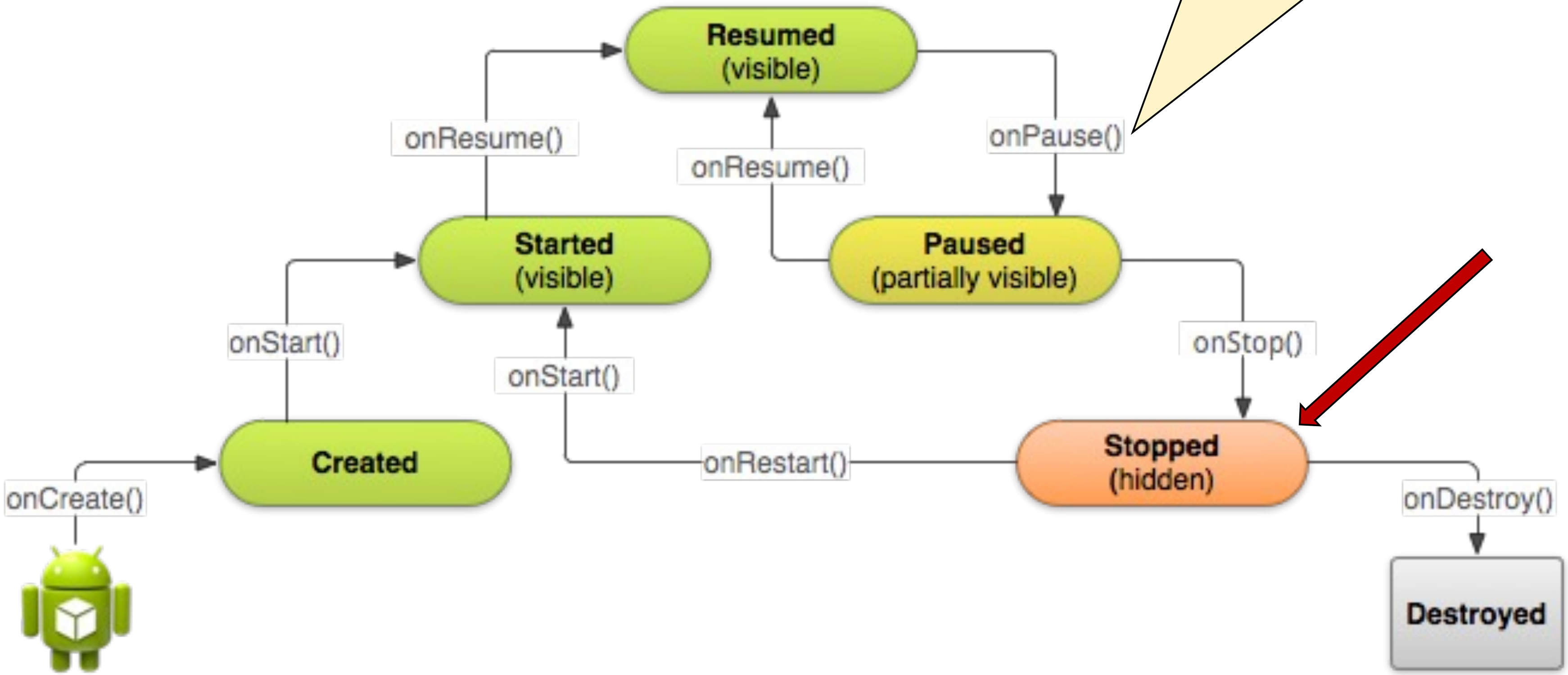
- More precisely, Android calls `onRestart()` and right after that `onStart()`
- The activity enters the *Started* state, but only briefly; the UI screen of the Activity becomes *visible* to the user
- Android then immediately calls `onResume()` and the Activity enters the *Resumed* state
- The user can again interact with the activity

Stopping and Restarting an Activity



From: Android Developer Fundamentals Course,
Google Developer Training

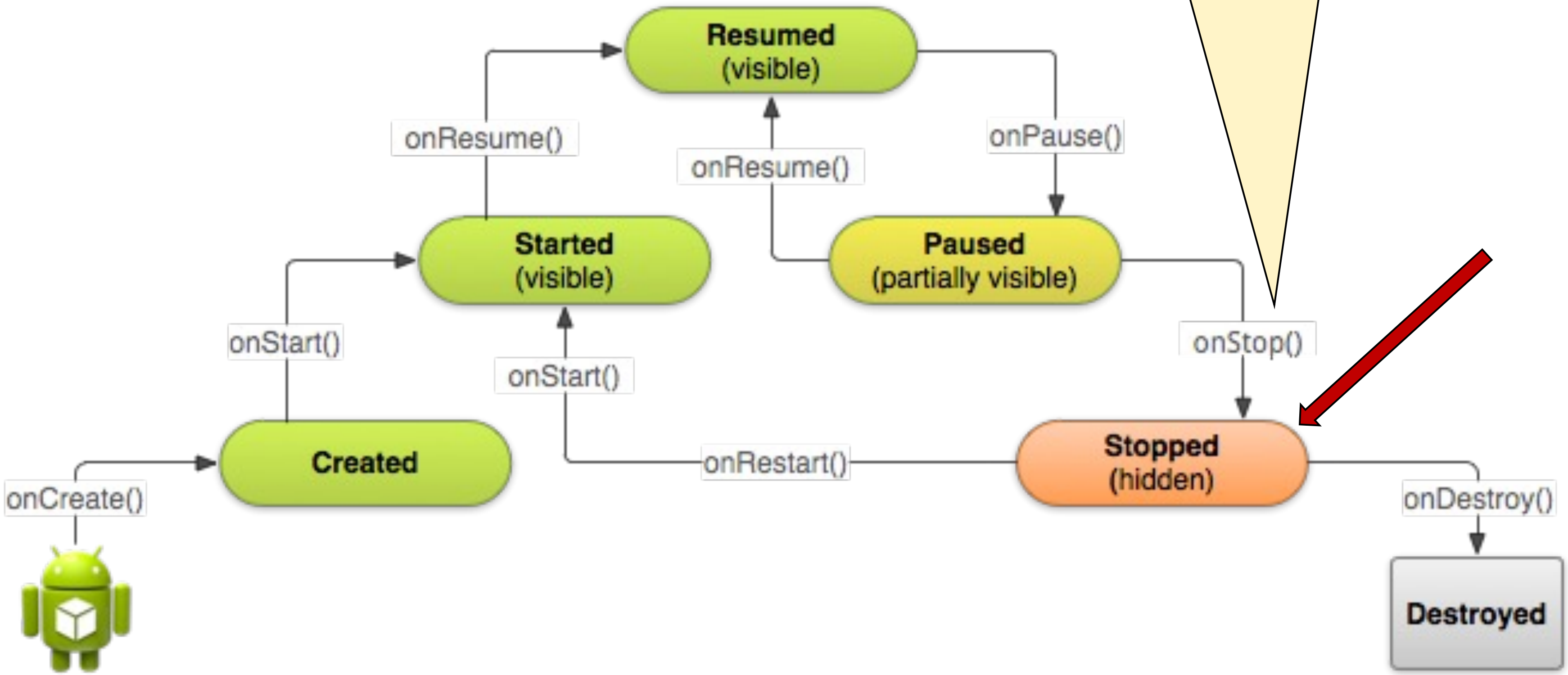
Stopping and Restarting an Activity



save application state
stop audio/animations/movie
release Android resources

From: Android Developer Fundamentals Course,
Google Developer Training

Stopping and Restarting an Activity

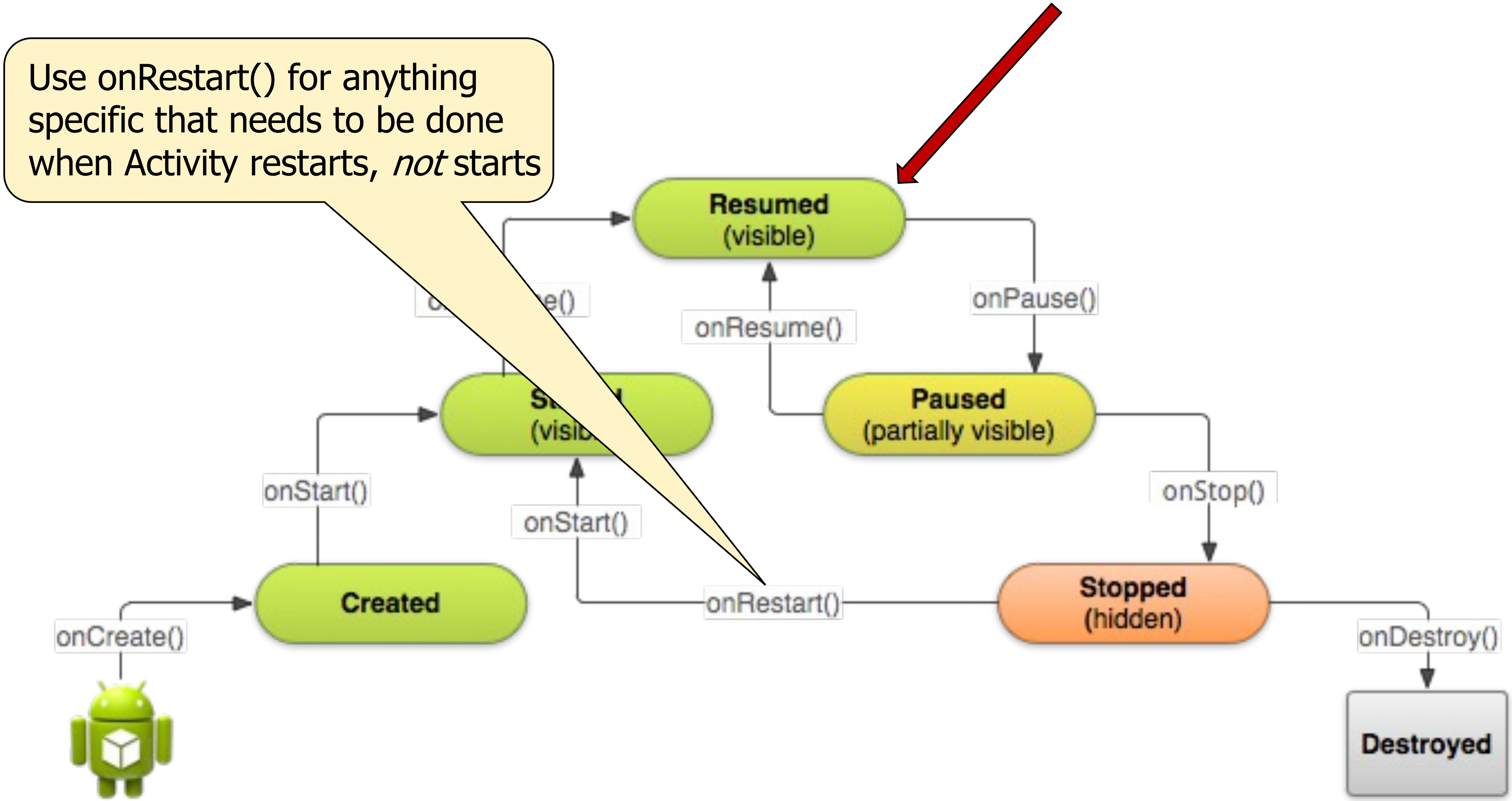


Use onStop() for CPU intensive operations, e.g., writing to a database

From: Android Developer Fundamentals Course,
Google Developer Training

Stopping and Restarting an Activity

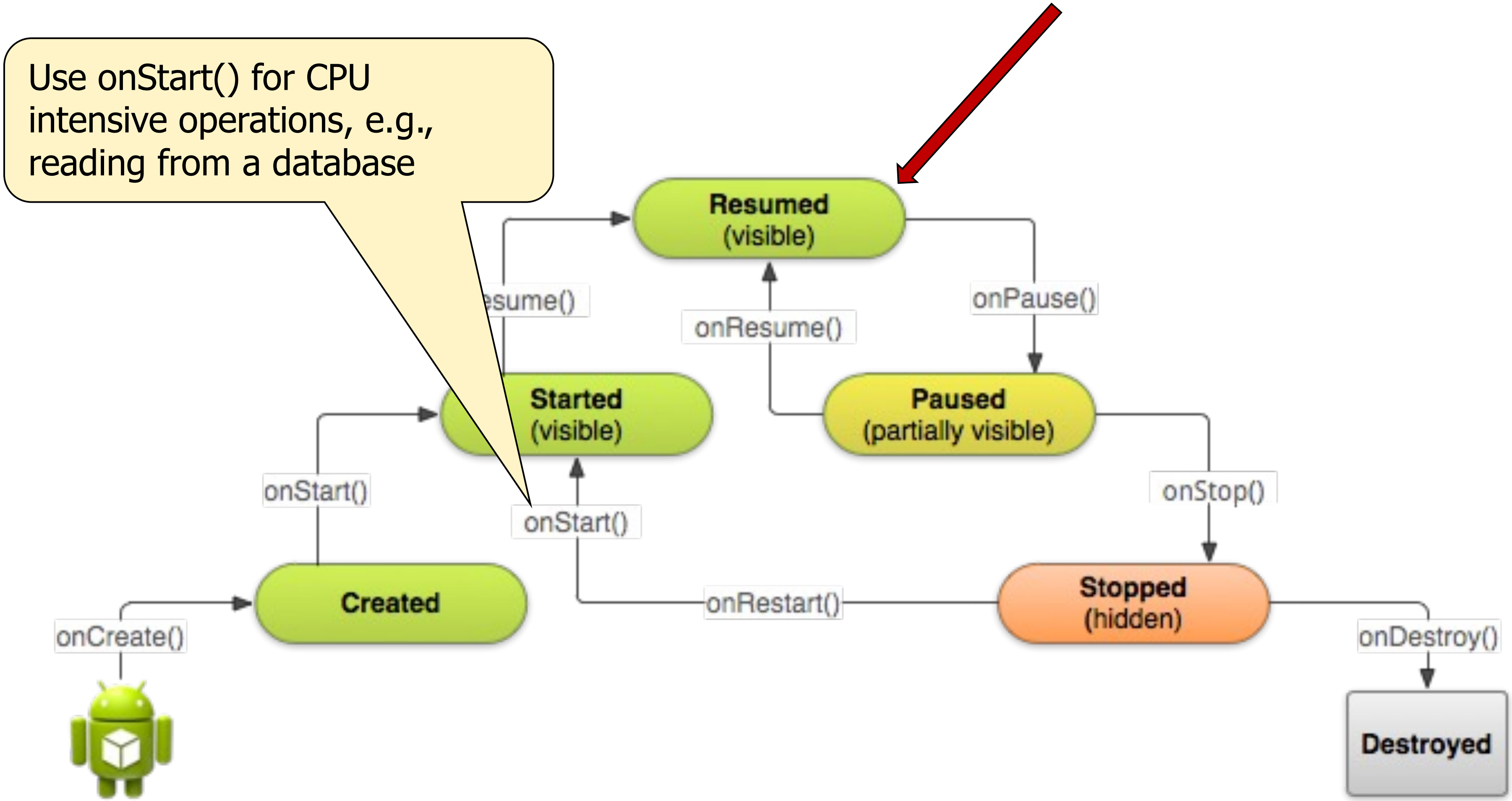
Use onRestart() for anything specific that needs to be done when Activity restarts, *not* starts



From: Android Developer Fundamentals Course,
Google Developer Training

Stopping and Restarting an Activity

Use onStart() for CPU intensive operations, e.g., reading from a database

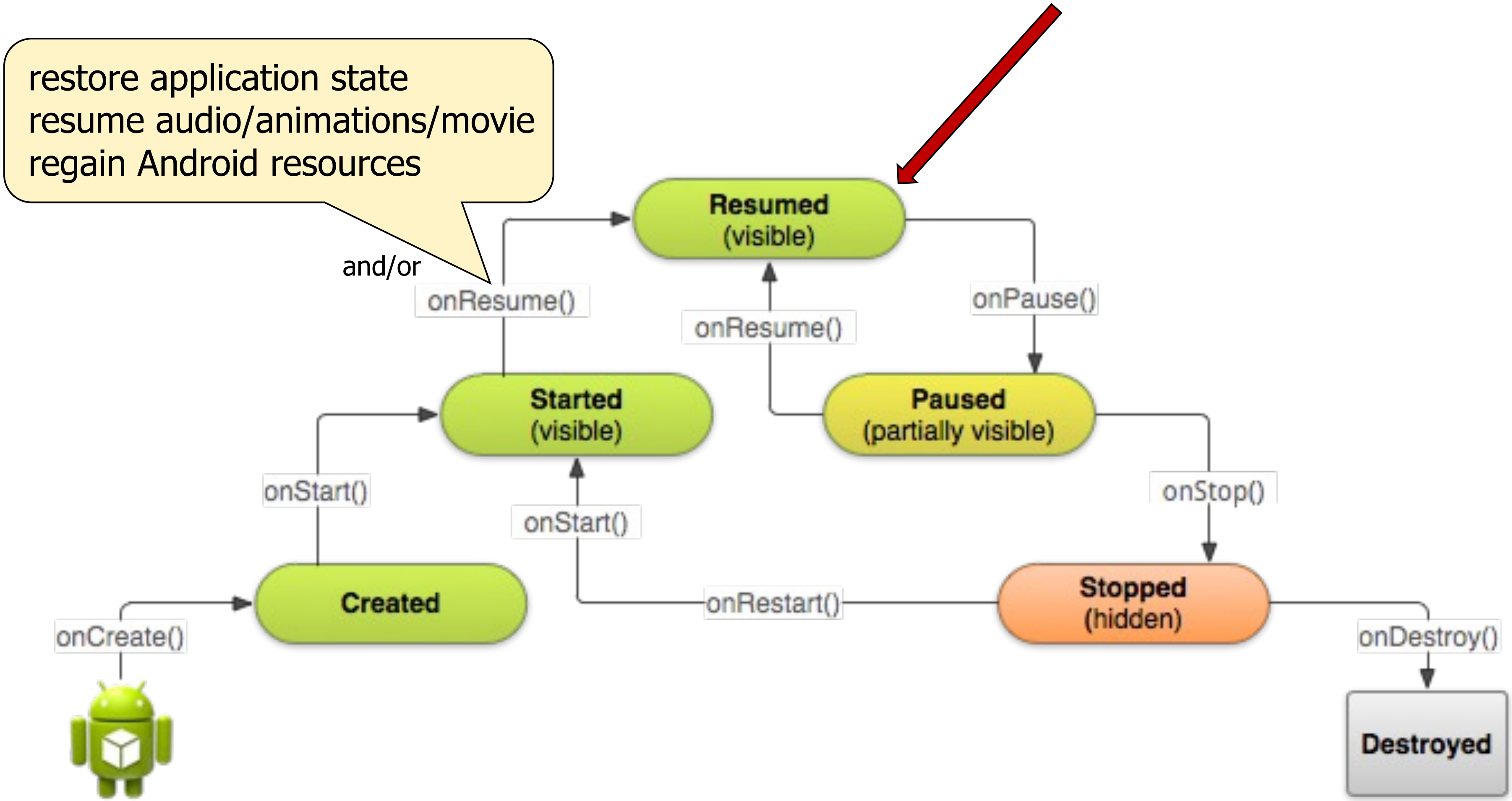


From: Android Developer Fundamentals Course,
Google Developer Training

Stopping and Restarting an Activity

restore application state
resume audio/animations/movie
regain Android resources

and/or



From: Android Developer Fundamentals Course,
Google Developer Training

Restarting a Killed Activity

- When a user presses a device’s back button, the current foreground activity instance is *destroyed permanently*
- Similarly, when the app is explicitly killed by the user, the activity is destroyed, as well
- However, when the user rotates the device (changes the screen’s orientation), Android kills the foreground activity and immediately *restarts* it

Restarting a Killed Activity

- To facilitate an activity restart, Android saves and then restores some (not all!), elements of the UI, e.g., contents of the EditTexts
- Some elements, modified by the application's listeners, are not saved/restored, e.g., text shown in TextViews
- Similarly, when Android kills a stopped activity due to low resources, it saves the UI state, as well, and restores it when the user comes back to the activity, later

Restarting a Killed Activity

- Consequently, for a possible activity's restart, the app should save its UI state data to a `Bundle` using `onSaveInstanceState(Bundle bundle)`.
- The `Bundle` class is a key-value store, similar to the `HashMap` class in Java, for saving/restoring UI state.

Restarting a Killed Activity

- `onSaveInstanceState()` is called after `onStop()` for apps targeting platforms starting with Pie. For earlier platforms, it will be called before `onStop()` (no guarantees if it is before or after `onPause()`).
- As already stated, for non-UI state saving steps, we should use `onPause()` and/or `onStop()`, *and not* `onSaveInstanceState()`

Restarting a Killed Activity

- In general, `onSaveInstanceState()` is invoked, when the user:
  - changed the orientation of the device (e.g., portrait to landscape)
  - pressed the home button
  - switched the current app with another one

Restarting a Killed Activity

- When an Activity is restarted by Android, the saved `Bundle` is passed as the argument to `onCreate()`, allowing the Activity to restore the exact state it was in, when it was killed
- Android also provides the `Bundle` data to the `onRestoreInstanceState()` callback

Restarting a Killed Activity

- In general, `onRestoreInstanceState()` is invoked right after `onStart()`, when an activity is restored, but only, when it was killed by the Android OS after the user:
  - changed the orientation of the device (e.g., portrait to landscape)
  - pressed the home button
  - switched the current app with another one (chose another from the list of apps)

Restarting a Killed Activity

- When the user changes the orientation of the device, the application is restarted to allow any changes in the layout and configuration of the app.

- To be precise, Android restarts the running Activity by calling `onPause()`, `onStop()`, and `onDestroy()`, and then immediately calling `onCreate()`, `onStart()`, and `onResume()`
  - `onCreate()` receives the saved `Bundle` as the argument

Destroying an Activity

- When an Activity is being destroyed, the `onDestroy()` method is usually called
- The `onDestroy()` method is called for one of two reasons:
    - The Activity completed its lifecycle *voluntarily* (e.g., user-initiated shutdown or orientation change)
    - The Activity is being killed by the OS because it needs the resources, but still has the time to gracefully destroy the Activity

Destroying an Activity

- To discover, call `isFinishing()`, which returns false if the Activity has been killed by Android and true otherwise.
- This method can be helpful in `onPause()` to know if the Activity is not going to resume right away (or in the near future).
- It may be able to use this as a hint to know how much instance state information to save or permanently persist.

Backward-Compatibile Activities

- When a new version of Android is released, there are many new APIs added, which are specifically designed for that version — and newer versions — provided those features are not deprecated or removed in future versions.
- The `Activity` class has received frequent updates with new features.
  - The downside of that means those features will not work on older versions of Android.

Backward-Compatibile Activities

- That is why `AppCompatActivity` class was introduced.
- `AppCompatActivity` provides the same functionality as the Activity class.
- To use AppCompatActivity, simply extend your custom Activity from AppCompatActivity instead of Activity and import the class as:

```
androidx.appcompat.app.AppCompatActivity
```

Activity Components as Fragments

- A `Fragment` is a subset (a fragment!) of a User Interface with its own lifecycle within an Activity.
  - A fragment is represented by the Fragment class (`androidx.fragment.app.Fragment`) and several supporting classes.
- A `Fragment` class instance must exist within an Activity instance (and its lifecycle).
- A `Fragment` does not have to be included in the same Activity class each time it's instantiated.

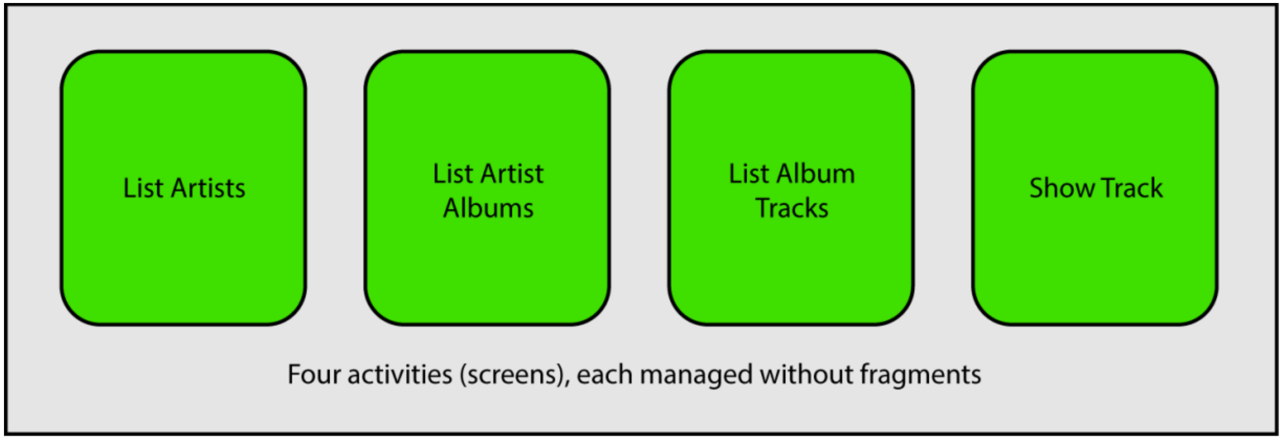
Activity Components as Fragments

- Consider an MP3 music player app. Following the one-screen-to-one-Activity rule:
  - List Artists Activity
  - List Artist Albums Activity
  - List Album Tracks Activity
  - Show Track Activity
- Each of these activities fills up an entire screen on a *smartphone*

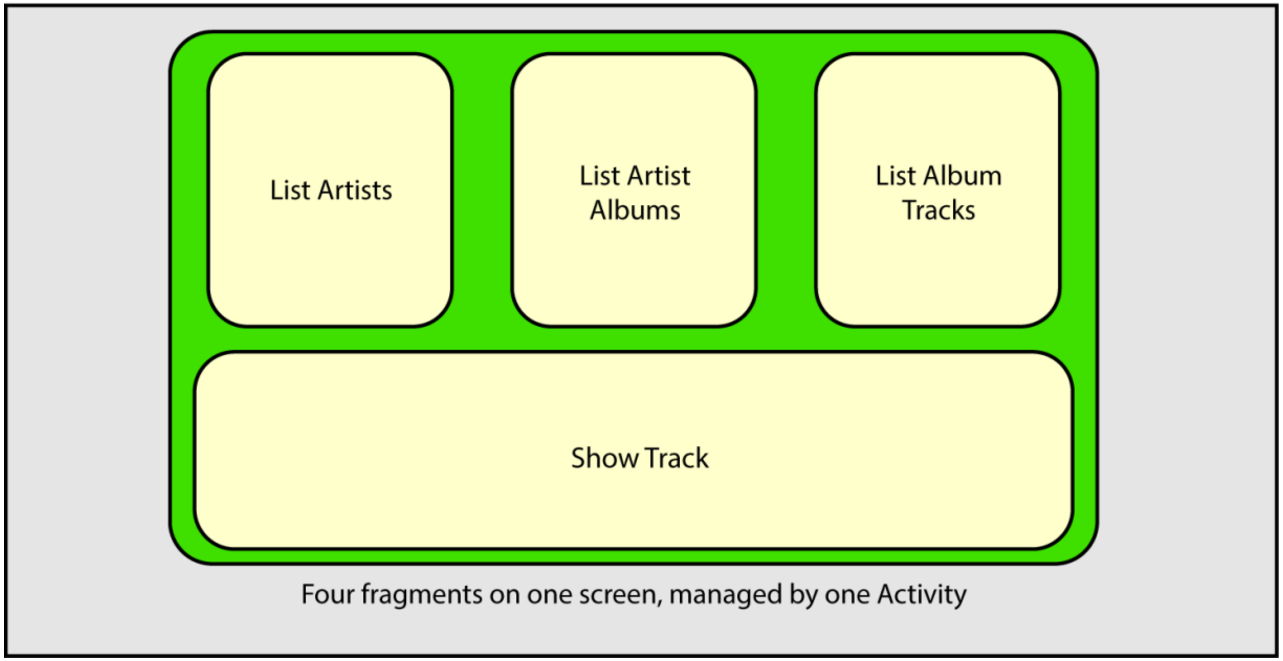
Activity Components as Fragments

- However, *all four* activities may fit on a single screen of a tablet (possibly a phablet):
  - Column 1 displays a list of artists.
    - Selecting an artist filters the second column.
  - Column 2 displays a list of that artist's albums.
    - Selecting an album filters the third column.
  - Column 3 displays a list of that album's tracks.
  - The bottom half of the screen, below all of the columns, displays the artist, album, or track art and details.

Activity Components as Fragments



Four activities (screens), each managed without fragments



Four fragments on one screen, managed by one Activity

Activity Components as Fragments

- We want to avoid having to build different activities for different-size devices.
- Subsets of screen features may be factored out as four fragments
- The layouts/app code can mix and match them on the fly these fragments, while still having only one code base

Activity Transitions with Intents

- Users transition between a number of different Activity instances.
- There are several ways in which an activity can be switched.
- Developers need to pay attention to the Activity lifecycle during these transitions.

Activity Transitions with Intents

- Ways to handle permanent Activity transitions:
  - `startActivity()` and `finish()`
- Ways to handle temporary transitions with plans to return a result:
  - `registerForActivityResult()` and `ActivityResultLauncher()`
  - `startActivityForResult()` has been deprecated

Activity Transitions with Intents

- Android applications can have multiple entry points.
- A specific Activity can be designated as the main Activity to launch by default.
- Other activities might be designated to launch under specific circumstances.

Launching a New Activity

- You can start activities in several ways. For example:
  - Use a Context object to call `startActivity()`.
  - `startActivity()` takes a single parameter, an `Intent`.
- An `Intent` (`android.content.Intent`) is an *asynchronous* message mechanism. It is used by Android to match task requests with the appropriate Activity or Service and to dispatch broadcast Intent events to the system.

Launching a New Activity

- An example of using `startActivity`:

```
startActivity( new Intent(getApplicationContext(),
               MyDrawActivity.class) );
```

Intents with Action and Data

- The internals of an `Intent` object are composed of two main parts:
  - The action to be performed, and
  - optionally, the data to be used
- You can also specify action/data pairs using Intent Action types and URI objects.
- Therefore, an Intent is basically saying “do this” (the action) to “that” (the URI describing on what resource the action is performed).

Intents with Action and Data

- The most common action types are defined in the `Intent` class, including:
  - ACTION_MAIN
    - Describes the main entry point of an Activity
  - ACTION_EDIT
    - Used in conjunction with a URI to the data edited
- You also find action types that generate integration points with activities in other applications:
  - For example, the Web browser or Phone Dialer

Intents For a Different App

- With the appropriate permissions, applications might also launch external activities within other applications.
  - For example, a Customer Relationship Management (CRM) app might launch the Contacts app to browse the Contacts database, choose a specific contact, and return that contact’s unique identifier to the CRM application for use.

Intents For a Different App

```
Uri number = Uri.parse( "tel:7065422911” );
Intent dial = new Intent( Intent.ACTION_DIAL, number );
startActivity(dial);
```

OR

```
String searchTerms = "uga school of computing";
Intent webSearch = new Intent( Intent.ACTION_WEB_SEARCH );
webSearch.putExtra( SearchManager.QUERY, searchTerms );
startActivity( webSearch );
```

Intents For a Different App

- A list of commonly used Google application intents is available at:
  - http://d.android.com/guide/components/intents-common.html (calendar, camera, email, maps, video, phone, text message, etc.)
  - http://www.openintents.org
- A growing list of intents is available from third-party applications and those within the Android SDK.

Additional Information in Intents

- It is possible to include additional data in an `Intent`.
- The Extras property of an Intent is stored in a `Bundle` object.
- The `Intent` class also has a number of helper methods for getting and setting name/value pairs for many common data types.

Additional Information in Intents

- For example:

```
Intent intent = new Intent(this, MyActivity.class);
intent.putExtra("SomeStringData",”Hi there!");
intent.putExtra("SomeBooleanData",false);
startActivity(intent);
```

- Then, in the `onCreate()` of the `MyActivity` class can access the data:

```
Bundle extras = getIntent().getExtras();
if (extras != null) {
    String myStr = extras.getString("SomeStringData");
    Boolean myBool = extras.getString("SomeBooleanData");
}
```

Navigation in an Application

- An app likely has a number of screens (each with its own Activity).
- There is a close relationship between activities and intents, and application navigation.
  - A menu paradigm may be used in several different ways for app’s navigation:
    - Main menu or list-style screen
    - Navigation-drawer-style screen
    - Master-detail-style screen
    - Click or Swipe actions
    - ActionBar-style navigation

Working with Services

- A `Service` (`android.app.Service`) can be thought of as a component that has no UI screen.
- An Android Service can be one of two things, or both:
  - It can be used to perform lengthy operations beyond the scope of a single Activity.
  - It can be the server of a client/server relationship for providing functionality through remote invocation via inter-process communication (IPC).
- A Service is often used to control long-running server operations.
- Generally, use a Service when no user input is required.

Working with Services

- Use a Service, if the task:
  - requires the use of a worker thread, or
  - might affect application responsiveness and performance, and is not time sensitive to the application
  - consider implementing a service to handle the task outside the main application and any individual Activity lifecycles.

Working with Services

- Examples of when to implement a Service:
  - A weather, email, or social network app
    - Routinely check for updates on the network
  - A game
    - Downloading and processing content for the next level before the user needs it
  - A photo or media app
    - To keep data in sync online
    - To package and upload new content in the background
  - A news application
    - For "preloading" content by continually downloading news stories in advance, to improve performance and responsiveness

Receiving and Broadcasting Intents

- Intents serve other purposes:
  - You can broadcast an Intent (via a call to `sendBroadcast()`) to the Android system, allowing any interested application (called a `BroadcastReceiver`) to receive that broadcast and act upon it.
  - Your application might send off as well as listen for Intent broadcasts.
  - Broadcasts are generally used to inform the system that something interesting has happened.

Receiving and Broadcasting Intents

- Your application can also share information using this same broadcast mechanism.
  - For example, an email application might broadcast an Intent whenever a new email arrives so that other applications (such as spam filters or antivirus apps) that might be interested in this type of event can react to it.

Summary

- We have learned important Android terminology.
- We have learned what the Application Context is and how to use it.
- We have learned the importance of the Activity lifecycle.
- We have learned how fragments improve the overall structure of an application.
- We have learned how to manage Activity transitions and organize navigation with intents.
- We have learned about the usefulness of services.
- We have learned about receiving and broadcasting intents.

References and More Information

- Android SDK reference regarding the application Context class:
  - http://d.android.com/reference/android/content/Context.html
- Android SDK reference regarding the Activity class:
  - http://d.android.com/reference/android/app/Activity.html
- Android SDK reference regarding the Fragment class:
  - http://d.android.com/reference/android/app/Fragment.html
- Android API guides: "Fragments":
  - http://d.android.com/guide/components/fragments.html

References and More Information

- Android tools: Support Library:
  - http://d.android.com/tools/support-library/index.html
- Android API guides: “Intents and Intent Filters”:
  - http://d.android.com/guide/components/intents-filters.html
- Android SDK Reference regarding the JobScheduler class:
  - http://d.android.com/reference/android/app/job/JobScheduler.html