UNDERSTANDING ANDROID APPLICATIONS AND ACTIVITY LIFECYCLES

ROMI FADILLAH RAHMAT

RESOURCE CONSTRAINT

- We have learned that Android applications run within processes and that they are comprised of multiple components in the form of activities, Services and Broadcast Receivers.
- It is important to keep in mind that these devices are still considered to be "resource constrained" by the standards of modern desktop and laptop based systems, particularly in terms of memory.
- As such, a key responsibility of the Android system is to ensure that these limited resources are managed effectively and that both the operating system and the applications running on it remain responsive to the user at all times. In order to achieve this, Android is given full control over the lifecycle and state of both the processes in which the applications run, and the individual components that comprise those applications.

IMPORTANT NOTE

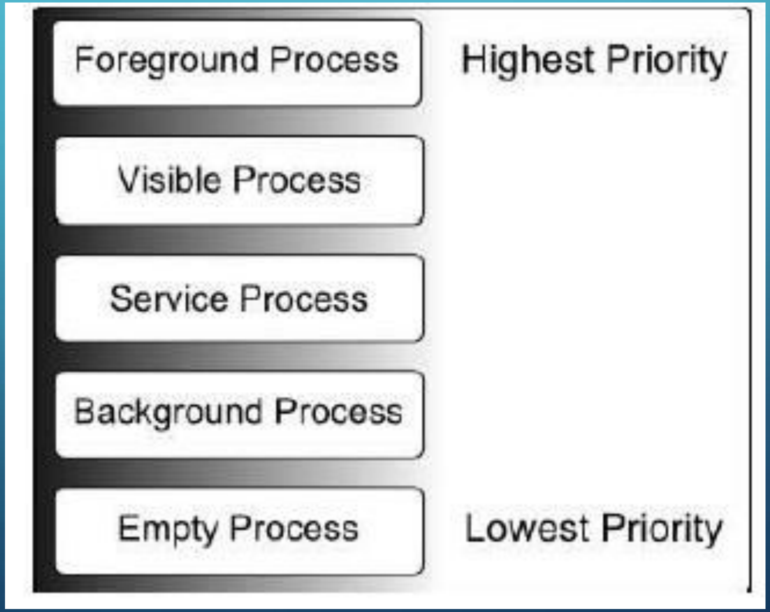
- An important factor in developing Android applications, therefore, is to gain an understanding of both the application and activity lifecycle management models of Android, and the ways in which an application can react to the state changes that are likely to be imposed upon it during its execution lifetime.

ANDROID APPLICATIONS AND RESOURCE MANAGEMENT

- Each running Android application is viewed by the operating system as a separate process.
- If the system identifies that resources on the device are reaching capacity it will take steps to terminate processes to free up memory.
- When making a determination as to which process to terminate in order to free up memory, the system takes into consideration both the *priority* and *state* of all currently running processes, combining these factors to create what is referred to by Google as an *importance hierarchy*. Processes are then terminated starting with the lowest priority and working up the hierarchy until sufficient resources have been liberated for the system to function.

ANDROID PROCESS STATES

- Processes host applications and applications are made up of components. Within an Android system, the current state of a process is defined by the highest-ranking active component within the application that it hosts. As outlined here, a process can be in one of the following five states at any given time:



FOREGROUND PROCESS

- These processes are assigned the highest level of priority. At any one time, there are unlikely to be more than one or two foreground processes active and these are usually the last to be terminated by the system. A process must meet one or more of the following criteria to qualify for foreground status:
  - Hosts an activity with which the user is currently interacting.
  - Hosts a Service connected to the activity with which the user is interacting.
  - Hosts a Service that has indicated, via a call to *startForeground()*, that termination would be disruptive to the user experience.
  - Hosts a Service executing either its *onCreate()*, *onResume()* or *onStart()* callbacks.
  - Hosts a Broadcast Receiver that is currently executing its *onReceive()* method.

VISIBLE PROCESS

- A process containing an activity that is visible to the user but is not the activity with which the user is interacting is classified as a “visible process”. This is typically the case when an activity in the process is visible to the user but another activity, such as a partial screen or dialog, is in the foreground.
- A process is also eligible for visible status if it hosts a Service that is, itself, bound to a visible or foreground activity.

SERVICE PROCESS

- Processes that contain a Service that has already been started and is currently executing.

BACKGROUND PROCESS

- A process that contains one or more activities that are not currently visible to the user, and does not host a Service that qualifies for *Service Process* status.
- Processes that fall into this category are at high risk of termination in the event that additional memory needs to be freed for higher priority processes.
- Android maintains a dynamic list of background processes, terminating processes in chronological order such that processes that were the least recently in the foreground are killed first.

EMPTY PROCESS

- Empty processes no longer contain any active applications and are held in memory ready to serve as hosts for newly launched applications.
- This is somewhat analogous to keeping the doors open and the engine running on a bus in anticipation of passengers arriving. Such processes are, obviously, considered the lowest priority and are the first to be killed to free up resources.

INTER PROCESS DEPENDENCIES

- The situation with regard to determining the highest priority process is slightly more complex than outlined in the preceding section for the simple reason that processes can often be inter-dependent.
- As such, when making a determination as to the priority of a process, the Android system will also take into consideration whether the process is in some way serving another process of higher priority (for example, a service process acting as the content provider for a foreground process).
- As a basic rule, the Android documentation states that a process can never be ranked lower than another process that it is currently serving.

ACTIVITY LIFE CYCLE

- As we have previously determined, the state of an Android process is determined largely by the status of the activities and components that make up the application that it hosts.
- It is important to understand, therefore, that these activities also transition through different states during the execution lifetime of an application.
- The current state of an activity is determined, in part, by its position in something called the *Activity Stack*.

ACTIVITY STACK

- For each application that is running on an Android device, the runtime system maintains an *Activity Stack*.
- When an application is launched, the first of the application's activities to be started is placed onto the stack. When a second activity is started, it is placed on the top of the stack and the previous activity is *pushed* down. The activity at the top of the stack is referred to as the *active* (or *running*) activity. When the active activity exits, it is *popped* off the stack by the runtime and the activity located immediately beneath it in the stack becomes the current active activity. The activity at the top of the stack might, for example, simply exit because the task for which it is responsible has been completed. Alternatively, the user may have selected a "Back" button on the screen to return to the previous activity, causing the current activity to be popped off the stack by the runtime system and therefore destroyed.
- A visual representation of the Android Activity Stack is illustrated below

ACTIVITY STACK

- New activities are pushed on to the top of the stack when they are started.
- The current active activity is located at the top of the stack until it is either pushed down the stack by a new activity, or popped off the stack when it exits or the user navigates to the previous activity.
- In the event that resources become constrained, the runtime will kill activities, starting with those at the bottom of the stack.
- The Activity Stack is what is referred to in programming terminology as a Last-In-First-Out (LIFO) stack in that the last item to be pushed onto the stack is the first to be popped off.

ACTIVITY STATES

- An activity can be in one of a number of different states during the course of its execution within an application:
  - **Active / Running** – The activity is at the top of the Activity Stack, is the foreground task visible on the device screen, has focus and is currently interacting with the user. This is the least likely activity to be terminated in the event of a resource shortage.
  - **Paused** – The activity is visible to the user but does not currently have focus (typically because this activity is partially obscured by the current *active* activity). Paused activities are held in memory, remain attached to the window manager, retain all state information and can quickly be restored to active status when moved to the top of the Activity Stack.
  - **Stopped** – The activity is currently not visible to the user (in other words it is totally obscured on the device display by other activities). As with paused activities, it retains all state and member information, but is at higher risk of termination in low memory situations.
  - **Killed** – The Activity has been terminated by the runtime system in order to free up memory and is no longer present on the Activity Stack. Such activities must be restarted if required by the application.

CONFIGURATION CHANGE

- We have looked at two of the causes for the change in state of an Android activity, namely the movement of an activity between the foreground and background, and termination of an activity by the runtime system in order to free up memory. In fact, there is a third scenario in which the state of an activity can dramatically change and this involves a change to the device configuration.
- By default, any configuration change that impacts the appearance of an activity (such as rotating the orientation of the device between portrait and landscape, or changing a system font setting) will cause the activity to be destroyed and recreated. The reasoning behind this is that such changes affect resources such as the layout of the user interface and simply destroying and recreating impacted activities is the quickest way for an activity to respond to the configuration change.

ACTIVITY CLASS

- With few exceptions, activities in an application are created as subclasses of either the Android *Activity* class, or another class that is, itself, subclassed from the Activity class (for example the *ActionBarActivity* or *FragmentActivity* classes).

DYNAMIC VS PERSISTENT STATE

- A key objective of Activity lifecycle management is ensuring that the state of the activity is saved and restored at appropriate times.
- When talking about *state* in this context we mean the data that is currently being held within the activity and the appearance of the user interface.
- The activity might, for example, maintain a data model in memory that needs to be saved to a database, content provider or file.
- Such state information, because it persists from one invocation of the application to another, is referred to as the *persistent state*.

DYNAMIC AND PERSISTENT STATE

- The appearance of the user interface (such as text entered into a text field but not yet committed to the application’s internal data model) is referred to as the *dynamic state*, since it is typically only retained during a single invocation of the application (and also referred to as *user interface state* or *instance state*).
- Understanding the differences between these two states is important because both the ways they are saved, and the reasons for doing so, differ.

THE DIFFERENCE : PERSISTANT VS DYNAMIC

- The purpose of saving the persistent state is to avoid the loss of data that may result from an activity being killed by the runtime system while in the background.
- The dynamic state, on the other hand, is saved and restored for reasons that are slightly more complex.
- The main purpose of saving dynamic state, therefore, is to give the perception of seamless switching between foreground and background activities, regardless of the fact that activities may actually have been killed and restarted without the user's knowledge.

EXAMPLE WHY WE USE DYNAMIC STATES

- An application contains an activity (which we will refer to as *Activity A*) containing a text field and some radio buttons. During the course of using the application, the user enters some text into the text field and makes a selection from the radio buttons. Before performing an action to save these changes, however, the user then switches to another activity causing *Activity A* to be pushed down the Activity Stack and placed into the background. After some time, the runtime system ascertains that memory is low and consequently kills *Activity A* to free up resources. As far as the user is concerned, however, *Activity A* was simply placed into the background and is ready to be moved to the foreground at any time. On returning *Activity A* to the foreground the user would, quite reasonably, expect the entered text and radio button selections to have been retained. In this scenario, however, a new instance of *Activity A* will have been created and, if the dynamic state was not saved and restored, the previous user input lost.

THE ANDROID ACTIVITY LIFECYCLE METHODS

- As previously explained, the Activity class contains a number of lifecycle methods which act as event handlers when the state of an Activity changes. The primary methods supported by the Android Activity class are as follows:
  - **onCreate(Bundle savedInstanceState)** – The method that is called when the activity is first created and the ideal location for most initialization tasks to be performed. The method is passed an argument in the form of a *Bundle* object that may contain dynamic state information (typically relating to the state of the user interface) from a prior invocation of the activity.
  - **onRestart()** – Called when the activity is about to restart after having previously been stopped by the runtime system.
  - **onDestroy()** – The activity is about to be destroyed, either voluntarily because the activity has completed its tasks and has called the *finish()* method or because the runtime is terminating it either to release memory or due to a configuration change (such as the orientation of the device changing). It is important to note that a call will not always be made to *onDestroy()* when an activity is terminated.

CONT'D

- **onStart()** – Always called immediately after the call to the *onCreate()* or *onRestart()* methods, this method indicates to the activity that it is about to become visible to the user. This call will be followed by a call to *onResume()* if the activity moves to the top of the activity stack, or *onStop()* in the event that it is pushed down the stack by another activity.
- **onResume()** – Indicates that the activity is now at the top of the activity stack and is the activity with which the user is currently interacting.
- **onPause()** – Indicates that a previous activity is about to become the foreground activity. This call will be followed by a call to either the *onResume()* or *onStop()* method depending on whether the activity moves back to the foreground or becomes invisible to the user. Steps should be taken within this method to store any *persistent data* required by the activity (such as data stored to a content provider, database or file). This method should also ensure that any CPU intensive tasks such as animation are stopped.
- **onStop()** – The activity is now no longer visible to the user. The two possible scenarios that may follow this call are a call to *onRestart()* in the event that the activity moves to the foreground again, or *onDestroy()* if the activity is being terminated.

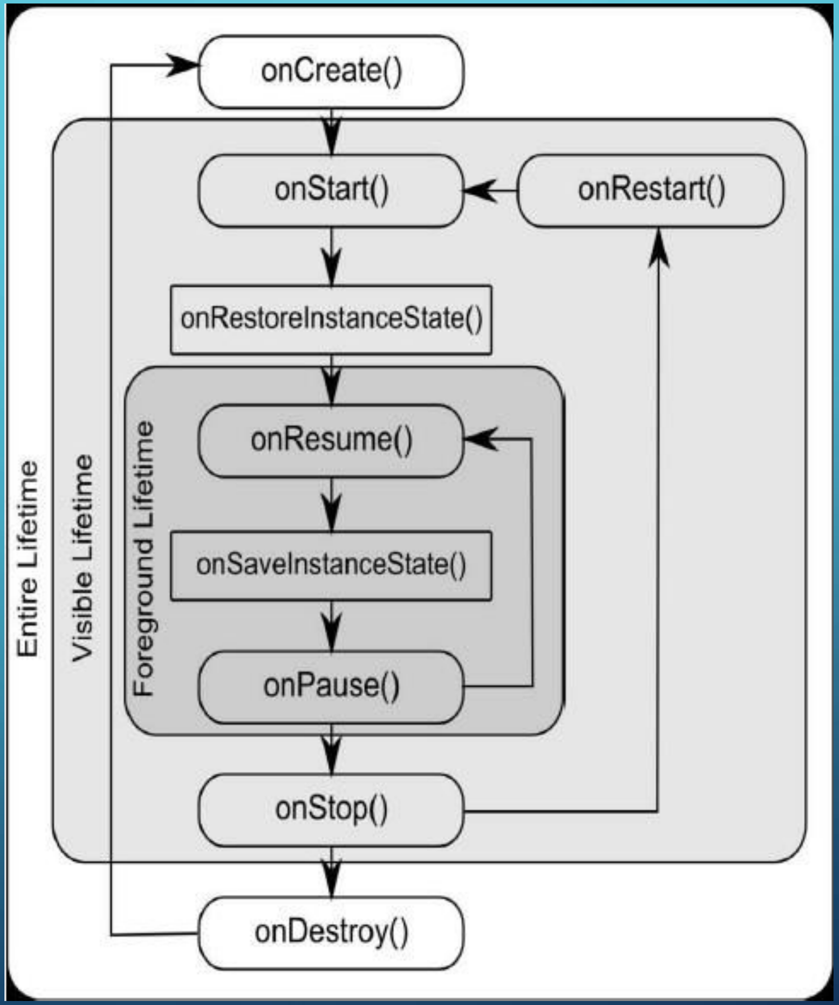
FOR DYNAMIC STATE

- In addition to the lifecycle methods outlined above, there are two methods intended specifically for saving and restoring the *dynamic state* of an activity:
  - **onRestoreInstanceState(Bundle savedInstanceState)** – This method is called immediately after a call to the *onStart()* method in the event that the activity is restarting from a previous invocation in which state was saved. As with *onCreate()*, this method is passed a Bundle object containing the previous state data. This method is typically used in situations where it makes more sense to restore a previous state after the initialization of the activity has been performed in *onCreate()* and *onStart()*.
  - **onSaveInstanceState(Bundle outState)** – Called before an activity is destroyed so that the current *dynamic state* (usually relating to the user interface) can be saved. The method is passed the Bundle object into which the state should be saved and which is subsequently passed through to the *onCreate()* and *onRestoreInstanceState()* methods when the activity is restarted. Note that this method is only called in situations where the runtime ascertains that dynamic state needs to be saved.

ACTIVITY LIFETIME

- The final topic to be covered involves an outline of the *entire*, *visible* and *foreground* lifetimes through which an activity will transition during execution:
  - **Entire Lifetime** –The term “entire lifetime” is used to describe everything that takes place within an activity between the initial call to the *onCreate()* method and the call to *onDestroy()* prior to the activity terminating.
  - **Visible Lifetime** – Covers the periods of execution of an activity between the call to *onStart()* and *onStop()*. During this period the activity is visible to the user though may not be the activity with which the user is currently interacting.
  - **Foreground Lifetime** – Refers to the periods of execution between calls to the *onResume()*and *onPause()* methods. It is important to note that an activity may pass through the *foreground* and *visible* lifetimes multiple times during the course of the *entire* lifetime.

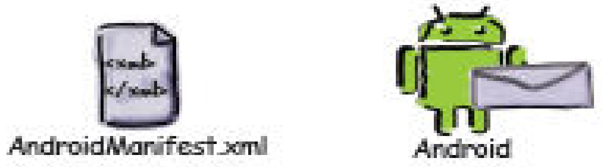
ILLUSTRATION



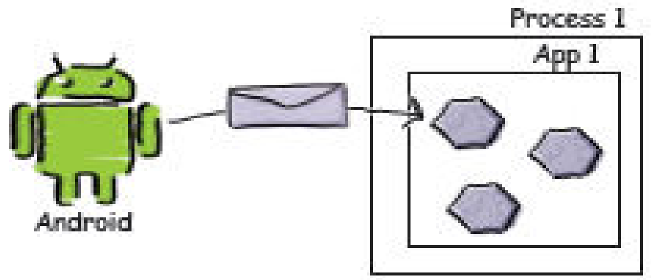
WHAT HAPPEN ACTUALLY ???

1 **The user decides she wants to run the app.**
She clicks on the icon for the app on her device.

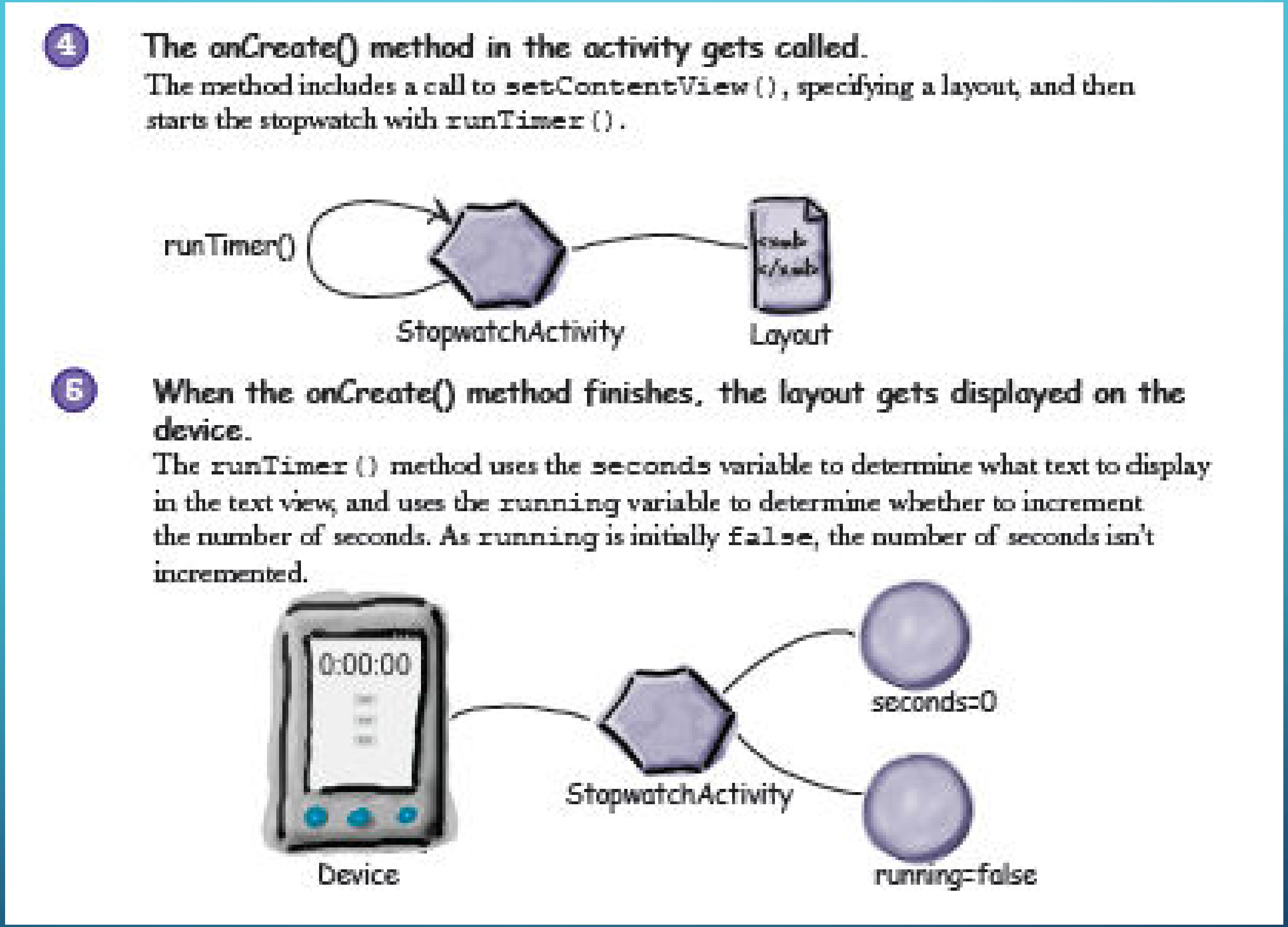
2 **The AndroidManifest.xml file for the app specifies which activity to use as the launch activity.**
An intent is constructed to start this activity using `startActivity(intent)`.



3 **Android checks if there's already a process running for the app, and if not, creates a new process.**
It then creates a new activity object—in this case, for `StopwatchActivity`.



WHAT HAPPEN ACTUALLY ???

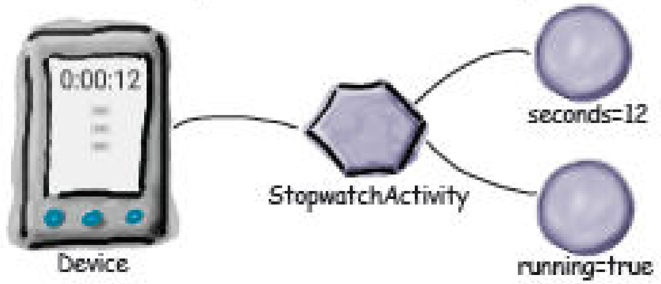


4 **The onCreate() method in the activity gets called.**
The method includes a call to `setContentView()`, specifying a layout, and then starts the stopwatch with `runTimer()`.

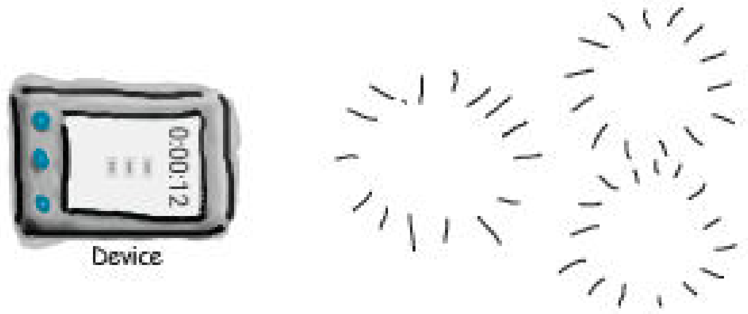
runTimer()
StopwatchActivity
Layout

5 **When the onCreate() method finishes, the layout gets displayed on the device.**
The `runTimer()` method uses the `seconds` variable to determine what text to display in the text view, and uses the `running` variable to determine whether to increment the number of seconds. As `running` is initially `false`, the number of seconds isn't incremented.

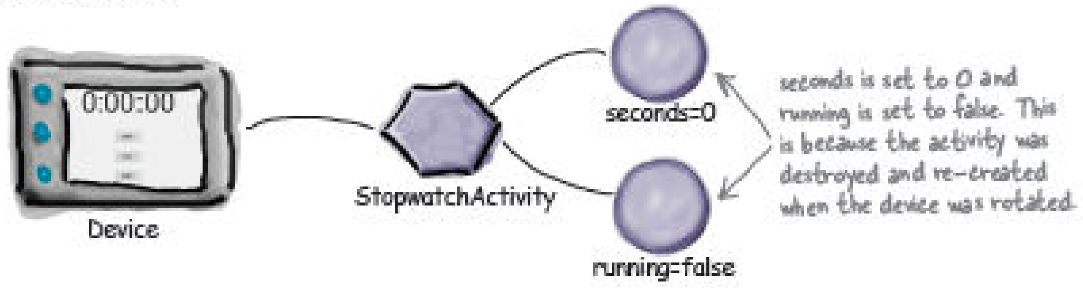
1. **The user starts the app, and clicks on the start button to set the stopwatch going.**
   The `runTimer()` method starts incrementing the number of seconds displayed in the `time_view` text view using the `seconds` and `running` variables.



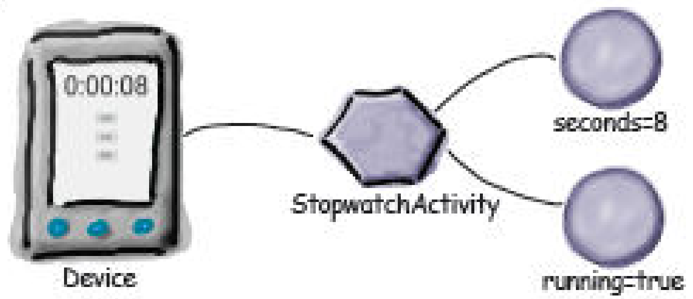
2. **The user rotates the device.**
   Android sees that the screen orientation and screen size has changed, and it destroys the activity, including any variables used by the `runTimer()` method.



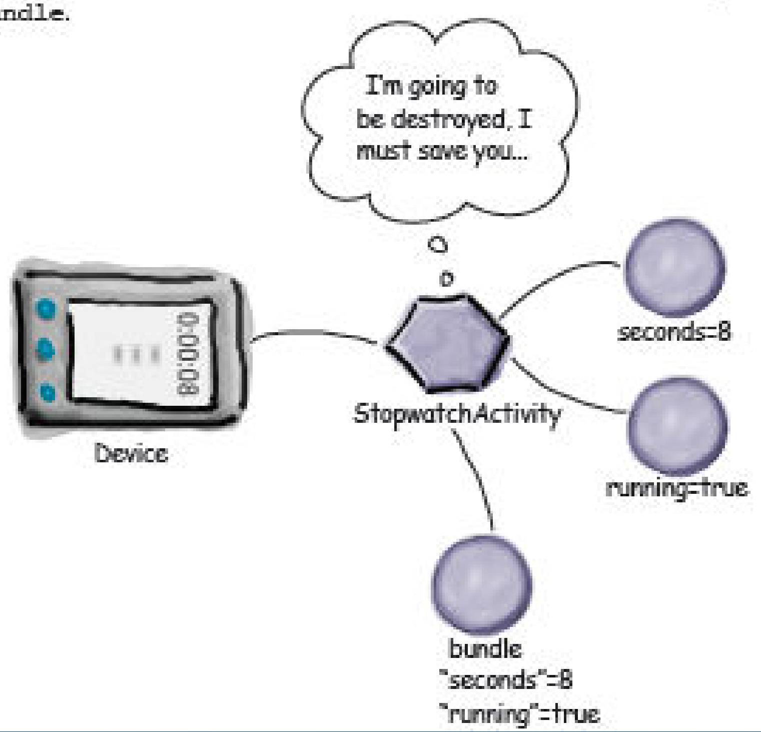
3. **StopwatchActivity is then re-created.**
   The `onCreate()` method runs again, and the `runTimer()` method gets called. As the activity has been re-created, the `seconds` and `running` variables are set to their default values.


0:00:00
Device
StopwatchActivity
seconds=0
running=false
seconds is set to 0 and running is set to false. This is because the activity was destroyed and re-created when the device was rotated.

1 **The user starts the app, and clicks on the start button to set the stopwatch going.**
The `runTimer()` method starts incrementing the number of seconds displayed in the `time_view` text view.

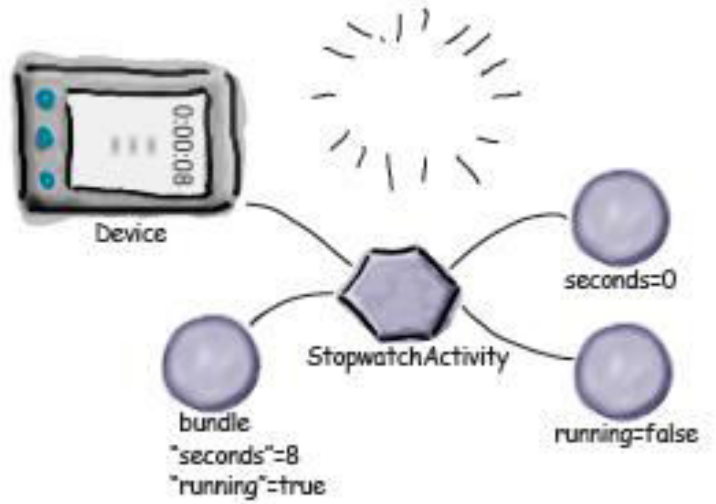


2 **The user rotates the device.**
Android views this as a configuration change, and gets ready to destroy the activity. Before the activity is destroyed, `onSaveInstanceState()` gets called. The `onSaveInstanceState()` method saves the seconds and running values to a `Bundle`.



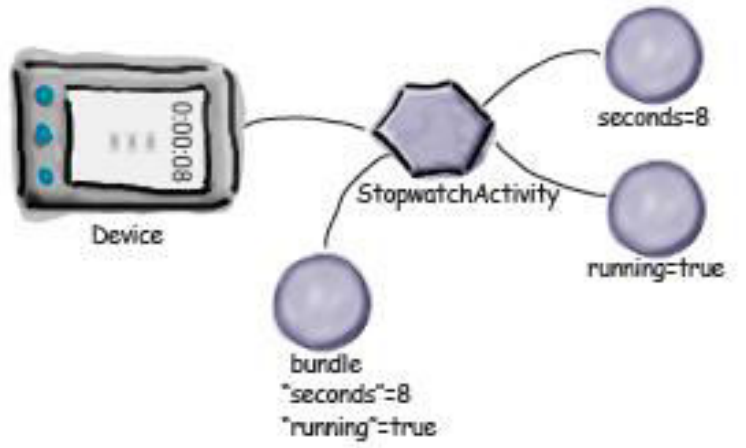
4 **Android destroys the activity, and then re-creates it.**

The `onCreate()` method gets called, and the `Bundle` gets passed to it.



6 **The Bundle contains the values of the seconds and running variables as they were before the activity was destroyed.**

Code in the `onCreate()` method set the current variables to the values in the `Bundle`.



6 **The runTimer() method gets called, and the timer picks up where it left off.**

The stopwatch gets displayed on the device.