4 the activity lifecycle

Being an Activity

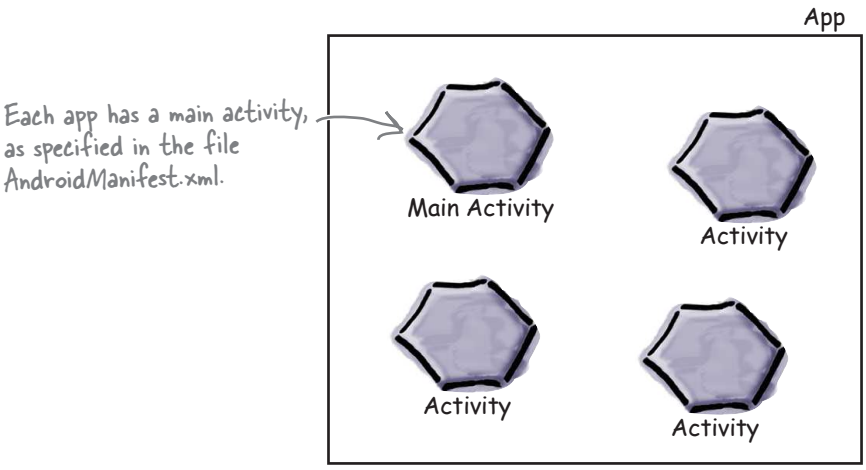
Activities form the foundation of every Android app.

So far you've seen how to create activities, and made one activity start another using an intent. But ***what's really going on beneath the hood?*** In this chapter, we're going to dig a little deeper into **the activity lifecycle**. What happens when an activity is **created** and **destroyed**? Which methods get called when an activity is **made visible and appears in the foreground**, and which get called when the activity **loses the focus and is hidden**? And **how do you save and restore your activity's state**? Read on to find out.

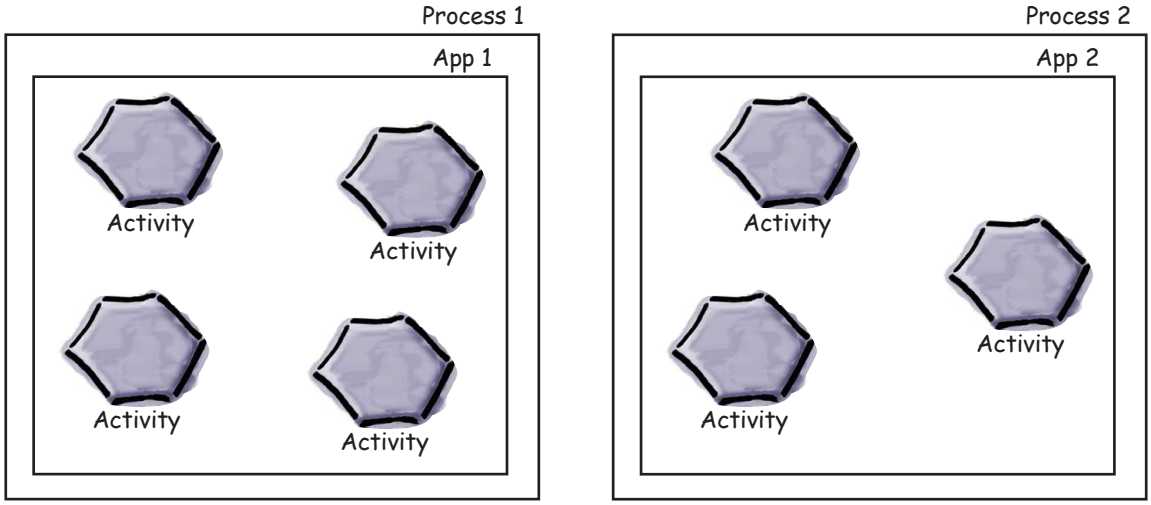
How do activities really work?

So far you've seen how to create apps that interact with the user, and apps that use multiple activities to perform tasks. Now that you have these core skills under your belt, it's time to take a deeper look at how activities *actually work*. Here's a recap of what you know so far, with a few extra details thrown in.

An app is a collection of activities, layouts, and other resources.
One of these activities is the main activity for the app.

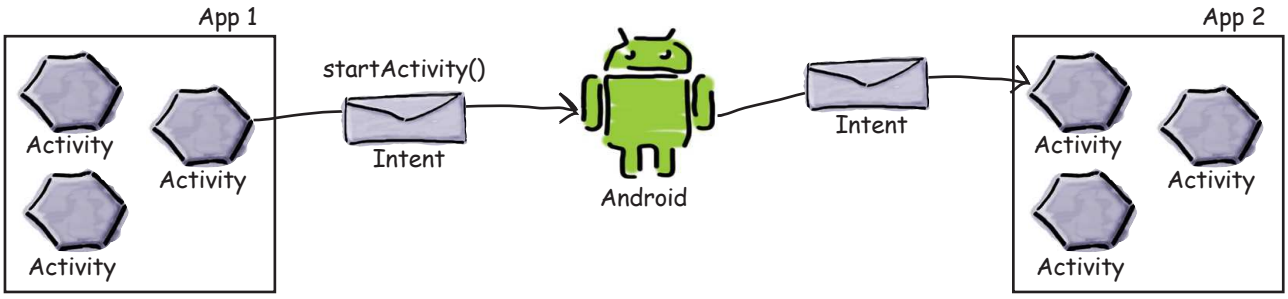


By default, each app runs within its own process.
This helps keep your apps safe and secure. You can read more about this in Appendix III (which covers the Android runtime, a.k.a. ART) at the back of this book.



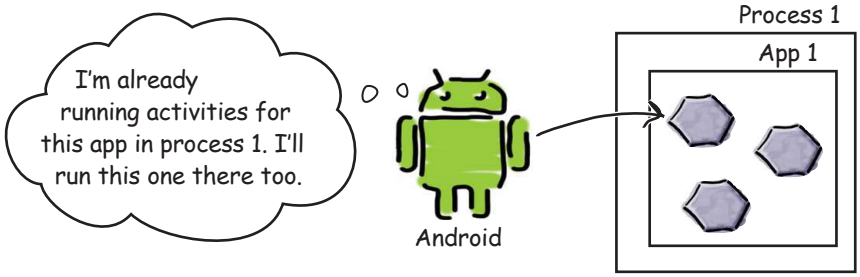
Your app can start an activity in another application by passing an intent with startActivity().

The Android system knows about all the device's installed apps and their activities, and uses the intent to start the correct activity.



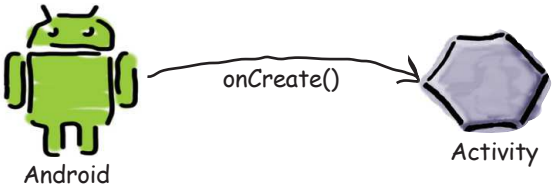
When an activity needs to start, Android checks whether there's already a process for that app.

If one exists, Android runs the activity in that process. If one doesn't exist, Android creates one.



When Android starts an activity, it calls its onCreate() method.

`onCreate()` is always run whenever an activity gets created.



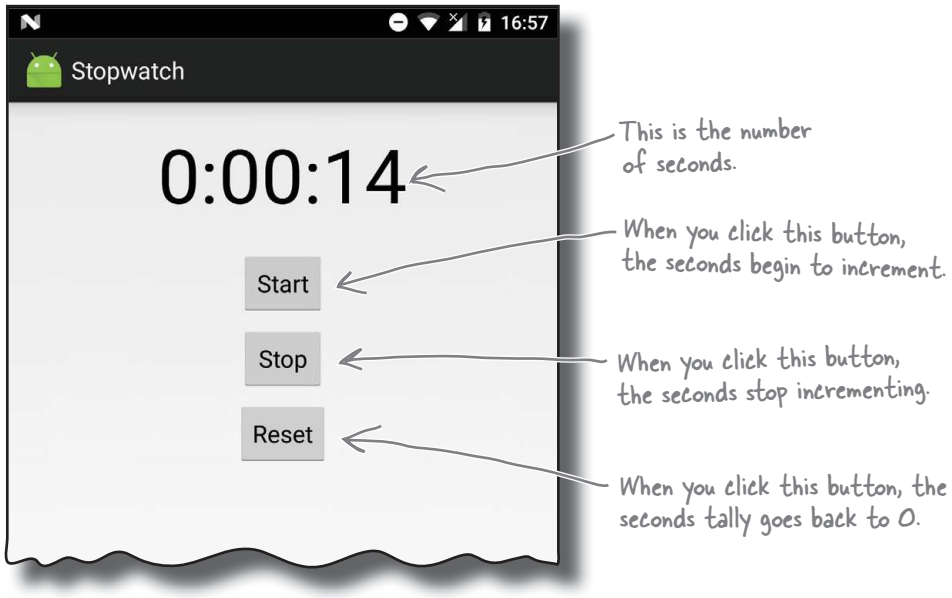
But there are still lots of things we don't yet know about how activities function. How long does an activity live for? What happens when your activity disappears from the screen? Is it still running? Is it still in memory? And what happens if your app gets interrupted by an incoming phone call? We want to be able to control the behavior of our activities in a *whole range of different circumstances*, but how?

stopwatch

The Stopwatch app

In this chapter, we're going to take a closer look at how activities work under the hood, common ways in which your apps can break, and how you can fix them using the activity lifecycle methods. We're going to explore the lifecycle methods using a simple Stopwatch app as an example.

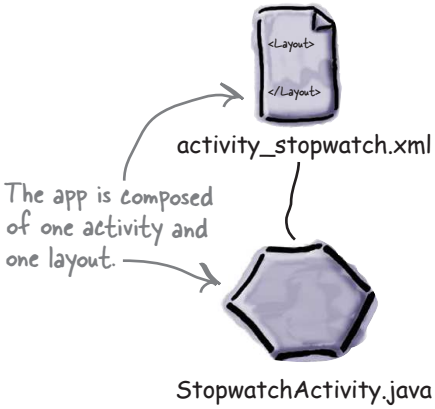
The Stopwatch app consists of a single activity and a single layout. The layout includes a text view showing you how much time has passed, a Start button that starts the stopwatch, a Stop button that stops it, and a Reset button that resets the timer value to 0.



Create a new project for the Stopwatch app

You have enough experience under your belt to build the app without much guidance from us. We're going to give you just enough code so you can build the app yourself, and then you can see what happens when you try to run it.

Start off by creating a new Android project for an application named "Stopwatch" with a company domain of "hfad.com", making the package name `com.hfad.stopwatch`. The minimum SDK should be API 19 so it can run on most devices. You'll need an empty activity called "StopwatchActivity" and a layout called "activity_stopwatch". **Make sure you uncheck the Backwards Compatibility (AppCompat) checkbox.**

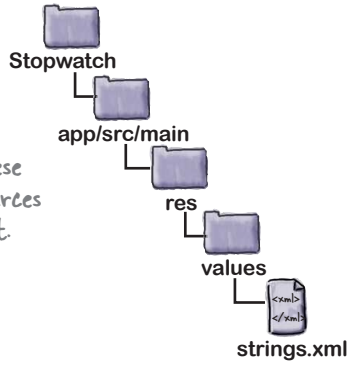


Add String resources

We're going to use three String values in our stopwatch layout, one for the text value of each button. These values are String resources, so they need to be added to *strings.xml*. Add the String values below to your version of *strings.xml*:

```
...
    <string name="start">Start</string>
    <string name="stop">Stop</string>
    <string name="reset">Reset</string>
...
```

We'll use these String resources in our layout.

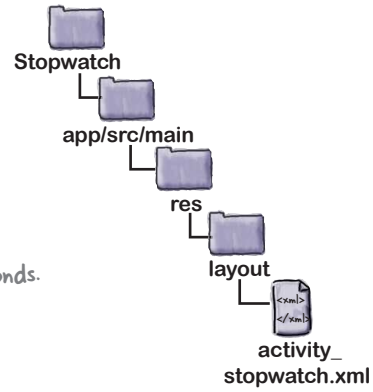


Next, let's update the code for our layout.

Update the stopwatch layout code

Here's the XML for the layout. It describes a single text view that's used to display the timer, and three buttons to control the stopwatch. Replace the XML currently in *activity_stopwatch.xml* with the XML shown here:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical"
    android:padding="16dp"
    tools:context=".StopwatchActivity">

    <TextView
        android:id="@+id/time_view"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_gravity="center_horizontal"
        android:textAppearance="@android:style/TextAppearance.Large"
        android:textSize="56sp" />
```



We'll use this text view to display the number of seconds.

These attributes make the stopwatch timer nice and big.

The layout code continues over the page.

activity_stopwatch.xml

The layout code (continued)

```
    <Button
        android:id="@+id/start_button"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_gravity="center_horizontal"
        android:layout_marginTop="20dp"
        android:onClick="onClickStart"
        android:text="@string/start" />

    <Button
        android:id="@+id/stop_button"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_gravity="center_horizontal"
        android:layout_marginTop="8dp"
        android:onClick="onClickStop"
        android:text="@string/stop" />

    <Button
        android:id="@+id/reset_button"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_gravity="center_horizontal"
        android:layout_marginTop="8dp"
        android:onClick="onClickReset"
        android:text="@string/reset" />
</LinearLayout>
```

This code is for the Start button.

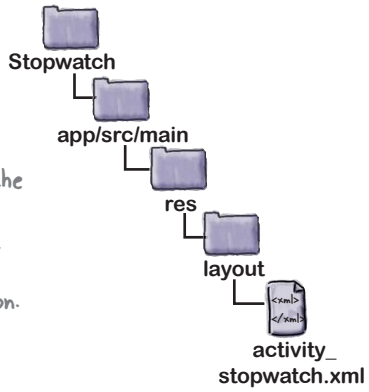
When it gets clicked, the Start button calls the onClickStart() method.

This is for the Stop button.

When it gets clicked, the Stop button calls the onClickStop() method.

This is for the Reset button.

When it gets clicked, the Reset button calls the onClickReset() method.



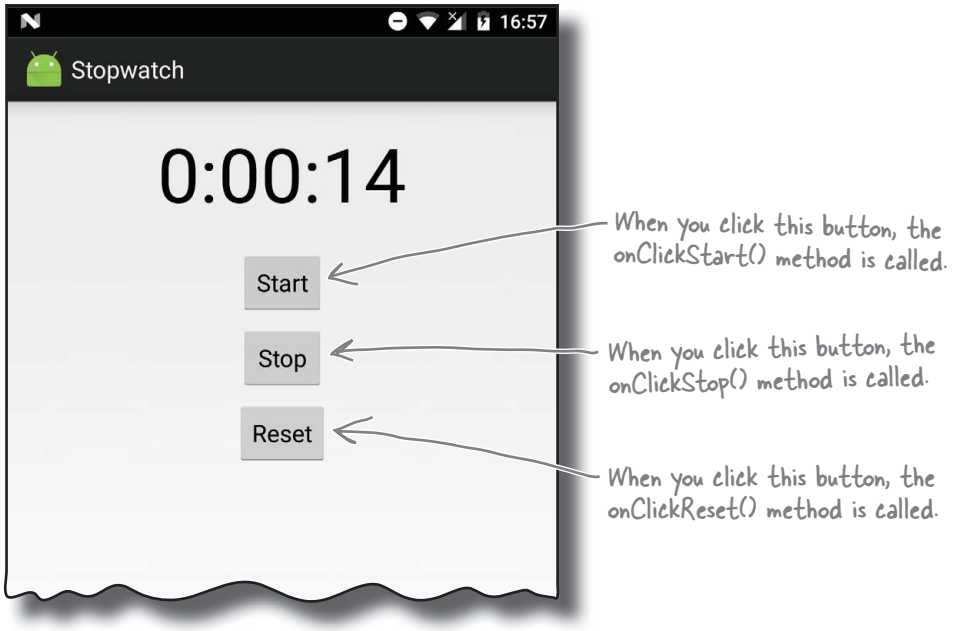
Do this!

Make sure you update the layout and *strings.xml* in your app before continuing.

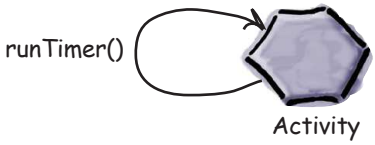
The layout is now done! Next, let's move on to the activity.

How the activity code will work

The layout defines three buttons that we'll use to control the stopwatch. Each button uses its `onClick` attribute to specify which method in the activity should run when the button is clicked. When the Start button is clicked, the `onClickStart()` method gets called, when the Stop button is clicked the `onClickStop()` method gets called, and when the Reset button is clicked the `onClickReset()` method gets called. We'll use these methods to start, stop, and reset the stopwatch.



We'll update the stopwatch using a method we'll create called `runTimer()`. The `runTimer()` method will run code every second to check whether the stopwatch is running, and, if it is, increment the number of seconds and display the number of seconds in the text view.

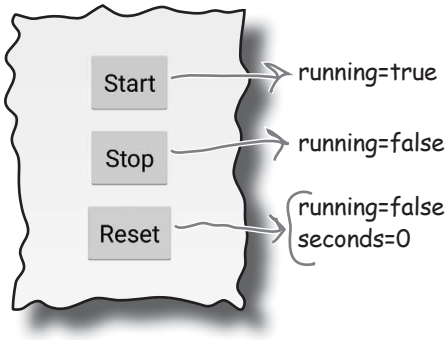


To help us with this, we'll use two private variables to record the state of the stopwatch. We'll use an `int` called `seconds` to track how many seconds have passed since the stopwatch started running, and a `boolean` called `running` to record whether the stopwatch is currently running.

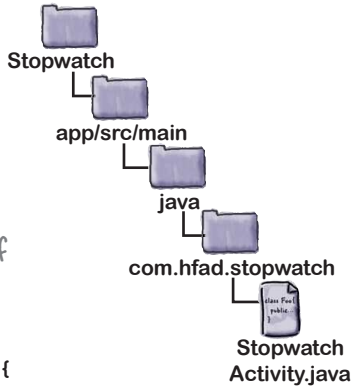
We'll start by writing the code for the buttons, and then we'll look at the `runTimer()` method.

Add code for the buttons

When the user clicks on the Start button, we'll set the `running` variable to `true` so that the stopwatch will start. When the user clicks on the Stop button, we'll set `running` to `false` so that the stopwatch stops running. If the user clicks on the Reset button, we'll set `running` to `false` and `seconds` to 0 so that the stopwatch is reset and stops running.



To do all that, replace the contents of *StopwatchActivity.java* with the code below:




```
package com.hfad.stopwatch;

import android.app.Activity;
import android.os.Bundle;
import android.view.View;

public class StopwatchActivity extends Activity {

    private int seconds = 0;
    private boolean running;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_stopwatch);
    }

    //Start the stopwatch running when the Start button is clicked.
    public void onClickStart(View view) {
        running = true;
    }

    //Stop the stopwatch running when the Stop button is clicked.
    public void onClickStop(View view) {
        running = false;
    }

    //Reset the stopwatch when the Reset button is clicked.
    public void onClickReset(View view) {
        running = false;
        seconds = 0;
    }
}
```

Make sure your activity extends the Activity class.

Use the seconds and running variables to record the number of seconds passed and whether the stopwatch is running.

Start the stopwatch.

This gets called when the Start button is clicked.

Stop the stopwatch.

This gets called when the Stop button is clicked.

Stop the stopwatch and set the seconds to 0.

This gets called when the Reset button is clicked.

The runTimer() method

The next thing we need to do is create the `runTimer()` method. This method will get a reference to the text view in the layout; format the contents of the `seconds` variable into hours, minutes, and seconds; and then display the results in the text view. If the `running` variable is set to `true`, it will increment the `seconds` variable.

The code for the `runTimer()` method is below. We'll add it to *StopwatchActivity.java* in a few pages:

```
private void runTimer() {
    final TextView timeView = (TextView)findViewById(R.id.time_view);
    ...
    int hours = seconds/3600;
    int minutes = (seconds%3600)/60;
    int secs = seconds%60;
    String time = String.format(Locale.getDefault(),
                  "%d:%02d:%02d", hours, minutes, secs);
    timeView.setText(time);
    if (running) {
        seconds++;
    }
    ...
}
```

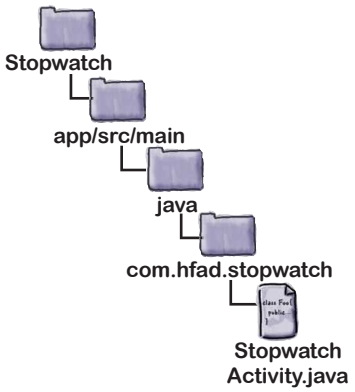
Get the text view.

We've left out a bit of code here. We'll look at it on the next page.

Format the seconds into hours, minutes, and seconds. This is plain Java code.

Set the text view text.

If running is true, increment the seconds variable.



We need this code to keep looping so that it increments the `seconds` variable and updates the text view every second. We need to do this in such a way that we don't block the main Android thread.

In non-Android Java programs, you can perform tasks like this using a background thread. In Androidville, that approach won't work—only the main Android thread can update the user interface, and if any other thread tries to do so, you get a `CalledFromWrongThreadException`.

The solution is to use a `Handler`. We'll look at this technique on the next page.

Handlers allow you to schedule code

A `Handler` is an Android class you can use to schedule code that should be run at some point in the future. You can also use it to post code that needs to run on a different thread than the main Android thread. In our case, we're going to use a `Handler` to schedule the stopwatch code to run every second.

To use the `Handler`, you wrap the code you wish to schedule in a `Runnable` object, and then use the `Handler` `post()` and `postDelayed()` methods to specify when you want the code to run. Let's take a closer look at these mehods.

The post() method

The `post()` method posts code that needs to be run as soon as possible (which is usually almost immediately). This method takes one parameter, an object of type `Runnable`. A `Runnable` object in Androidville is just like a `Runnable` in plain old Java: a job you want to run. You put the code you want to run in the `Runnable`'s `run()` method, and the `Handler` will make sure the code is run as soon as possible. Here's what the method looks like:

```
final Handler handler = new Handler();
handler.post(Runnable);
```

← You put the code you want to run in the Runnable's run() method.

The postDelayed() method

The `postDelayed()` method works in a similar way to the `post()` method except that you use it to post code that should be run in the future. The `postDelayed()` method takes two parameters: a `Runnable` and a `long`. The `Runnable` contains the code you want to run in its `run()` method, and the `long` specifies the number of milliseconds you wish to delay the code by. The code will run as soon as possible after the delay. Here's what the method looks like:

```
final Handler handler = new Handler();
handler.postDelayed(Runnable, long);
```

← Use this method to delay running code by a specified number of milliseconds.

On the next page, we'll use these methods to update the stopwatch every second.

The full runTimer() code

To update the stopwatch, we're going to repeatedly schedule code using the `Handler` with a delay of 1 second each time. Each time the code runs, we'll increment the `seconds` variable and update the text view.

Here's the full code for the `runTimer()` method, which we'll add to *StopwatchActivity.java* in a couple of pages:

```
private void runTimer() {
    final TextView timeView = (TextView)findViewById(R.id.time_view);
    final Handler handler = new Handler();
    handler.post(new Runnable() {
        @Override
        public void run() {
            int hours = seconds/3600;
            int minutes = (seconds%3600)/60;
            int secs = seconds%60;
            String time = String.format(Locale.getDefault(),
                        "%d:%02d:%02d", hours, minutes, secs);
            timeView.setText(time);
            if (running) {
                seconds++;
            }
            handler.postDelayed(this, 1000);
        }
    });
}
```

Create a new Handler.

Call the post() method, passing in a new Runnable. The post() method processes code without a delay, so the code in the Runnable will run almost immediately.

The Runnable run() method contains the code you want to run—in our case, the code to update the text view.

Post the code in the Runnable to be run again after a delay of 1,000 milliseconds. As this line of code is included in the Runnable run() method, it will keep getting called.

Using the `post()` and `postDelayed()` methods in this way means that the code will run as soon as possible after the required delay, which in practice means almost immediately. While this means the code will lag slightly over time, it's accurate enough for the purposes of exploring the lifecycle methods in this chapter.

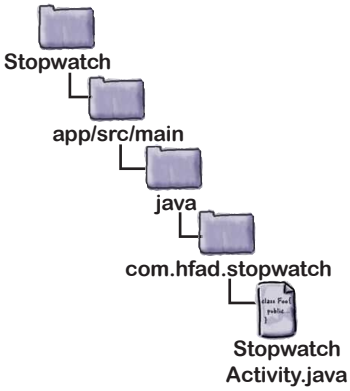
We want the `runTimer()` method to start running when `StopwatchActivity` gets created, so we'll call it in the activity `onCreate()` method:

```
protected void onCreate(Bundle savedInstanceState) {
    ...
    runTimer();
}
```

We'll show you the full code for `StopwatchActivity` on the next page.

The full StopwatchActivity code

Here's the full code for *StopwatchActivity.java*. Update your code to match our changes below.




```
package com.hfad.stopwatch;

import android.app.Activity;
import android.os.Bundle;
import android.view.View;
import java.util.Locale;
import android.os.Handler;
import android.widget.TextView;

public class StopwatchActivity extends Activity {
    //Number of seconds displayed on the stopwatch.
    private int seconds = 0;
    //Is the stopwatch running?
    private boolean running;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_stopwatch);
        runTimer();
    }

    //Start the stopwatch running when the Start button is clicked.
    public void onClickStart(View view) {
        running = true;
    }

    //Stop the stopwatch running when the Stop button is clicked.
    public void onClickStop(View view) {
        running = false;
    }
```

We're using these extra classes so we need to import them.

Use the seconds and running variables to record the number of seconds passed and whether the stopwatch is running.

We're using a separate method to update the stopwatch. We're starting it when the activity is created.

This gets called when the Start button is clicked.

Start the stopwatch.

This gets called when the Stop button is clicked.

Stop the stopwatch.

The activity code (continued)

```
    //Reset the stopwatch when the Reset button is clicked.
    public void onClickReset(View view) {
        running = false;
        seconds = 0;
    }

    //Sets the number of seconds on the timer.
    private void runTimer() {
        final TextView timeView = (TextView)findViewById(R.id.time_view);
        final Handler handler = new Handler();
        handler.post(new Runnable() {
            @Override
            public void run() {
                int hours = seconds/3600;
                int minutes = (seconds%3600)/60;
                int secs = seconds%60;
                String time = String.format(Locale.getDefault(),
                        "%d:%02d:%02d", hours, minutes, secs);
                timeView.setText(time);
                if (running) {
                    seconds++;
                }
                handler.postDelayed(this, 1000);
            }
        });
    }
}
```

This gets called when the Reset button is clicked.

Stop the stopwatch and set the seconds to 0.

Get the text view.

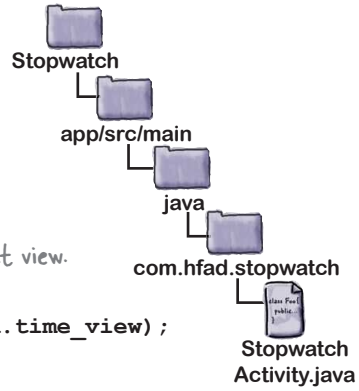
Use a Handler to post code.

Format the seconds into hours, minutes, and seconds.

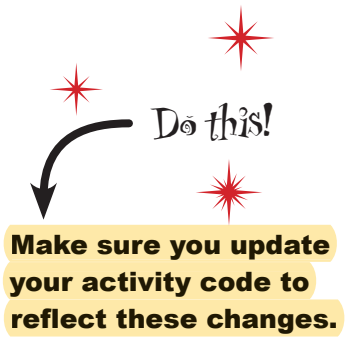
Set the text view text.

If running is true, increment the seconds variable.

Post the code again with a delay of 1 second.



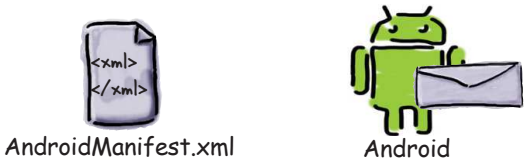
Let's look at what happens when the code runs.



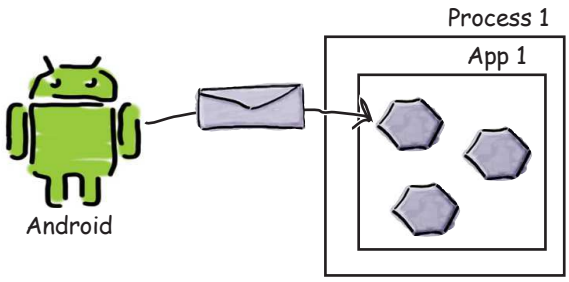
What happens when you run the app

1. **The user decides she wants to run the app.**
On her device, she clicks on the app's icon.

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

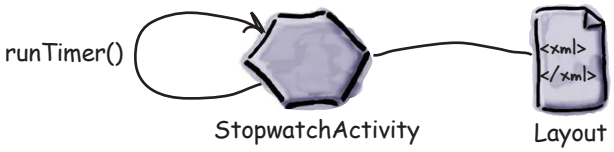


3. **Android checks to see whether there's already a process running for the app, and if not, creates a new process.**
It then creates a new activity object—in this case, for `StopwatchActivity`.

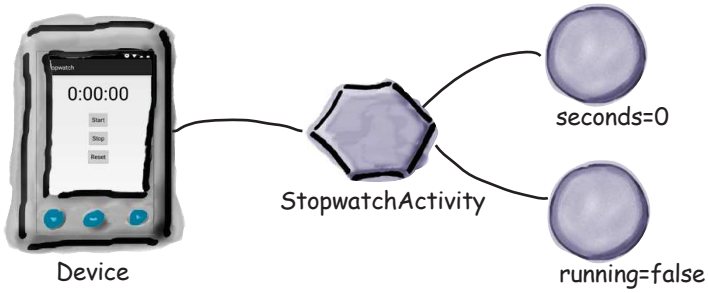


The story continues

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()`.



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.



there are no Dumb Questions

Q: Why does Android run each app inside a separate process?

A: For security and stability. This approach prevents one app from accessing the data of another. It also means if one app crashes, it won't take others down with it.

Q: Why have an `onCreate()` method in our activity? Why not just put that code inside a constructor?

A: Android needs to set up the environment for the activity after it's constructed. Once the environment is ready, Android calls `onCreate()`. That's why code to set up the screen goes inside `onCreate()` instead of a constructor.

Q: Couldn't I just write a loop in `onCreate()` to keep updating the timer?

A: No, `onCreate()` needs to finish before the screen will appear. An endless loop would prevent that from happening.

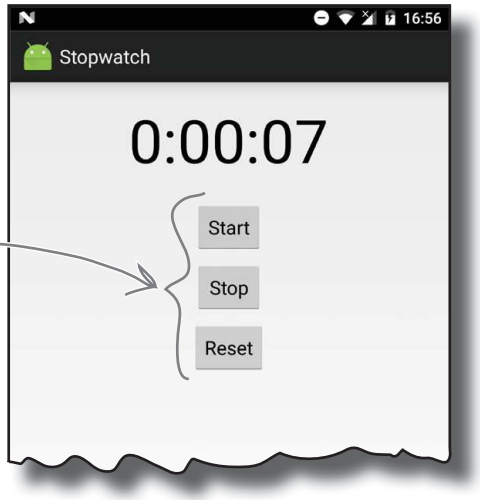
Q: `runTimer()` looks really complicated. Do I really need to do all this?

A: It's a little complex, but whenever you need to post code that runs in a loop, the code will look similar to `runTimer()`.

test drive

Test drive the app

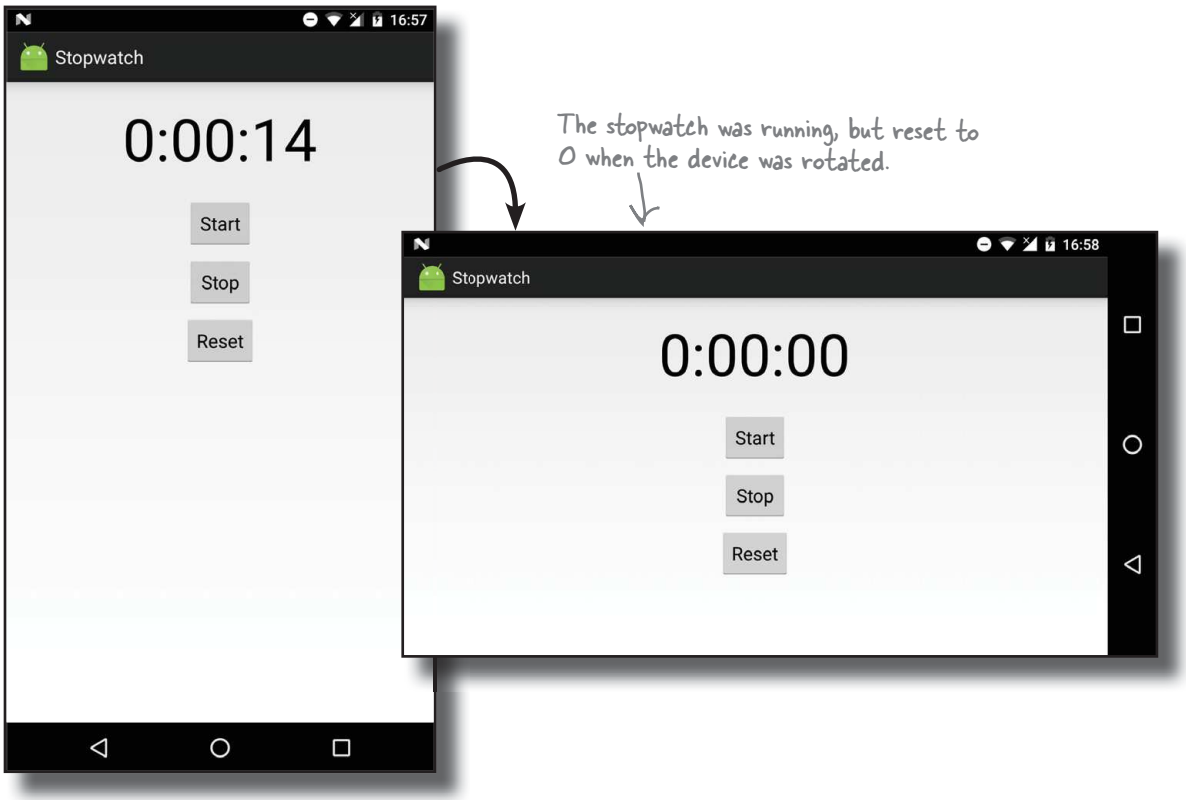
When we run the app in the emulator, the app works great. We can start, stop, and reset the stopwatch without any problems at all—the app works just as you'd expect.



These buttons work as you'd expect. The Start button starts the stopwatch, the Stop button stops it, and the Reset button sets the stopwatch back to 0.

But there's just one problem...

When we ran the app on a physical device, the app worked OK until someone rotated the device. When the device was rotated, the stopwatch set itself back to 0.



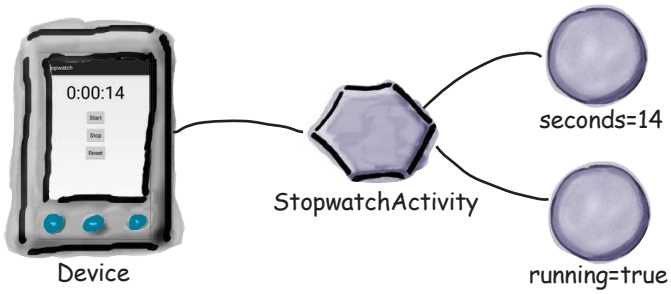
In Androidville, it's surprisingly common for apps to break when you rotate the device. Before we fix the problem, let's take a closer look at what caused it.

What just happened?

So why did the app break when the user rotated the screen?
Let's take a closer look at what really happened.

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.

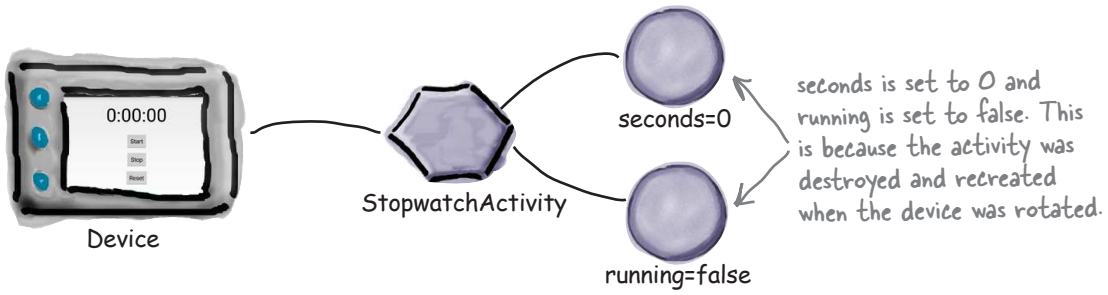


2 **The user rotates the device.**

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 recreated.**

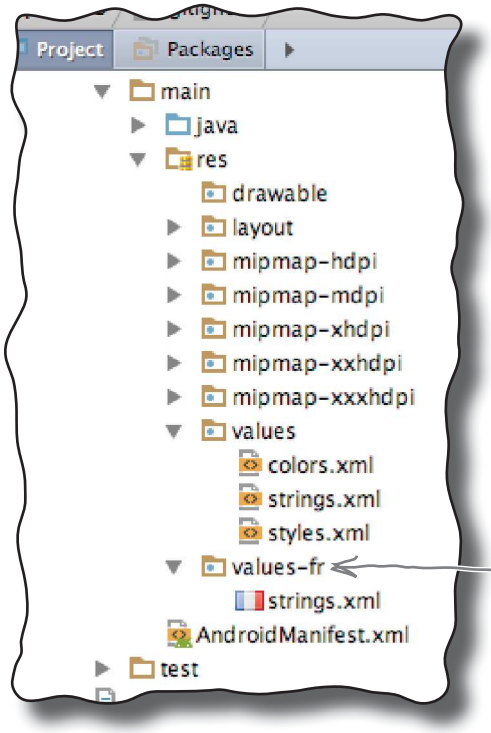
The `onCreate()` method runs again, and the `runTimer()` method gets called. As the activity has been recreated, the `seconds` and `running` variables are set to their default values.



Rotating the screen changes the device configuration

When Android runs your app and starts an activity, it takes into account the **device configuration**. By this we mean the configuration of the physical device (such as the screen size, screen orientation, and whether there's a keyboard attached) and also configuration options specified by the user (such as the locale).

Android needs to know what the device configuration is when it starts an activity because the configuration can impact what resources are needed for the application. A different layout might need to be used if the device screen is landscape rather than portrait, for instance, and a different set of String values might need to be used if the locale is France.



Android apps can contain multiple resource files in the app/src/main/res folder. For instance, if the device locale is set to France, Android will use the strings.xml file in the values-fr folder.

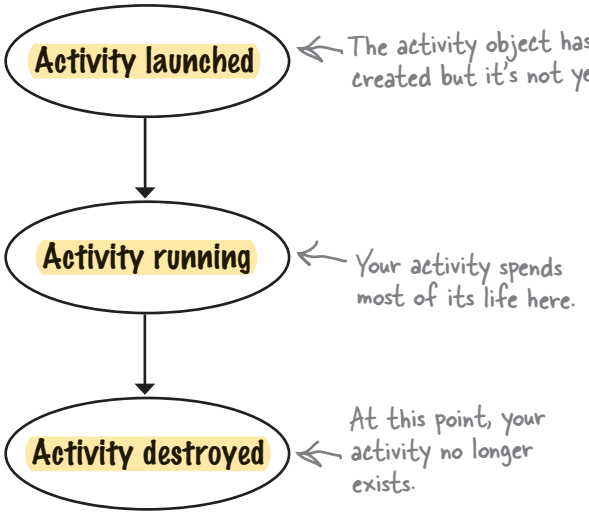
The device configuration includes options specified by the user (such as the locale), and options relating to the physical device (such as the orientation and screen size). A change to any of these options results in the activity being destroyed and then recreated.

When the device configuration changes, anything that displays a user interface needs to be updated to match the new configuration. If you rotate your device, Android spots that the screen orientation and screen size have changed, and classes this as a change to the device configuration. It destroys the current activity, and then recreates it so that resources appropriate to the new configuration get picked up.

The states of an activity

When Android creates and destroys an activity, the activity moves from being launched to running to being destroyed.

The main state of an activity is when it's ***running*** or ***active***. An activity is running when it's in the foreground of the screen, it has the focus, and the user can interact with it. The activity spends most of its life in this state. An activity starts running after it has been launched, and at the end of its life, the activity is ***destroyed***.



When an activity moves from being launched to being destroyed, it triggers key activity lifecycle methods: the `onCreate()` and `onDestroy()` methods. These are lifecycle methods that your activity inherits, and which you can override if necessary.

The `onCreate()` method gets called immediately after your activity is launched. This method is where you do all your normal activity setup such as calling `setContentView()`. You should always override this method. If you *don't* override it, you won't be able to tell Android what layout your activity should use.

The `onDestroy()` method is the final call you get before the activity is destroyed. There are a number of situations in which an activity can get destroyed—for example, if it's been told to finish, if the activity is being recreated due to a change in device configuration, or if Android has decided to destroy the activity in order to save space.

We'll take a closer look at how these methods fit into the activity states on the next page.

An activity is running when it's in the foreground of the screen.

onCreate() gets called when the activity is first created, and it's where you do your normal activity setup.

onDestroy() gets called just before your activity gets destroyed.

The activity lifecycle: from create to destroy

Here's an overview of the activity lifecycle from birth to death. As you'll see later in the chapter, we've left out some of the details, but at this point we're just focusing on the `onCreate()` and `onDestroy()` methods.

1. **The activity gets launched.**
   The activity object is created and its constructor is run.

2. **The onCreate() method runs immediately after the activity is launched.**
   The `onCreate()` method is where any initialization code should go, as this method always gets called after the activity has launched but before it starts running.

3. **An activity is running when it's visible in the foreground and the user can interact with it.**
   This is where an activity spends most of its life.

4. **The onDestroy() method runs immediately before the activity is destroyed.**
   The `onDestroy()` method enables you to perform any final cleanup such as freeing up resources.

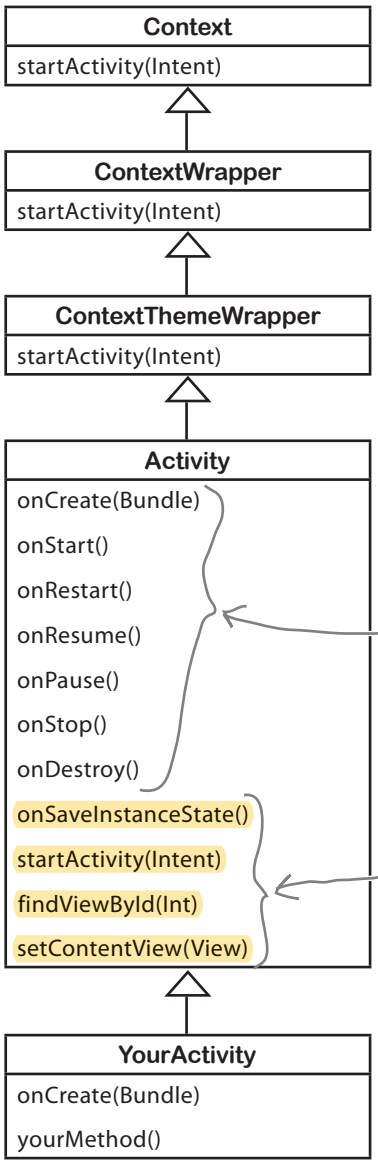
5. **After the onDestroy() method has run, the activity is destroyed.**
   The activity ceases to exist.

If your device is extremely low on memory, onDestroy() might not get called before the activity is destroyed.

The `onCreate()` and `onDestroy()` methods are two of the activity lifecycle methods. So where do these methods come from?

Your activity inherits the lifecycle methods

As you saw earlier in the book, your activity extends the `android.app.Activity` class. It's this class that gives your activity access to the Android lifecycle methods. Here's a diagram showing the class hierarchy:



Context abstract class
(android.content.Context)
An interface to global information about the application environment. Allows access to application resources, classes, and operations.

ContextWrapper class
(android.content.ContextWrapper)
A proxy implementation for the `Context`.

ContextThemeWrapper class
(android.view.ContextThemeWrapper)
Allows you to modify the theme from what's in the `ContextWrapper`.

Activity class
(android.app.Activity)
The `Activity` class implements default versions of the lifecycle methods. It also defines methods such as `findViewById(Int)` and `setContentView(View)`.

These are the activity lifecycle methods. You'll find out more about them through the rest of the chapter.

These aren't lifecycle methods, but they're still very useful. You've already used most of them in earlier chapters.

YourActivity class
(com.hfad.foo)
Most of the behavior of your activity is handled by superclass methods your activity inherits. All you do is override the methods you need.

Now that you know more about the activity lifecycle methods, let's see how you deal with device configuration changes.

Save the current state...

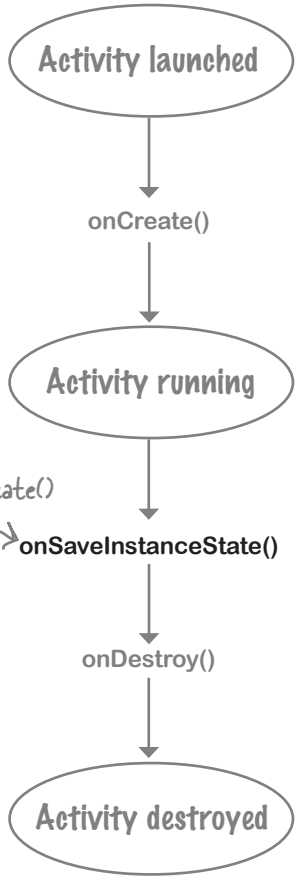
As you saw, our app went wrong when the user rotated the screen. The activity was destroyed and recreated, which meant that local variables used by the activity were lost. So how do we get around this issue?

The best way of dealing with configuration changes is to save the current state of the activity, and then reinstate it in the `onCreate()` method of the activity.

To save the activity's current state, you need to implement the `onSaveInstanceState()` method. This method gets called before the activity gets destroyed, which means you get an opportunity to save any values you want to retain before they get lost.

The `onSaveInstanceState()` method takes one parameter, a `Bundle`. A `Bundle` allows you to gather together different types of data into a single object:

```
public void onSaveInstanceState(Bundle savedInstanceState) {
}
```



The onSaveInstanceState() method gets called before onDestroy(). It gives you a chance to save your activity's state before the activity is destroyed.

The `onCreate()` method gets passed the `Bundle` as a parameter. This means that if you add the values of the `running` and `seconds` variables to the `Bundle`, the `onCreate()` method will be able to pick them up when the activity gets recreated. To do this, you use `Bundle` methods to add name/value pairs to the `Bundle`. These methods take the form:

```
bundle.put*("name", value)
```

where `bundle` is the name of the `Bundle`, `*` is the type of value you want to save, and `name` and `value` are the name and value of the data. As an example, to add the `seconds` `int` value to the `Bundle`, you'd use:

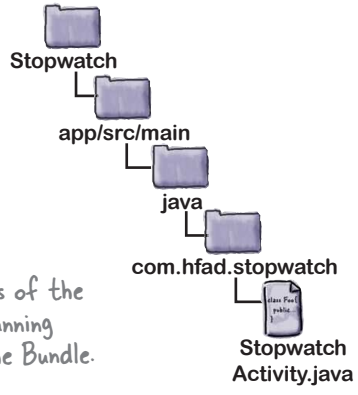
```
bundle.putInt("seconds", seconds);
```

You can save multiple name/value pairs of data to the `Bundle`.

Here's our `onSaveInstanceState()` method in full (we'll add it to *StopwatchActivity.java* a couple of pages ahead):

```
@Override
public void onSaveInstanceState(Bundle savedInstanceState) {
    savedInstanceState.putInt("seconds", seconds);
    savedInstanceState.putBoolean("running", running);
}
```

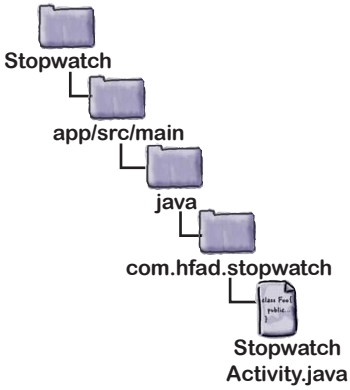
Save the values of the seconds and running variables to the Bundle.



Once you've saved variable values to the `Bundle`, you can use them in our `onCreate()` method.

...then restore the state in onCreate()

As we said earlier, the `onCreate()` method takes one parameter, a `Bundle`. If the activity's being created from scratch, this parameter will be null. If, however, the activity's being recreated and there's been a prior call to `onSaveInstanceState()`, the `Bundle` object used by `onSaveInstanceState()` will get passed to the activity:




```
protected void onCreate(Bundle savedInstanceState) {
    ...
}
```

You can get values from `Bundle` by using methods of the form:

```
bundle.get*("name");
```

Instead of *, use Int, String, and so on, to specify the type of data you want to get.

where `bundle` is the name of the `Bundle`, `*` is the type of value you want to get, and `name` is the name of the name/value pair you specified on the previous page. As an example, to get the `seconds` value from the `Bundle`, you'd use:

```
int seconds = bundle.getInt("seconds");
```

Putting all of this together, here's what our `onCreate()` method now looks like (we'll add this to *StopwatchActivity.java* on the next page):

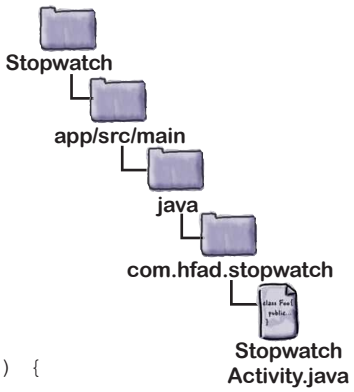
```
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_stopwatch);
    if (savedInstanceState != null) {
        seconds = savedInstanceState.getInt("seconds");
        running = savedInstanceState.getBoolean("running");
    }
    runTimer();
}
```

Retrieve the values of the seconds and running variables from the Bundle.

We'll look at the full code to save and restore `StopwatchActivity`'s state on the next page.

The updated StopwatchActivity code

We've updated our `StopwatchActivity` code so that if the user rotates the device, its state gets saved via the `onSaveInstanceState()` method, and restored via the `onCreate()` method. Update your version of *StopwatchActivity.java* to include our changes (below in bold):




```
...

public class StopwatchActivity extends Activity {
    //Number of seconds displayed on the stopwatch.
    private int seconds = 0;
    //Is the stopwatch running?
    private boolean running;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_stopwatch);
        if (savedInstanceState != null) {
            seconds = savedInstanceState.getInt("seconds");
            running = savedInstanceState.getBoolean("running");
        }
        runTimer();
    }

    @Override
    public void onSaveInstanceState(Bundle savedInstanceState) {
        savedInstanceState.putInt("seconds", seconds);
        savedInstanceState.putBoolean("running", running);
    }
...
```

Restore the activity's state by getting values from the Bundle.

Save the state of the variables in the activity's onSaveInstanceState() method.

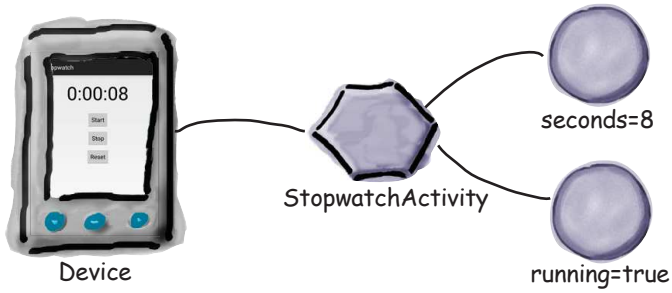
We've left out some of the activity code, as we don't need to change it.

So how does this work in practice?

What happens when you run the app

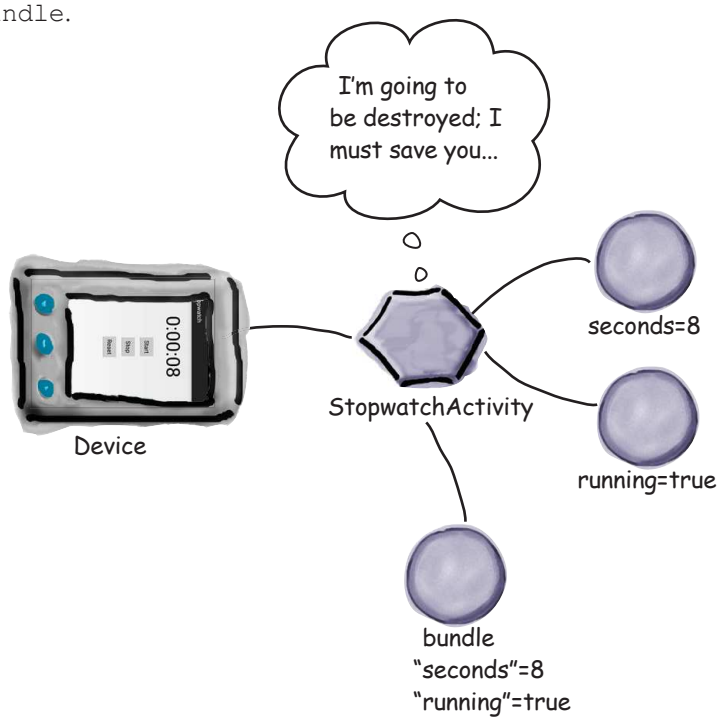
1 **The user starts the app, and clicks on the Start button to set the stopwatch going.**

The `runTimer()` method starts incrementing the number of seconds displayed in the `time_view` text view.



2 **The user rotates the device.**

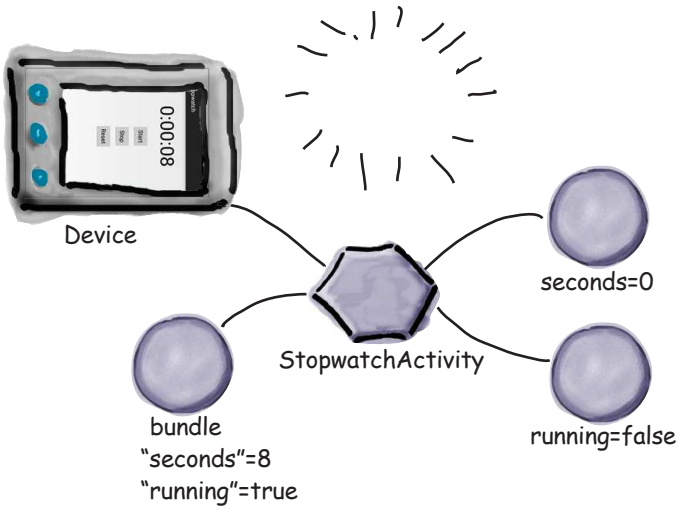
Android views this as a configuration change, and gets ready to destroy the activity. Before the activity is destroyed, `onSaveInstanceState()` gets called. The `onSaveInstanceState()` method saves the `seconds` and `running` values to a `Bundle`.



The story continues

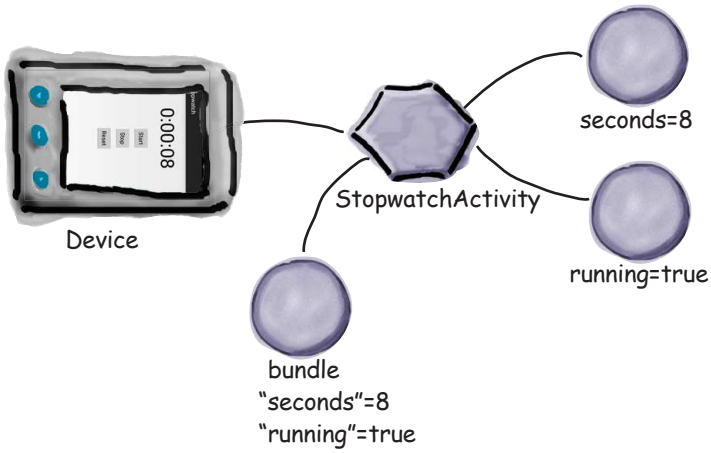
3 **Android destroys the activity, and then recreates it.**

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



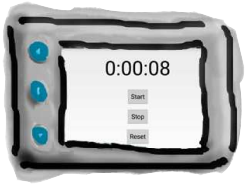
4 **The Bundle contains the values of the seconds and running variables as they were before the activity was destroyed.**

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



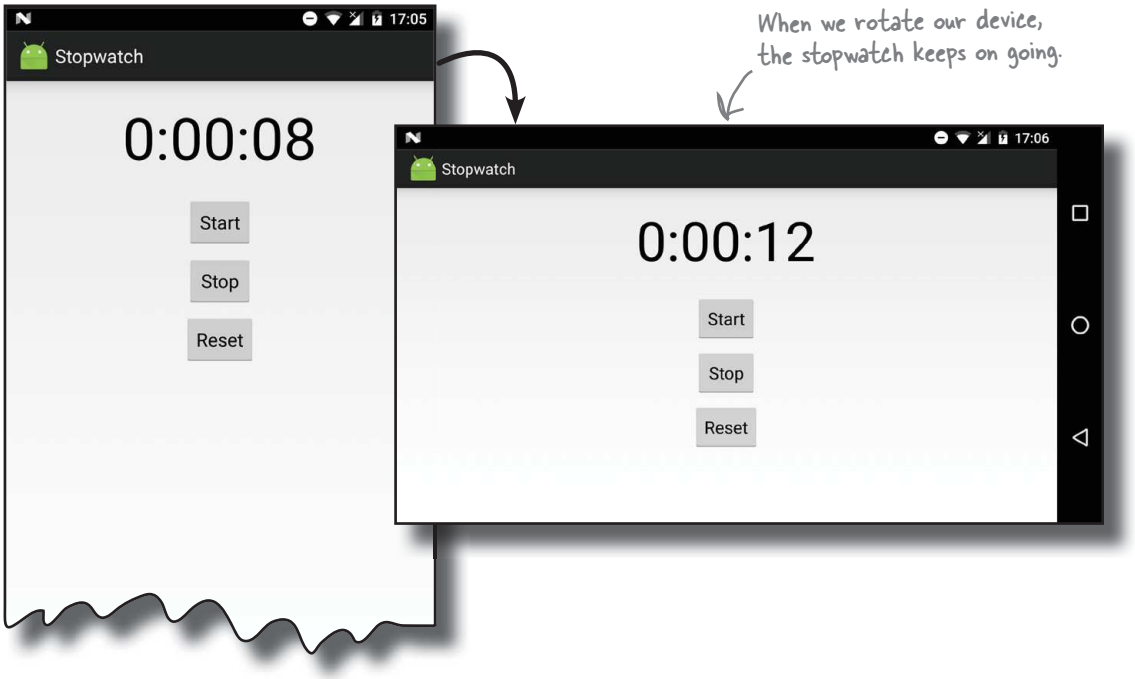
5 **The runTimer() method gets called, and the timer picks up where it left off.**

The running stopwatch gets displayed on the device and continues to increment.



Test drive the app

Make the changes to your activity code, then run the app. When you click on the Start button, the timer starts, and it continues when you rotate the device.



there are no Dumb Questions

Q: Why does Android want to recreate an activity just because I rotated the screen?

A: The `onCreate()` method is normally used to set up the screen. If your code in `onCreate()` depended upon the screen configuration (for example, if you had different layouts for landscape and portrait), then you would want `onCreate()` to be called every time the configuration changed. Also, if the user changed their locale, you might want to recreate the UI in the local language.

Q: Why doesn't Android automatically store every instance variable automatically? Why do I have to write all of that code myself?

A: You might not want every instance variable stored. For example, you might have a variable that stores the current screen width. You would want that variable to be recalculated the next time `onCreate()` is called.

Q: Is a `Bundle` some sort of Java map?

A: No, but it's designed to work like a `java.util.Map`. A `Bundle` has additional abilities compared to a `Map`. `Bundle`s can be sent between processes, for example. That's really useful, because it allows the Android OS to stay in touch with the state of an activity.

There's more to an activity's life than create and destroy

So far we've looked at the create and destroy parts of the activity lifecycle (and a little bit in between), and you've seen how to deal with configuration changes such as screen orientation. But there are other events in an activity's life that you might want to deal with to get the app to behave in the way you want.

As an example, suppose the stopwatch is running and you get a phone call. Even though the stopwatch isn't visible, it will continue running. But what if you want the stopwatch to stop while it's hidden, and resume once the app is visible again?

Even if you don't really want your stopwatch to behave like this, just play along with us. It's a great excuse to look at more lifecycle methods.

Start, stop, and restart

Fortunately, it's easy to handle actions that relate to an activity's visibility if you use the right lifecycle methods. In addition to the `onCreate()` and `onDestroy()` methods, which deal with the overall lifecycle of the activity, there are other lifecycle methods that deal with an activity's visibility.

Specifically, there are three key lifecycle methods that deal with when an activity becomes visible or invisible to the user: `onStart()`, `onStop()`, and `onRestart()`. Just as with `onCreate()` and `onDestroy()`, your activity inherits them from the Android `Activity` class.

`onStart()` gets called when your activity becomes visible to the user.

`onStop()` gets called when your activity has stopped being visible to the user. This might be because it's completely hidden by another activity that's appeared on top of it, or because the activity is going to be destroyed. If `onStop()` is called because the activity's going to be destroyed, `onSaveInstanceState()` gets called before `onStop()`.

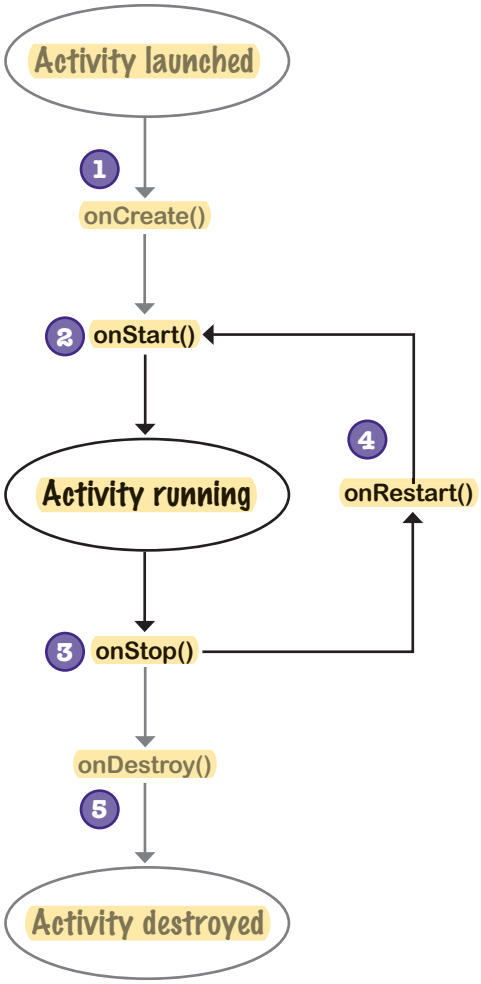
`onRestart()` gets called after your activity has been made invisible, before it gets made visible again.

We'll take a closer look at how these fit in with the `onCreate()` and `onDestroy()` methods on the next page.

An activity has a state of stopped if it's completely hidden by another activity and isn't visible to the user. The activity still exists in the background and maintains all state information.

The activity lifecycle: the visible lifetime

Let's build on the lifecycle diagram you saw earlier in the chapter, this time including the `onStart()`, `onStop()`, and `onRestart()` methods (the bits you need to focus on are in bold):



1. **The activity gets launched, and the onCreate() method runs.**
   Any activity initialization code in the `onCreate()` method runs. At this point, the activity isn't yet visible, as no call to `onStart()` has been made.

2. **The onStart() method runs. It gets called when the activity is about to become visible.**
   After the `onStart()` method has run, the user can see the activity on the screen.

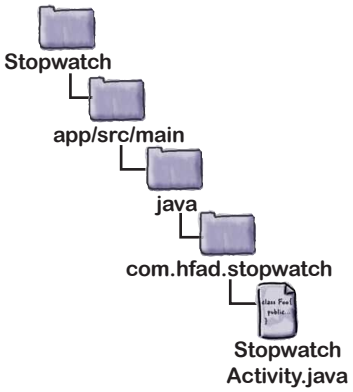
3. **The onStop() method runs when the activity stops being visible to the user.**
   After the `onStop()` method has run, the activity is no longer visible.

4. **If the activity becomes visible to the user again, the onRestart() method gets called followed by onStart().**
   The activity may go through this cycle many times if the activity repeatedly becomes invisible and then visible again.

5. **Finally, the activity is destroyed.**
   The `onStop()` method will get called before `onDestroy()`.

We need to implement two more lifecycle methods

There are two things we need to do to update our Stopwatch app. First, we need to implement the activity's `onStop()` method so that the stopwatch stops running when the app isn't visible. Once we've done that, we need to implement the `onStart()` method so that the stopwatch starts again when the app is visible. Let's start with the `onStop()` method.



Implement onStop() to stop the timer

You override the `onStop()` method in the Android `Activity` class by adding the following method to your activity:

```
@Override
protected void onStop() {
    super.onStop();
    ...
}
```

This calls the onStop() method in the activity's superclass, android.app.Activity.

The line of code:

```
super.onStop();
```

calls the `onStop()` method in the `Activity` superclass. You need to add this line of code whenever you override the `onStop()` method to make sure that the activity gets to perform any other actions in the superclass `onStop()` method. If you bypass this step, Android will generate an exception. This applies to all of the lifecycle methods. If you override any of the `Activity` lifecycle methods in your activity, you must call the superclass method or Android will give you an exception.

We need to get the stopwatch to stop when the `onStop()` method is called. To do this, we need to set the value of the `running` boolean to `false`. Here's the complete method:

```
@Override
protected void onStop() {
    super.onStop();
    running = false;
}
```

So now the stopwatch stops when the activity is no longer visible. The next thing we need to do is get the stopwatch to start again when the activity becomes visible.

When you override any activity lifecycle method in your activity, you need to call the Activity superclass method. If you don't, you'll get an exception.

Sharpen your pencil

Now it's your turn. Change the activity code so that if the stopwatch was running before **`onStop()`** was called, it starts running again when the activity regains the focus. Hint: you may need to add a new variable.

```
public class StopwatchActivity extends Activity {
    private int seconds = 0;
    private boolean running;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_stopwatch);
        if (savedInstanceState != null) {
            seconds = savedInstanceState.getInt("seconds");
            running = savedInstanceState.getBoolean("running");
        }
        runTimer();
    }

    @Override
    public void onSaveInstanceState(Bundle savedInstanceState) {
        savedInstanceState.putInt("seconds", seconds);
        savedInstanceState.putBoolean("running", running);
    }

    @Override
    protected void onStop() {
        super.onStop();
        running = false;
    }
```

Here's the first part of the activity code. You'll need to implement the onStart() method and change other methods slightly too.

Sharpen your pencil Solution

Now it's your turn. Change the activity code so that if the stopwatch was running before **`onStop()`** was called, it starts running again when the activity regains the focus. Hint: you may need to add a new variable.

```
public class StopwatchActivity extends Activity {
    private int seconds = 0;
    private boolean running;
    private boolean wasRunning;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_stopwatch);
        if (savedInstanceState != null) {
            seconds = savedInstanceState.getInt("seconds");
            running = savedInstanceState.getBoolean("running");
            wasRunning = savedInstanceState.getBoolean("wasRunning");
        }
        runTimer();
    }

    @Override
    public void onSaveInstanceState(Bundle savedInstanceState) {
        savedInstanceState.putInt("seconds", seconds);
        savedInstanceState.putBoolean("running", running);
        savedInstanceState.putBoolean("wasRunning", wasRunning);
    }

    @Override
    protected void onStop() {
        super.onStop();
        wasRunning = running;
        running = false;
    }

    @Override
    protected void onStart() {
        super.onStart();
        if (wasRunning) {
            running = true;
        }
    }
```

We added a new variable, wasRunning, to record whether the stopwatch was running before the onStop() method was called so that we know whether to set it running again when the activity becomes visible again.

We'll restore the state of the wasRunning variable if the activity is recreated.

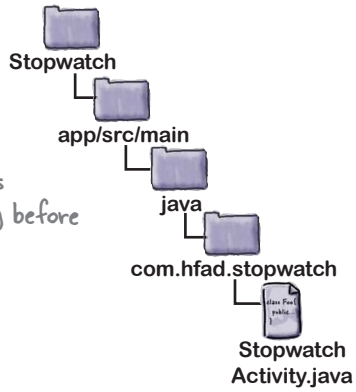
Save the state of the wasRunning variable.

Record whether the stopwatch was running when the onStop() method was called.

Implement the onStart() method. If the stopwatch was running, set it running again.

The updated StopwatchActivity code

We've updated our activity code so that if the stopwatch was running before it lost the focus, it starts running again when it gets the focus back. Make the following changes (in bold) to your version of *StopwatchActivity.java*:




```
public class StopwatchActivity extends Activity {
    private int seconds = 0;
    private boolean running;
    private boolean wasRunning;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_stopwatch);
        if (savedInstanceState != null) {
            seconds = savedInstanceState.getInt("seconds");
            running = savedInstanceState.getBoolean("running");
            wasRunning = savedInstanceState.getBoolean("wasRunning");
        }
        runTimer();
    }

    @Override
    public void onSaveInstanceState(Bundle savedInstanceState) {
        savedInstanceState.putInt("seconds", seconds);
        savedInstanceState.putBoolean("running", running);
        savedInstanceState.putBoolean("wasRunning", wasRunning);
    }

    @Override
    protected void onStop() {
        super.onStop();
        wasRunning = running;
        running = false;
    }

    @Override
    protected void onStart() {
        super.onStart();
        if (wasRunning) {
            running = true;
        }
    }
...
```

A new variable, wasRunning, records whether the stopwatch was running before the onStop() method was called.

Restore the state of the wasRunning variable if the activity is recreated.

Save the state of the wasRunning variable.

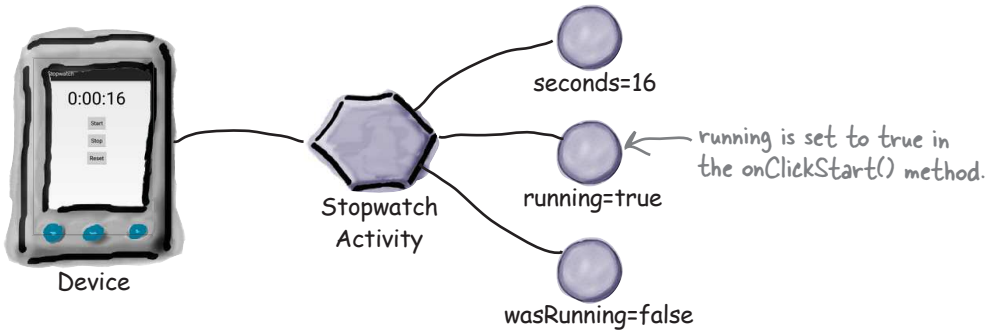
Record whether the stopwatch was running when the onStop() method was called.

Implement the onStart() method. If the stopwatch was running, we'll set it running again.

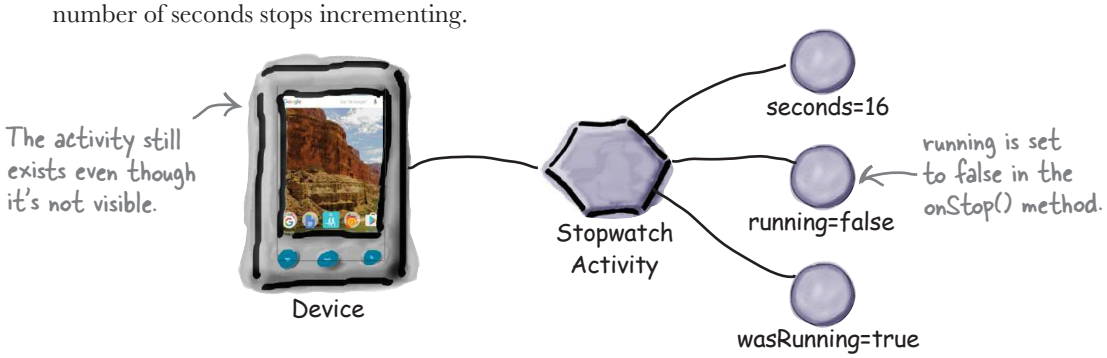
We've left out some of the activity code as we don't need to change it.

What happens when you run the app

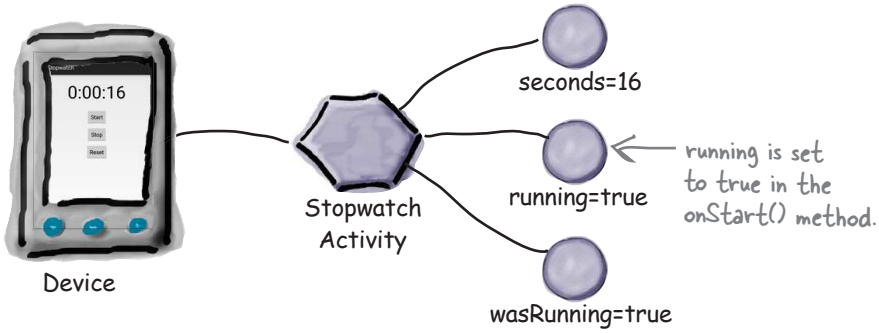
1. **The user starts the app, and clicks the Start button to set the stopwatch going.**
The `runTimer()` method starts incrementing the number of seconds displayed in the `time_view` text view.



2. **The user navigates to the device's home screen so the Stopwatch app is no longer visible.**
The `onStop()` method gets called, `wasRunning` is set to `true`, `running` is set to `false`, and the number of seconds stops incrementing.



3. **The user navigates back to the Stopwatch app.**
The `onStart()` method gets called, `running` is set to `true`, and the number of seconds starts incrementing again.



Test drive the app

Save the changes to your activity code, then run the app. When you click on the Start button the timer starts: it stops when the app is no longer visible, and it starts again when the app becomes visible again.

there are no Dumb Questions

Q: Could we have used the `onRestart()` method instead of `onStart()` to set the stopwatch running again?

A: `onRestart()` is used when you only want code to run when an app becomes visible after having previously been invisible. It doesn't run when the activity becomes visible for the first time. In our case, we wanted the app to still work when we rotated the device.

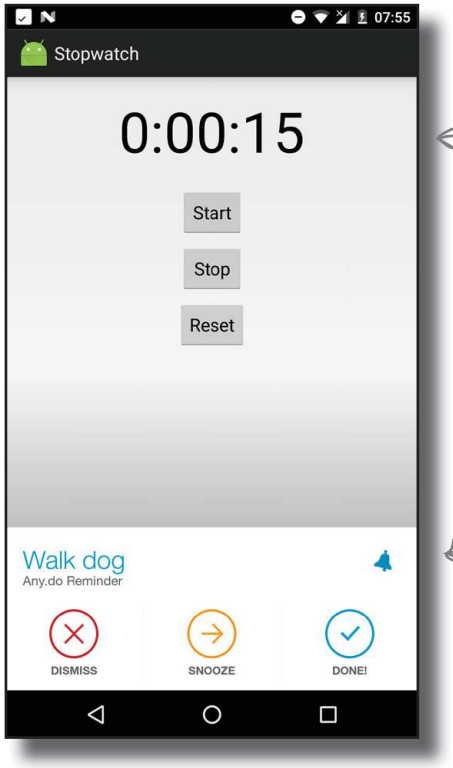
Q: Why should that make a difference?

A: When you rotate the device, the activity is destroyed and a new one is created in its place. If we'd put code to set the stopwatch running again in the `onRestart()` method instead of `onStart()`, it wouldn't have run when the activity was recreated. The `onStart()` method gets called in both situations.

What if an app is only partially visible?

So far you've seen what happens when an activity gets created and destroyed, and you've also seen what happens when an activity becomes visible, and when it becomes invisible. But there's one more situation we need to consider: when an activity is visible but doesn't have the focus.

When an activity is visible but doesn't have the focus, the activity is paused. This can happen if another activity appears on top of your activity that isn't full-size or that's transparent. The activity on top has the focus, but the one underneath is still visible and is therefore paused.



The stopwatch activity is still visible, but it's partially obscured and no longer has the focus. When this happens, it pauses.

This is an activity from another app that's appeared on top of the stopwatch.

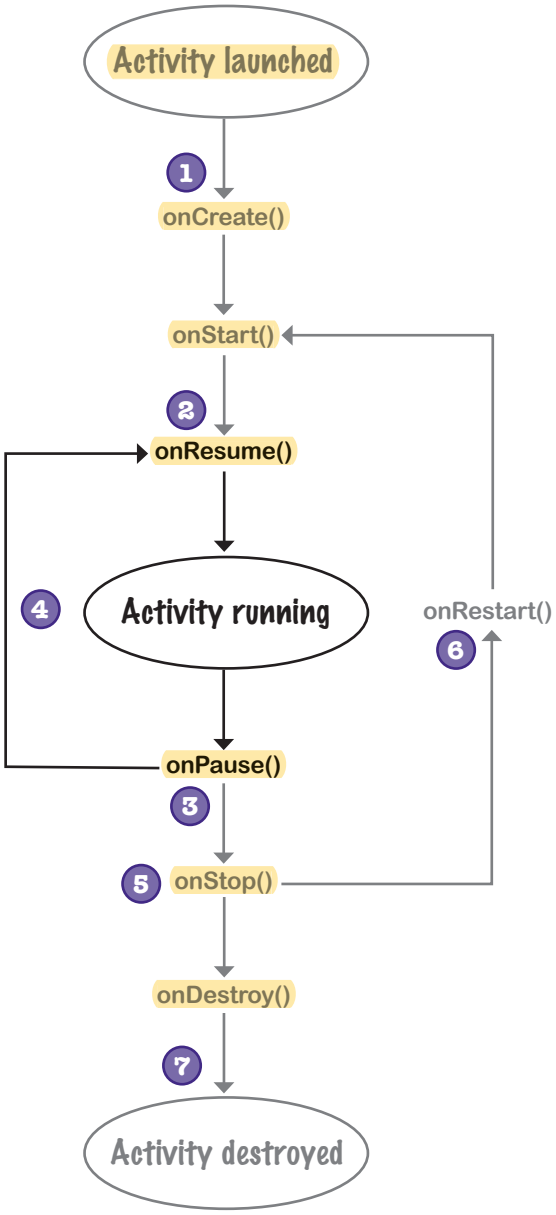
An activity has a state of paused if it's lost the focus but is still visible to the user. The activity is still alive and maintains all its state information.

There are two lifecycle methods that handle when the activity is paused and when it becomes active again: `onPause()` and `onResume()`. `onPause()` gets called when your activity is visible but another activity has the focus. `onResume()` is called immediately before your activity is about to start interacting with the user. If you need your app to react in some way when your activity is paused, you need to implement these methods.

You'll see on the next page how these methods fit in with the rest of the lifecycle methods you've seen so far.

The activity lifecycle: the foreground lifetime

Let's build on the lifecycle diagram you saw earlier in the chapter, this time including the `onResume()` and `onPause()` methods (the new bits are in bold):



1. **The activity gets launched, and the onCreate() and onStart() methods run.**
   At this point, the activity is visible, but it doesn't have the focus.

2. **The onResume() method runs. It gets called when the activity is about to move into the foreground.**
   After the `onResume()` method has run, the activity has the focus and the user can interact with it.

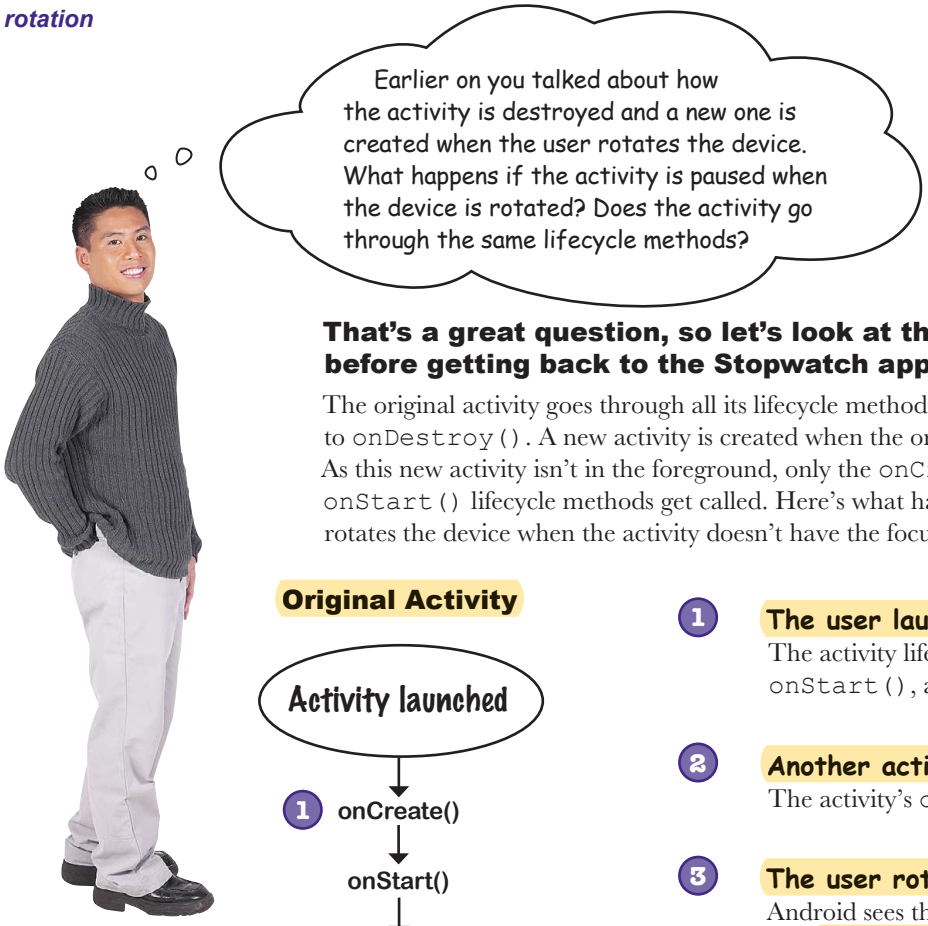
3. **The onPause() method runs when the activity stops being in the foreground.**
   After the `onPause()` method has run, the activity is still visible but doesn't have the focus.

4. **If the activity moves into the foreground again, the onResume() method gets called.**
   The activity may go through this cycle many times if the activity repeatedly loses and then regains the focus.

5. **If the activity stops being visible to the user, the onStop() method gets called.**
   After the `onStop()` method has run, the activity is no longer visible.

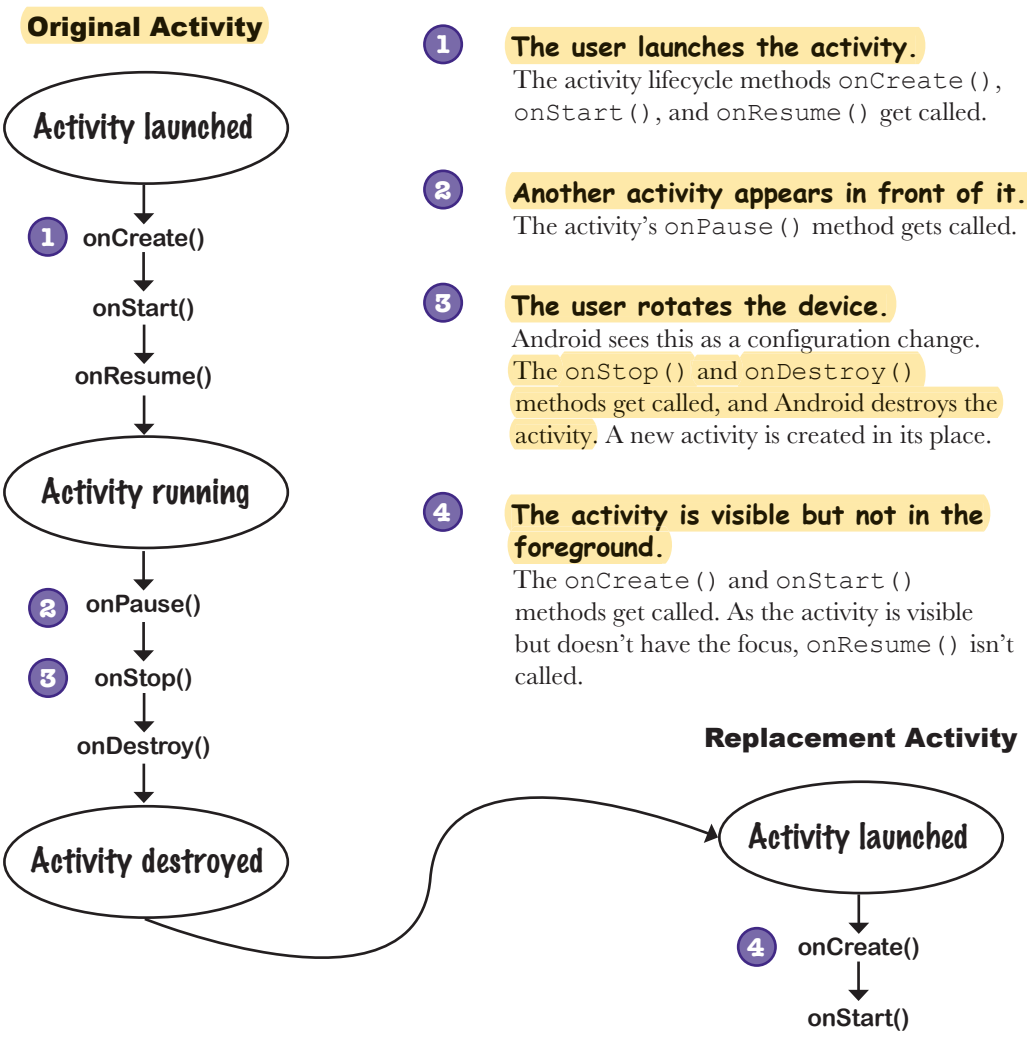
6. **If the activity becomes visible to the user again, the onRestart() method gets called, followed by onStart() and onResume().**
   The activity may go through this cycle many times.

7. **Finally, the activity is destroyed.**
   As the activity moves from running to destroyed, the `onPause()` and `onStop()` methods get called before the activity is destroyed.



That's a great question, so let's look at this in more detail before getting back to the Stopwatch app.

The original activity goes through all its lifecycle methods, from `onCreate()` to `onDestroy()`. A new activity is created when the original is destroyed. As this new activity isn't in the foreground, only the `onCreate()` and `onStart()` lifecycle methods get called. Here's what happens when the user rotates the device when the activity doesn't have the focus::



1. **The user launches the activity.**
   The activity lifecycle methods `onCreate()`, `onStart()`, and `onResume()` get called.

2. **Another activity appears in front of it.**
   The activity's `onPause()` method gets called.

3. **The user rotates the device.**
   Android sees this as a configuration change. The `onStop()` and `onDestroy()` methods get called, and Android destroys the activity. A new activity is created in its place.

4. **The activity is visible but not in the foreground.**
   The `onCreate()` and `onStart()` methods get called. As the activity is visible but doesn't have the focus, `onResume()` isn't called.

I see, the replacement activity doesn't reach a state of "running" because it's not in the foreground. But what if you navigate away from the activity completely so it's not even visible? If the activity's stopped, do onResume() and onPause() get called before onStop()?

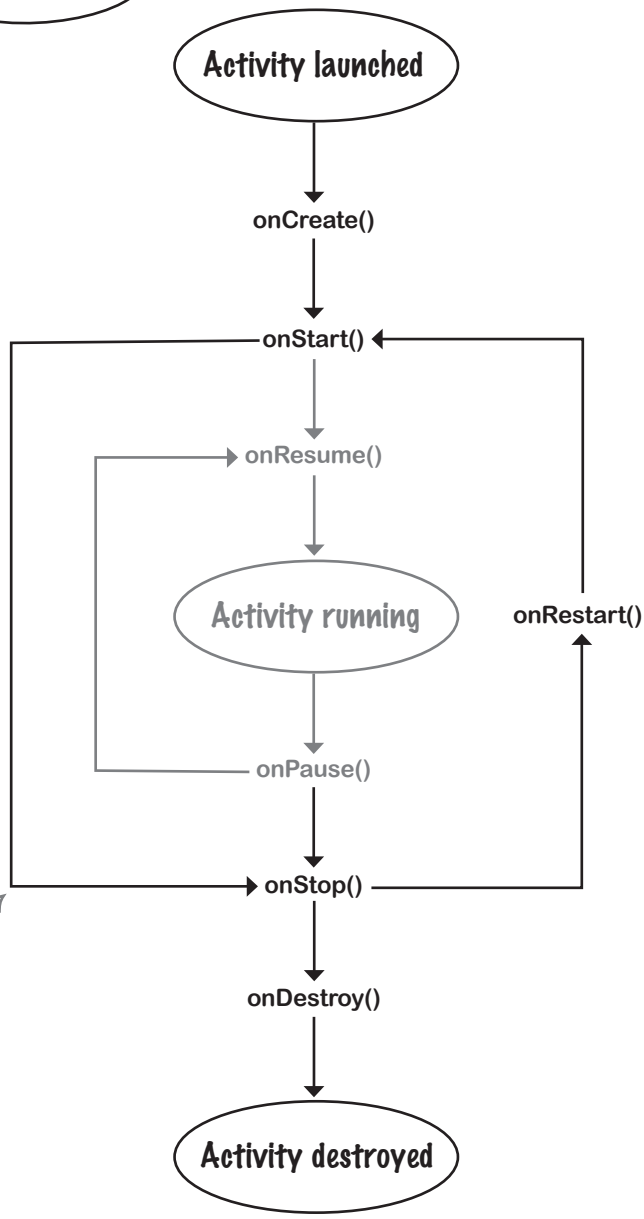
Activities can go straight from onStart() to onStop() and bypass onPause() and onResume().

If you have an activity that's visible, but never in the foreground and never has the focus, the `onPause()` and `onResume()` methods ***never get called***.

The `onResume()` method gets called when the activity appears in the foreground and has the focus. If the activity is only visible behind other activities, the `onResume()` method doesn't get called.

Similarly, the `onPause()` method gets called only when the activity is no longer in the foreground. If the activity is never in the foreground, this method won't get called.

If an activity stops or gets destroyed before it appears in the foreground, the onStart() method is followed by the onStop() method. onResume() and onPause() are bypassed.

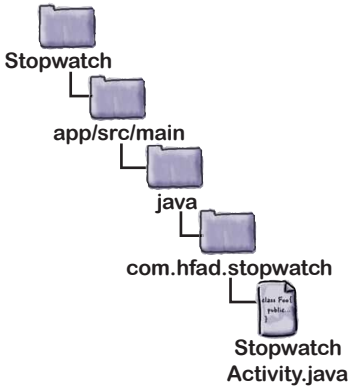


Stop the stopwatch if the activity's paused

Let's get back to the Stopwatch app.

So far we've made the stopwatch stop if the Stopwatch app isn't visible, and made it start again when the app becomes visible again. We did this by overriding the `onStop()` and `onStart()` methods like this:

```
@Override
protected void onStop() {
    super.onStop();
    wasRunning = running;
    running = false;
}

@Override
protected void onStart() {
    super.onStart();
    if (wasRunning) {
        running = true;
    }
}
```



Let's get the app to have the same behavior if the app is only partially visible. We'll get the stopwatch to stop if the activity is paused, and start again when the activity is resumed. So what changes do we need to make to the lifecycle methods?

We want the Stopwatch app to stop running when the activity is paused, and start it again (if it was running) when the activity is resumed. In other words, we want it to behave the same as when the activity is stopped or started. This means that instead of repeating the code we already have in multiple methods, we can use one method when the activity is paused or stopped, and another method when the activity is resumed or started.

Implement the onPause() and onResume() methods

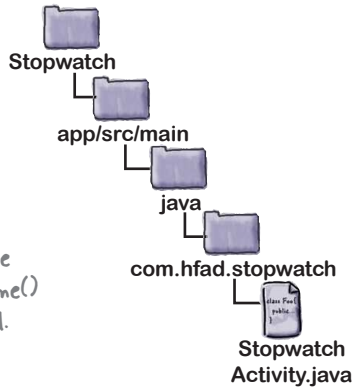
We'll start with when the activity is resumed or started.

When the activity is resumed, the activity's `onResume()` lifecycle method is called. If the activity is started, the activity's `onResume()` method is called after calling `onStart()`. The `onResume()` method is called irrespective of whether the activity is resumed or started, which means that if we move our `onStart()` code to the `onResume()` method, our app will behave the same irrespective of whether the activity is resumed or started. This means we can remove our `onStart()` method, and replace it with the `onResume()` method like this:

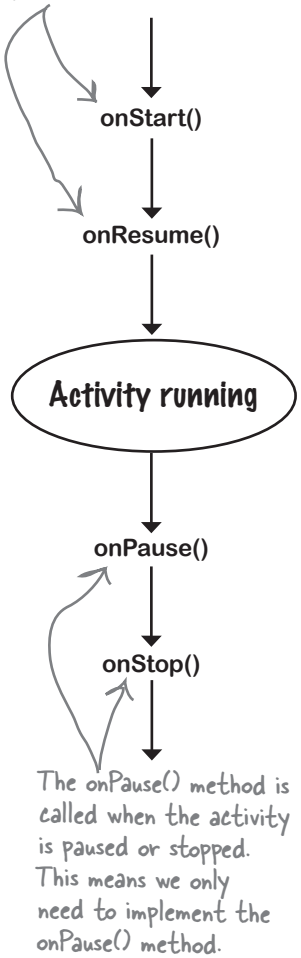
```
@Override
protected void onStart() {
    super.onStart();
    if (wasRunning) {
        running = true;
    }
}
@Override
protected void onResume() {
    super.onResume();
    if (wasRunning) {
        running = true;
    }
}
```

Delete the onStart() method.

Add the onResume() method.



The onResume() method is called when the activity is started or resumed. As we want the app to do the same thing irrespective of whether it's started or resumed, we only need to implement the onResume() method.



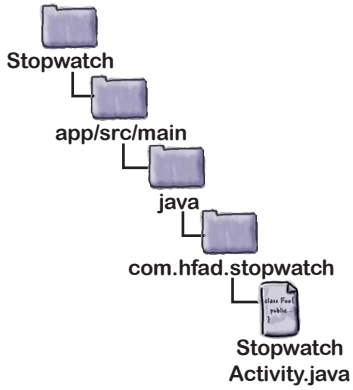
We can do something similar when the activity is paused or stopped.

When the activity is paused, the activity's `onPause()` lifecycle method is called. If the activity is stopped, the activity's `onPause()` method is called prior to calling `onStop()`. The `onPause()` method is called irrespective of whether the activity is paused or stopped, which means we can move our `onStop()` code to the `onPause()` method:

```
@Override
protected void onStop() {
    super.onStop();
    wasRunning = running;
    running = false;
}
@Override
protected void onPause() {
    super.onPause();
    wasRunning = running;
    running = false;
}
```

Delete the onStop() method.

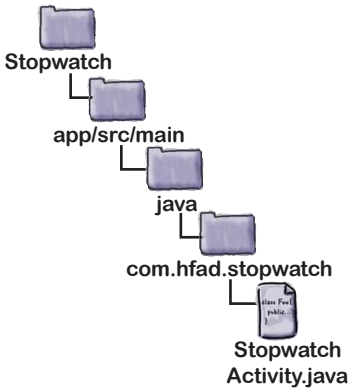
Add the onPause() method.



The onPause() method is called when the activity is paused or stopped. This means we only need to implement the onPause() method.

The complete StopwatchActivity code

Here's the full *StopwatchActivity.java* code for the finished app (with changes in bold):




```
package com.hfad.stopwatch;

import android.app.Activity;
import android.os.Bundle;
import android.view.View;
import java.util.Locale;
import android.os.Handler;
import android.widget.TextView;

public class StopwatchActivity extends Activity {
    //Number of seconds displayed on the stopwatch.
    private int seconds = 0;
    //Is the stopwatch running?
    private boolean running;
    private boolean wasRunning;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_stopwatch);
        if (savedInstanceState != null) {
            seconds = savedInstanceState.getInt("seconds");
            running = savedInstanceState.getBoolean("running");
            wasRunning = savedInstanceState.getBoolean("wasRunning");
        }
        runTimer();
    }

    @Override
    public void onSaveInstanceState(Bundle savedInstanceState) {
        savedInstanceState.putInt("seconds", seconds);
        savedInstanceState.putBoolean("running", running);
        savedInstanceState.putBoolean("wasRunning", wasRunning);
    }
```

Use seconds, running, and wasRunning respectively to record the number of seconds passed, whether the stopwatch is running, and whether the stopwatch was running before the activity was paused.

Get the previous state of the stopwatch if the activity's been destroyed and recreated.

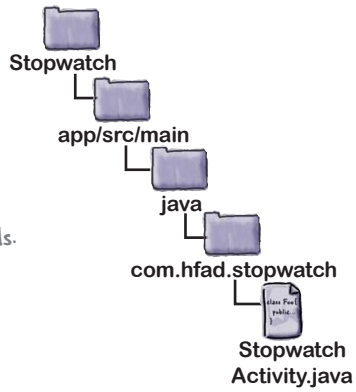
Save the state of the stopwatch if it's about to be destroyed.

The activity code continues over the page.

The activity code (continued)

```
@Override
protected void onStop() {
    super.onStop();
    wasRunning = running;
    running = false;
}

@Override
protected void onStart() {
    super.onStart();
    if (wasRunning) {
        running = true;
    }
}
```

Delete these two methods.




```
@Override
protected void onPause() {
    super.onPause();
    wasRunning = running;
    running = false;
}

@Override
protected void onResume() {
    super.onResume();
    if (wasRunning) {
        running = true;
    }
}

//Start the stopwatch running when the Start button is clicked.
public void onClickStart(View view) {
    running = true;
}
```

If the activity's paused, stop the stopwatch.

If the activity's resumed, start the stopwatch again if it was running previously.

This gets called when the Start button is clicked.

The activity code continues over the page.

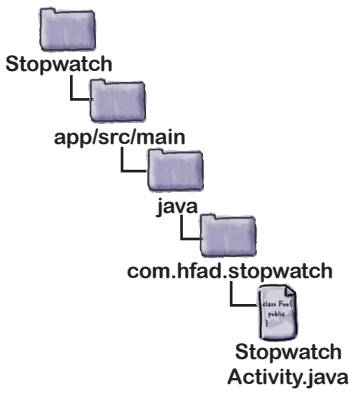
The activity code (continued)

```
    //Stop the stopwatch running when the Stop button is clicked.
    public void onClickStop(View view) {
        running = false;
    }

    //Reset the stopwatch when the Reset button is clicked.
    public void onClickReset(View view) {
        running = false;
        seconds = 0;
    }

    //Sets the number of seconds on the timer.
    private void runTimer() {
        final TextView timeView = (TextView)findViewById(R.id.time_view);
        final Handler handler = new Handler();
        handler.post(new Runnable() {
            @Override
            public void run() {
                int hours = seconds/3600;
                int minutes = (seconds%3600)/60;
                int secs = seconds%60;
                String time = String.format(Locale.getDefault(),
                            "%d:%02d:%02d", hours, minutes, secs);
                timeView.setText(time);
                if (running) {
                    seconds++;
                }
                handler.postDelayed(this, 1000);
            }
        });
    }
}
```

This gets called when the Stop button is clicked.

This gets called when the Reset button is clicked.

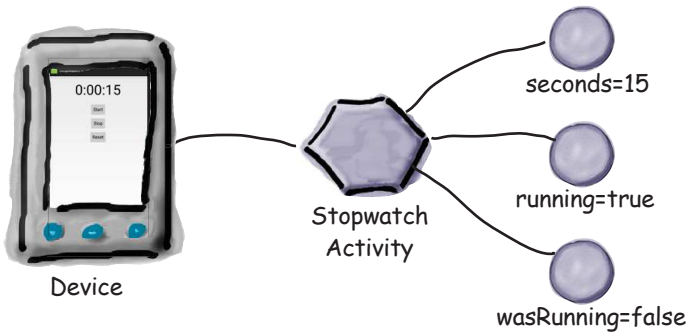
The runTimer() method uses a Handler to increment the seconds and update the text view.



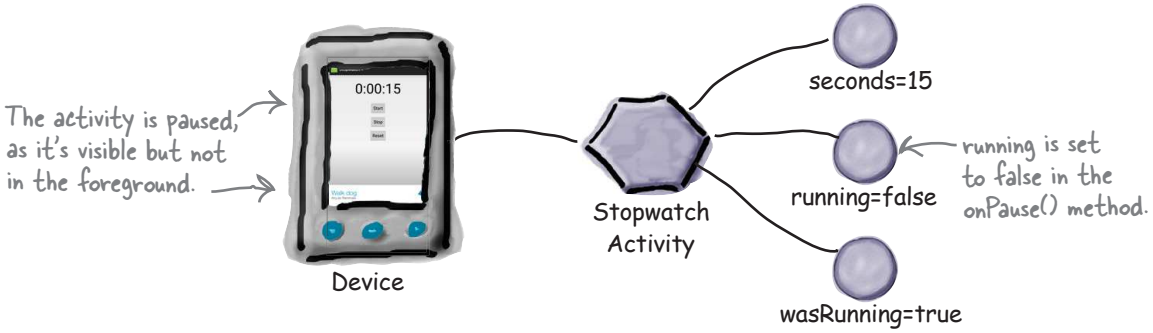
Let's go through what happens when the code runs.

What happens when you run the app

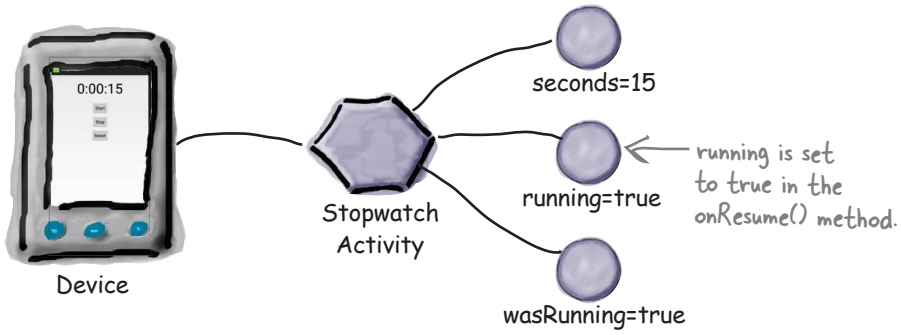
1. **The user starts the app, and clicks on the Start button to set the stopwatch going.**
The `runTimer()` method starts incrementing the number of seconds displayed in the `time_view` text view.



2. **Another activity appears in the foreground, leaving StopwatchActivity partially visible.**
The `onPause()` method gets called, `wasRunning` is set to `true`, `running` is set to `false`, and the number of seconds stops incrementing.



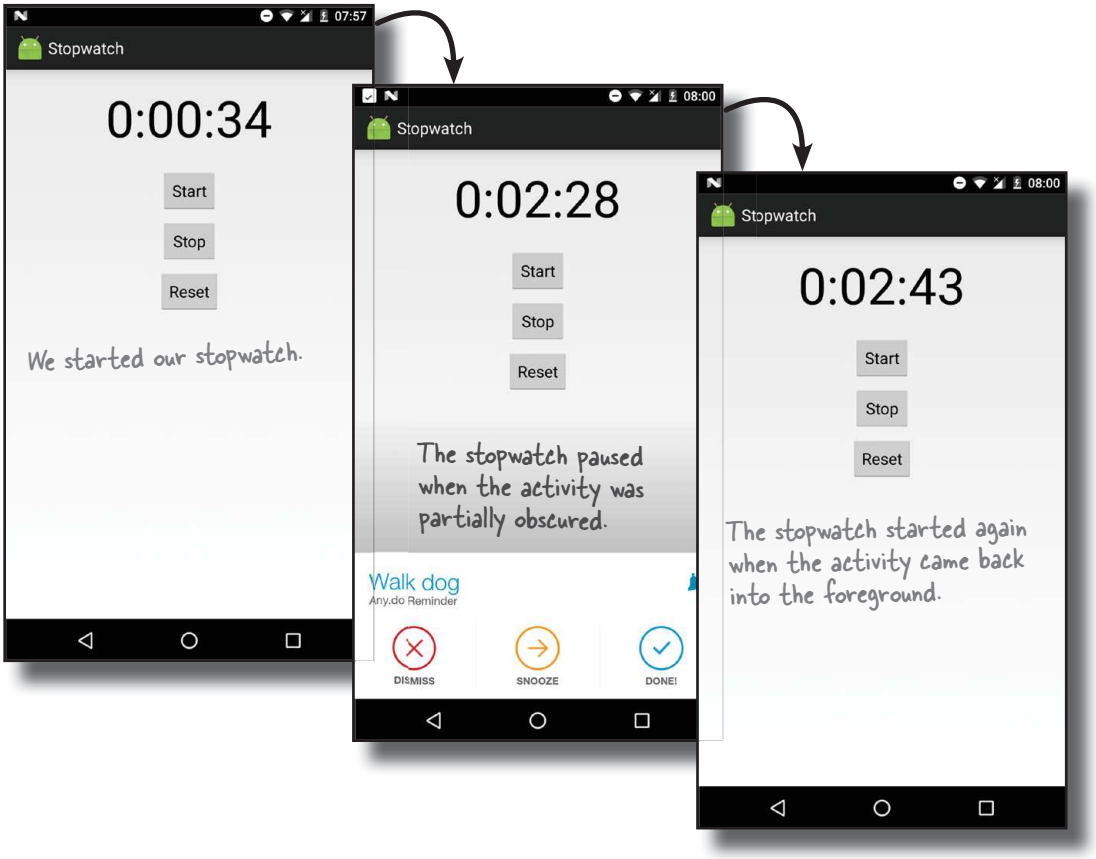
3. **When StopwatchActivity returns to the foreground, the onResume() method gets called, running is set to true, and the number of seconds starts incrementing again.**



test drive

Test drive the app

Save the changes to your activity code, then run the app. When you click on the Start button, the timer starts; it stops when the app is partially obscured by another activity; and it starts again when the app is back in the foreground.



BE the Activity

On the right, you'll see some activity code. Your job is to play like you're the activity and say which code will run in each of the situations below. We've labeled the code we want you to consider. We've done the first one to start you off.

User starts the activity and starts using it.

Code segments A, G, D. The activity is created, then made visible, then receives the focus.

User starts the activity, starts using it, then switches to another app.

This one's tough.

User starts the activity, starts using it, rotates the device, switches to another app, then goes back to the activity.

```
...
class MyActivity extends Activity{

    protected void onCreate(
              Bundle savedInstanceState) {
    A //Run code A
      ...
    }

    protected void onPause() {
    B //Run code B
      ...
    }

    protected void onRestart() {
    C //Run code C
      ...
    }

    protected void onResume() {
    D //Run code D
      ...
    }

    protected void onStop() {
    E //Run code E
      ...
    }

    protected void onRecreate() {
    F //Run code F
      ...
    }

    protected void onStart() {
    G //Run code G
      ...
    }

    protected void onDestroy() {
    H //Run code H
      ...
    }
}
```

solution

BE the Activity Solution

On the right, you'll see some activity code. Your job is to play like you're the activity and say which code will run in each of the situations below. We've labeled the code we want you to consider. We've done the first one to start you off.

User starts the activity and starts using it.

Code segments A, G, D. The activity is created, then made visible, then receives the focus.

User starts the activity, starts using it, then switches to another app.

Code segments A, G, D, B, E. The activity is created, then made visible, then receives the focus. When the user switches to another app, it loses the focus and is no longer visible to the user.

User starts the activity, starts using it, rotates the device, switches to another app, then goes back to the activity.

Code segments A, G, D, B, E, H, A, G, D, B, E, C, G, D. First, the activity is created, made visible, and receives the focus. When the device is rotated, the activity loses the focus, stops being visible, and is destroyed. It's then created again, made visible, and receives the focus. When the user switches to another app and back again, the activity loses the focus, loses visibility, becomes visible again, and regains the focus.

```
...
class MyActivity extends Activity{

    protected void onCreate(
              Bundle savedInstanceState) {
    A //Run code A
      ...
    }

    protected void onPause() {
    B //Run code B
      ...
    }

    protected void onRestart() {
    C //Run code C
      ...
    }

    protected void onResume() {
    D //Run code D
      ...
    }

    protected void onStop() {
    E //Run code E
      ...
    }

    protected void onRecreate() {
    F //Run code F
      ...
    }

    protected void onStart() {
    G //Run code G
      ...
    }

    protected void onDestroy() {
    H //Run code H
      ...
    }
}
```

There's no lifecycle method called onRecreate().

Your handy guide to the lifecycle methods

Method	When it's called	Next method
onCreate()	When the activity is first created. Use it for normal static setup, such as creating views. It also gives you a `Bundle` that contains the previously saved state of the activity.	`onStart()`
onRestart()	When your activity has been stopped but just before it gets started again.	`onStart()`
onStart()	When your activity is becoming visible. It's followed by `onResume()` if the activity comes into the foreground, or `onStop()` if the activity is made invisible.	`onResume()` or `onStop()`
onResume()	When your activity is in the foreground.	`onPause()`
onPause()	When your activity is no longer in the foreground because another activity is resuming. The next activity isn't resumed until this method finishes, so any code in this method needs to be quick. It's followed by `onResume()` if the activity returns to the foreground, or `onStop()` if it becomes invisible.	`onResume()` or `onStop()`
onStop()	When the activity is no longer visible. This can be because another activity is covering it, or because this activity is being destroyed. It's followed by `onRestart()` if the activity becomes visible again, or `onDestroy()` if the activity is being destroyed.	`onRestart()` or `onDestroy()`
onDestroy()	When your activity is about to be destroyed or because the activity is finishing.	None

toolbox

Your Android Toolbox

You've got Chapter 4 under your belt and now you've added the activity lifecycle to your toolbox.

You can download the full code for the chapter from https://tinyurl.com/HeadFirstAndroid.

BULLET POINTS

- Each app runs in its own process by default.
- Only the main thread can update the user interface.
- Use a `Handler` to schedule code or post code to a different thread.
- A device configuration change results in the activity being destroyed and recreated.
- Your activity inherits the lifecycle methods from the `android.app.Activity` class. If you override any of these methods, you need to call up to the method in the superclass.
- `onSaveInstanceState(Bundle)` enables your activity to save its state before the activity gets destroyed. You can use the `Bundle` to restore state in `onCreate()`.
- You add values to a `Bundle` using `bundle.put*("name", value)`. You retrieve values from the bundle using `bundle.get*("name")`.
- `onCreate()` and `onDestroy()` deal with the birth and death of the activity.
- `onRestart()`, `onStart()`, and `onStop()` deal with the visibility of the activity.
- `onResume()` and `onPause()` handle when the activity gains and loses the focus.