CP3406_CP5307 Codelab 5.2: Complex Lifecycle Situations

Contents

1. Introduction and app overview
2. Avoid lifecycle mistakes
3. Simulate app shutdown and use onSaveInstanceState()
4. Explore configuration changes
5. Summary

1. Introduction and app overview

In the last codelab, you learned about the `Activity` and `Fragment` lifecycles, and you explored the methods that are called when the lifecycle state changes in activities and fragments. In this codelab, you will explore the activity lifecycle in greater detail, including how to start and stop background tasks when an app moves through the lifecycle, handle app shutdown, and handle configuration changes.

In this codelab, you expand on the DessertClicker app from the previous codelab. You add a backgroundtimer, then convert the app to use the Android lifecycle library.

2. Avoid lifecycle mistakes

In the previous codelab, you learned how to observe the activity and fragment lifecycles by overriding various lifecycle callbacks, and logging when the system invokes those callbacks. In this task, you explore a more complex example of managing lifecycle tasks in the DessertClicker app. You use a timer that prints a log statement every second, with the count of the number of seconds it has been running.

Step 1: Set up DessertTimer

1. Open the DessertClicker app from the last codelab.
2. Create a new Kotlin Class in your package called `DessertTimer.kt`.
3. Copy and paste in the following code:

```
class DessertTimer {

    // The number of seconds counted since the timer started
    var secondsCount = 0

    /**
     * [Handler] is a class meant to process a queue of messages (known as
[android.os.Message]s)
     * or actions (known as [Runnable]s)
     */
    private var handler = Handler(Looper.getMainLooper())
    private lateinit var runnable: Runnable
    fun startTimer() {
        // Create the runnable action, which prints out a log and increments
        // the seconds counter
        runnable = Runnable {
            secondsCount++
            Timber.i("Timer is at : $secondsCount")
            // postDelayed re-adds the action to the queue of actions the
               Handler is cycling
            // through. The delayMillis param tells the handler to run the
            // runnable in 1 second (1000ms)
            handler.postDelayed(runnable, 1000)
        }

        // This is what initially starts the timer
        handler.postDelayed(runnable, 1000)

        // Note that the Thread the handler runs on is determined by a class
        // called Looper.
    }

    fun stopTimer() {
        // Removes all pending posts of runnable from the handler's queue,
        // effectively stopping the timer
        handler.removeCallbacks(runnable)
    }
}
```

4. Notice that the `DessertTimer` class includes `startTimer()` and `stopTimer()`, which start and stop the timer. When `startTimer()` is running, the timer prints a log message every second, with the total count of the seconds the time has been running. The `stopTimer()` method, in turn, stops the timer and the log statements.

Note: The `DessertTimer` class uses a background thread for the timer, with associated `Runnable` and `Handler` classes. You don't need to know about these things for this codelab.

5. Open `MainActivity.kt`. At the top of the class, just below the `dessertsSold` variable, add a variable for the timer:

```
private lateinit var dessertTimer: DessertTimer
```

6. Scroll down to `onCreate()` and create a new `DessertTimer` object, just after the call to `setOnClickListener()`:

```
dessertTimer = DessertTimer()
```

Now that you have a dessert timer object, consider where you should start and stop the timer to get it to run **only** when the activity is on-screen. You look at a few options in the next steps.

Step 2: Start and stop the timer

The `onStart()` method is called just before the activity becomes visible. The `onStop()` method is called after the activity stops being visible. These callbacks seem like good candidates for when to start and stop the timer.

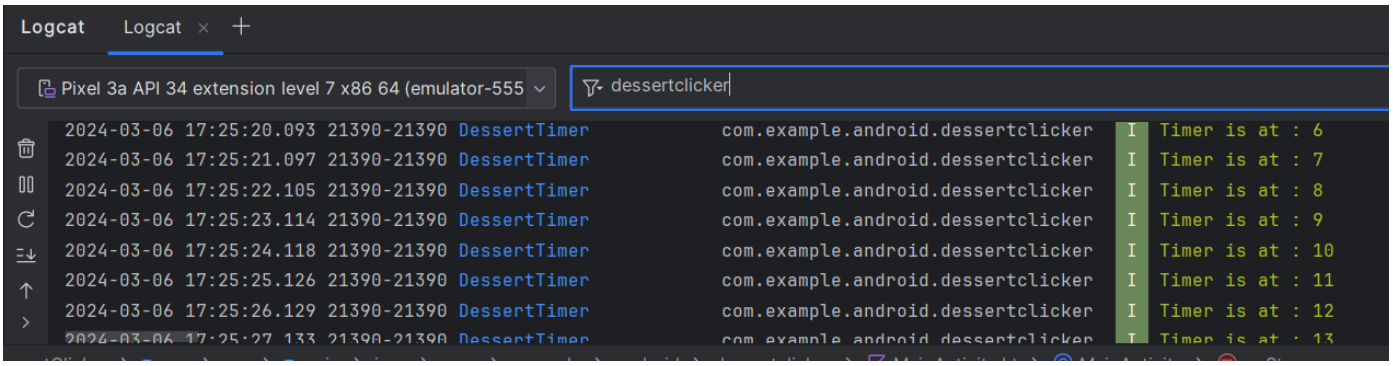
1. In the `MainActivity` class, start the timer in the `onStart()` callback:

```
override fun onStart() {
   super.onStart()
   dessertTimer.startTimer()

   Timber.i("onStart called")
}
```

2. Stop the timer in `onStop()`:

```
override fun onStop() {
   super.onStop()
   dessertTimer.stopTimer()

   Timber.i("onStop Called")
}
```

3. Compile and run the app. In Android Studio, click the **Logcat** pane. In the Logcat search box, enter `dessertclicker`, which will filter by both the `MainActivity` and `DessertTimer` classes.



```
2024-03-06 17:25:20.093 21390-21390 DessertTimer     com.example.android.dessertclicker  I  Timer is at : 6
2024-03-06 17:25:21.097 21390-21390 DessertTimer     com.example.android.dessertclicker  I  Timer is at : 7
2024-03-06 17:25:22.105 21390-21390 DessertTimer     com.example.android.dessertclicker  I  Timer is at : 8
2024-03-06 17:25:23.114 21390-21390 DessertTimer     com.example.android.dessertclicker  I  Timer is at : 9
2024-03-06 17:25:24.118 21390-21390 DessertTimer     com.example.android.dessertclicker  I  Timer is at : 10
2024-03-06 17:25:25.126 21390-21390 DessertTimer     com.example.android.dessertclicker  I  Timer is at : 11
2024-03-06 17:25:26.129 21390-21390 DessertTimer     com.example.android.dessertclicker  I  Timer is at : 12
2024-03-06 17:25:27.133 21390-21390 DessertTimer     com.example.android.dessertclicker  I  Timer is at : 13
```

Notice that once the app starts, the timer also starts running immediately.

4. Click the **Back** button and notice that the timer stops again. The timer stops because both the activity and the timer it controls have been destroyed.

5. Use the recents screen to return to the app. Notice in Logcat that the timer restarts from 0.
6. Click the **Share** button. Notice in Logcat that the timer is still running.
7. Click the **Home** button. Notice in Logcat the timer stops running.
8. Use the recents screen to return to the app. Notice in Logcat the timer starts up again from where it left off because we called `startTimer()` in the `onStart()` method.
9. In `MainActivity`, in the `onStop()` method, comment out the call to `stopTimer()`. Commenting out `stopTimer()` demonstrates the case where you start an operation in `onStart()`, but forget to stop it again in `onStop()`.
10. Compile and run the app, and click the Home button after the timer starts. Even though the app is in the background, the timer is running, and continually using system resources. Having the timer continue may unnecessarily use computing resources on your phone, and probably not the behavior you want.

The general pattern is when you set up or start something in a callback, you stop or remove that thing in a corresponding callback. This way, you avoid having anything running when it's no longer needed.

Note: In some cases, such as music playback, you want to keep the thing running. There are proper and efficient ways to keep something running, but they are beyond the scope of this lesson.

11. Uncomment the line in `onStop()` where you stop the timer.
12. Cut and paste the `startTimer()` call from `onStart()` to `onCreate()`. This change demonstrates the case where you both initialize and start a resource in `onCreate()`, rather than using `onCreate()` to initialize it and `onStart()` to start it.
13. Compile and run the app. Notice that the timer starts running, as you would expect.
14. Click Home to stop the app. The timer stops running, as you would expect.
15. Use the recents screen to return to the app. Notice that the timer does **not** start again in this case, because `onCreate()` is only called when the app starts—it's not called when an app returns to the foreground.

Key points to remember:

- When you set up a resource in a lifecycle callback, also tear the resource down.
- Do setup and teardown in corresponding methods.
- If you set up something in `onStart()`, stop or tear it down again in `onStop()`.

3. Simulate app shutdown and use onSaveInstanceState()

What happens to your app and its data if Android shuts down that app while it is in the background? This tricky edge case is important to understand.

When your app goes into the background, it's not destroyed, it's only stopped and waiting for the user to return to it. But one of the Android OS's main concerns is keeping the activity that's in the foreground running smoothly. For example, if your user is using a GPS app to help them catch a bus, it's important to render that GPS app quickly and keep showing the directions. It's less important to keep the DessertClicker app, which the user might not have looked at for a few days, running smoothly in the background.

Android regulates background apps so that the foreground app can run without problems. For example, Android limits the amount of processing that apps running in the background can do.

Sometimes Android even shuts down an entire app process, which includes every activity associated with the app. Android does this kind of shutdown when the system is stressed and in danger of visually lagging, so no additional callbacks or code is run at this point. Your app's process is simply shut down, silently, in the background. But to the user, it doesn't look like the app has been closed. When the user navigates back to an app that the Android OS has shut down, Android restarts that app.

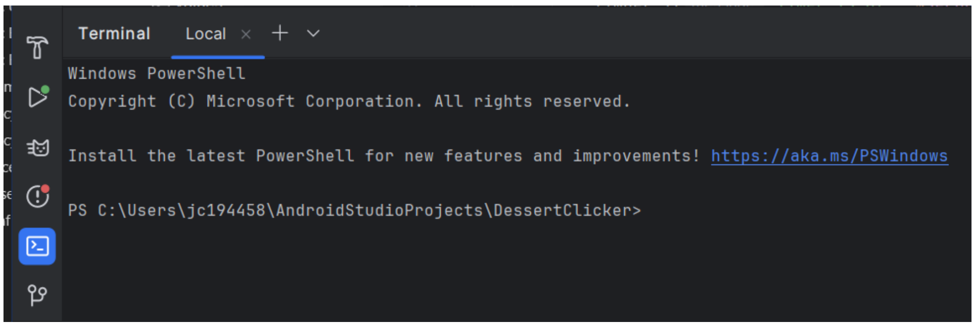
In this task, you simulate an Android process shutdown and examine what happens to your app when it starts up again.

Note: Before you start, make sure you are running an emulator or device that supports API 28 or higher.

Step 1: Use adb to simulate a process shutdown

The Android Debug Bridge (`adb`) is a command-line tool that lets you send instructions to emulators and devices attached to your computer. In this step, you use `adb` to close your app's process and see what happens when Android shuts down your app.

1. Compile and run your app. Click the cupcake a few times.
2. Press the Home button to put your app into the background. Your app is now stopped, and the app is subject to being closed if Android needs the resources that the app is using.
3. In Android Studio, click the **Terminal** tab to open the command-line terminal.



```
Windows PowerShell
Copyright (C) Microsoft Corporation. All rights reserved.

Install the latest PowerShell for new features and improvements! https://aka.ms/PSWindows

PS C:\Users\jc194458\AndroidStudioProjects\DessertClicker>
```

4. Type `adb` and press Return.

If you see a lot of output that begins with `Android Debug Bridge version X.XX.X` and ends with `tags to be used by logcat (see logcat --help)`, everything is fine. If instead you see `adb: command not found`, make sure the `adb` command is available in your execution path.

5. Copy and paste this comment into the command line and press Return:

```
adb shell am kill com.example.android.dessertclicker
```

This command tells any connected devices or emulators to send a STOP message to terminate the process with the `dessertclicker` package name, but only if the app is in the background. Because your app was in the background, nothing shows on the device or emulator screen to indicate that your process has been stopped. In Android Studio, click the **Run** tab to see the `onStop()` method called. Click the **Logcat** tab to see that the `onDestroy()` callback was never run—your activity simply ended.

6. Use the recents screen to return to the app. Your app appears in recents whether it has been put into the background or has been stopped altogether. When you use the recents screen to return to the app, the activity is started up again. The activity goes through the entire set of startup lifecycle callbacks, including `onCreate()`.
7. Notice that when the app restarted, it resets your "score" (both the number of desserts sold and the total dollars) to the default values (0). If Android shut down your app, why didn't it save your state?

When the OS restarts your app for you, Android tries its best to reset your app to the state it had before. Android takes the state of some of your views and saves it in a bundle whenever you navigate away from the activity. Some examples of data that's automatically saved are the text in an EditText (as long as they have an ID set in the layout), and the back stack of your activity.

However, sometimes the Android OS doesn't know about all your data. For example, if you have a custom variable like `revenue` in the DessertClicker app, the Android OS doesn't know about this data or its importance to your activity. You need to add this data to the bundle yourself.

Step 2: Use onSaveInstanceState() to save bundle data

The `onSaveInstanceState()` method is the callback you use to save any data that you might need if the Android OS destroys your app. Recall in the lifecycle callback diagram, `onSaveInstanceState()` is called after the activity has been stopped. It's called every time your app goes into the background.

Think of the `onSaveInstanceState()` call as a safety measure; it gives you a chance to save a small amount of information to a bundle as your activity exits the foreground. The system saves this data now because if it waited until it was shutting down your app, the OS might be under resource pressure. Saving the data each time ensures that update data in the bundle is available to restore, if it is needed.

1. In `MainActivity`, override the `onSaveInstanceState()` callback, and add a `Timber` log statement. Note: It may already be overloaded so check the class first.

```
override fun onSaveInstanceState(outState: Bundle) {
   super.onSaveInstanceState(outState)

   Timber.i("onSaveInstanceState Called")
}
```

Note: There are two overrides for `onSaveInstanceState()`, one with just an `outState` parameter, and another that includes `outState` and `outPersistentState` parameters. Use the one shown in the code above, with the single `outState` parameter.

2. Compile and run the app, and click the **Home** button to put it into the background. Notice that the `onSaveInstanceState()` callback occurs just after both `onPause()` and `onStop()`:

3. At the top of the file, just before the class definition, add these constants in a companion object:

```
companion object {

    const val KEY_TIMER_SECONDS = "timer_seconds_key"

    const val KEY_REVENUE = "revenue_key"

    const val KEY_DESSERT_SOLD = "dessert_sold_key"

}
```

You will use these keys for both saving and retrieving data from the instance state bundle.

4. Scroll down to `onSaveInstanceState()`, and notice the `outState` parameter, which is of type `Bundle`.

A bundle is a collection of key-value pairs, where the keys are always strings. You can put primitive values, such as `int` and `boolean` values, into the bundle. Because the system keeps this bundle in RAM, it's a best practice to keep the data in the bundle small. The size of this bundle is also limited, though the size varies from device to device. Generally you should store far less than 100k, otherwise you risk crashing your app with the `TransactionTooLargeException` error.

5. In `onSaveInstanceState()`, put the `revenue` value (an integer) into the bundle with the `putInt()` method:

```
outState.putInt(KEY_REVENUE,  revenue)
```

The `putInt()` method (and similar methods from the `Bundle` class like `putFloat()` and `putString()` takes two arguments: a string for the key (the `KEY_REVENUE` constant), and the actual value to save.

6. Repeat the same process with the number of desserts sold, and the status of the timer:

```
outState.putInt(KEY_DESSERT_SOLD, dessertsSold)
outState.putInt(KEY_TIMER_SECONDS, dessertTimer.secondsCount)
```

Step 3: Use onCreate() to restore bundle data

1. Scroll up to `onCreate()`, and examine the method signature:

```
override  fun  onCreate(savedInstanceState:  Bundle?)  {
```

Notice that `onCreate()` gets a `Bundle` each time it is called. When your activity is restarted due to a process shut-down, the bundle that you saved is passed to `onCreate()`. If your activity was starting fresh, this bundle in `onCreate()` is `null`. So if the bundle is not `null`, you know you're "re-creating" the activity from a previously known point.

Note: If the activity is being re-created, the `onRestoreInstanceState()` callback is called after `onStart()`, also with the bundle. Most of the time, you restore the activity state in `onCreate()`. But because `onRestoreInstanceState()` is called after `onStart()`, if you ever need to restore some state after `onCreate()` is called, you can use `onRestoreInstanceState()`.

2. Add this code to `onCreate()`, after the `DessertTimer` setup:

```
if (savedInstanceState != null) {
   revenue = savedInstanceState.getInt(KEY_REVENUE, 0)
}
```

The test for `null` determines whether there is data in the bundle, or if the bundle is `null`, which in turn tells you if the app has been started fresh or has been re-created after a shutdown. This test is a common pattern for restoring data from the bundle.

Notice that the key you used here (`KEY_REVENUE`) is the same key you used for `putInt()`. To make sure you use the same key each time, it is a best practice to define those keys as constants. You use `getInt()` to get data out of the bundle, just as you used `putInt()` to put data into the bundle. The `getInt()` method takes two arguments:

- A string that acts as the key, for example `"key_revenue"` for the revenue value.
- A default value in case no value exists for that key in the bundle.

The integer you get from the bundle is then assigned to the `revenue` variable, and the UI will use that value.

3. Add `getInt()` methods to restore the number of desserts sold and the value of the timer:

```
if (savedInstanceState != null) {
   revenue = savedInstanceState.getInt(KEY_REVENUE, 0)
   dessertsSold = savedInstanceState.getInt(KEY_DESSERT_SOLD, 0)
   dessertTimer.secondsCount =
       savedInstanceState.getInt(KEY_TIMER_SECONDS, 0)
}
```

4. Compile and run the app.
5. Stop the app by selecting **Run** > **Stop ‘app’** from the Android Studio menu. Now the app is installed on your device/emulator but the app is not running.
6. Open app launcher in your device/emulator and select **Dessert Clicker** app. Press the cupcake at least five times until it switches to a donut. Click Home to put the app into the background.

7. In the Android Studio **Terminal** tab, run `adb` to shut down the app's process.

```
adb shell am kill com.example.android.dessertclicker
```

8. Use the recents screen to return to the app. Notice that this time the app returns with the correct revenue and desserts sold values from the bundle. But also notice that the dessert has returned to a cupcake. There's one more thing left to do to ensure that the app returns from a shutdown exactly the way it was left.
9. In `MainActivity`, examine the `showCurrentDessert()` method. Notice that this method determines which dessert image should be displayed in the activity based on the current number of desserts sold and the list of desserts in the `allDesserts` variable.

```
for (dessert in allDesserts) {
   if (dessertsSold >= dessert.startProductionAmount) {
       newDessert = dessert
   }
    else break
}
```

This method relies on the number of desserts sold to choose the right image. Therefore, you don't need to do anything to store a reference to the image in the bundle in `onSaveInstanceState()`. In that bundle, you're already storing the number of desserts sold.

10. In `onCreate()`, in the block that restores the state from the bundle, call `showCurrentDessert()`:

```
 if (savedInstanceState != null) {
   revenue = savedInstanceState.getInt(KEY_REVENUE, 0)
   dessertsSold = savedInstanceState.getInt(KEY_DESSERT_SOLD, 0)
   dessertTimer.secondsCount =
      savedInstanceState.getInt(KEY_TIMER_SECONDS, 0)
   showCurrentDessert()
}
```

11. Compile and run the app.
12. Stop the app by selecting **Run** > **Stop ‘app'** from the Android Studio menu. Now the app is installed on your device/emulator but the app is not running.
13. Open app launcher and select **Dessert Clicker** app. Click on the dessert icon a few times.
14. Put the app into the background. Use `adb` to shut down the process.

```
adb shell am kill com.example.android.dessertclicker
```

15. Use the recents screen to return to the app. Note now that both the values for desserts told, total revenue, and the dessert image are correctly restored.

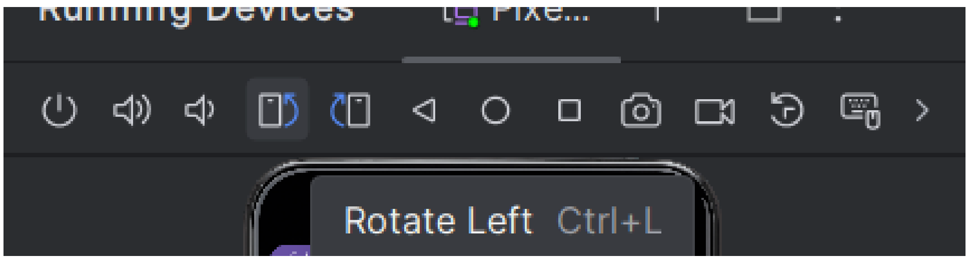
4. Explore configuration changes

There's one last special case in managing the activity and fragment lifecycle that is important to understand: how configuration changes affect the lifecycle of your activities and fragments.

A *configuration change* happens when the state of the device changes so radically that the easiest way for the system to resolve the change is to completely shut down and rebuild the activity. For example, if the user changes the device language, the whole layout might need to change to accommodate different text directions. If the user plugs the device into a dock or adds a physical keyboard, the app layout may need to take advantage of a different display size or layout. And if the device orientation changes, for example if the device is rotated from portrait to landscape or back the other way, then the layout may need to change to fit the new orientation.

Step 1: Explore device rotation and the lifecycle callbacks

1. Compile and run your app, and open Logcat.
2. Rotate the device or emulator to landscape mode. You can rotate the emulator left or right with the rotation buttons, or with the `Control` and arrow keys (`Command` and arrow keys on a Mac).



3. Examine the output in Logcat. Filter the output on `MainActivity`.

Notice that when the device or emulator rotates the screen, the system calls all the lifecycle callbacks to shut down the activity. Then, as the activity is re-created, the system calls all the lifecycle callbacks to start the activity.

4. In `MainActivity`, comment out the entire `onSaveInstanceState()` method.
5. Compile and run your app again. Click the cupcake a few times, and rotate the device or emulator. This time, when the device is rotated and the activity is shut down and re-created, the activity starts up with default values.

When a configuration change occurs, Android uses the same instance state bundle that you learned about in the previous task to save and restore the state of the app. As with a process shutdown, use `onSaveInstanceState()` to put your app's data into the bundle. Then restore the data in `onCreate()`, to avoid losing activity state data if the device is rotated.

6. In `MainActivity`, uncomment the `onSaveInstanceState()` method, run the app, click the cupcake, and rotate the app or device. Notice this time the dessert data is retained across activity rotation.

5. Summary

Lifecycle tips

- If you set up or start something in a lifecycle callback, stop or remove that thing in the corresponding callback. By stopping the thing, you make sure it doesn't keep running when it's no longer needed. For example, if you set up a timer in `onStart()`, you need to pause or stop the timer in `onStop()`.
- Use `onCreate()` only to initialize the parts of your app that run once, when the app first starts. Use `onStart()` to start the parts of your app that run both when the app starts, and each time the app returns to the foreground.

Process shutdowns and saving activity state

- Android regulates apps running in the background so that the foreground app can run without problems. This regulation includes limiting the amount of processing that apps in the background can do, and sometimes even shutting down your entire app process.
- The user cannot tell if the system has shut down an app in the background. The app still appears in the recents screen and should restart in the same state in which the user left it.
- The Android Debug Bridge (`adb`) is a command-line tool that lets you send instructions to emulators and devices attached to your computer. You can use `adb` to simulate a process shutdown in your app.
- When Android shuts down your app process, the `onDestroy()` lifecycle method is not called. The app just stops.

Preserving activity and fragment state

- When your app goes into the background, just after `onStop()` is called, app data is saved to a bundle. Some app data, such as the contents of an `EditText`, is automatically saved for you.
- The bundle is an instance of `Bundle`, which is a collection of keys and values. The keys are always strings.
- Use the `onSaveInstanceState()` callback to save other data to the bundle that you want to retain, even if the app was automatically shut down. To put data into the bundle, use the bundle methods that start with `put`, such as `putInt()`.
- You can get data back out of the bundle in the `onRestoreInstanceState()` method, or more commonly in `onCreate()`. The `onCreate()` method has a `savedInstanceState` parameter that holds the bundle.
- If the `savedInstanceState` variable contains `null`, the activity was started without a state bundle and there is no state data to retrieve.
- To retrieve data from the bundle with a key, use the `Bundle` methods that start with `get`, such as `getInt()`.

Configuration changes

- A *configuration change* happens when the state of the device changes so radically that the easiest way for the system to resolve the change is to shut down and rebuild the activity.

- The most common example of a configuration change is when the user rotates the device from portrait to landscape mode, or from landscape to portrait mode. A configuration change can also occur when the device language changes or a hardware keyboard is plugged in.
- When a configuration change occurs, Android invokes all the activity lifecycle's shutdown callbacks. Then Android restarts the activity from scratch, running all the lifecycle startup callbacks.
- When Android shuts down an app because of a configuration change, it restarts the activity with the state bundle that is available to `onCreate()`.
- As with process shutdown, save your app's state to the bundle in `onSaveInstanceState()`.