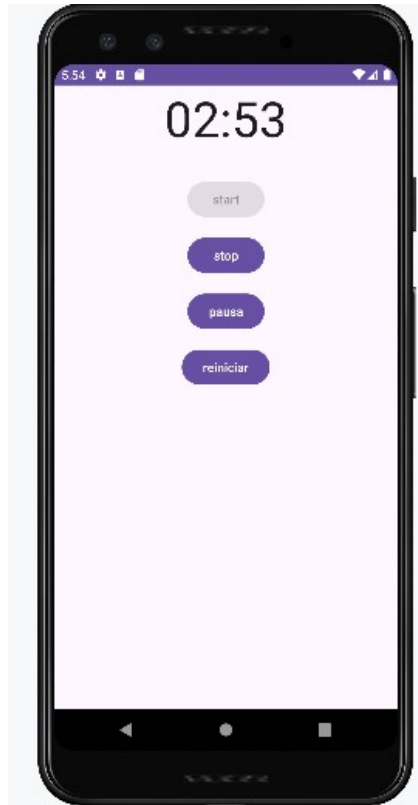


# PROYECTO 3

## CRONÓMETRO



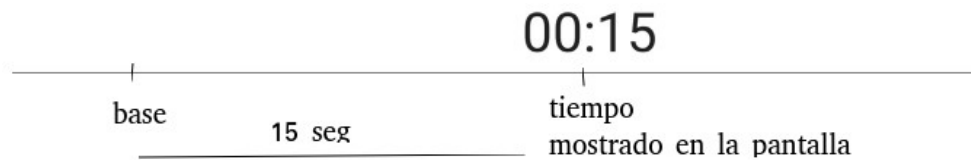
En este proyecto haremos un sencillo cronómetro que tiene tres botones: iniciar, pausar y resetear. Pese a lo sencillo de su comportamiento, veremos que es necesario gestionar correctamente lo que le sucede a la app cuando se produce un cambio estructural en el dispositivo, y también cuando otras apps se ponen en marcha y la nuestra pasa a segundo plano.

Durante su desarrollo se tratarán estos conceptos:

- El componente Chronometer
- El método onCreate
- Control del Chronometer
- Cambios estructurales
- Introducción al View Model
- Respuesta a la puesta de la app en segundo plano
- Respuesta a la minimización parcial de la app
- El ciclo de vida de las Activity
- Tests de instrumentación

## 1 – El componente Chronometer

La base de la app que vamos a hacer es el **Chronometer**. Se trata de una vista que se actualiza tras cada segundo, y nos muestra el número de segundos transcurridos desde un instante de tiempo llamado **base**



- Abre Android Studio y crea un proyecto llamado **Cronometro**, usando la plantilla **Empty Views Activity**
- Abre el archivo **res/layout/activity\_main.xml** en la vista de código fuente, borra lo que hay y pon un **LinearLayout** con orientación vertical que centre horizontalmente su contenido, así:

```
1. <LinearLayout
2.     xmlns:android="http://schemas.android.com/apk/res/android"
3.     xmlns:app="http://schemas.android.com/apk/res-auto"
4.     android:layout_width="match_parent"
5.     android:layout_height="match_parent"
6.     android:orientation="vertical"
7.     android:gravity="center_horizontal">
8. </LinearLayout>
```

- Añade este componente **Chronometer**

```
1. <Chronometer
2.     android:id="@+id/chrRelej"
3.     android:textSize="56sp"
4.     android:layout_width="wrap_content"
5.     android:layout_height="wrap_content"
6. </Chronometer>
```

- Añade al archivo **strings.xml** los strings **start**, **stop**, **pausa** y **reset** con valores **Start**, **Stop**, **Pausa** y **Reset** respectivamente.
- Añade cuatro botones uno debajo de otro después del cronómetro y deja margen a todo, de forma que la haya una separación entre todos los componentes de la interfaz, de esta forma:

```
1. <LinearLayout
2.     xmlns:android="http://schemas.android.com/apk/res/android"
3.     xmlns:app="http://schemas.android.com/apk/res-auto"
4.     android:layout_width="match_parent"
5.     android:layout_height="match_parent"
6.     android:orientation="vertical"
7.     android:gravity="center_horizontal">
8.     <Chronometer
9.         android:id="@+id/chrRelej"
10.        android:textSize="56sp"
11.        android:layout_marginTop="16dp"
12.        android:layout_width="wrap_content"
13.        android:layout_height="wrap_content"/>
```

```

14. <Button
15.     android:id="@+id/btnStart"
16.     android:layout_marginTop="32dp"
17.     android:layout_width="wrap_content"
18.     android:layout_height="wrap_content"
19.     android:text="@string/start"/>
20. <Button
21.     android:id="@+id/btnStop"
22.     android:layout_marginTop="16dp"
23.     android:layout_width="wrap_content"
24.     android:layout_height="wrap_content"
25.     android:text="@string/stop"/>
26. <Button
27.     android:id="@+id/btnPausa"
28.     android:layout_marginTop="16dp"
29.     android:layout_width="wrap_content"
30.     android:layout_height="wrap_content"
31.     android:text="@string/pausa"/>
32. <Button
33.     android:id="@+id/btnReset"
34.     android:layout_marginTop="16dp"
35.     android:layout_width="wrap_content"
36.     android:layout_height="wrap_content"
37.     android:text="@string/reset"/>
38. </LinearLayout>

```

- Vamos a usar el atributo **android:enabled** para hacer que los botones de stop, pausa y reset estén deshabilitados al iniciarse la app

```

1. <Button
2.     android:id="@+id/btnStop"
3.     android:layout_marginTop="16dp"
4.     android:layout_width="wrap_content"
5.     android:layout_height="wrap_content"
6.     android:enabled="false"
7.     android:text="@string/stop"/>
8. <Button
9.     android:id="@+id/btnPausa"
10.    android:layout_marginTop="16dp"
11.    android:layout_width="wrap_content"
12.    android:layout_height="wrap_content"
13.    android:enabled="false"
14.    android:text="@string/pausa"/>
15. <Button
16.     android:id="@+id/btnReiniciar"
17.     android:layout_marginTop="16dp"
18.     android:layout_width="wrap_content"
19.     android:layout_height="wrap_content"
20.     android:enabled="false"
21.     android:text="@string/reiniciar"/>

```

## 2 – El método onCreate

Cuando el usuario pulsa el icono de la app para ponerla en marcha, el sistema Android crea la Activity usando su método constructor (este paso nosotros no lo vemos), y justo después, llama al método **onCreate**

- **onCreate** sirve para realizar la parte de inicialización que tiene que ver con la creación de la interfaz, como indicar su layout, obtener referencias a los objetos que lo forman y definir los eventos a los que responderán los componentes de la interfaz.

- Abre el archivo **build.gradle.kts (Module)** y habilita el **view binding**
- Abre **MainActivity** y crea una variable de instancia llamada **binding**, que será inicializada con el método **ActivityMainBinding.inflate** en haz un método auxiliar llamado **inicializarViewBinding**. Dicho método se llamará en **onCreate**

```

1. class MainActivity : AppCompatActivity() {
2.     private lateinit var binding:ActivityMainBinding
3.     override fun onCreate(savedInstanceState: Bundle?) {
4.         super.onCreate(savedInstanceState)
5.         inicializarViewBinding()
6.     }
7.     private fun inicializarViewBinding(){
8.         binding=ActivityMainBinding.inflate(layoutInflater)
9.         setContentView(binding.root)
10.    }
11. }

```

- A continuación, vamos a añadir un método llamado **inicializarEventos**, que también será llamado en **onCreate**. En él, llamaremos al método **setOnClickListener** de los botones. Para que el código fuente sea pequeño, crearemos métodos auxiliares **iniciar()**, **detener()**, **reiniciar()** y **pausa()** que serán llamados en los **setOnClickListener** respectivos.

```

1. override fun onCreate(savedInstanceState: Bundle?) {
2.     super.onCreate(savedInstanceState)
3.     inicializarViewBinding()
4.     inicializarEventos()
5. }
6. private fun inicializarEventos(){
7.     binding.btnStart.setOnClickListener { iniciar() }
8.     binding.btnStop.setOnClickListener { detener() }
9.     binding.btnPausa.setOnClickListener { pausa() }
10.    binding.btnReiniciar.setOnClickListener { reiniciar() }
11. }
12. private fun iniciar(){
13. }
14. private fun detener(){
15. }
16. private fun pausa(){
17. }
18. private fun reiniciar(){
19. }
20.

```

- También vamos a añadir un método auxiliar para habilitar o deshabilitar los botones. Teniendo en cuenta que cuando se habilita **btnStart** se deshabilitan los otros y viceversa, vamos a hacer un método llamado **habilitarStart**, que recibirá un **boolean**, y según dicho valor, el botón **btnStart** se habilitará o no, y los demás botones adoptarán el estado contrario al de **btnStart**.

```

1. private fun habilitarStart(b:Boolean){
2.     binding.btnStart.isEnabled=b
3.     binding.btnStop.isEnabled=!b
4.     binding.btnPausa.isEnabled=!b
5.     binding.btnReiniciar.isEnabled=!b
6. }

```

Todos los componentes tienen una propiedad llamada **isEnabled** que permite habilitar o deshabilitar el componente. Cuando un componente está deshabilitado, no emite eventos cuando se pulsa sobre él.

### 3 – Control del Chronometer

El componente **Chronometer** tiene los siguientes métodos:

- **start:** Hace que el cronómetro comience a funcionar, actualizándose la interfaz a cada segundo.
- **stop:** Hace que el cronómetro deje de funcionar y la interfaz no se actualizará más hasta que vuelva a llamarse al método **start**

Además de estos métodos, el **Chronometer** posee una variable de instancia llamada **base**, que es el instante de tiempo que se toma como 0:00 a la hora de mostrar el tiempo. Es posible obtener el instante de tiempo actual, usando el método **SystemClock.elapsedRealTime()**

- Crea una **enum** llamada **Situacion** que tenga las constantes: **DETENIDO**, **INICIADO** y **PAUSA**, que son las diferentes situaciones en las que puede estar el cronómetro de nuestra app.

```
1. enum class Situacion {  
2.     DETENIDO, INICIADO, PAUSA  
3. }
```

- Añade a **MainActivity** estas variables de instancia:

| Variable de instancia | Tipo de dato | Significado  |
|-----------------------|--------------|--|
| situacion             | Situacion    | Variable que guarda cómo se encuentra el cronómetro de la app. Por defecto, está detenido. |
| Base                  | Long         | Instante de tiempo a partir del cual el cronómetro empieza a contar segundos desde 0:00    |

```
1. class MainActivity : AppCompatActivity() {  
2.     private lateinit var binding:ActivityMainBinding  
3.     private var situacion:Situacion=Situacion.DETENIDO  
4.     private var base:Long= 0L  
5.     // resto omitido  
6. }
```

- A la hora de programar los métodos **iniciar**, **detener**, **pausa** y **reset**, es bueno añadir a **MainActivity** este método auxiliar, que simplificará el trabajo:
  - o **reset()**: Hace que el cronómetro **chrRelej** se ponga a 0:00

```

1. private fun reset(){
2.     base = SystemClock.elapsedRealtime()
3.     chrReloj.base=base
4. }

```

☞ ¿Por qué usamos la variable **base** y no guardamos el valor directamente **chrReloj.base**? Al producirse cambios estructurales la **Activity** se borra, con lo que **chrReloj** se elimina de la memoria con su correspondiente base, y no habría ninguna forma de acceder a **chrReloj.base** tras el reinicio. La variable de instancia **base** también se borra, pero veremos que es posible “salvarla” de ser borrada.

- Programa el método **iniciar** haciendo en él estas cosas:
  - o Deshabilitaremos **btnStart** (y se habilitarán los otros) usando el método auxiliar **habilitarStart** que hemos hecho y pondremos a **true** la variable de instancia **iniciado**.
  - o Haremos que el **Chronometer** comience a funcionar, tomando como base (momento 0:00) el instante de tiempo en el que el botón es pulsado. Para ello, llamaremos al método auxiliar **reset()**
  - o Hacemos que la variable de instancia **situacion** sea la constante **Situacion.INICIADO**

```

1. private fun iniciar(){
2.     habilitarStart(false)
3.     reset()
4.     binding.chrReloj.start()
5.     situacion=Situacion.INICIADO
6. }

```

- Pon en marcha la app y comprueba que el botón **btnStart** funciona correctamente
- Programa ahora el método **reiniciar()**, de forma que la base del **Chronometer** sea el instante en que se pulsa dicho botón. Bastará con llamar al método **reset()** para que eso suceda.

```

1. fun reiniciar() {
2.     reset()
3. }

```

- Pon en marcha la app y comprueba que los botones **btnStart** y **btnReiniciar** funcionan correctamente.
- Programa ahora el método **detener()**, de forma que se detenga el cronómetro, y se vuelva a habilitar el botón **btnStart** y deshabilitar los otros dos. También se pondrá a **Situacion.DETENIDO** la propiedad **situación** y se llamará a **reset** (esto último es para que al pulsar **btnStop** el tiempo se muestre a 0:00)

```

1. private fun detener(){
2.     habilitarStart(true)
3.     binding.chrReloj.stop()
4.     reset()
5.     situacion=Situacion.DETENIDO
6. }

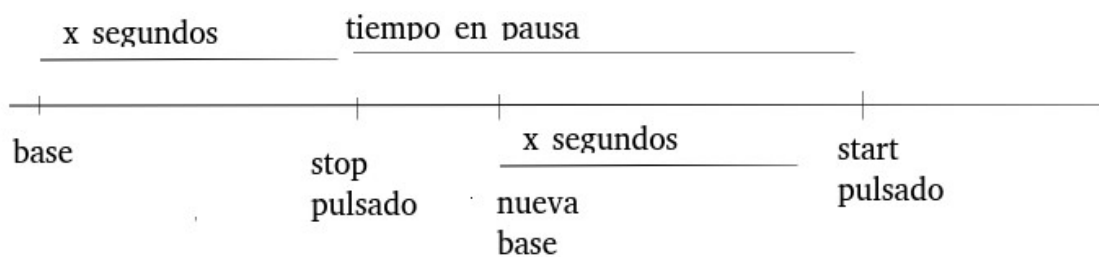
```

- Programa ahora el método **pausar()**, de forma que se detenga el cronómetro, y se vuelva a habilitar el botón **btnStart** y deshabilitar los otros dos. También se pondrá a **Situacion.PAUSA** la propiedad **situacion**.

```
1. private fun pausar(){
2.     habilitarStart(true)
3.     binding.chrReloj.stop()
4.     situacion=Situacion.PAUSA
5. }
```

- Pon en marcha la app y comprueba que el botón **btnStop** detiene el reloj, pero si volvemos a pulsar **btnStart**, el reloj se inicia desde 0:00.

☞ ¿Por qué pasa esto? En el método **setOnClickListener** de **btnStart** se define como base el momento en que pulsamos **btnStart**, pero esto hace que siempre que se pulse **btnStart** el tiempo empiece en 0:00. Lo que buscamos es que el cronómetro continúe por donde estaba la siguiente vez que se pulse **btnStart**. La situación está descrita en este dibujo:



Si pasaron X segundos desde que se pulsa **btnStart** hasta que se pulsa **btnStop**, queremos que cuando se pulse otra vez **btnStart** el cronómetro continúe por X. Para ello, la base (que es el instante que se toma como 0:00), debe ser el instante donde se vuelve a pulsar **btnStart** menos los X segundos que ya habían transcurrido. De esa forma, al volver a pulsarse **btnStart**, se considerarán transcurridos X segundos y el cronómetro continuará por donde se quedó.

- Para solucionar el problema, vamos a introducir una variable de instancia en la clase que será la cantidad de tiempo transcurrido entre que se pulsa **btnStart** y **btnStop**.

```
1. class MainActivity : AppCompatActivity() {
2.     private lateinit var binding: MainActivityBinding
3.     private var situacion:Situacion = Situacion.DETENIDO
4.     private var base:Long= 0L
5.     private var tiempoTranscurrido:Long = 0L
6.     // resto omitido
7. }
```

- Vamos a añadir dos métodos auxiliares que nos ayuden a programar correctamente la pausa del reloj, teniendo en cuenta el gráfico anterior.
  - **actualizarTiempoTranscurrido**: actualiza el valor de la variable **tiempoTranscurrido**, calculando el instante actual y restándole la base del cronómetro, tal y como se explica en el dibujo anterior (o sea, este método calcula la X del dibujo).
  - **resetTiempoTranscurrido**: Hace que el cronómetro marque la cantidad de segundos que indica la variable **tiempoTranscurrido**. Según la explicación anterior, la nueva base del cronómetro será el instante actual menos la variable **tiempoTranscurrido**

```

1. private fun actualizarTiempoTranscurrido(){
2.     tiempoTranscurrido = SystemClock.elapsedRealtime() - base
3. }
4. private fun resetTiempoTranscurrido(){
5.     base = SystemClock.elapsedRealtime()-tiempoTranscurrido
6.     chrReloj.base=base
7. }

```

- Ayudándonos de la nueva variable **tiempoTranscurrido** y el método **resetTiempoTranscurrido**, podemos actualizar los métodos **pausar()** e **iniciar()** de esta forma:
  - En **pausar()**, asignaremos a **tiempoTranscurrido** la cantidad de tiempo X que se ve en el dibujo de la página anterior. Dicha cantidad se calcula como el momento en que se pulsa el botón menos la base del cronómetro.
  - En **iniciar()**, ahora mismo estamos tomando como base (momento 0:00) el instante en que se pulsa **btnStart**. Pero si **btnStart** se pulsa cuando la situación es **PAUSA**, hay que llamar a **resetTiempoTranscurrido**, porque queremos que el reloj empiece a contar a partir del valor de **tiempoTranscurrido**.

```

1. class MainActivity : AppCompatActivity() {
2.     // resto omitido
3.     private fun iniciar(){
4.         habilitarStart(false)
5.         if(situacion==Situacion.DETENIDO){
6.             reset()
7.         }else if(situacion == Situacion.PAUSA){
8.             resetTiempoTranscurrido()
9.         }
10.        binding.chrReloj.start()
11.        situacion=Situacion.INICIADO
12.    }
13.    private fun pausar(){
14.        habilitarStart(true)
15.        binding.chrReloj.stop()
16.        resetTiempoTranscurrido()
17.        situacion = Situacion.PAUSA
18.    }
19. }

```

- Ejecuta la app y comprueba que los botones parecen funcionar correctamente.



## 4 – Cambios estructurales

Un **cambio estructural** se produce cuando el usuario hace un cambio en la configuración que invalida drásticamente las condiciones de ejecución de la aplicación (por ejemplo, girar el móvil, cambiar de lenguaje, conectar un teclado físico, etc son cambios estructurales)

Cuando se produce un cambio estructural, la **MainActivity** es eliminada, y creada nuevamente desde cero (o sea, es **recreada**) aplicando los nuevos archivos de layout, idiomas, etc

Al ser eliminada, todas las variables de instancia de **MainActivity** (que están en la memoria) se pierden, y es responsabilidad del programador hacer algo para que el estado de la app se conserve tras el reinicio, sin que el usuario se percate de ello.

- Gira el móvil mientras el cronómetro está funcionando y verás que la app se resetea y todo vuelve a su estado inicial.

## 5 – Introducción al View Model

Para evitar que los datos que están en la memoria se pierdan cuando una **Activity** es recreada debido a un cambio estructural, la técnica actual consiste en utilizar un **view model**.

Un **view model** es una clase que programamos nosotros a la que añadimos como variables de instancia las variables del **Activity** que no queramos que se pierdan ante un cambio estructural (o sea, las variables que forman el **estado** de la interfaz), y también los métodos que procesan dichas variables.

El **view model** tiene la característica de que no se pierde cuando se produce un cambio estructural, y puede ser recuperado tal como estaba antes del cambio.

- Abre el archivo **build.gradle.ktx (Module)** y añade la siguiente línea en la sección de dependencias, sincronizando después el proyecto:

```
implementation("androidx.lifecycle:lifecycle-viewmodel-ktx:2.8.6")
```

- Elimina de **MainActivity** las variables de instancia **situación**, **base** y **tiempoTranscurrido**

```
1. class MainActivity: AppCompatActivity() {  
2.     var situacion:Situacion = Situacion.DETENIDO  
3.     var base:Long=0L  
4.     var tiempoTranscurrido:Long = 0L  
5.     // resto omitido  
6. }
```

- Crea una clase llamada **MainActivityViewModel** y haz que herede de la clase **ViewModel**, así:

```
1. class MainActivityViewModel : ViewModel() {
2. }
```

Para saber qué variables de instancia ponemos a nuestro **view model**, es necesario descubrir las variables que definen el **estado** de **MainActivity** y que permiten reestablecerlo. Lo normal es que sean las variables de instancia que no sean objetos de la interfaz.

- Añade a **MainActivityViewModel** las variables de instancia **situacion**, **base** y **tiempoTranscurrido**

```
1. class MainActivityViewModel : ViewModel() {
2.     var situacion:Situacion = Situacion.DETENIDO
3.     var base:Long=0L
4.     var tiempoTranscurrido:Long = 0L
5. }
```

- Vamos a mover al **MainActivityViewModel** los métodos de **MainActivity** que tienen que ver con las variables de instancia **situacion**, **base** y **tiempoTranscurrido**. Los adaptaremos de esta forma:
  - **marcarTiempoTranscurrido**: Se queda igual, ya que en su interior, solo aparecen un método del sistema y la variable de instancia **base**
  - **reset** y **resetTiempoTranscurrido**: Como el cronómetro es un objeto de la interfaz, y por tanto, se queda en **MainActivity**, es necesario que dichos métodos (que requieren el cronómetro) reciban un cronómetro como parámetro.
  - Los métodos ahora deben ser públicos (en Kotlin es el comportamiento por defecto cuando no se pone ningún modificador)

```
1. class MainActivityViewModel : ViewModel() {
2.     var situacion:Situacion = Situacion.DETENIDO
3.     var base:Long = 0L
4.     var tiempoTranscurrido:Long = 0L
5.     fun reset(c: Chronometer){
6.         base = SystemClock.elapsedRealtime()
7.         c.base=base
8.     }
9.     fun actualizarTiempoTranscurrido(){
10.         tiempoTranscurrido = SystemClock.elapsedRealtime() - base
11.     }
12.     fun resetTiempoTranscurrido(c: Chronometer){
13.         base = SystemClock.elapsedRealtime()-tiempoTranscurrido
14.         c.base=base
15.     }
16. }
```

- Vete a **MainActivity** y crea una variable de instancia llamada **viewModel**, de tipo **MainActivityViewModel**, que será el **view model** usado por **MainActivity**

```

1. class MainActivity : AppCompatActivity() {
2.     private lateinit var viewModel: MainActivityViewModel
3.     // resto omitido
4. }

```

- Haz un método auxiliar llamado **inicializarViewModel**, inicialice la propiedad **viewModel** llamando al método **get** de la clase **ViewModelProvider**. Dicho método será llamado en **onCreate**

```

1. override fun onCreate(savedInstanceState: Bundle?) {
2.     super.onCreate(savedInstanceState)
3.     inicializarViewModel()
4.     inicializarViewBinding()
5.     inicializarEventos()
6. }
7. fun inicializarViewModel(){
8.     viewModel = ViewModelProvider(this).get(MainActivityViewModel::class.java)
9. }

```

La clase **ViewModelProvider** se encarga de gestionar todos los **view model** de la app, creándolos la primera vez que son requeridos, y devolviendo los que ya había en caso de que se produzca un cambio estructural.

- Usando **ViewModelProvider(this)** obtenemos un **ViewModelProvider** que gestiona los **view model** de **MainActivity**
- Su método **get** nos permite obtener un **view model** del tipo pasado como parámetro (la clase **MainActivityViewModel** se pasa como parámetro escribiendo **MainActivityViewModel::class.java**). Dicho **view model** se crea la primera vez que se llama a **get**, y es devuelto en las siguientes llamadas.

- Programa el método **detener** haciendo que se acceda al **view model** para acceder a la variable **situacion** y también para poner a 0 el cronómetro

```

1. private fun detener(){
2.     habilitarStart(true)
3.     binding.chrReloj.stop()
4.     viewModel.reset(binding.chrReloj)
5.     viewModel.situacion=Situacion.DETENIDO
6. }

```

- Programa el método **reiniciar** haciendo también que se acceda al **view model** para llamar al método **reset**

```

1. private fun reiniciar(){
2.     viewModel.reset(binding.chrReloj)
3. }

```

- Programa el método **pausar** para que también haga uso del **view model**

```

1. private fun pausar(){
2.     habilitarStart(true)
3.     binding.chrReloj.stop()
4.     viewModel.actualizarTiempoTranscurrido()
5.     viewModel.situacion=Situacion.PAUSA
6. }

```

- Por último, programa el método **iniciar** para que haga uso del **view model**

```

1. private fun iniciar(){
2.     habilitarStart(false)
3.     if(viewModel.situacion==Situacion.DETENIDO){
4.         viewModel.reset(binding.chrReloj)
5.     }else if(viewModel.situacion==Situacion.PAUSA){
6.         viewModel.resetTiempoTranscurrido(binding.chrReloj)
7.     }
8.     binding.chrReloj.start()
9.     viewModel.situacion=Situacion.INICIADO
10. }

```

- Ejecuta la app, veremos que al girar el móvil el estado sigue sin actualizarse.

Aunque el **view model** se conserva, puede pasarnos (como ocurre en nuestra app) que nos falte actualizar la interfaz con los datos del **view model** para que la apariencia de la app siga siendo la previa al cambio estructural.

- En **MainActivity** sobrescribe el método **onStart**, de forma que en dicho método se mire la variable **situacion** y según la misma se haga esto:
  - Si está detenido, simplemente se habilita **btnStart** y se deshabilitan los demás
  - Si está iniciado, se deshabilita **btnStart**, se pone como base la que tuviese el **view model**, y se inicia el cronómetro
  - Si está en pausa, se deshabilita **btnStart** y se toma como base el instante actual menos el tiempo transcurrido (de esa forma, el reloj mostrará en pantalla el tiempo transcurrido)

```

1. override fun onStart() {
2.     super.onStart()
3.     when(viewModel.situacion){
4.         Situacion.DETENIDO -> {
5.             habilitarStart(true)
6.         }
7.         Situacion.INICIADO -> {
8.             habilitarStart(false)
9.             binding.chrReloj.base=viewModel.base
10.            binding.chrReloj.start()
11.        }
12.        Situacion.PAUSA -> {
13.            habilitarStart(true)
14.            binding.chrReloj.base = SystemClock.elapsedRealtime() - viewModel.tiempoTranscurrido
15.        }
16.    }
17. }

```

En el método **onStart** se realizan las tareas que conducen a establecer el estado de la app

Dicho método es llamado cuando la **MainActivity** va a hacerse visible (bien porque ha sido creada, o porque ha sido minimizada y posteriormente restaurada)

- Ejecuta la app y comprueba que al girar el móvil todo funciona correctamente

## **6 – Respuesta a la puesta de la app en segundo plano**

Además de los cambios estructurales, otra situación que puede producirse es que nuestra app sea puesta en segundo plano. Esto ocurre cuando el usuario abre otra app, que desplaza a la nuestra de la pantalla.

En esta situación, **MainActivity** no es borrada de la memoria y es conservada. No obstante, si el sistema detecta que queda poca memoria libre, puede borrar **MainActivity** si lo considera necesario.

Si nos interesa programar acciones cuando nuestra app vaya a ser puesta en segundo plano y posteriormente, cuando sea restaurada, sobreescribiremos estos métodos:

- **onStop:** Este método es llamado cuando nuestra app ya no está en primer plano. Aquí haremos las acciones necesarias para gestionar la minimización de la app.
- **onRestart:** Este método es llamado cuando nuestra app va a ser puesta en primer plano. En él realizaremos las gestiones necesarias para restaurar la app tras haber sido puesta en segundo plano.

También en esta situación hay que tener en cuenta al método **onStart**, porque es llamado siempre que la **Activity** se va a hacer visible para establecer su estado. Tras ser llamado **onRestart** el sistema Android llama a **onStart**

- Pon la app en marcha y abre otra app (por ejemplo, el navegador de Internet), de forma que el cronómetro quede en segundo plano. Deja que pasen unos segundos y restaura el cronómetro. Comprueba que parece que ha seguido funcionando y que se muestra el tiempo total, como si hubiese estado funcionando todo el rato.

Cuando una app pasa a segundo plano **MainActivity** no se destruye, y las variables de instancia y el estado de los botones se conservan en la memoria.

En nuestro caso, la **base** del **Chronometer** sigue siendo la previa a la minimización, y por ese motivo, al restaurarlo coge esa base y muestra el tiempo como si no se hubiese producido minimización nunca.

- Vamos a hacer que el cronómetro se detenga si se produce una minimización. Para ello, sobreescribe **onStop** para que, en caso de que el reloj esté iniciado, lo detenga y guarde el valor actual de los segundos en **tiempoTranscurrido**, como si se hubiera hecho una pausa.

```

1. override fun onStop() {
2.     super.onStop()
3.     if(viewModel.situacion==Situacion.INICIADO){
4.         viewModel.actualizarTiempoTranscurrido()
5.         binding.chrReloj.stop()
6.     }
7. }

```

- Para hacer que al restaurar la app el cronómetro vuelva a funcionar, vamos a sobrescribir el método **onRestart** haciendo que, en caso de que el reloj estuviese funcionando, el cronómetro empiece a contar a partir de **tiempoTranscurrido** y se ponga en marcha.

```

1. override fun onRestart() {
2.     super.onRestart()
3.     if(viewModel.situacion==Situacion.INICIADO){
4.         viewModel.resetTiempoTranscurrido(binding.chrReloj)
5.         binding.chrReloj.start()
6.     }
7. }

```

- Ejecuta la app y pon a funcionar el cronómetro. Comprueba que si abres otra app y la dejas en primer plano unos segundos, al volver al cronómetro, este continúa por donde se quedó.

## **6 – Respuesta a la minimización parcial de la app**

También puede suceder que nuestra app deje de estar en primer plano, pero no sea completamente minimizada, sino que se quede parcialmente visible. Es lo que sucede si por ejemplo, abrimos el asistente de Google que aparece en la parte inferior de la pantalla.

Para responder a esta situación tenemos estos métodos que podemos sobrescribir:

- **onPause:** Se llama cuando la app deja de estar en primer plano para hacerse parcialmente visible
- **onResume:** Se llama cuando la app vuelve a estar en primer plano y el usuario puede interactuar con ella.

En esta situación, el sistema no llama a **onStart** tras **onResume**, ya que la app está parcialmente visible.

- Sobreescribe los métodos **onPause** y **onResume** de forma que hagan lo mismo que **onStop** y **onRestart**

```

1. override fun onPause() {
2.     super.onPause()
3.     if(viewModel.situacion==Situacion.INICIADO){
4.         viewModel.actualizarTiempoTranscurrido()
5.         binding.chrReloj.stop()
6.     }
7. }

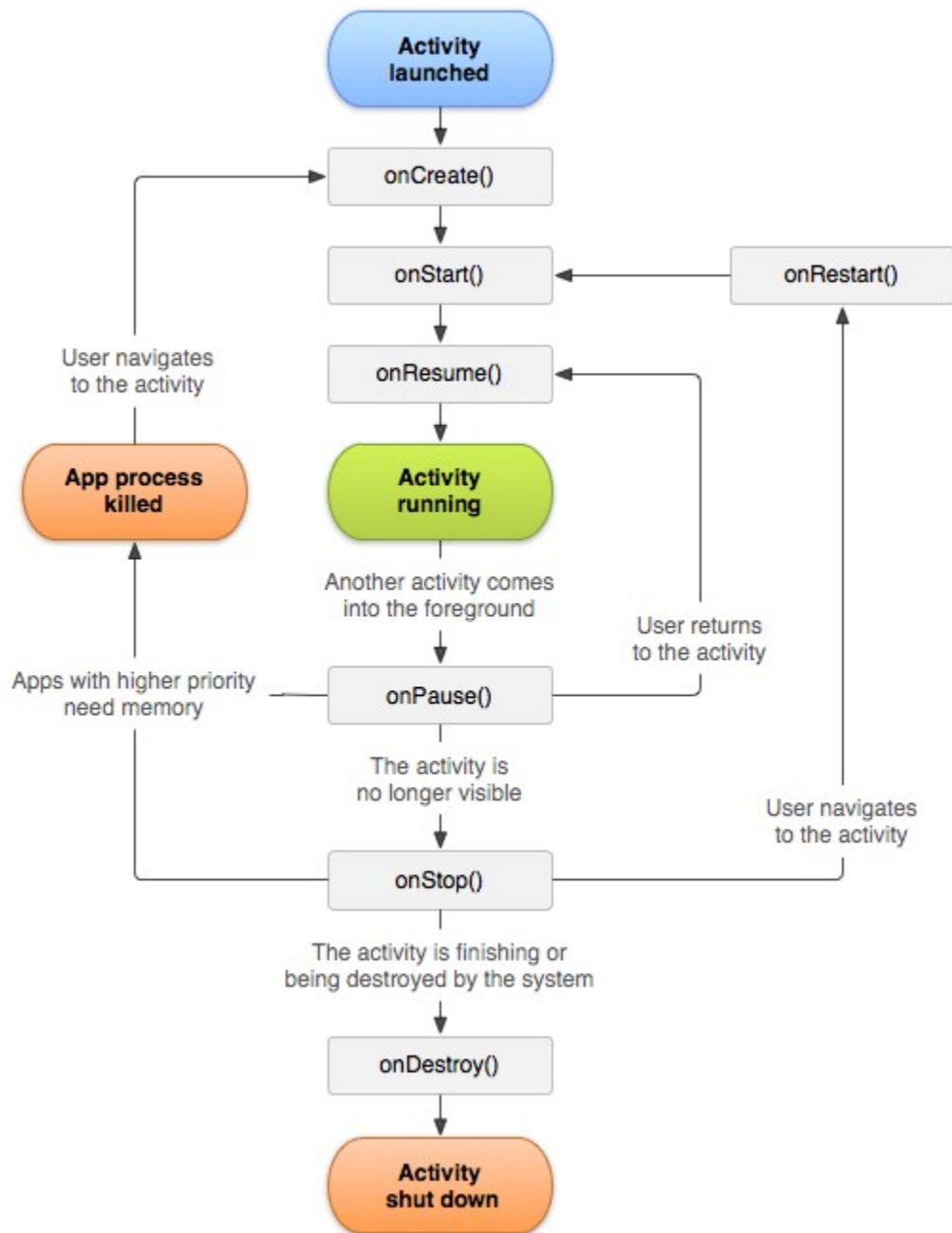
```

```
8. override fun onResume() {  
9.     super.onResume()  
10.    if(viewModel.situacion==Situacion.INICIADO){  
11.        viewModel.resetTiempoTranscurrido(binding.chrReloj)  
12.        binding.chrReloj.start()  
13.    }  
14. }
```

- Abre la app (si tu emulador no posee el asistente de Google, usa un dispositivo físico conectado al PC) y abre el asistente de Google. Comprueba que el cronómetro se hace parcialmente visible y que permanece detenido hasta que vuelve a estar en primer plano.

## **7 – El ciclo de vida de las Activity**

A lo largo de este proyecto hemos visto un montón de métodos como **onCreate**, **onStart**, **onStop**, etc. Estos métodos son llamados por el sistema Android según lo que le pasa a la Activity. La relación que hay entre ellos es el **ciclo de vida de la Activity**, que se muestra en el siguiente gráfico:



- Abre el código fuente de **MainActivity** y comprueba que los métodos **onStop** y **onPause** hacen exactamente lo mismo. Comprueba que le pasa lo mismo a **onRestart** y **onResume**.
- Observa que en el ciclo de vida después de **onStart** se llama a **onResume** y que después de **onPause** se llama a **onStop**.
- Borra los métodos **onStop** y **onRestart** y deja solamente **onPause** y **onResume**

Puesto que después de **onPause** se llama a **onStop**, si el código de ambos está repetido, podemos dejar solamente el **onPause**. Lo mismo sucede con los otros.



- Comprueba que la app sigue funcionando correctamente

Android nos proporciona los métodos del ciclo de vida para que programemos en ellos lo que sucede cuando cambia el estado de la Activity. Muchas veces podemos elegir el método que queramos entre varios que parece que hacen lo mismo y no habrá ningún problema. La diferencia entre unos y otros en muchos casos son muy sutiles y se deben a si la Activity está dibujada en la pantalla o no, o si está en memoria o no.

## **8 – Tests de instrumentación**

Siempre que se hacen programas son muy importantes los tests, para comprobar que todo funciona correctamente. Actualmente se busca que los tests se hagan de forma automática, como por ejemplo en los **tests unitarios** de las clases con **JUnit**.

En las apps de Android existen los **tests de instrumentación**, que son pruebas automatizadas que se hacen sobre la interfaz de usuario una vez que la app se está ejecutando en el emulador.

Usaremos la librería **Espresso** (incluida por defecto en el proyecto) para hacer tests de instrumentación de nuestra app.



- Abre la carpeta **kotlin+java** y verás que uno de los paquetes de código fuente aparece con la palabra **androidTest**.
- Abre dicho paquete y comprueba que hay un archivo llamado **ExampleInstrumentedTest** con un código similar a este:

```

16  @RunWith(AndroidJUnit4::class)
17  class ExampleInstrumentedTest {
18      @Test
19      fun useAppContext() {
20          // Context of the app under test.
21          val appContext = InstrumentationRegistry.getInstrumentation().targetContext
22          assertEquals("expected: "com.hfad.cronometro", appContext.packageName)
23      }
24  }

```

El archivo **ExampleInstrumentedTest** contiene tests de instrumentación. Cada uno de sus métodos anotado con **@Test** es un test.

Podemos ejecutar individualmente los tests con el botón  que acompaña a cada método, o todos a la vez con el botón .

- Pulsa el botón para ejecutar el test de ejemplo llamado **useAppContext** y comprueba que tras unos momentos, el test se ejecuta y aparece que ha sido superado correctamente.

| Tests                     | Duration | Pixel_3_API_30 |
|---------------------------|----------|----------------|
| ✓ Test Results            | 29 ms    | 1/1            |
| ✓ ExampleInstrumentedTest | 29 ms    | 1/1            |
| ✓ useAppContext           | 29 ms    | ✓              |

- Vamos a crear nuestro propio test de instrumentación para ver si **MainActivity** funciona correctamente de forma automatizada (hasta ahora lo hemos probado todo de forma manual). Para ello, abre **MainActivity.kt**, pulsa el botón derecho del ratón y después **Generate → Test**
- Rellena la ventana que aparece con estos datos:
  - Testing library: **JUnit4** (si sale una advertencia diciendo que no se encuentra, pulsa el botón **Fix** para descargarla)
  - Marca las opciones: **setUp/@Before** y **tearDown/@After**
- Al pulsar **OK** saldrá el lugar donde quieres guardar el test. Como se trata de un test de instrumentación, elige la carpeta **androidTests** y pulsa finalmente **OK**



- Observa que se genera un código fuente similar a este

```

1. @RunWith(AndroidJUnit4::class) // añade esta línea si Android Studio no te la ha puesto
2. class MainActivityTest {
3.     @Before
4.     fun setUp() {
5.     }
6.     @After
7.     fun tearDown() {
8.     }
9. }

```

- El método **setUp** sirve para definir código que se ejecutará antes de cada test que añadamos al archivo
- El método **tearDown** sirve para definir código que se ejecutará después de cada test que añadamos al archivo

- Añade una variable de instancia llamada **escenario** cuyo tipo será **ActivityScenario<MainActivity>**. Esta variable se inicializará dentro del método **setUp** con el método **launch**, al que pasaremos el objeto **MainActivity::class.java**, que representa la clase **MainActivity**. También cerraremos dicho objeto en el método **tearDown**.

```

1. @RunWith(AndroidJUnit4::class)
2. class MainActivityTest {
3.     private lateinit var escenario:ActivityScenario<MainActivity>
4.     @Before
5.     fun setUp() {
6.         escenario=launch(MainActivity::class.java)
7.     }
8.     @After
9.     fun tearDown() {
10.    }
11. }

```

La clase **ActivityScenario<MainActivity>** nos permite abrir y controlar de forma automatizada una pantalla de la clase **MainActivity**

- Su método **launch** abre una ventana de tipo **MainActivity** que podremos controlar y examinar mediante código fuente
- Su método **close** cierra dicha ventana y libera los recursos asociados.

- Vamos a añadir un test que compruebe que el cronómetro pone 00:00

```

1. @Test
2. fun comprobarEstadoInicialInterfaz(){
3.     onView(withId(R.id.chrReloj)) // selecciona el componente chrReloj
4.     .check(matches(withText("00:00"))) // comprueba que su texto es 00:00
5.
6. }

```

La función **onView** nos devuelve un componente de la interfaz que se está ejecutando en el emulador, y encadenando la llamada al método **check** podemos hacer una comprobación sobre el mismo.

- Completa el test anterior para comprobar que **btnStart** está habilitado y los demás deshabilitados

```

1. @Test
2. fun comprobarEstadoInicialInterfaz(){
3.     onView(withId(R.id.chrReloj))
4.     .check(matches(withText("00:00")))
5.     onView(withId(R.id.btnStart))
6.     .check(matches(isEnabled()))
7.     onView(withId(R.id.btnStop))
8.     .check(matches(isNotEnabled()))
9.     onView(withId(R.id.btnReset))
10.    .check(matches(isNotEnabled()))
11. }

```

- Ejecuta el test (con su método play) y comprueba que se abre el emulador con una **MainActivity** y que tras un breve momento, sale el resultado del test. *Puedes introducir algún fallo (por ejemplo, cambiar **btnStop** para que esté habilitado para comprobar que el test da fallo si no se cumple lo que está programado en él*
- Vamos a añadir un segundo test que haga click en el botón **btnStart** y compruebe que **btnStart** se desactiva y los demás se activan

```

1. @Test
2. fun comprobarPulsacionStart(){
3.     onView(withId(R.id.btnStart))
4.         .perform(click())
5.     onView(withId(R.id.btnStart))
6.         .check(matches(isNotEnabled()))
7.     onView(withId(R.id.btnStop))
8.         .check(matches(isEnabled()))
9.     onView(withId(R.id.btnReset))
10.        .check(matches(isEnabled()))
11. }

```

- Vamos a añadir un tercer test que pulse el botón **btnStart**, deje pasar 5 segundos, y después compruebe que el cronómetro marca 00:05

```

1. @Test
2. fun comprobarTexto10seg(){
3.     onView(withId(R.id.btnStart))
4.         .perform(click())
5.     Thread.sleep(5000)
6.     onView(withId(R.id.chrReloj))
7.         .check(matches(withText("00:05")))
8. }

```

- Añade un cuarto test que pulse el botón **btnStart**, deje pasar 5 segundos, pulse el botón **btnStop**, deje pasar otros 5 segundos, vuelva a pulsar **btnStart** y deje pasar 2 segundos más. El test comprobará que el cronómetro pone 00:07

```

1. @Test
2. fun comprobarStop(){
3.     onView(withId(R.id.btnStart))
4.         .perform(click())
5.     Thread.sleep(5000)
6.     onView(withId(R.id.btnStop))
7.         .perform(click())
8.     Thread.sleep(5000)
9.     onView(withId(R.id.btnStart))
10.        .perform(click())
11.     Thread.sleep(2000)
12.     onView(withId(R.id.chrReloj))
13.         .check(matches(withText("00:07")))
14. }

```

- Por último, añadimos un test que pulse **btnStart**, deje pasar 5 segundos, simule un cambio estructural y tras recrearse la app, el cronómetro siga poniendo 5 segundos

```
1. @Test
2. fun comprobarRecreacion(){
3.     onView(withId(R.id.btnStart))
4.         .perform(click())
5.     Thread.sleep(5000)
6.     escenario.recreate()
7.     onView(withId(R.id.chrReloj))
8.         .check(matches(withText("00:05")))
9. }
```

## 9 – Ejercicios

Ejercicio 1: Realiza una app que elija un color aleatorio entre estas opciones: rojo, amarillo, verde, azul, negro, naranja. Se mostrará en el centro de la app el mensaje “Este texto está escrito en color (*poner aquí el nombre del color*)” usando el color elegido aleatoriamente.

Hacer que la app conserve el mensaje y su color correctamente a los cambios estructurales.

Ejercicio 2: Realiza una app que elija un número aleatorio entre 0 y 10 y el usuario tiene 4 intentos para acertarlo. Habrá un **EditText** donde el usuario escribirá un número, un **Button** para jugar y un **TextView** para mostrar el número de vidas. En caso de que el número que escribe el usuario sea menor, se mostrará un **Toast** indicando que el número introducido es menor que el número secreto, y en caso de que sea mayor, se indicará también en un **Toast**. Si el usuario acierta el número, se mostrará un **Toast** indicando que ha ganado, y si se acaban las vidas, se mostrará otro para decir que ha perdido (mostrándose el número secreto). La app responderá correctamente ante los cambios estructurales.

Ejercicio 3: Realiza la app descrita en los siguientes apartados

- a) Haz una app cuya interfaz contenga estos elementos
- Un **EditText** que permita introducir una cantidad de segundos
  - Un **Chronometer** que inicialmente estará invisible
  - Un **Button** que al ser pulsado hará que el cronómetro se haga visible (y el **EditText** y el **Button** se deshabiliten) y comience a contar hacia atrás la cantidad de segundos indicada. Al llegar el cronómetro a 0, se mostrará un **Toast** con el mensaje “Cuenta atrás finalizada” y se volverá a habilitar el **EditText** y el **Button** y el cronómetro se hará invisible.

*Pista: Consulta en Internet la documentación de la clase Chronometer para conseguir que el cronómetro cuente hacia atrás y también para saber cuándo llega a 0.*

- b) Haz que cuando se produzca un cambio estructural, el cronómetro siga funcionando correctamente
- c) Haz que si la app pasa a segundo plano, el cronómetro continúe por donde se quedó cuando la app se restaure.

Ejercicio 4: Arregla la app InformeMatrículas para que funcione correctamente cuando haya cambios estructurales.