



Laboratorio

El Ciclo de Vida de una Activity

Versión: 1.0.0
Junio de 2017



[Miguel Muñoz Serafín](#)
@msmdotnet





CONTENIDO

INTRODUCCIÓN

EJERCICIO 1: GUARDANDO EL ESTADO DE LA ACTIVITY

Tarea 1. Crear un proyecto Android.

Tarea 2. Examinar los métodos del ciclo de vida de la Activity.

Tarea 3. Examinar las transiciones de estado.

Tarea 4. Agregar un contador de Clics.

Tarea 5. Manejar el estado utilizando un objeto Bundle.

Tarea 6. Persistir datos complejos.

EJERCICIO 2: VALIDANDO TU ACTIVIDAD

Tarea 1. Agregar los componentes de la Capa de acceso a Servicio.

Tarea 2. Agregar la funcionalidad para validar la actividad.

RESUMEN



Introducción

Las *Activities* son el bloque de construcción fundamental de las aplicaciones Android y pueden existir en varios estados diferentes. El ciclo de vida de una *Activity* comienza con la instanciación y termina con la destrucción incluyendo varios estados intermedios. Cuando una *Activity* cambia de estado se genera un evento del ciclo de vida de la *Activity*. El método que maneja ese evento es invocado notificando a la *Activity* del cambio de estado inminente y permitiéndole ejecutar código para adaptarse a ese cambio.

En este laboratorio examinaremos el ciclo de vida de las *Activities* y explicaremos la responsabilidad que tiene una *Activity* durante cada uno de los cambios de estado para poder crear aplicaciones confiables y consientes del ambiente operativo Android.

Objetivos

Al finalizar este laboratorio, los participantes serán capaces de:

- Describir el ciclo de vida de una *Activity*.
- Describir los estados de una *Activity*.
- Describir los métodos asociados al ciclo de vida de una *Activity*.
- Conservar el estado de una *Activity*.

Requisitos

Para la realización de este laboratorio es necesario contar con lo siguiente:

- Un equipo de desarrollo con Visual Studio. Los pasos descritos en este laboratorio fueron realizados con Visual Studio 2017 y Windows 10 Professional, sin embargo, los participantes pueden utilizar la versión de Visual Studio 2015 que ya tengan instalada.
- Xamarin para Visual Studio.

Tiempo estimado para completar este laboratorio: **60 minutos**.

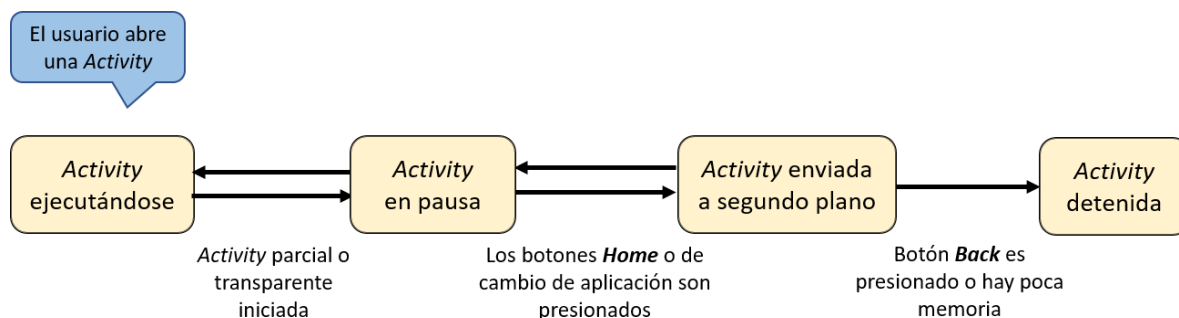


Ejercicio 1: Guardando el estado de la Activity

Las *Activities* son un concepto de programación inusual específico de Android. En el desarrollo tradicional de aplicaciones, normalmente existe un método estático *main* que es ejecutado para iniciar la aplicación. Con Android, sin embargo, las cosas son diferentes, las aplicaciones Android pueden ser lanzadas desde cualquier *Activity* registrada dentro de una aplicación. En la práctica, la mayoría de las aplicaciones solo tendrán una *Activity* particular que es especificada como el punto de entrada de la aplicación. Sin embargo, si una aplicación colapsa o es terminada por el sistema operativo, el sistema operativo puede intentar reiniciar la aplicación desde la última *Activity* abierta o desde cualquier otra que esté en la pila de *Activities* previas. Adicionalmente, el sistema operativo puede pausar *Activities* cuando no estén activas y haya poca memoria. Es importante tener cuidado de permitir que la aplicación restaure correctamente su estado en caso de que se reinicie una *Activity*, especialmente si la *Activity* depende de los datos de instancias previas.

Estados de la activity

El sistema operativo Android regula las *Activities* según su estado. Esto ayuda a Android a identificar *Activities* que ya no están en uso, permitiendo que el sistema operativo recupere memoria y recursos. El siguiente diagrama ilustra los estados que una *Activity* puede atravesar durante su vida útil.



Los estados pueden ser divididos en 4 principales grupos de la siguiente forma:

1. **Activo, En ejecución o Reanudada.** Las *Activities* se consideran *Activas, En ejecución o Reanudadas* si están en primer plano. En este estado, la *Activity* se encuentra en la parte superior de la pila de *Activities* siendo considerada como la *Activity* de mayor prioridad en Android. La *Activity* de mayor prioridad sólo será destruida por el sistema operativo en situaciones extremas, como por ejemplo si la *Activity* intenta utilizar más memoria que la



disponible en el dispositivo ya que esto podría hacer que la interfaz de usuario deje de responder.

2. **En pausa.** Se considera que la *Activity* está *en Pausa* cuando el dispositivo se va a dormir o cuando una *Activity* sigue estando visible, es decir, otra *Activity* está visible por encima de esta y esa *Activity* es parcialmente transparente o no cubre toda la pantalla. Las *Activities* en pausa continúan vivas, es decir, mantienen su estado y permanecen conectadas al administrador de ventanas. Una *Activity* en pausa es considerada como la segunda *Activity* de mayor prioridad en Android y, como tal, sólo será destruida por el sistema operativo si el hecho de eliminarla satisface los requerimientos de recursos necesarios para mantener la *Activity* activa estable y responsiva. Android puede eliminarla en situaciones donde la memoria sea extremadamente baja.
3. **Detenida o en segundo plano.** La *Activity* que está completamente opacada por otra *Activity* es considerada *Detenida* o en segundo plano. Una *Activity* detenida también permanece viva y mantiene la información de su estado, pero no está conectada al administrador de ventanas. Las *Activities* detenidas tienen la prioridad más baja y, como tal, el sistema operativo eliminará primero las *Activities* en ese estado para satisfacer las necesidades de recursos de *Activities* de mayor prioridad. La *Activity* detenida ya no está visible para el usuario y el sistema puede eliminarla cuando necesite memoria en alguna otra parte.
4. **Reiniciada.** Es posible que una *Activity* que se encuentre en un estado de pausa o detenida sea eliminada de la memoria por Android. Si el usuario navega de regreso a la *Activity*, debe ser reiniciada, restaurada a su estado previamente guardado y luego ser mostrada al usuario.

Recreación de la *Activity* en respuesta a los cambios de la configuración

Para complicar más las cosas, Android contempla otro comportamiento conocido como *Cambios de Configuración (Configuration Changes)*. Los cambios de configuración son ciclos rápidos de destrucción/recreación de la *Activity* que ocurren cuando alguna configuración del dispositivo cambia en tiempo de ejecución tal como el cambio de idioma (la *Activity* tiene que recargar los recursos alternativos), la orientación de la pantalla (la *Activity* necesita reconstruirse en modo horizontal o vertical), cuando el teclado es mostrado (y la *Activity* debe cambiar su tamaño), etc.

Los cambios de configuración siguen causando los mismos cambios de estado de la *Activity* que podrían ocurrir durante la detención y el reinicio de una *Activity*. Sin embargo, para asegurarse de que una aplicación siga siendo responsiva y se desempeñe correctamente durante los cambios de configuración, es importante que se procesen esos cambios de estado lo más rápidamente posible. Debido a esto, Android tiene una API específica que puede ser utilizada para mantener el estado durante los cambios de configuración.

El SDK de Android y, por extensión, el Framework Xamarin.Android, proporcionan un modelo potente para administrar el estado de las *Activities* dentro de una aplicación.

En este ejercicio conocerás la forma de guardar eficientemente el estado de una aplicación Android durante el ciclo de vida de la *Activity*.

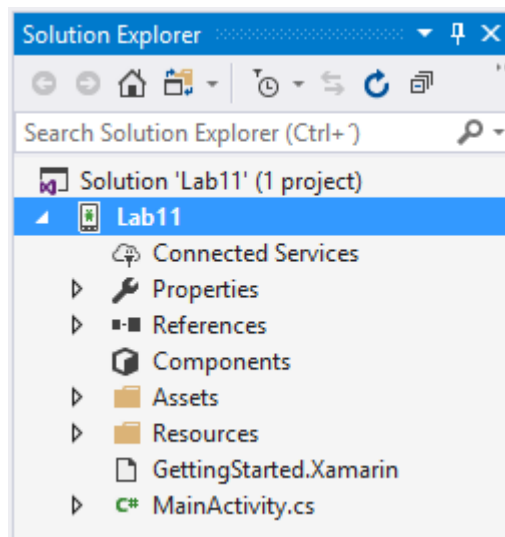


Tarea 1. Crear un proyecto Android.

En esta tarea crearás un proyecto Android que te permitirá examinar el ciclo de vida de la aplicación. La aplicación tendrá 2 pantallas. La pantalla principal tendrá un botón que permitirá mostrar la segunda pantalla. Un segundo botón permitirá llevar un contador de clics que el usuario haya realizado y finalmente un cuadro de texto donde el usuario podrá proporcionar un valor arbitrario.

1. Abre Visual Studio bajo el contexto del Administrador.
2. Utiliza la plantilla **Blank App (Android)** para crear una solución con un proyecto *Xamarin.Android* llamado **Lab11**.

El explorador de soluciones deberá mostrar algo similar a lo siguiente.



3. Utiliza la plantilla **Activity** para agregar a la aplicación una nueva *Activity* llamada *SecondActivity.cs*.
4. Modifica el archivo *Strings.xml* para definir 3 valores cadena. El contenido debe ser similar al siguiente.

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
  <string name="ApplicationName">Lab11</string>
  <string name="StartActivity_Text">Iniciar segunda Activity</string>
  <string name="ClicksCounter_Text">Contador de Clics: %1$d.</string>
</resources>
```

5. Guarda todos los cambios.
6. Abre el archivo *Main.axml* en el diseñador de Android.
7. Agrega un elemento **Button** a la superficie de diseño.
8. Asigna el valor **@+id/StartActivity** a la propiedad **id** del botón agregado.



9. Asigna el valor **@string/StartActivity_Text** a la propiedad **text** del botón agregado.
10. Agrega un segundo elemento **Button** a la superficie de diseño.
11. Asigna el valor **@+id/ClicksCounter** a la propiedad **id** del segundo botón agregado.
12. Asigna el valor **@string/ClicksCounter_Text** a la propiedad **text** del segundo botón agregado.
13. Agrega un elemento **Plain Text** a la superficie de diseño.
14. Asigna el valor **@+id/ManualCounter** a la propiedad **id** del **Plain Text** agregado.

El código AXML será similar al siguiente.

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="match_parent"
    android:layout_height="match_parent">
    <Button
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:id="@+id/StartActivity"
        android:text="@string/StartActivity_Text" />
    <Button
        android:text="@string/ClicksCounter_Text"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:id="@+id/ClicksCounter" />
    <EditText
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:id="@+id/ManualCounter" />
</LinearLayout>
```

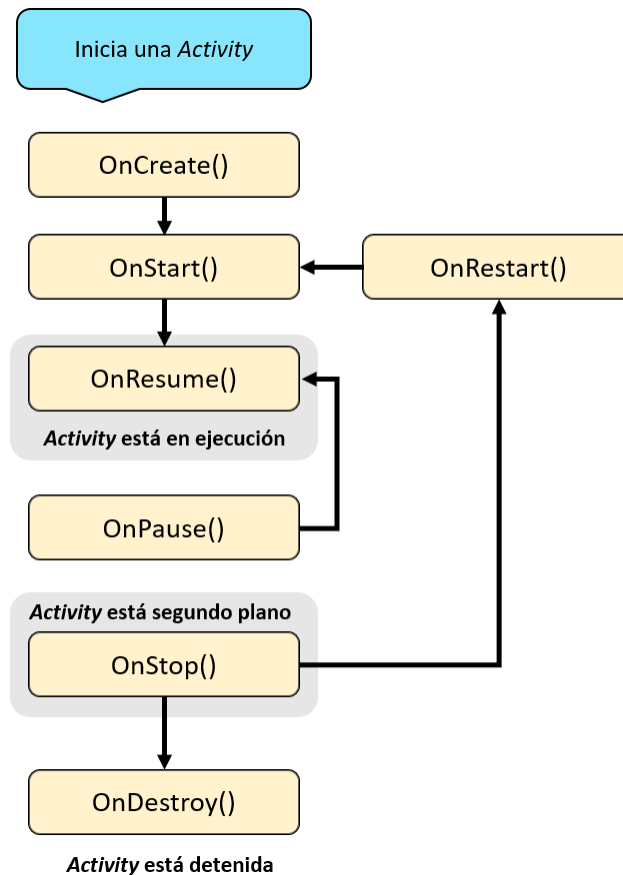


15. Guarda los cambios.

Tarea 2. Examinar los métodos del ciclo de vida de la Activity.

El ciclo de vida de una *Activity* es implementado como una colección de métodos expuestos en la clase *Activity* y que el sistema operativo invoca a lo largo del ciclo de vida de la *Activity*. Estos métodos permiten a los desarrolladores implementar la funcionalidad necesaria para satisfacer los requisitos de administración de estado de cada *Activity* dentro de una aplicación y manejar apropiadamente la administración de recursos.

Cuando el estado de una *Activity* está cambiando, la *Activity* es notificada por el sistema operativo mediante la invocación de métodos específicos en esa *Activity*. El siguiente diagrama ilustra esos métodos en relación con el ciclo de vida de la *Activity*.



Como desarrolladores, podemos manejar los cambios de estado sobrescribiendo esos métodos dentro de la *Activity*. Es importante tomar en cuenta que todos los métodos del ciclo de vida se invocan en el hilo de la interfaz de usuario y que por lo tanto bloquean al sistema operativo de realizar la próxima tarea relacionada con la interfaz de usuario tal como ocultar la *Activity* actual, mostrar una nueva *Activity*, etc. Por este motivo, el código en estos métodos debe ser lo más breve posible para hacer que una aplicación tenga un desempeño eficiente. Las tareas de larga duración deben ejecutarse en un hilo de fondo (background thread).

Es extremadamente importante para el desarrollador de aplicaciones analizar los requerimientos de cada *Activity* para determinar qué métodos expuestos por el ciclo de vida de la *Activity* necesitan ser implementados. De no hacerlo, puede originar la inestabilidad de las aplicaciones, fallas, uso excesivo de recursos y, posiblemente, la inestabilidad del sistema operativo.

En esta tarea examinemos cada uno de los métodos del ciclo de vida y su uso.

OnCreate es el primer método que es invocado cuando se crea una *Activity*. *OnCreate* siempre es sobrescrito para realizar cualquier inicialización de arranque que pueda ser requerida por una *Activity* como:

- Crear Vistas
- Inicializar variables



- Enlazar datos estáticos a listas

1. Abre el archivo *MainActivity.cs*.
2. Agrega el siguiente código al inicio del método *OnCreate* para que al ejecutarse envíe un mensaje al log del dispositivo.

```
Android.Util.Log.Debug("Lab11Log", "Activity A - OnCreate");
```

3. Quita el comentario a la línea de código que establece el contenido de la *Activity*.

```
SetContentView (Resource.Layout.Main);
```

4. Agrega el siguiente código al final del método *OnCreate* para implementar el manejador del evento clic del botón que muestra la segunda pantalla.

```
FindViewById<Button>(Resource.Id.StartActivity).Click += (sender, e) =>
{
    var ActivityIntent =
        new Android.Content.Intent(this, typeof(SecondActivity));
    StartActivity(ActivityIntent);
};
```

El código del método *OnCreate* será similar al siguiente.

```
protected override void OnCreate(Bundle bundle)
{
    Android.Util.Log.Debug("Lab11Log", "Activity A - OnCreate");

    base.OnCreate(bundle);

    // Set our view from the "main" layout resource
    SetContentView (Resource.Layout.Main);

    FindViewById<Button>(Resource.Id.StartActivity).Click += (sender, e) =>
    {
        var ActivityIntent =
            new Android.Content.Intent(this, typeof(SecondActivity));
        StartActivity(ActivityIntent);
    };
}
```

OnStart es el método que siempre es invocado por Android después de finalizar *OnCreate*. Las *Activities* pueden sobrescribir este método si necesitan realizar tareas específicas como actualizar los valores actuales de vistas dentro de la *Activity* justo antes de que una *Activity* se haga visible.

5. Agrega el siguiente método a la clase *MainActivity* para enviar al log del dispositivo un mensaje que indique el momento en que se está ejecutando el método *OnStart*.

```
protected override void OnStart()
{
```



```
        Android.Util.Log.Debug("Lab11Log", "Activity A - OnStart");  
        base.OnStart();  
    }
```

OnResume es el método invocado por el sistema inmediatamente después del método *OnStart* cuando la *Activity* se encuentra lista para comenzar a interactuar con el usuario. Las *Activities* deben sobrescribir este método para realizar tareas tales como:

- a. Aumento en la velocidad de Frames (una tarea común en la creación de juegos).
- b. Iniciar animaciones.
- c. Escuchar actualizaciones del GPS.
- d. Mostrar cualquier alerta o dialogo relevante.
- e. Conectar con manejadores de evento externos.

OnResume es el único método del ciclo de vida que tiene la garantía de ejecutarse después del método *OnPause* cuando traemos a la *Activity* de regreso a la vida.

6. Agrega el siguiente método a la clase *MainActivity* para enviar al log del dispositivo un mensaje que indique el momento en que se está ejecutando el método *OnResume*.

```
protected override void OnResume()  
{  
    Android.Util.Log.Debug("Lab11Log", "Activity A - OnResume");  
    base.OnResume();  
}
```

OnPause es el método que es invocado cuando el sistema está a punto de poner la *Activity* en segundo plano o cuando la *Activity* queda parcialmente oculta. Las *Activities* deben sobrescribir este método si necesitan:

- Guardar los cambios pendientes para persistencia de la información.
- Destruir o liberar otros objetos que consumen recursos.
- Reducir la velocidad de fotogramas y pausar animaciones.
- Anular el registro de los manejadores de eventos externos o de los manejadores de notificaciones (es decir, los que están vinculados a un servicio). Esto debe hacerse para evitar fugas de memoria de la *Activity*.
- De la misma manera, si la *Activity* ha mostrado diálogos o alertas, deben limpiarse con el método *Dismiss()*.

7. Agrega el siguiente método a la clase *MainActivity* para enviar al log del dispositivo un mensaje que indique el momento en que se está ejecutando el método *OnPause*.

```
protected override void OnPause()  
{  
    Android.Util.Log.Debug("Lab11Log", "Activity A - OnPause");  
    base.OnPause();  
}
```



Existen dos posibles métodos del ciclo de vida que pueden ser invocados después de *OnPause*:

- a) *OnResume* en caso de que la *Activity* regrese al primer plano.
- b) *OnStop* en caso de que la *Activity* sea colocada en segundo plano.

OnStop es el método que es invocado cuando la *Activity* ya no es visible para el usuario. Esto ocurre en las siguientes situaciones:

- Se está iniciando una nueva *Activity* que cubre esta *Activity*.
- Una *Activity* existente se está llevando a primer plano.
- La *Activity* está siendo destruida.

En situaciones de poca memoria, el método *OnStop* no siempre podría ser invocado, por ejemplo, cuando Android requiere de muchos recursos y no puede enviar apropiadamente la *Activity* a segundo plano. Por esta razón, es mejor no confiar en que *OnStop* pueda ser invocado cuando se prepara una *Activity* para la destrucción.

8. Agrega el siguiente método a la clase *MainActivity* para enviar al log del dispositivo un mensaje que indique el momento en que se está ejecutando el método *OnStop*.

```
protected override void OnStop()
{
    Android.Util.Log.Debug("Lab11Log", "Activity A - OnStop");
    base.OnStop();
}
```

Los siguientes métodos del ciclo de vida que pueden ser invocados después de *OnStop* son *OnDestroy* si la *Activity* será destruida o bien *OnRestart* si la *Activity* vuelve a primer plano para interactuar con el usuario.

OnDestroy es el método final que es invocado en una instancia de una *Activity* antes de que sea destruida y se elimine completamente de memoria. En situaciones extremas, Android puede matar el proceso de la aplicación que hospeda la *Activity*, lo que ocasionará que *OnDestroy* no sea invocado. La mayoría de las *Activities* no implementan este método porque la mayoría de las acciones de limpieza y destrucción se realizan en los métodos *OnPause* y *OnStop*. Normalmente el método *OnDestroy* se sobrescribe para finalizar los procesos de larga ejecución que podrían provocar pérdidas de recursos. Un ejemplo de esto podrían ser hilos en segundo plano que se hayan iniciado en el método *OnCreate*.

Después de que una *Activity* haya sido destruida, no habrá más invocación de métodos del ciclo de vida.

9. Agrega el siguiente método a la clase *MainActivity* para enviar al log del dispositivo un mensaje que indique el momento en que se está ejecutando el método *OnDestroy*.



```
protected override void OnDestroy()  
{  
    Android.Util.Log.Debug("Lab11Log", "Activity A - OnDestroy");  
    base.OnDestroy();  
}
```

OnRestart es el método que es invocado después de que la *Activity* ha sido detenida y antes de que se esté iniciando nuevamente. Un buen ejemplo de esto sería cuando el usuario presiona el botón de *Home* mientras está en una *Activity* de la aplicación. Cuando esto sucede los métodos *OnPause* y *OnStop* son invocados y la *Activity* es movida a segundo plano, pero no es destruida. Si el usuario restaura la aplicación mediante el administrador de tareas o una aplicación similar, Android invocará al método *OnRestart* de la *Activity*.

No hay una guía general para especificar qué tipo de lógica debe ser implementada en *OnRestart*. Esto es debido a que *OnStart* siempre es invocado independientemente de si la *Activity* está siendo creada o está siendo reiniciada, por lo que cualquier recurso requerido por la *Activity* debe inicializarse en *OnStart*, en lugar de *OnRestart*.

10. Agrega el siguiente método a la clase *MainActivity* para enviar al log del dispositivo un mensaje que indique el momento en que se está ejecutando el método *OnRestart*.

```
protected override void OnRestart()  
{  
    Android.Util.Log.Debug("Lab11Log", "Activity A - OnRestart");  
    base.OnRestart();  
}
```

11. Modifica el código de la clase *SecondActivity* para determinar el momento en que se ejecuten los métodos correspondientes al ciclo de vida de la *Activity*. El código de la clase será similar al siguiente.

```
public class SecondActivity : Activity  
{  
    protected override void OnCreate(Bundle bundle)  
    {  
        Android.Util.Log.Debug("Lab11Log", "Activity B - OnCreate");  
        base.OnCreate(bundle);  
    }  
    protected override void OnStart()  
    {  
        Android.Util.Log.Debug("Lab11Log", "Activity B - OnStart");  
        base.OnStart();  
    }  
    protected override void OnResume()  
    {  
        Android.Util.Log.Debug("Lab11Log", "Activity B - OnResume");  
        base.OnResume();  
    }  
    protected override void OnPause()  
    {  
        Android.Util.Log.Debug("Lab11Log", "Activity B - OnPause");  
    }  
}
```

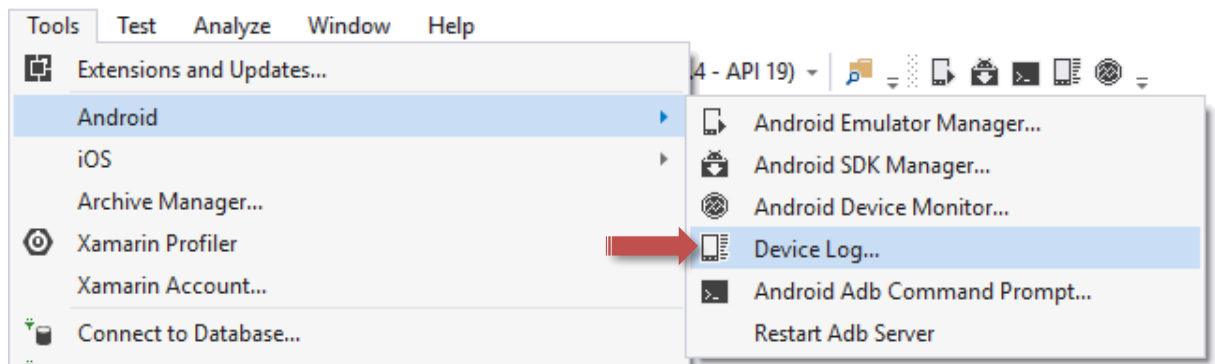


```
        base.OnPause();
    }
    protected override void OnStop()
    {
        Android.Util.Log.Debug("Lab11Log", "Activity B - OnStop");
        base.OnStop();
    }
    protected override void OnDestroy()
    {
        Android.Util.Log.Debug("Lab11Log", "Activity B - OnDestroy");
        base.OnDestroy();
    }
    protected override void OnRestart()
    {
        Android.Util.Log.Debug("Lab11Log", "Activity B - OnRestart");
        base.OnRestart();
    }
}
```

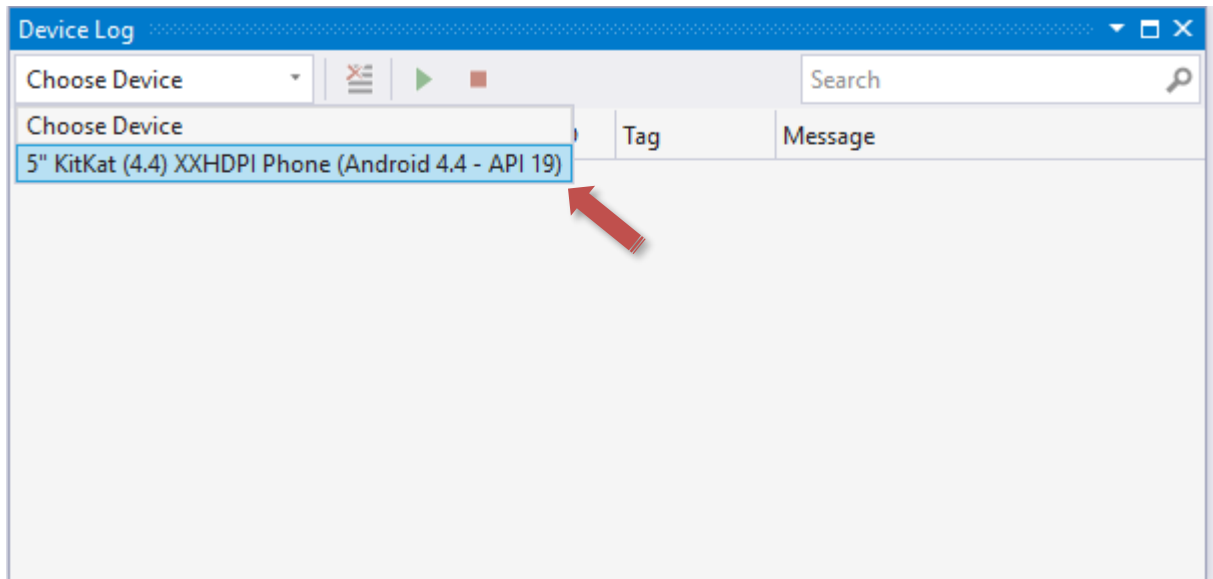
Tarea 3. Examinar las transiciones de estado.

Cada método en *MainActivity* y *SecondActivity* escriben un mensaje a la ventana *Output* de Visual Studio y al Log del dispositivo Android para indicar el estado de la *Activity*. En esta tarea ejecutarás la aplicación y examinarás las transiciones entre los distintos estados de las *Activities*.

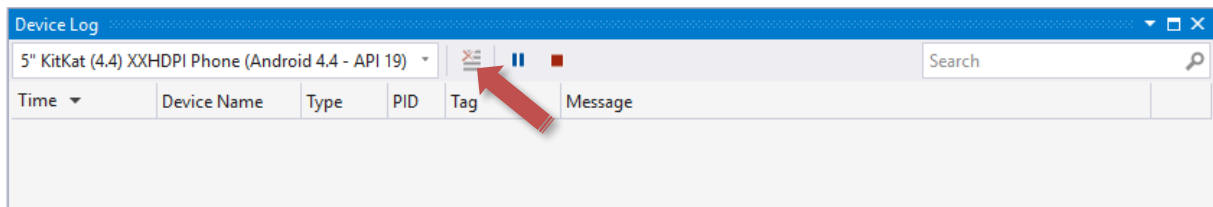
1. Selecciona la opción **Tools > Android > Device Log...** para abrir la ventana de log del dispositivo Android.



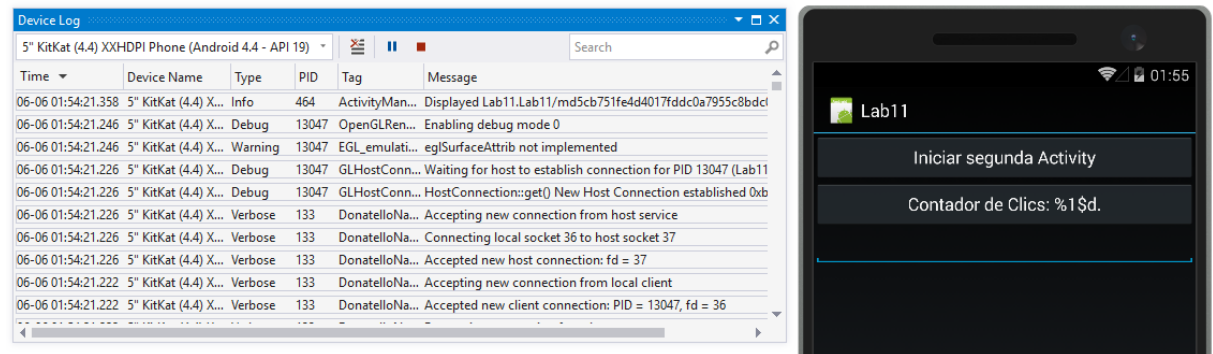
2. En la ventana **Device Log** selecciona el dispositivo en el que ejecutarás la aplicación.



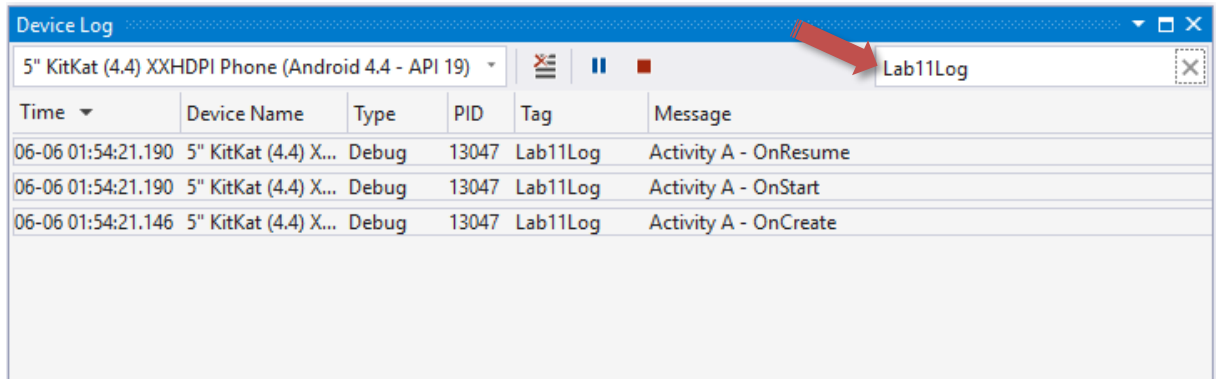
3. En caso de que la ventana **Device Log** te muestre algunos mensajes, haz clic en el botón para eliminarlos.



4. Ejecuta la aplicación. Cuando la aplicación haya iniciado, podrás ver distintos mensajes en la ventana **Device Log**.

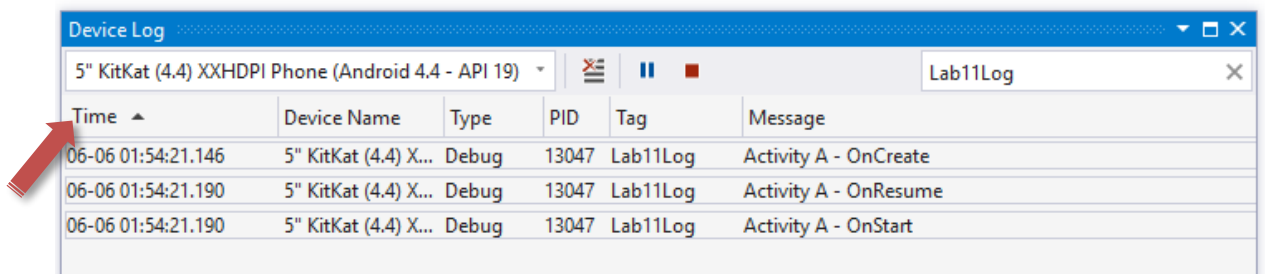


5. En el cuadro **Search** escribe **Lab11Log** para filtrar los mensajes.



Time	Device Name	Type	PID	Tag	Message
06-06 01:54:21.190	5" KitKat (4.4) X...	Debug	13047	Lab11Log	Activity A - onResume
06-06 01:54:21.190	5" KitKat (4.4) X...	Debug	13047	Lab11Log	Activity A - onStart
06-06 01:54:21.146	5" KitKat (4.4) X...	Debug	13047	Lab11Log	Activity A - onCreate

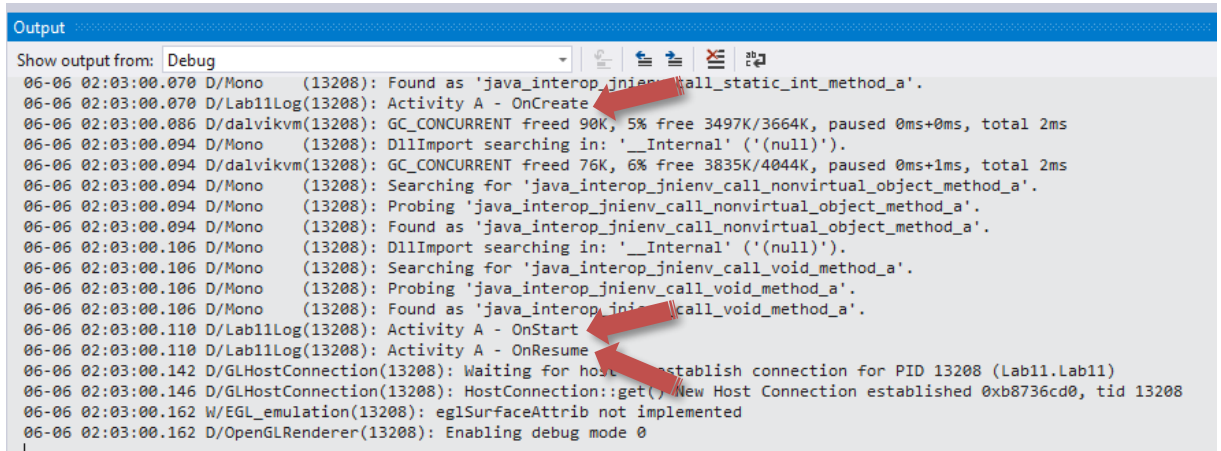
6. Haz clic en la columna **Time** para ordenar los mensajes de forma ascendente.



Time	Device Name	Type	PID	Tag	Message
06-06 01:54:21.146	5" KitKat (4.4) X...	Debug	13047	Lab11Log	Activity A - onCreate
06-06 01:54:21.190	5" KitKat (4.4) X...	Debug	13047	Lab11Log	Activity A - onResume
06-06 01:54:21.190	5" KitKat (4.4) X...	Debug	13047	Lab11Log	Activity A - onStart

Puedes observar los cambios de estado de la **Activity A**.

7. Selecciona la opción **View > Output** de la barra de menús de Visual Studio para abrir la ventana **Output**. Examina el orden en el que se ejecutaron los métodos.



Show output from:	Debug
06-06 02:03:00.070 D/Mono (13208): Found as 'java_interop_jnienv_call_static_int_method_a'.	
06-06 02:03:00.070 D/Lab11Log(13208): Activity A - onCreate	
06-06 02:03:00.086 D/dalvikvm(13208): GC_CONCURRENT freed 90K, 5% free 3497K/3664K, paused 0ms+0ms, total 2ms	
06-06 02:03:00.094 D/Mono (13208): DllImport searching in: '__Internal' ('(null)').	
06-06 02:03:00.094 D/dalvikvm(13208): GC_CONCURRENT freed 76K, 6% free 3835K/4044K, paused 0ms+1ms, total 2ms	
06-06 02:03:00.094 D/Mono (13208): Searching for 'java_interop_jnienv_call_nonvirtual_object_method_a'.	
06-06 02:03:00.094 D/Mono (13208): Probing 'java_interop_jnienv_call_nonvirtual_object_method_a'.	
06-06 02:03:00.094 D/Mono (13208): Found as 'java_interop_jnienv_call_nonvirtual_object_method_a'.	
06-06 02:03:00.106 D/Mono (13208): DllImport searching in: '__Internal' ('(null)').	
06-06 02:03:00.106 D/Mono (13208): Searching for 'java_interop_jnienv_call_void_method_a'.	
06-06 02:03:00.106 D/Mono (13208): Probing 'java_interop_jnienv_call_void_method_a'.	
06-06 02:03:00.106 D/Mono (13208): Found as 'java_interop_jnienv_call_void_method_a'.	
06-06 02:03:00.110 D/Lab11Log(13208): Activity A - onStart	
06-06 02:03:00.110 D/Lab11Log(13208): Activity A - onResume	
06-06 02:03:00.142 D/GLHostConnection(13208): Waiting for host to establish connection for PID 13208 (Lab11.Lab11)	
06-06 02:03:00.146 D/GLHostConnection(13208): HostConnection::get() New Host Connection established 0xb8736cd0, tid 13208	
06-06 02:03:00.162 W/EGl_emulation(13208): eglSurfaceAttrib not implemented	
06-06 02:03:00.162 D/OpenGLRenderer(13208): Enabling debug mode 0	

El orden fue: **onCreate**, **onStart**, **onResume**.

8. Haz Clic en el botón **Iniciar segunda Activity**. Podrás ver el estado **onPause** y **onStop** de la **Activity A** mientras que la **Activity B** transita por sus distintos estados.

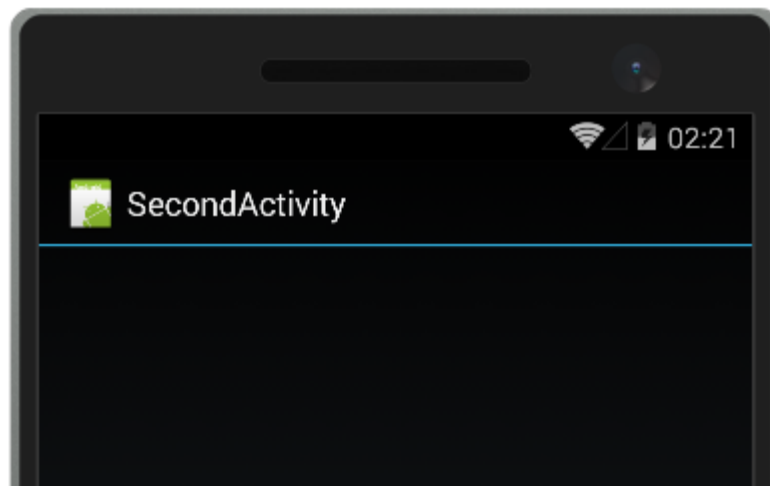
Activity A – **onPause**, **Activity B** – **onCreate**, **Activity B** – **onStart**, **Activity B** – **onResume**, **Activity A** – **onStop**.



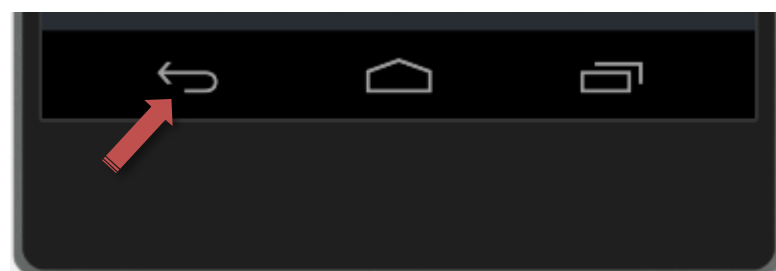
06-06 02:17:34.726	5" KitKat (4.4) X...	Debug	13208	Lab11Log	Activity A - onPause
06-06 02:17:34.742	5" KitKat (4.4) X...	Debug	13208	Lab11Log	Activity B - onCreate
06-06 02:17:34.742	5" KitKat (4.4) X...	Debug	13208	Lab11Log	Activity B - onStart
06-06 02:17:34.742	5" KitKat (4.4) X...	Debug	13208	Lab11Log	Activity B - onResume
06-06 02:17:35.198	5" KitKat (4.4) X...	Debug	13208	Lab11Log	Activity A - onStop

```
Output
Show output from: Debug
06-06 02:17:34.726 D/Lab11Log(13208): Activity A - onPause
06-06 02:17:34.742 D/Lab11Log(13208): Activity B - onCreate
06-06 02:17:34.742 D/Lab11Log(13208): Activity B - onStart
06-06 02:17:34.742 D/Lab11Log(13208): Activity B - onResume
06-06 02:17:34.782 W/EGL_emulation(13208): eglSurfaceAttrib not implemented
06-06 02:17:35.198 D/Lab11Log(13208): Activity A - onStop
```

Como resultado, la *Activity B* es iniciada y mostrada en lugar de la *Activity A*.



- Haz clic en el botón **Back** del dispositivo.



La *Activity B* es destruida y la *Activity A* es reanudada.

Activity B – onPause, *Activity A* – OnRestart, *Activity A* – onStart, *Activity A* – onResume,
Activity B – onStop, *Activity B* – onDestroy.



```
Output
Show output from: Debug
06-06 02:17:34.726 D/Lab11Log(13208): Activity A - onPause
06-06 02:17:34.742 D/Lab11Log(13208): Activity B - onCreate
06-06 02:17:34.742 D/Lab11Log(13208): Activity B - onStart
06-06 02:17:34.742 D/Lab11Log(13208): Activity B - onResume
06-06 02:17:34.782 W/EGL_emulation(13208): eglSurfaceAttrib not implemented
06-06 02:17:35.198 D/Lab11Log(13208): Activity A - onStop
06-06 02:28:39.670 D/Lab11Log(13208): Activity B - onPause
06-06 02:28:39.670 D/Lab11Log(13208): Activity A - onRestart
06-06 02:28:39.670 D/Lab11Log(13208): Activity A - onStart
06-06 02:28:39.670 D/Lab11Log(13208): Activity A - onResume
06-06 02:28:39.694 W/EGL_emulation(13208): eglSurfaceAttrib not implemented
06-06 02:28:40.062 D/Lab11Log(13208): Activity B - onStop
06-06 02:28:40.062 D/Lab11Log(13208): Activity B - onDestroy
```

Muchos dispositivos Android tienen dos botones distintos: un botón "**Back**" y un botón "**Home**". Aunque parecen tener el mismo efecto de poner una aplicación en segundo plano, hay una sutil diferencia entre los dos botones. Cuando un usuario hace clic en el botón **Back**, le está diciendo a Android que ha terminado con la *Activity*. Android destruirá la *Activity*. Por el contrario, cuando el usuario hace clic en el botón **Home**, la *Activity* se coloca simplemente en segundo plano. Android no destruirá la *Activity*.

Cuando una *Activity* es detenida o destruida, el sistema proporciona una oportunidad de guardar el estado de la *Activity* para poder recuperarlo posteriormente. Este estado almacenado se conoce como **Estado de Instancia**. Android ofrece tres opciones para almacenar el *Estado de Instancia* durante el ciclo de vida de la *Activity*:

- Almacenar valores primitivos en un diccionario conocido como **Bundle** que Android utilizará para salvar el estado.
- Crear una clase personalizada que contendrá valores complejos como bitmaps. Android utilizará esta clase personalizada para guardar el estado.
- Procesar los cambios de configuración durante el ciclo de vida y asumir la responsabilidad completa para mantener el estado en la *Activity*.

Tarea 4. Agregar un contador de Clics.

En esta tarea realizarás cambios en la aplicación para mostrar el número de veces que el usuario hace clic en un botón.

1. Abre el archivo *MainActivity*.



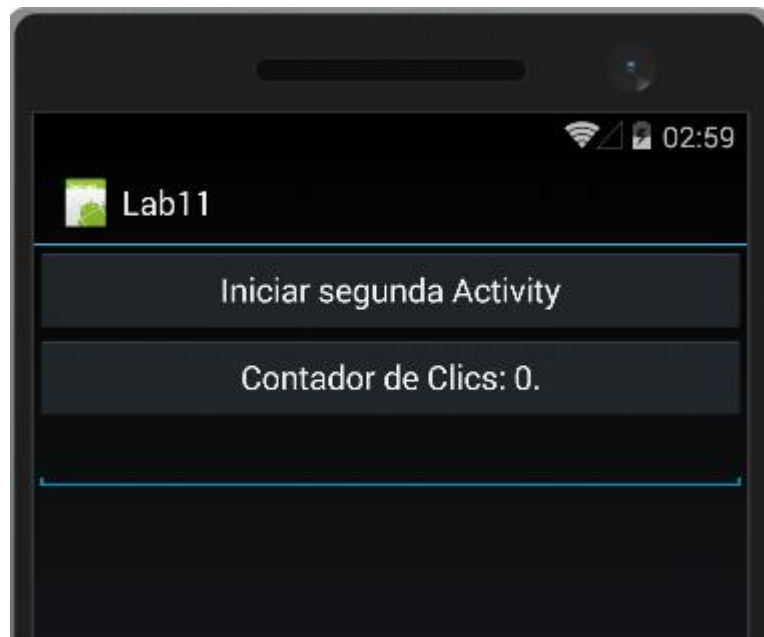
2. Agrega el siguiente código a nivel de clase para declarar una variable que almacenará el número de veces que el usuario hace clic en un botón.

```
public class MainActivity : Activity
{
    int Counter = 0;
```

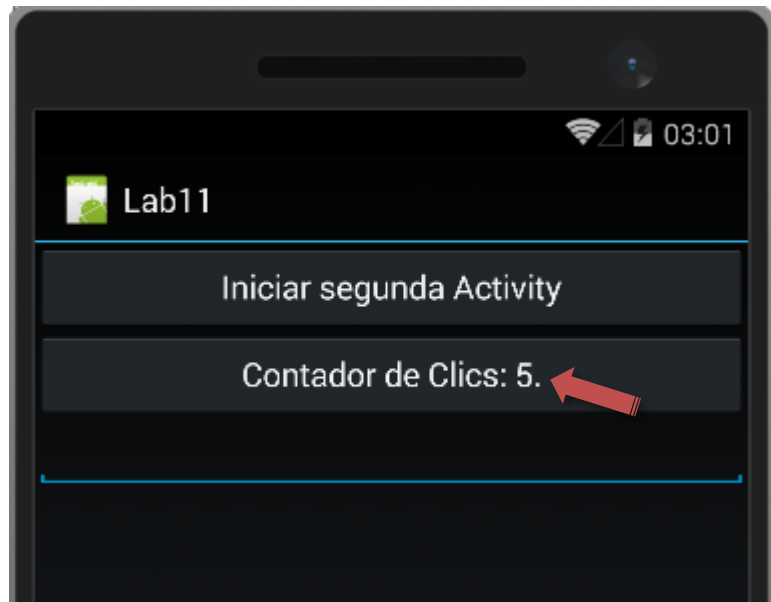
3. Agrega el siguiente código al final del método *OnCreate* para manejar el evento *Click* del botón *ClickCounter*.

```
var ClickCounter =
    FindViewById<Button>(Resource.Id.ClicksCounter);
ClickCounter.Text =
    Resources.GetString(Resource.String.ClicksCounter_Text, Counter);
ClickCounter.Click += (sender, e) =>
{
    Counter++;
    ClickCounter.Text =
        Resources.GetString(Resource.String.ClicksCounter_Text, Counter);
};
```

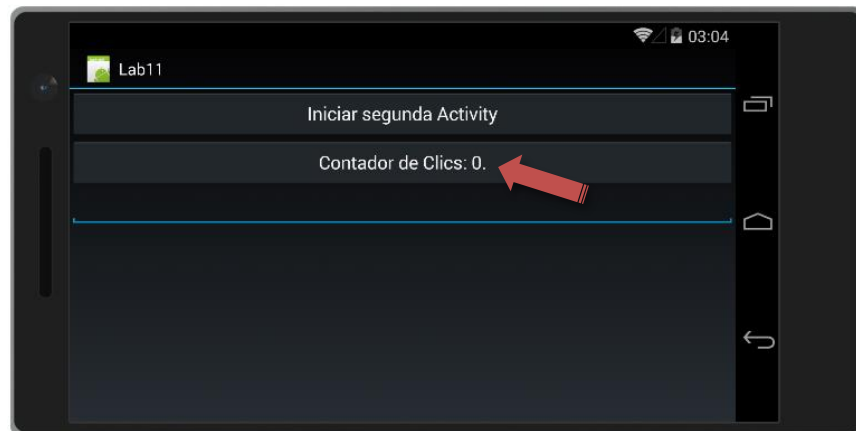
4. Ejecuta la aplicación. Se mostrará una pantalla similar a la siguiente.



5. Haz clic 5 veces sobre el botón contador de Clics. Puedes notar que el contador se va incrementando correctamente con cada Clic.



6. Gira el dispositivo hacia el modo horizontal. Puedes notar que el contador se ha perdido.



Examinando la salida de la aplicación puedes ver que la *Activity A* fue pausada, detenida, destruida, recreada, reiniciada y reanudada durante la rotación de modo vertical a horizontal.

```
Output
Show output from: Debug
06-06 02:59:11.834 W/EGL_emulation(13628): eglSurfaceAttrib not implemented
06-06 02:59:11.838 D/OpenGLRenderer(13628): Enabling debug mode 0
06-06 03:04:15.266 D/Lab11Log(13628): Activity A - onPause
06-06 03:04:15.270 D/Lab11Log(13628): Activity A - onStop
06-06 03:04:15.274 D/Lab11Log(13628): Activity A - onDestroy
06-06 03:04:15.278 D/Lab11Log(13628): Activity A - onCreate
06-06 03:04:15.282 D/Lab11Log(13628): Activity A - onStart
06-06 03:04:15.286 D/Lab11Log(13628): Activity A - onResume
06-06 03:04:15.342 W/EGL_emulation(13628): eglSurfaceAttrib not implemented
```



Debido a que la Activity es destruida y recreada nuevamente cuando el dispositivo es girado, su estado de instancia es perdido.

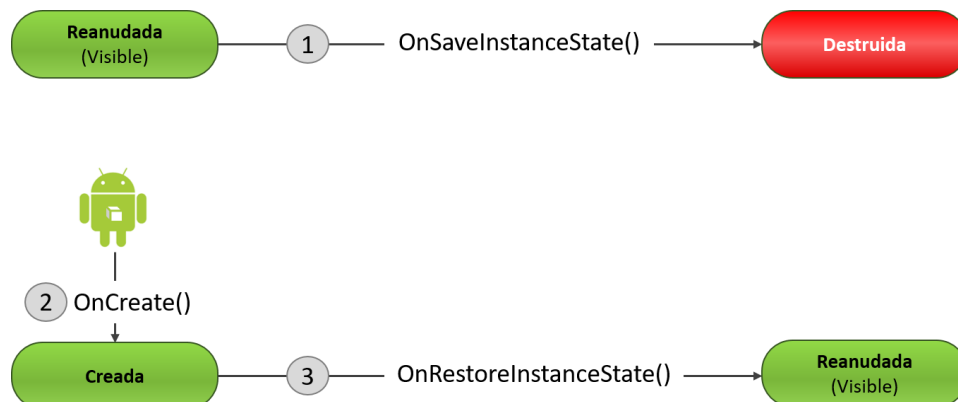
Tarea 5. Manejar el estado utilizando un objeto Bundle.

La opción primaria para conservar el estado de instancia es utilizar un objeto diccionario clave/valor conocido como **Bundle**. Cuando una Activity es creada, el método *OnCreate* recibe un *Bundle* como parámetro, este parámetro puede ser utilizado para restaurar el estado de instancia. No es recomendado usar un *Bundle* para trabajar con datos complejos que no puedan ser serializados como pares clave/valor de forma fácil y rápida (como mapas de bits). Un *Bundle* debe ser utilizado para valores simples como cadenas o enteros.

Una Activity proporciona métodos para ayudar a guardar y recuperar el estado de instancia en el *Bundle*:

- **OnSaveInstanceState**. Este metodo es invocado por Android cuando la Activity está siendo destruida. Las Activities pueden implementar este método si necesitan persistir cualquier elemento de estado clave/valor.
- **OnRestoreInstanceState**. Este metodo es invocado después de que el método *OnCreate* ha finalizado y proporciona otra oportunidad para que una Activity restaure su estado después de que se haya completado la inicialización.

El siguiente diagrama ilustra cómo son usados estos métodos.



1. Agrega el siguiente código en la clase *MainActivity* para definir el método *OnSaveInstanceState*. Este método será invocado cuando la Activity esté siendo detenida. Antes de que la Activity sea destruida, Android invocará automáticamente a este método y le pasará un objeto **Bundle** que la Activity puede utilizar para almacenar su estado. Cuando un dispositivo experimenta un cambio de configuración, una Activity puede sobrescribir el



método *OnSaveInstanceState* y utilizar el objeto *Bundle* que le es pasado para conservar ahí su estado.

```
protected override void OnSaveInstanceState(Bundle outState)
{
    base.OnSaveInstanceState(outState);
}
```

2. Agrega el siguiente código dentro del método *OnSaveInstanceState* para almacenar el valor del contador como un valor entero.

```
protected override void OnSaveInstanceState(Bundle outState)
{
    outState.PutInt("CounterValue", Counter);
    Android.Util.Log.Debug("Lab11Log", "Activity A - OnSaveInstanceState");
    base.OnSaveInstanceState(outState);
}
```

Nota: Es importante invocar siempre a la implementación base del método *OnSaveInstanceState* para que el estado de la arquitectura de la vista también pueda ser salvado.

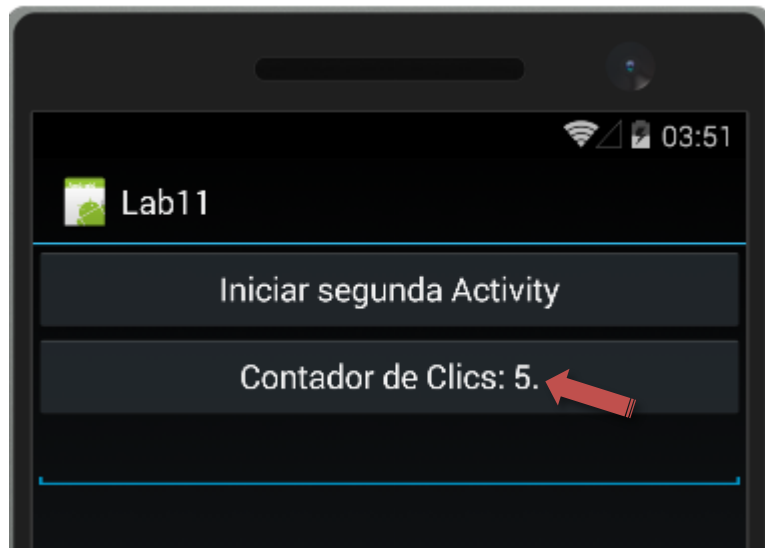
Cuando una *Activity* es recreada y reanudada, Android le pasa de regreso el objeto *Bundle* en el método **OnCreate**. Nota que el método *OnCreate* define un parámetro *Bundle* que es un diccionario para almacenar y pasar información de estado y objetos entre *Activities*. Si el *Bundle* no es nulo, nos indica que la *Activity* se está reiniciando y debe restaurar el estado de la instancia anterior.

3. Agrega el siguiente código justo antes de la línea que define la variable *ClickCounter*. El código restaurará el valor de *Counter* desde el objeto *Bundle* que le es pasado.

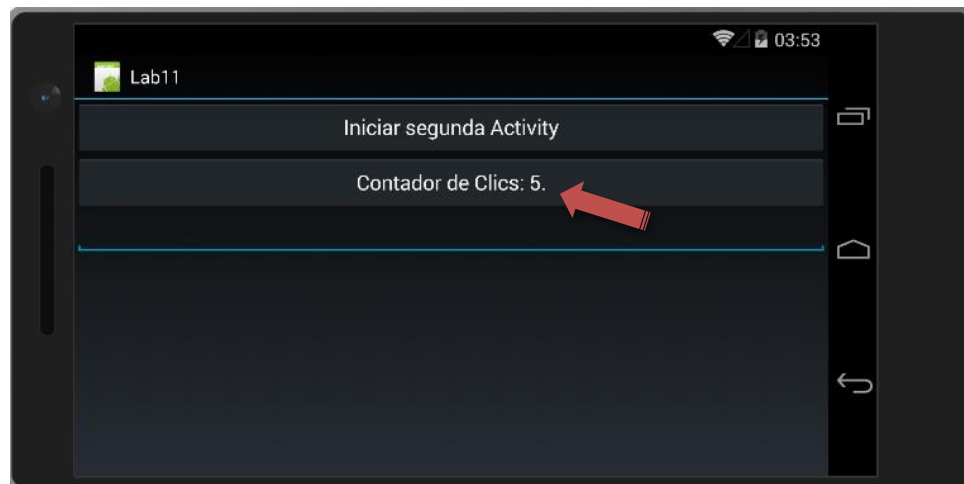
```
if(bundle!=null)
{
    Counter = bundle.GetInt("CounterValue", 0);
    Android.Util.Log.Debug("Lab11Log", "Activity A - Recovered Instance State");
}

var ClickCounter =
    FindViewById<Button>(Resource.Id.ClicksCounter);
```

4. Ejecuta la aplicación.
7. Haz clic 5 veces sobre el botón contador de Clics. Puedes notar que el contador se va incrementando correctamente con cada Clic.



5. Gira el dispositivo hacia el modo horizontal. *¡Puedes notar que el contador se ha mantenido!*



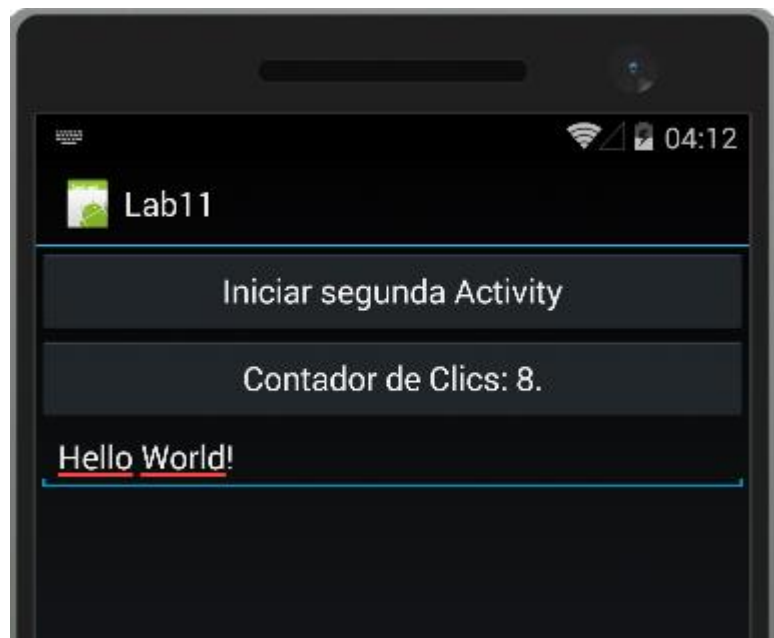
6. Examina el resultado en la ventana Output.

```
Output
Show output from: Debug
06-06 03:50:47.458 W/EGL_emulation(14117): eglSurfaceAttrib not implemented
06-06 03:50:47.462 D/OpenGLRenderer(14117): Enabling debug mode 0
06-06 03:53:14.534 D/Lab11Log(14117): Activity A - onPause
06-06 03:53:14.542 D/Lab11Log(14117): Activity A - onSaveInstanceState
06-06 03:53:14.546 D/Lab11Log(14117): Activity A - onStop
06-06 03:53:14.550 D/Lab11Log(14117): Activity A - onDestroy
06-06 03:53:14.562 D/Lab11Log(14117): Activity A - onCreate
06-06 03:53:14.566 D/Mono (14117): DllImport searching in: '__Internal' ('(null)').
06-06 03:53:14.566 D/Mono (14117): Searching for 'java_interop_jnienv_call_int_method_a'.
06-06 03:53:14.566 D/Mono (14117): Probing 'java_interop_jnienv_call_int_method_a'.
06-06 03:53:14.566 D/Mono (14117): Found as 'java_interop_jnienv_call_int_method_a'.
06-06 03:53:14.566 D/Lab11Log(14117): Activity A - Recovered Instance State
06-06 03:53:14.566 D/Lab11Log(14117): Activity A - onStart
06-06 03:53:14.574 D/Lab11Log(14117): Activity A - onResume
06-06 03:53:14.574 W/InputStreamConnectionWrapper(14117): beginBatchEdit on inactive InputConnection
06-06 03:53:14.574 W/InputStreamConnectionWrapper(14117): endBatchEdit on inactive InputConnection
06-06 03:53:14.610 W/EGL_emulation(14117): eglSurfaceAttrib not implemented
```



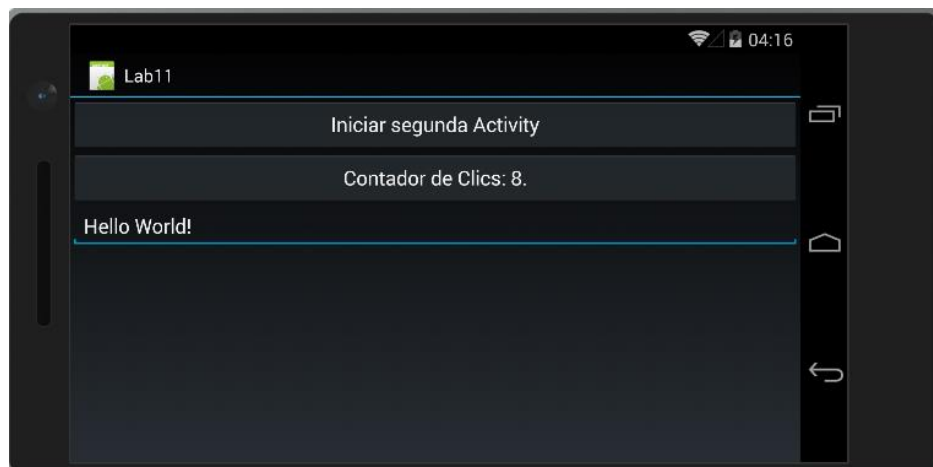
Antes de que el método *OnStop* fuera invocado, el nuevo método *OnSaveInstanceState* fue invocado para guardar el valor de la variable *Counter* en un *Bundle*. Android pasó este *Bundle* de regreso cuando invocó al método *OnCreate* y fuimos capaces de utilizarlo para restaurar el valor original de *Counter*.

7. Gira nuevamente la pantalla a modo vertical. El contador se mantiene.
8. Haz clic 3 veces en el botón contador de clics. Nota que el contador se sigue incrementando.
9. Escribe un dato en el cuadro de texto.



¿Qué crees que suceda con el valor del cuadro de texto al girar en horizontal el dispositivo? Recuerda que únicamente estamos guardando el valor del Contador en el **Bundle**. ¿El valor se perderá al girar el dispositivo?

10. Gira el dispositivo a modo horizontal.





Puedes notar que el valor que escribiste se mantiene.

Sobrescribir el método *OnSaveInstanceState* es un mecanismo apropiado para guardar datos transitorios en una *Activity* a través de cambios de orientación, como el Contador en el ejemplo anterior. Sin embargo, la implementación predeterminada de *OnSaveInstanceState* se encargará de guardar los datos transitorios de cada vista de la interfaz de usuario, siempre y cuando cada vista tenga asignado un ID.

El elemento *EditText* que agregaste tiene un **id** que asignaste previamente y por lo tanto puede ser persistido.

11. Regresa a Visual Studio y detén la aplicación.
12. Elimina la propiedad **id** del elemento *EditText*. El código AXML del elemento *EditText* será similar al siguiente.

```
<EditText
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
/>
```

13. Ejecuta la aplicación.
14. Escribe un dato en el cuadro de texto.
15. Gira el dispositivo. Puedes notar que el valor no es persistido.
16. Regresa a Visual Studio y detén la aplicación.

El método *OnRestoreInstanceState* es invocado después de *OnStart*. Proporciona a una *Activity* la oportunidad de restaurar cualquier estado que haya sido almacenado previamente en un *Bundle* durante la ejecución previa del método *OnSaveInstanceState*. Este es el mismo *Bundle* que se proporciona a *OnCreate*.

El método *OnRestoreInstanceState* existe para proporcionar cierta flexibilidad cuando el estado debe ser restaurado. A veces es más apropiado esperar hasta que todas las inicializaciones se realicen antes de restaurar el estado de la instancia. Además, una subclase de una *Activity* existente podría querer únicamente restaurar ciertos valores del estado de instancia. En la mayoría de los casos, no es necesario sobrescribir *OnRestoreInstanceState*, ya que la mayoría de las *Activities* pueden restaurar el estado utilizando el *Bundle* proporcionado a *OnCreate*.

Limitaciones del Bundle

Aunque *OnSaveInstanceState* hace que sea fácil guardar datos transitorios, tiene algunas limitaciones:

- No es invocado en todos los casos. Por ejemplo, al presionar *Home* o *Back* para salir de una *Activity*, no se activará el método *OnSaveInstanceState*.



- El *Bundle* pasado como parámetro en *OnSaveInstanceState* no está diseñado para objetos grandes, tales como imágenes.
- Los datos guardados mediante el *Bundle* son serializados, lo que puede provocar retardos.

El estado del *Bundle* es útil para datos simples que no utilicen mucha memoria, mientras que los datos de instancia que no son de configuración son útiles para datos más complejos o datos que son costosos de recuperar, tal como los datos de una llamada a un servicio web o a los datos de una consulta compleja a la base de datos. Los datos de instancia que no son de configuración se guardan en un objeto según sea necesario.

Tarea 6. Persistir datos complejos.

Si para reiniciar una *Activity* se deben recuperar grandes conjuntos de datos, restablecer una conexión de red o realizar otras operaciones intensivas, un reinicio completo debido a un cambio de configuración podría dar como resultado una experiencia lenta para el usuario. Además, quizá no se pueda restaurar completamente el estado de la *Activity* con el *Bundle* que el sistema almacena con el método *OnSaveInstanceState*. El *Bundle* no está diseñado para transferir objetos grandes (como mapas de bits) y los datos que contiene deben serializarse y, luego, deserializarse, lo que puede consumir mucha memoria y hacer que el cambio de configuración sea lento. En dicha situación, podemos reducir la carga que implica reiniciar la *Activity* reteniendo un objeto **Fragment** cuando la *Activity* se reinicie debido a un cambio de configuración. Este fragmento puede contener referencias a los objetos con estado que deseamos retener.

Cuando el sistema Android reconstruye la *Activity* debido a un cambio de configuración, los fragmentos de la *Activity* que son marcados para ser retenidos no son destruidos.

En esta tarea agregaremos código a la aplicación para persistir datos complejos durante un cambio de configuración en tiempo de ejecución utilizando un objeto **Fragment**.

1. Utiliza la plantilla **Android > Class** para agregar a la aplicación Android una nueva clase llamada **Complex**.
2. Modifica la clase *Complex* para que herede de la clase *Fragment*.

```
public class Complex : Fragment
{
}
```

3. Agrega el siguiente código para definir los miembros de la clase *Complex*.

```
public class Complex : Fragment
{
    public int Real { get; set; }
    public int Imaginary { get; set; }

    public override string ToString()
    {
```



```
        return $"{Real} + {Imaginary}i";  
    }  
}
```

4. Agrega a la clase *Complex* el siguiente código para indicar a Android que retenga la instancia actual del fragmento cuando la *Activity* sea recreada.

```
public override void OnCreate(Bundle savedInstanceState)  
{  
    base.OnCreate(savedInstanceState);  
    RetainInstance = true;  
}
```

El método *OnCreate* es invocado únicamente cuando el fragmento se está construyendo.

De manera predeterminada, los fragmentos son destruidos y recreados junto con su *Activity* cuando un cambio en la configuración ocurre. Al establecer el valor de la propiedad ***RetainInstance*** en true, nos permite evitar el ciclo de destrucción y recreación indicando al sistema retener la instancia actual del fragmento cuando la *Activity* sea recreada.

Nota: Aunque podemos almacenar cualquier objeto, nunca debemos pasar un objeto que esté vinculado con la *Activity*, por ejemplo, un *Drawable*, un *Adapter*, un *View* u otros objetos asociados con un *Context*. Si lo hacemos, perderemos todas las vistas y los recursos de la instancia de la *Activity* original. (Pérdida de recursos significa que la aplicación retiene los recursos y no se les puede recolectar como elementos no utilizados, por lo que se puede perder mucha memoria).

5. Abre el archivo *MainActivity*.
6. Agrega el siguiente código a la clase *MainActivity* para declarar una variable de tipo *Complex* a nivel de clase.

```
public class MainActivity : Activity  
{  
    Complex Data;  
    int Counter = 0;
```

7. Agrega el siguiente código justo antes de la línea `if (bundle != null)`.

```
// Utilizar FragmentManager para recuperar el Fragmento  
Data = (Complex)this.FragmentManager.FindFragmentByTag("Data");  
if(Data == null)  
{  
    // No ha sido almacenado, agregar el fragmento a la Activity  
    Data = new Complex();  
    var FragmentTransaction = this.FragmentManager.BeginTransaction();  
    FragmentTransaction.Add(Data, "Data");  
    FragmentTransaction.Commit();  
}
```



8. Agrega el siguiente código antes de la línea que define al manejador del evento *Click* del botón *ClickCounter*.

```
ClickCounter.Text += $"{Data.ToString()}";
```

9. Agrega el siguiente código dentro del código del manejador del evento *Click* del botón *ClickCounter*.

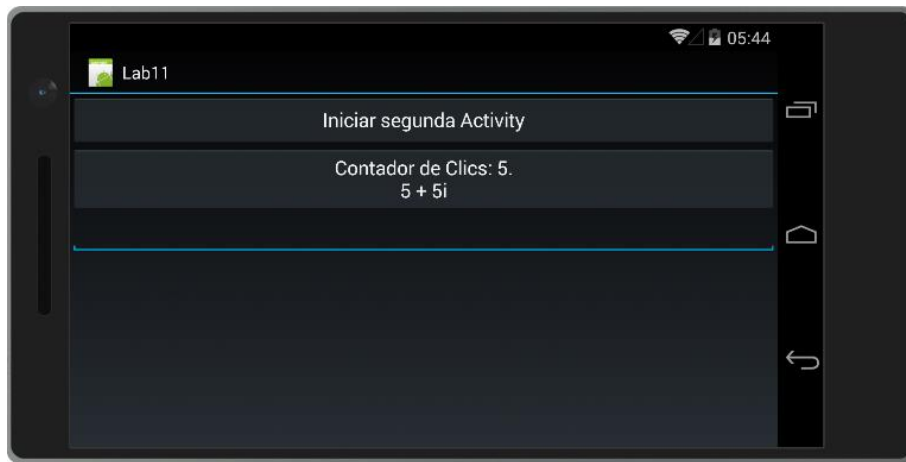
```
ClickCounter.Text += $"{Data.ToString()}";  
ClickCounter.Click += (sender, e) =>  
{  
    Counter++;  
    ClickCounter.Text =  
        Resources.GetString(Resource.String.ClicksCounter_Text, Counter);  
  
    // Modificar con cualquier valor solo para verificar la persistencia  
    Data.Real++;  
    Data.Imaginary++;  
    // Mostrar el valor de los miembros  
    ClickCounter.Text += $"{Data.ToString()}";  
};
```

10. Ejecuta la aplicación.
11. Haz 5 veces clic en el botón contador de Clics. Puedes notar que los valores del Contador y del objeto *Complex* son mostrados correctamente.





12. Gira el dispositivo al modo horizontal. Puedes notar que los valores se han mantenido.



Algo importante de comentar es que el uso de Fragmentos fue introducido en Android 3.0. Antes de Android 3.0, la estrategia recomendada para persistir objetos complejos era sobrescribiendo el método *OnRetainNonConfigurationInstance*.



Ejercicio 2: Validando tu actividad

En este ejercicio agregarás funcionalidad a tu laboratorio con el único propósito de enviar una evidencia de la realización del mismo.

La funcionalidad que agregarás consumirá un ensamblado que representa una Capa de acceso a servicio (SAL) que será consumida por tu aplicación Android.

Es importante que realices cada laboratorio del diplomado ya que esto te dará derecho a obtener el diploma final del mismo.

Tarea 1. Agregar los componentes de la Capa de acceso a Servicio.

En esta tarea agregarás una referencia al ensamblado **SALLab11.dll** que implementa la capa de acceso a servicio. El archivo **SALLab11.dll** se encuentra disponible junto con este documento.

1. En el proyecto Xamarin.Android, agrega una referencia del ensamblado **SALLab11.dll**.

Este componente realiza una conexión a un servicio de Azure Mobile, por lo tanto, será necesario agregar el paquete NuGet **Microsoft.Azure.Mobile.Client**.

2. En el proyecto Xamarin.Android, instala el paquete NuGet **Microsoft.Azure.Mobile.Client**.

Tarea 2. Agregar la funcionalidad para validar la actividad.

El componente DLL que agregaste te permite registrar tu actividad en la plataforma de TI Capacitación y Microsoft. El componente se comunica con la plataforma de TI Capacitación para autenticarte y posteriormente envía un registro a la plataforma Microsoft.

1. Agrega la funcionalidad necesaria para que, al lanzar tu aplicación, se muestre la pantalla principal con los datos de validación de la actividad.
2. Agrega la funcionalidad necesaria para que el servicio de validación solo sea ejecutado una sola vez.

Cuando tu actividad se haya validado exitosamente puedes ver el estatus en el siguiente enlace: <https://ticapacitacion.com/evidencias/xamarin30>.

Nota: Es probable que recibas un correo similar al siguiente.

Tu código de lab no es válido, revisa que estés utilizando un código de reto válido. Si tienes preguntas o dudas por favor contacta a dxaudmx@microsoft.com

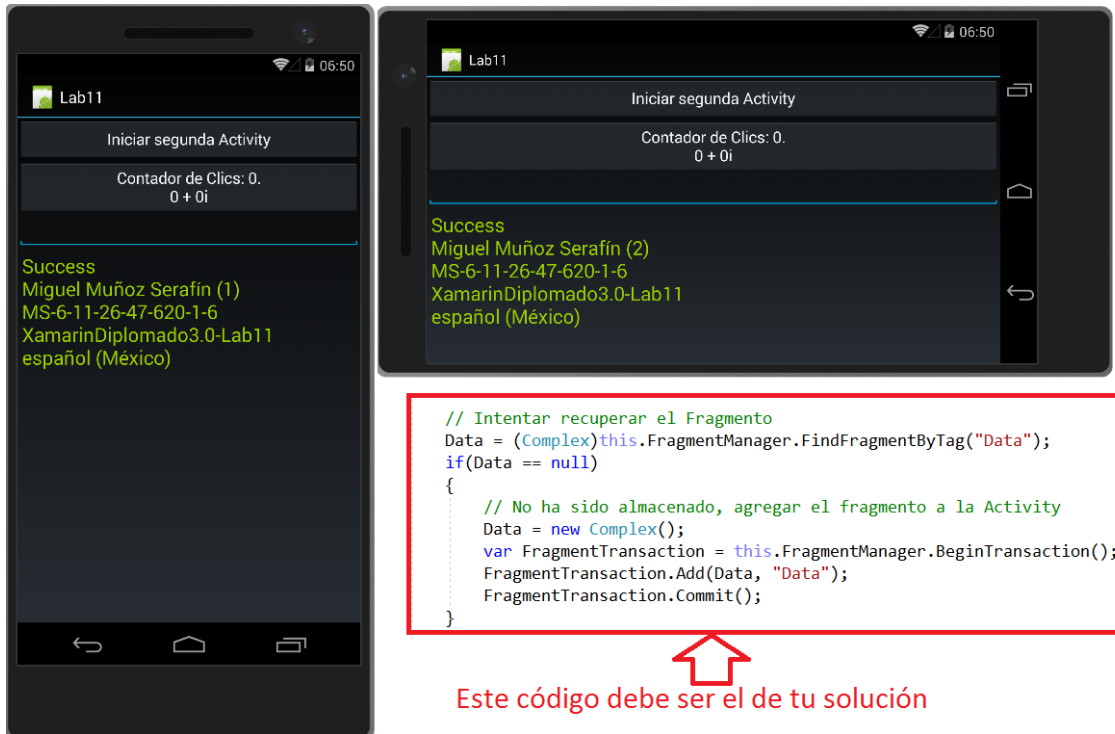
Puedes hacer caso omiso al mensaje.



Para verificar que hayas realizado tu actividad, sube una imagen mostrando lo siguiente:

- La imagen de la pantalla principal en modo vertical.
- La imagen de la pantalla principal en modo horizontal.
- El código C# donde muestres como se hace persistente la información de validación de la actividad.

La imagen deberá ser similar a la siguiente:



La evidencia donde debes subir la imagen es “Imagen laboratorio 11”.

Los puntos que se evalúan en la evidencia son los siguientes:

- La imagen debe mostrar la pantalla principal en modo vertical con los datos de validación.
- La imagen debe mostrar la pantalla principal en modo horizontal con los datos de validación.
- La imagen debe mostrar el código C# donde muestre como se hace la persistencia de la información de autenticación.

Si encuentras problemas durante la realización de este laboratorio, puedes solicitar apoyo en los grupos de Facebook siguientes:

<https://www.facebook.com/groups/iniciandoconxamarin/>

<https://www.facebook.com/groups/xamarindiplomadoitc/>



Resumen

En este laboratorio examinamos el ciclo de vida de las *Activities* y explicamos la responsabilidad que tiene una *Activity* durante cada uno de los cambios de estado para poder crear aplicaciones confiables y consientes del ambiente operativo Android.

En este laboratorio también adquiriste los conocimientos sobre el ciclo de vida de la *Activity* necesarios para persistir los datos de estado.

¿Qué te pareció este laboratorio?

Comparte tus comentarios en twitter y Facebook utilizando el hashtag **#XamarinDiplomado**.