Laboratorio

Trabajando con Tareas

Versión: 1.0.0
Agosto de 2017

CONTENIDO

Introducción

La clase ***Task*** se encuentra en el corazón de ***Task Parallel Library* (*TPL*)** del .NET Framework. Como su nombre lo indica, utilizamos la clase *Task* para representar una *tarea* o, en otras palabras, una unidad de trabajo. La clase *Task* nos permite realizar múltiples *tareas* al mismo tiempo, cada una en un hilo diferente. Detrás de escena, *TPL* administra el *Grupo de hilos* (*Thread pool*) y asigna tareas a los hilos. Podemos implementar funcionalidad multitarea sofisticada utilizando la biblioteca *Task Parallel* para encadenar tareas, pausar tareas, esperar a que las tareas se completen antes de continuar y realizar muchas otras operaciones.

En este laboratorio presentaremos una introducción al manejo de *tareas* con Visual C#.

Objetivos

Al finalizar este laboratorio, los participantes serán capaces de:

- Crear tareas.
- Utilizar expresiones lambda para crear tareas.
- Controlar la ejecución de las tareas.
- Devolver el valor de una tarea.
- Cancelar tareas de larga duración.

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 diseñados con Visual Studio Enterprise 2017 sobre una máquina con Windows 10 Pro.

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

Ejercicio 1: Creando Tareas

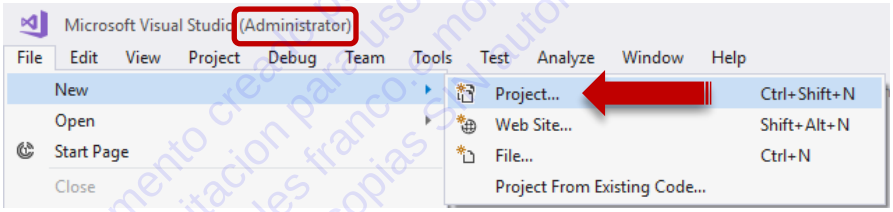
En este ejercicio, crearás una aplicación con interfaz de usuario gráfica para ejemplificar la creación de tareas utilizando la clase *Task*, *Delegados* y *Expresiones lambda*.

Tarea 1. Crear una aplicación gráfica.

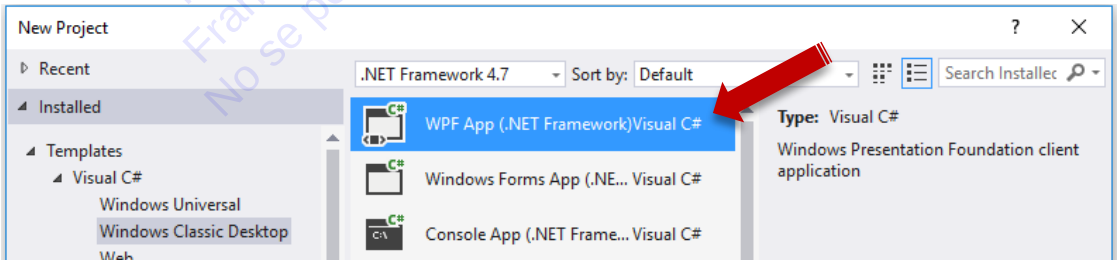
En esta tarea, crearás una aplicación gráfica *Windows Presentation Foundation* que será utilizada para realizar cada una de las tareas de este laboratorio. Si lo deseas, puedes crear alguna otra aplicación gráfica como, por ejemplo, Xamarin.Android o Xamarin.iOS.

Realiza los siguientes pasos para crear una aplicación ***Windows Presentation Foundation***.

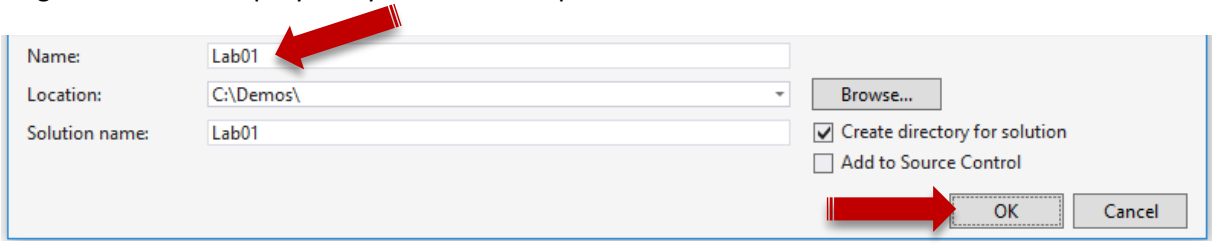
1. Abre Visual Studio en el contexto del Administrador.
2. Selecciona la opción ***File*** > ***New*** > ***Project***.



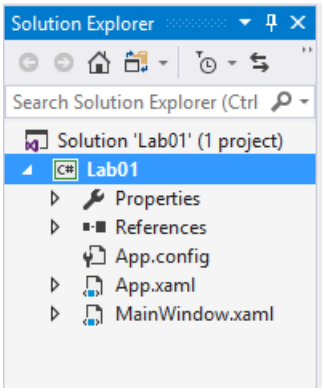
3. En la ventana ***New Project*** selecciona la plantilla ***WPF App (.NET Framework)*** para crear una nueva aplicación ***Windows Presentation Foundation***.



4. Asigna un nombre al proyecto y haz clic en ***OK*** para crear la solución.



El *Explorador de Soluciones* será similar al siguiente.



5. Haz doble clic sobre el archivo ***MainWindow.xaml*** para abrirlo en el diseñador.

6. Agrega el siguiente código en la vista XAML dentro del elemento ***Grid*** para definir un control ***Label***.

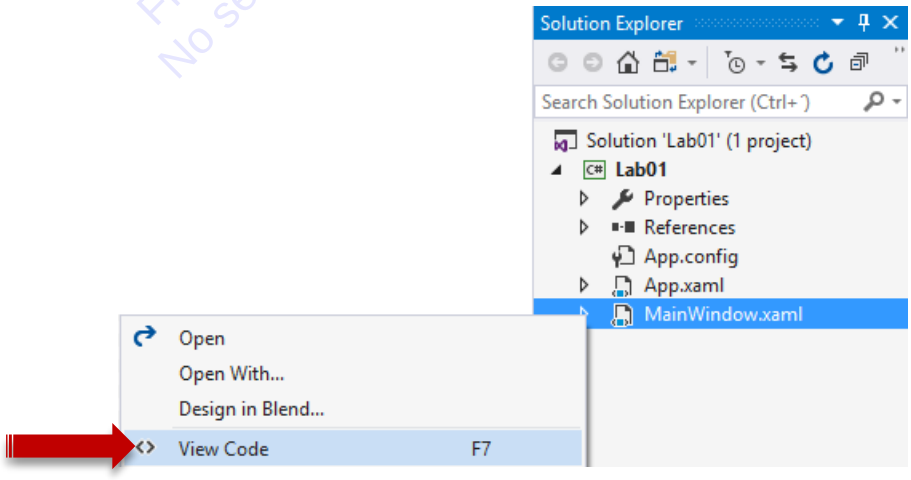
```
<Label x:Name="Messages" />
```

7. Guarda los cambios.

Tarea 2. Crear una tarea.

En esta tarea utilizarás la clase ***Task*** para crear tareas utilizando *Delegados* y métodos anónimos.

1. Selecciona la opción ***View Code*** del menú contextual del archivo ***MainWindows.xaml***.



2. Agrega el siguiente código para definir el método ***CreateTask***. En este método agregaremos el código que realizaremos en esta tarea.

```
void CreateTask()
{

}
```

3. Agrega el siguiente código en el método ***CreateTask*** para definir una variable que represente una tarea. Utilizamos la clase *Task* para representar una tarea o, en otras palabras, una unidad de trabajo.

```
void CreateTask()
{
    Task T;
}
```

Un objeto *Task* ejecuta un bloque de código.

4. Agrega el siguiente método a la clase ***MainWindow***. Este método contiene un bloque de código que podemos asociar a una tarea. El método permite mostrar al usuario un cuadro de dialogo con un mensaje.

```
void ShowMessage()
{
    MessageBox.Show("Ejecutando el método ShowMessage");
}
```

Para que la tarea pueda ejecutar un método, podemos crear un delegado ***Action*** que envuelva a ese método.

5. Agrega el siguiente código en el método *CreateTask* para envolver al método *ShowMessage* en un delegado ***Action***.

```
void CreateTask()
{
    Task T;
    var Code = new Action(ShowMessage);

}
```

El bloque de código que debe ejecutar la tarea puede ser especificado como un parámetro del constructor.

6. Agrega el siguiente código para crear una tarea especificando en su constructor el bloque de código que podrá ejecutar.

```
void CreateTask()
{
    Task T;
    var Code = new Action(ShowMessage);

    T = new Task(Code);
}
```

Un delegado proporciona un mecanismo para hacer referencia a un bloque de código o a un método. La clase ***Action*** es un tipo de la biblioteca de clases del .NET Framework que nos permite convertir un método en un delegado. El método no puede devolver un valor, pero puede tomar parámetros.

La biblioteca de clases del .NET Framework también proporciona la clase ***Func***, que permite definir un delegado que puede devolver un resultado.

Utilizar un delegado ***Action*** requiere que hayamos definido un método que contenga el código que deseamos ejecutar en una tarea. Sin embargo, si el único propósito de este método es proporcionar la lógica para una tarea y no se vuelve a utilizar en otro lado, podemos encontrarnos creando y teniendo que recordar los nombres de una gran cantidad de métodos. Esto hace que el mantenimiento sea más difícil.

Un enfoque más común es utilizar *métodos anónimos*. Un método anónimo es un método sin nombre. Proporcionamos el código de un método anónimo directamente (inline), en el punto que necesitamos utilizarlo. Podemos utilizar la palabra reservada ***delegate*** para convertir un método anónimo en delegado.

7. Agrega el siguiente código para crear una tarea mediante la utilización de un delegado anónimo.

```
Task T2 = new Task(delegate
{
    MessageBox.Show("Ejecutando una tarea en un delegado anónimo");
});
```

Tarea 3. Utilizar expresiones Lambda para crear tareas.

Una expresión lambda es una sintaxis abreviada que proporciona una manera simple y concisa para definir métodos anónimos como delegados. Cuando creamos una instancia *Task*, podemos utilizar una expresión lambda para definir al delegado que deseamos asociar con la tarea.

En esta tarea, conocerás la sintaxis de las expresiones lambda y la forma de utilizar expresiones lambda para crear tareas.

1. Agrega el siguiente código al final del método *CreateTask*.

```
Task T3 = new Task();
```

Puedes notar que el constructor de la clase *Task* requiere de un bloque de código a ejecutar, este bloque de código puede ser un delegado.

2. Modifica el código anterior para especificar un delegado que invoque a un método con nombre.

```
Task T3 = new Task(delegate { ShowMessage(); });
```

Si deseamos que el delegado invoque un método con nombre o una simple línea de código, podemos utilizar una expresión lambda. Una expresión lambda proporciona una notación abreviada para definir un delegado que puede tomar parámetros y devolver un resultado. Una expresión lambda tiene la siguiente forma:

(*Parámetros de entrada*) **=>** *Expresión*

En este caso:

- El operador lambda, **=>**, se lee como "va hacia".
- La parte izquierda del operador lambda incluye las variables que deseamos pasarle a la expresión. Si no requerimos ninguna entrada, por ejemplo, si estamos invocando un método que no toma parámetros, incluimos paréntesis vacíos () en el lado izquierdo del operador lambda.
- El lado derecho del operador lambda incluye la expresión que deseamos evaluar. Alternativamente, podemos invocar a un método del lado derecho del operador lambda.

3. Modifica la línea de código anterior para especificar ahora una expresión lambda que representa a un delegado que invoca a un método con nombre (método no anónimo).

```
Task T3 = new Task( () => ShowMessage() );
```

4. Agrega el siguiente código que ejemplifica el uso de una expresión lambda que representa a un delegado que invoca a un método anónimo.

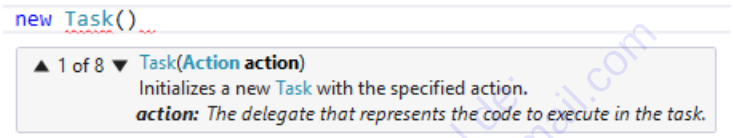
```
Task T4 = new Task(() => MessageBox.Show("Ejecutando la Tarea 4"));
```

Una expresión lambda puede ser una simple expresión o llamada a función como se muestra en el ejemplo anterior, o puede referenciar a un bloque de código más largo. Para hacer esto, especificamos el código entre llaves (como el cuerpo de un método) a la derecha del operador lambda.

5. Agrega el siguiente código para ejemplificar el uso de expresiones lambda para representar un bloque de código.

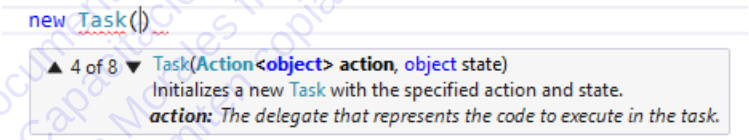
```
Task T5 = new Task(() =>
{
    DateTime CurrentDate = DateTime.Today;
    DateTime StartDate = CurrentDate.AddDays(30);
    MessageBox.Show($"Tarea 5. Fecha calculada:{StartDate}");
});
```

Algo importante que podemos notar es que algunas de las sobrecargas del constructor de la clase *Task* como la del código anterior, espera un tipo *Delegate* ***Action***.



El tipo *Delegate* ***Action*** encapsula un método que no tiene parámetros y no devuelve un valor, por lo cual, debemos poner paréntesis vacíos a la izquierda del operador lambda, tal y como se muestra en el ejemplo anterior.

El constructor de la clase *Task*, también tiene sobrecargas que aceptan un tipo *Delegate* ***Action<object>***.



El parámetro ***state*** representa el valor del parámetro ***object*** del delegado ***Action<object>***.

6. Agrega el siguiente código para ejemplificar el uso de expresiones lambda que utilizan parámetro. El código utiliza una sintaxis del constructor de la clase *Task* que acepta un parámetro.

```
Task T6 = new Task((message) =>
MessageBox.Show(message.ToString()), "Expresión lambda con parámetros.");
```

En este caso, al ejecutar la tarea, el parámetro ***message*** tendrá el valor "***Expresión lambda con parámetros.***".

A medida que los delegados se vuelven más complejos, las expresiones lambda ofrecen una forma mucho más concisa y fácil de entender para expresar delegados anónimos y métodos anónimos. Debido a esto, las expresiones lambda son el enfoque recomendado cuando trabajamos con tareas.

Para obtener más información acerca de expresiones lambda, se recomienda consultar el siguiente enlace:

Lambda Expressions (C# Programming Guide)
http://go.microsoft.com/fwlink/?LinkID=267836

Ejercicio 2: Controlando la Ejecución de las Tareas

La biblioteca ***Task Parallel***, ofrece diversos enfoques que pueden ser utilizados para iniciar tareas. Existen también diferentes maneras en que podemos pausar la ejecución del código hasta que una o más tareas se hayan completado.

En este ejercicio conocerás la forma de iniciar la ejecución de una tarea y los distintos mecanismos que podemos utilizar para esperar a que la ejecución de una tarea finalice.

Tarea 1. Iniciar una tarea.

Cuando el código de una aplicación inicia una tarea, la biblioteca ***Task Parallel***, asigna un hilo (*thread*) a la tarea y se empieza a ejecutar esa tarea. La tarea se ejecuta en un hilo independiente, por lo que el código no necesita esperar a que la tarea se complete. En vez de eso, la tarea y el código que invoca la tarea, continúan ejecutándose en paralelo. Veamos esto en acción.

1. Agrega el siguiente código al inicio del archivo ***MainWindow.xaml.cs*** para importar el espacio de nombres que contiene la clase ***Thread***.

   ```
   using System.Threading;
   ```

2. Agrega el siguiente código a la clase ***MainWindow*** para definir un método que agregue al contenido del control ***Messages*** el mensaje recibido como parámetro y el identificador del hilo de ejecución actual.

   ```
   void AddMessage(string message)
   {
       Messages.Content +=
           $"Mensaje: {message}, " +
           $"Hilo actual: {Thread.CurrentThread.ManagedThreadId}\n";
   }
   ```

3. Dentro del método *CreateTask*, agrega el siguiente código para definir una tarea que invoque al método ***AddMessage***.

   ```
   Task T7 = new Task(() => AddMessage("Ejecutando la tarea"));
   ```

   Cuando el código de la aplicación inicia una tarea, la biblioteca ***Task Parallel*** asigna un hilo a la tarea y esta se empieza a ejecutar. Para iniciar una tarea, podemos utilizar el método ***Start*** del objeto *Task*.

4. Agrega el siguiente código debajo de la instrucción anterior para iniciar la tarea.

```
Task T7 = new Task(() => AddMessage("Ejecutando la tarea"));
T7.Start();
```

La tarea se ejecutará en un hilo independiente, por lo que el código principal de la aplicación no necesita esperar a que la tarea sea completada. La tarea y el hilo principal se ejecutarán en paralelo.

5. Agrega el siguiente código debajo del código anterior para que el hilo principal invoque al método **AddMessage**.

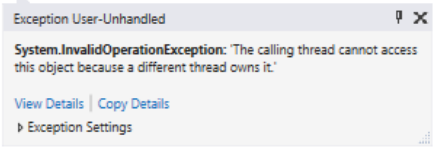
```
AddMessage("En el hilo principal");
```

6. Agrega el siguiente código en el constructor de la clase ***MainWindow*** para invocar al método *CreateTask*.

```
public MainWindow()
{
    InitializeComponent();
    CreateTask();
}
```

7. Ejecuta la aplicación. Podrás notar que se genera una excepción.

```
void AddMessage(string message)
{
    Messages.Content +=
        $"Mensaje: {message}, " +
        $"Hilo actual: {Thread.CurrentThread.ManagedThreadId}\n";
}
```



Esta excepción es generada cuando un *thread* distinto al *thread* principal intenta actualizar los elementos de la interfaz de usuario, en este caso el hilo distinto fue el de la tarea lanzada por el hilo principal.

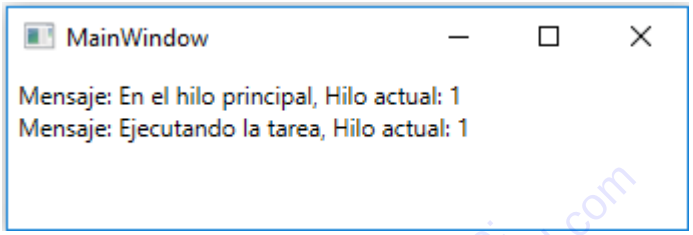
En aplicaciones WPF puedes resolver este problema a través del objeto ***Dispatcher***.

8. Modifica el código del método ***AddMessage*** para permitir que un hilo distinto al hilo principal pueda modificar los elementos de la interfaz de usuario.

```
void AddMessage(string message)
{
    this.Dispatcher.Invoke(() =>
    {
```

```
            Messages.Content +=
                $"Mensaje: {message}, " +
                $"Hilo actual: {Thread.CurrentThread.ManagedThreadId}\n";
        });
    }
```

9. Ejecuta nuevamente la aplicación. Puedes notar que ahora se muestran los mensajes correspondientes.



Puedes notar que el hilo principal es el que modifica los elementos de la interfaz de usuario (Hilo actual: 1).

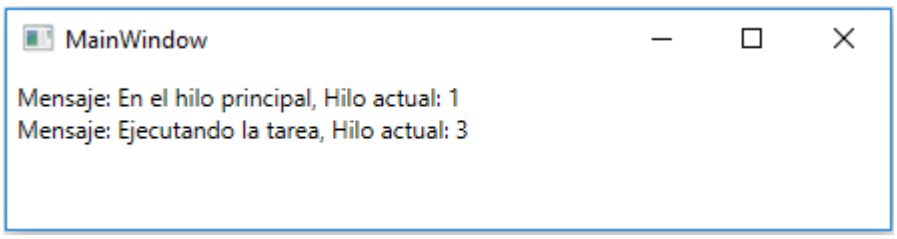
10. Regresa a Visual Studio y detén la ejecución.

11. Modifica el código del método ***AddMessage*** para que muestre el identificador del *thread* que invoca al método ***AddMessage***.

```
void AddMessage(string message)
{
    int CurrentThreadID = Thread.CurrentThread.ManagedThreadId;

    this.Dispatcher.Invoke(() =>
    {
        Messages.Content +=
            $"Mensaje: {message}, " +
            $"Hilo actual: {CurrentThreadID}\n";
    });
}
```

12. Ejecuta nuevamente la aplicación. Puedes notar que ahora se muestra el identificador del thread del hilo principal y el del hilo alterno.



En la imagen anterior, también puedes notar que, en este caso, el hilo principal fue ejecutado antes que el hilo alterno.

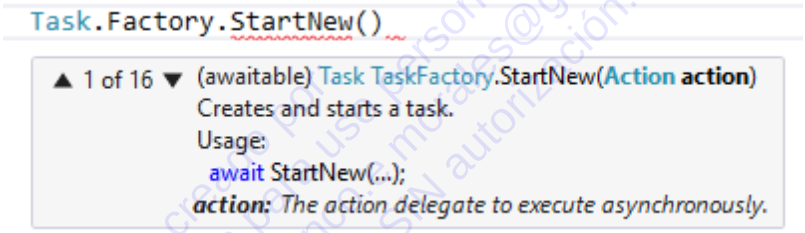
Tarea 2. Iniciar una tarea con la clase TaskFactory.

Alternativamente, podemos utilizar la clase estática ***TaskFactory*** para crear y poner en cola una tarea con una sola línea de código. La clase *TaskFactory* es expuesta a través de la propiedad estática ***Factory*** de la clase *Task*.

1. Agrega el siguiente código al final del método *CreateTask*.

```
var T8 =
    Task.Factory.StartNew(() => AddMessage("Tarea iniciada con TaskFactory"));
```

El método ***Task.Factory.StartNew*** es ampliamente configurable y tiene distintas sobrecargas.

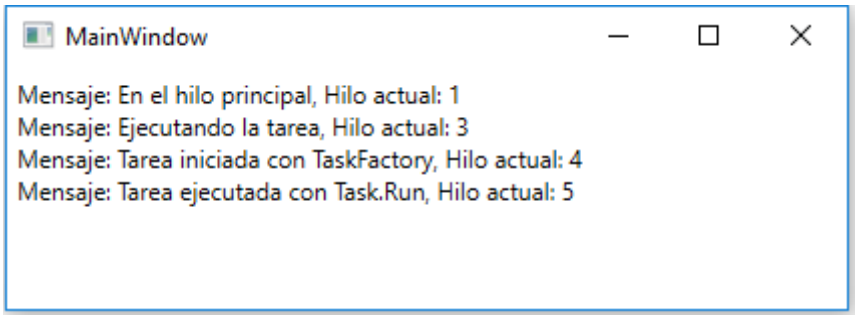


Si simplemente deseamos ejecutar algún código sin utilizar opciones adicionales de ejecución, podemos utilizar el método estático ***Task.Run*** como un atajo del método ***Task.Factory.StartNew***.

2. Agrega el siguiente código para ejemplificar el uso del método ***Task.Run***.

```
var T9 = Task.Run(() => AddMessage("Tarea ejecutada con Task.Run"));
```

3. Ejecuta la aplicación. Puedes notar los mensajes de cada tarea.



Tarea 3. Esperar la ejecución de las tareas.

En algunos casos, es posible que necesitemos interrumpir la ejecución de código hasta que se haya completado una determinada tarea.

Normalmente esto se hace si el código depende del resultado de una o más tareas, o si se tienen que manejar las excepciones que una tarea puede producir.

En esta tarea conocerás los distintos mecanismos que podemos utilizar para esperar a que la ejecución de una tarea finalice: ***Task.Wait***, ***Task.WaitAll*** y ***Task.WaitAny***.

Task.Wait

Uno de los mecanismos que podemos utilizar para para esperar a que la ejecución de una tarea finalice es utilizando el método *Task.Wait*.

Si queremos esperar a que una sola tarea complete su ejecución, utilizamos el método ***Task.Wait***.

1. Agrega el siguiente método a la clase ***MainWindow***. El método envía una cadena de texto a la ventana ***Output*** de *Visual Studio*.

```
void WriteToOutput(string message)
{
    System.Diagnostics.Debug.WriteLine(
        $"Mensaje: {message}, " +
        $"Hilo actual: {Thread.CurrentThread.ManagedThreadId}");
}
```

2. Agrega el siguiente código al método *CreateTask* para definir la ejecución de una tarea.

```
var T10 = Task.Run(() =>
{
    WriteToOutput("Iniciando tarea 10...");
    // Simular un proceso de dura 10 segundos
    Thread.Sleep(10000); // El hilo es suspendido por 10000 milisegundos
    WriteToOutput("Fin de la tarea 10.");
});
```

3. Agrega el siguiente código para que el hilo principal notifique que está esperando a que la tarea termine su ejecución.

```
WriteToOutput("Esperando a la tarea 10.");
```

4. Agrega el siguiente código para esperar a que la tarea finalice su ejecución.

```
T10.Wait();
```

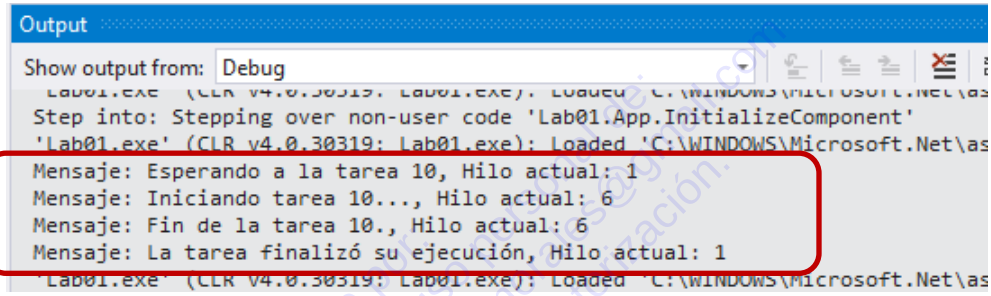
5. Agrega el siguiente código para que el hilo principal notifique que la tarea ha finalizado su ejecución.

```
WriteToOutput("La tarea finalizó su ejecución");
```

6. Ejecuta la aplicación.

7. Selecciona la opción ***View*** > ***Output*** de la barra de menús de Visual Studio para mostrar la ventana *Output*.

8. Puedes notar los siguientes mensajes.



Puedes notar que el hilo principal (Hilo actual: 1) detiene su ejecución y espera a que la tarea (Hilo actual: 6) termine su ejecución tal y como lo esperábamos.

¿Qué sucede si cambias las llamadas al método ***WriteToOutput*** por ***AddMessage*** y ejecutas la aplicación?

¿Se muestran los mensajes en la pantalla sin ningún problema? ¿Sí? ¿No? ¿Por qué?

Task.WaitAll

Si queremos esperar a que múltiples tareas finalicen su ejecución, debemos agregar las tareas a un arreglo.

Para esperar a que todas las tareas finalicen su ejecución, utilizamos el método estático ***Task.WaitAll*** pasándole el arreglo de tareas como parámetro.

9. Agrega el siguiente método a la clase ***MainWindow***.

```
void RunTask(byte taskNumber)
{
    WriteToOutput($"Iniciando tarea {taskNumber}.");
```

```
    // Simular un proceso de dura 10 segundos
    Thread.Sleep(10000); // El hilo es suspendido por 10000 milisegundos
    WriteToOutput($"Finalizando tarea {taskNumber}.");
}
```

El código anterior será ejecutado por una tarea y mostrará el momento en que inicia y el momento en que finaliza la ejecución, simulando la ejecución de un proceso que dura 10 segundos.

10. Agrega el siguiente código para definir un método llamado ***RunTaskGroup***.

```
void RunTaskGroup()
{

}
```

11. Agrega el siguiente código dentro del método *RunTaskGroup* para definir un arreglo de tareas, cada una invocando al método *RunTask*.

```
Task[] TaskGroup = new Task[]
{
    Task.Run(() => RunTask(1)),
    Task.Run(() => RunTask(2)),
    Task.Run(() => RunTask(3)),
    Task.Run(() => RunTask(4)),
    Task.Run(() => RunTask(5))
};
```

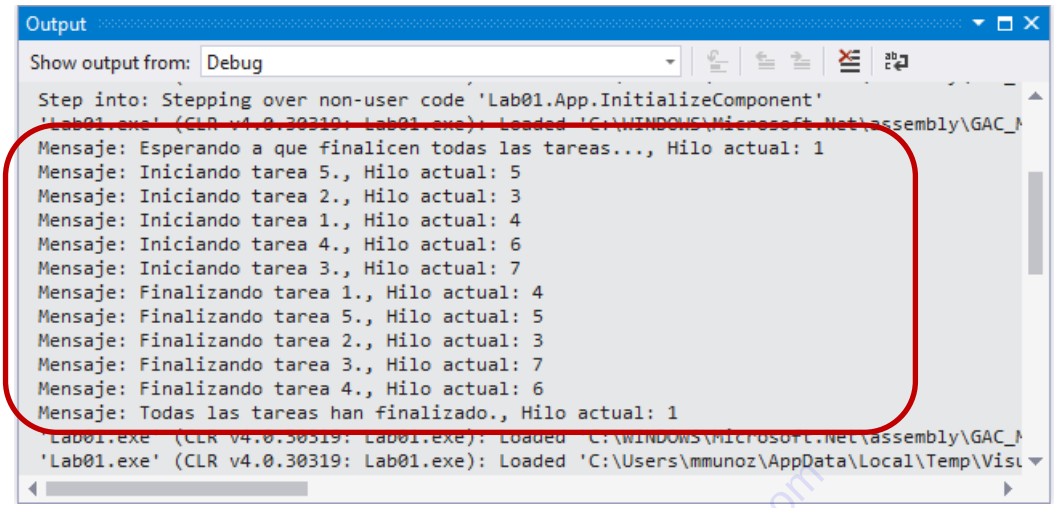
12. Agrega ahora el siguiente código para esperar a que todas las tareas finalicen su ejecución. Nota que estamos utilizando el método estático ***Task.WaitAll*** pasándole el arreglo de tareas como parámetro.

```
WriteToOutput("Esperando a que finalicen todas las tareas...");
Task.WaitAll(TaskGroup);
WriteToOutput("Todas las tareas han finalizado.");
```

13. Modifica el código del constructor ***MainWindow*** para invocar al método *RunTaskGroup* en lugar del método *CreateTask*.

```
public MainWindow()
{
    InitializeComponent();
    //CreateTask();
    RunTaskGroup();
}
```

14. Ejecuta la aplicación. La ventana ***Output*** mostrará algo similar a lo siguiente.



Puedes notar que no hay un orden de ejecución de cada tarea, sin embargo, el mensaje "***Todas las tareas han finalizado…***" se muestra después de que todas las tareas del arreglo finalizaron. En otras palabras, con ***Task.WaitAll***, el hilo principal esperó a que todas las tareas del arreglo finalizaran su ejecución.

Task.WaitAny

Si queremos esperar a que al menos una tarea de un grupo de tareas finalice su ejecución, debemos agregar las tareas a un arreglo.

Para esperar a que al menos una de las tareas finalice su ejecución, utilizamos el método estático ***Task.WaitAny*** pasándole el arreglo de tareas como parámetro.

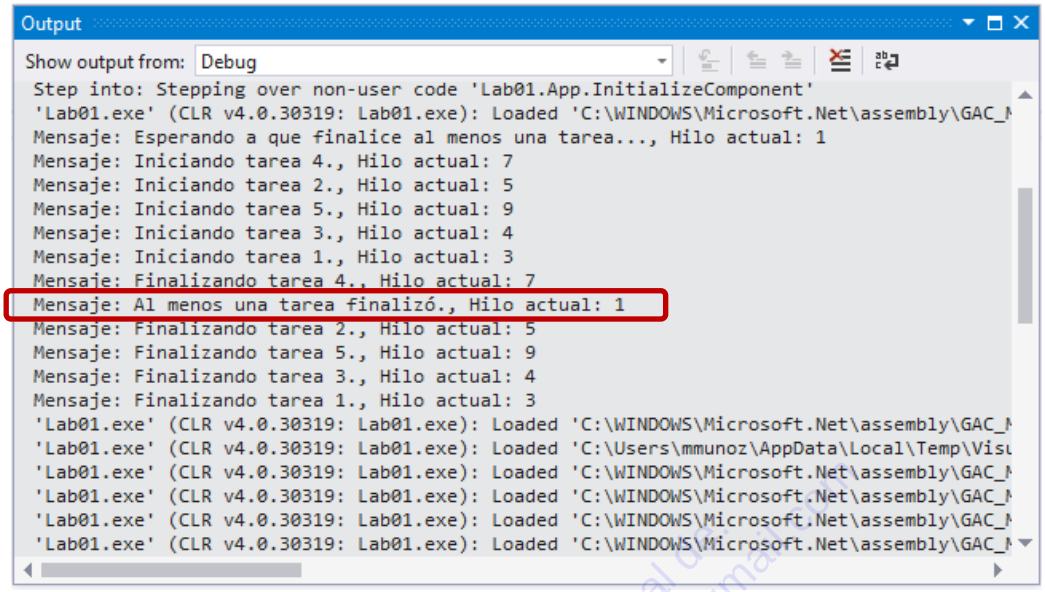
15. Modifica las siguientes líneas de código para esperar a que finalice al menos una tarea del arreglo.

```
WriteToOutput("Esperando a que finalicen todas las tareas...");
Task.WaitAll(TaskGroup);
WriteToOutput("Todas las tareas han finalizado.");
```

Las líneas anteriores deberán quedar modificadas de la siguiente forma.

```
WriteToOutput("Esperando a que finalice al menos una tarea...");
Task.WaitAny(TaskGroup);
WriteToOutput("Al menos una tarea finalizó.");
```

16. Ejecuta la aplicación. La ventana Output mostrará algo similar a lo siguiente.



Puedes notar que ahora el mensaje fue mostrado al finalizar al menos una tarea. En otras palabras, con ***Task.WaitAny***, el hilo principal esperó a que al menos una tarea del arreglo finalizara su ejecución.

Ejercicio 3: Retornando un valor de una Tarea

Para que las tareas puedan ser efectivas en escenarios del mundo real, tenemos que ser capaces de crear tareas que puedan devolver valores o resultados al código que inicie la tarea. La clase no genérica ***Task***, no permite hacer esto. Sin embargo, la biblioteca ***Task Parallel*** también incluye la clase genérica ***Task<TResult>*** que podemos utilizar cuando necesitemos devolver algún valor.

En este ejercicio, utilizarás la clase ***Task<TResult>*** para definir tareas que devuelvan valores.

Tarea 1. Crear una tarea que devuelva un valor.

Cuando creamos una instancia de ***Task<TResult>***, utilizamos el *Parámetro de Tipo* (*TResult*) para especificar el tipo del resultado que devolverá la tarea.

1. Agrega el siguiente código para definir un método llamado ***ReturnTaskvalue***.

```
void ReturnTaskValue()
{

}
```

2. Dentro del método *ReturnTaskvalue*, agrega el siguiente código que define una variable que permitirá ejecutar una tarea que devuelve un valor entero.

```
Task<int> T;
```

   Utilizamos el parámetro de tipo ***TResult*** (en este caso ***int***) para especificar el tipo de resultado que devolverá la tarea.

3. Agrega el siguiente código para ejecutar una tarea que devuelve un valor entero.

```
T = Task.Run<int>();
```

   Cuando creamos una instancia de ***Task<TResult>***, utilizamos el parámetro de tipo (***TResult***) para especificar el tipo del resultado que devolverá la tarea. El constructor espera el código que será ejecutado y que debe devolver un valor entero.

4. Modifica la línea anterior para especificar el código que será ejecutado en la tarea. Este código devuelve un número aleatorio positivo menor a 1000.

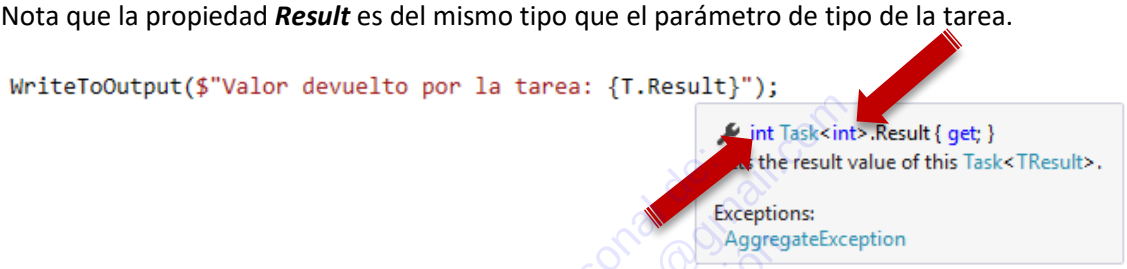
```
T = Task.Run<int>( () => new Random().Next(1000) );
```

La clase *Task<TResult>* expone una propiedad de sólo lectura llamada ***Result***. Después de que la tarea termine su ejecución, podemos utilizar la propiedad ***Result*** para recuperar el valor devuelto por la tarea.

5. Agrega el siguiente código para mostrar el valor devuelto por la tarea.

```
WriteToOutput($"Valor devuelto por la tarea: {T.Result}");
```

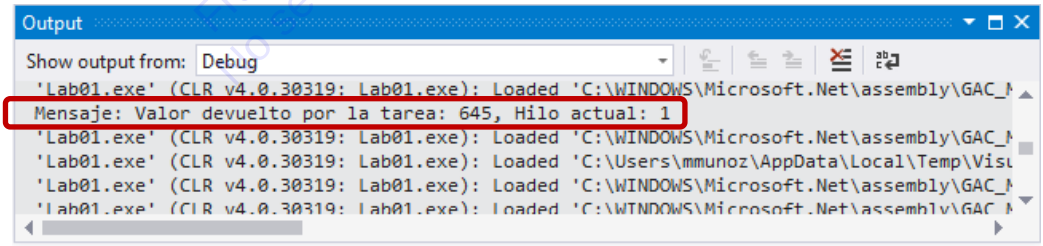
Nota que la propiedad ***Result*** es del mismo tipo que el parámetro de tipo de la tarea.



6. Modifica el código del constructor *MainWindow* para invocar sólo al método *ReturnTaskValue*.

```
public MainWindow()
{
    InitializeComponent();
    //CreateTask();
    //RunTaskGroup();
    ReturnTaskValue();
}
```

7. Ejecuta la aplicación. Puedes notar que se muestra el resultado devuelto por la tarea.



Es importante mencionar que cuando el hilo principal accede a la propiedad *Result* y la tarea aún no ha terminado su ejecución, el hilo principal esperará hasta que el resultado esté disponible antes de poder acceder a él.

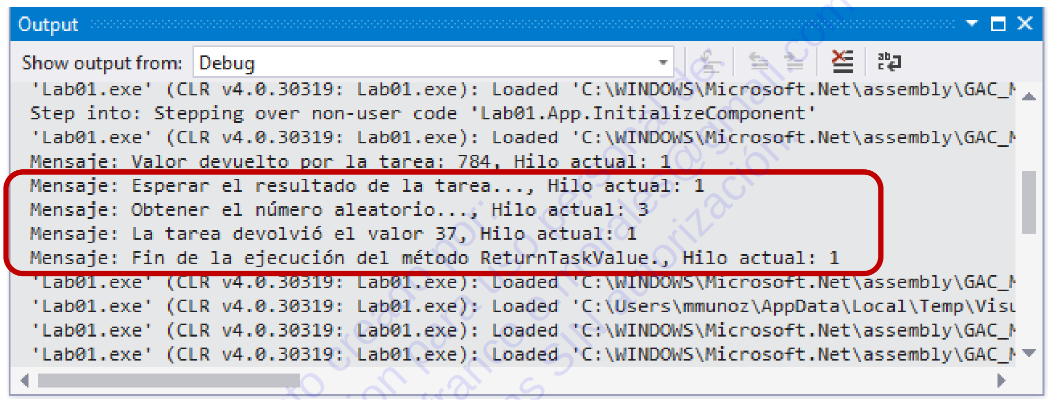
8. Agrega el siguiente código en el método *ReturnTaskValue* para definir una nueva tarea.

```
Task<int> T2 = Task.Run<int>(() =>
{
    WriteToOutput("Obtener el número aleatorio...");
    Thread.Sleep(10000); // Simular un proceso largo.
    return new Random().Next(1000);
});
```

9. Agrega ahora el siguiente código para mostrar el resultado de la tarea.

```
WriteToOutput("Esperar el resultado de la tarea...");
WriteToOutput($"La tarea devolvió el valor {T2.Result}");
WriteToOutput("Fin de la ejecución del método ReturnTaskValue.");
```

10. Ejecuta la aplicación. Observa el resultado de la tarea.



Puedes notar que el mensaje "***Fin de la ejecución…***" se muestra después de mostrar el resultado obtenido, en otras palabras, el hilo principal esperó a que el hilo secundario devolviera el resultado antes de continuar la ejecución.

Ejercicio 4: Cancelando Tareas de larga duración

Debido a su naturaleza asíncrona, las tareas se utilizan a menudo para realizar operaciones de larga duración sin bloquear el hilo de la interfaz de usuario. En algunos casos, podríamos querer dar a los usuarios la oportunidad de cancelar una tarea si ellos ya no desean esperar.

En este ejercicio, aprenderás a cancelar tareas de larga duración.

Tarea 1. Modificar la interfaz de usuario.

En esta tarea modificarás la interfaz de usuario de la aplicación para permitir lanzar y detener una tarea.

1. Haz doble clic sobre el archivo ***MainWindow.xaml*** para abrirlo en el diseñador.

2. Modifica el contenido del elemento ***Grid*** para agregar un botón que permita iniciar una tarea y un botón que permita detener la tarea. El código XAML será similar al siguiente.

```
<Grid>
    <Grid.RowDefinitions>
        <RowDefinition Height="*"/>
        <RowDefinition Height="auto"/>
    </Grid.RowDefinitions>
    <Label Grid.Row="0" x:Name="Messages" />
    <StackPanel Grid.Row="1" Orientation="Horizontal"
                HorizontalAlignment="Center" >
        <Button x:Name="StartTask" Content="Iniciar tarea"
                Margin="5" Padding="5"
                Click="StartTask_Click"/>
        <Button x:Name="CancelTask" Content="Cancelar tarea"
                Margin="5" Padding="5"
                Click="CancelTask_Click"/>
    </StackPanel>
</Grid>
```

3. Comenta el código dentro del constructor de la clase ***MainWindow***.

```
public MainWindow()
{
    InitializeComponent();
    //CreateTask();
    //RunTaskGroup();
    //ReturnTaskValue();
}
```

4. Agrega el siguiente código para definir los manejadores de evento clic de los botones agregados.

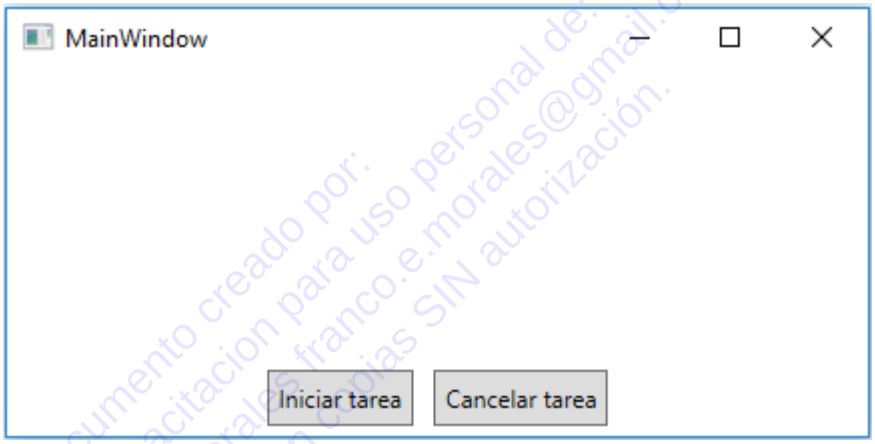
```
private void StartTask_Click(object sender, RoutedEventArgs e)
{

}

private void CancelTask_Click(object sender, RoutedEventArgs e)
{


}
```

5. Ejecuta la aplicación. La nueva interfaz será similar a la siguiente.



Tarea 2. Cancelar una tarea.

En algunos casos, podríamos querer dar a los usuarios la oportunidad de cancelar una tarea si ellos ya no desean esperar, sin embargo, sería peligroso interrumpir simplemente la tarea sobre demanda ya que esto podría dejar los datos de la aplicación en un estado desconocido. En lugar de eso, la biblioteca ***Task Parallel*** utiliza ***Tokens de Cancelación*** (***Cancellation Tokens***) que soportan un modelo de cancelación cooperativo.

1. Dentro de la clase ***MainWindow***, agrega el siguiente código a nivel de clase para definir una variable que almacenará un objeto ***CancellationTokenSource***, una variable para almacenar un *Token de Cancelación* y una variable para definir la tarea que podría ser cancelada.

```
CancellationTokenSource CTS;
CancellationToken CT;
Task LongRunningTask;
```

Examinemos la forma en que funciona un proceso de cancelación.

Creamos el origen del *Token de Cancelación*.

2. Agrega el siguiente código dentro del método **StartTask_Click**. Este método será ejecutado cuando el usuario haga clic en el botón ***Iniciar tarea***. El código genera el objeto origen para el *Token de Cancelación*.

```
CTS = new CancellationTokenSource();
```

Obtenemos el Token de Cancelación.

3. Agrega el siguiente código para obtener el *Token de Cancelación* desde la propiedad ***Token*** del objeto *CancellationTokenSource*.

```
CT = CTS.Token;
```

Creamos la tarea y pasamos el Token de Cancelación al método delegado.

4. Agrega el siguiente código para crear la tarea y pasar el *Token de Cancelación* como un argumento al método delegado.

```
LongRunningTask = Task.Run(() =>
{
    DoLongRunningTask(CT);
});
```

5. Agrega el siguiente código a la clase ***MainWindow*** para definir el método que contiene el código que será ejecutado en la tarea.

```
void DoLongRunningTask(CancellationToken ct)
{

}
```

En el hilo que creó la tarea, solicitamos la cancelación invocando al método *Cancel* del objeto *CancellationTokenSource*

6. Agrega el siguiente código dentro del método ***CancelTask_Click***. Este código será ejecutado cuando el usuario presione el botón ***Cancelar tarea***.

```
private void CancelTask_Click(object sender, RoutedEventArgs e)
{
    CTS.Cancel();
}
```

En el método de la tarea, podemos verificar el estatus del *Token de Cancelación* en cualquier momento.

7. Agrega el siguiente código dentro del método *DoLongRunningTask* para verificar el estatus del *Token de Cancelación* y determinar si se ha solicitado cancelar la tarea.

```
void DoLongRunningTask(CancellationToken ct)
{

    if(ct.IsCancellationRequested)
    {
        // Lógica para finalizar la tarea.
    }
}
```

En este punto, si se ha solicitado que la tarea sea cancelada, podemos finalizar la lógica de la tarea de forma apropiada, probablemente deshaciendo cualquier cambio que hayamos realizado en la tarea.

Por lo general, en el método de la tarea, podemos verificar en cualquier momento si se ha solicitado la cancelación de la tarea examinando el valor de la propiedad ***IsCancellationRequested*** del *Token de Cancelación*, Por ejemplo, si la lógica de la tarea itera sobre una colección, podríamos verificar la cancelación después de cada iteración.

8. Agrega el siguiente código al inicio del método *DoLongRunningTask* para simular un proceso que itera sobre una colección y verifica si se ha solicitado cancelar la tarea.

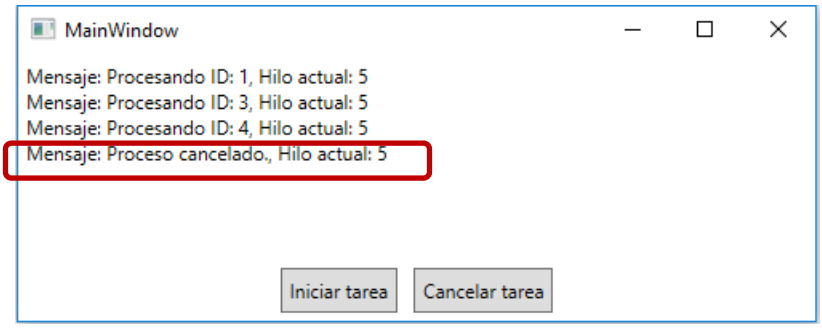
```
void DoLongRunningTask(CancellationToken ct)
{
    int[] IDs = { 1, 3, 4, 7, 11, 18, 29, 47, 76, 100 };
    for(int i=0; i < IDs.Length && !ct.IsCancellationRequested; i++)
    {
        AddMessage($"Procesando ID: {IDs[i]}");
        Thread.Sleep(2000); // Simular un proceso largo.
    }

    if(ct.IsCancellationRequested)
    {
        // Finalizar el procesamiento
        AddMessage("Proceso cancelado.");
    }
}
```

9. Ejecuta la aplicación.

10. Haz clic en el botón ***Iniciar tarea*** y espera a que se muestren al menos unos 3 mensajes.

11. Haz clic en el botón ***Cancelar tarea***. Podrás notar que el procesamiento de la tarea es cancelado y se muestra el mensaje correspondiente.



Este enfoque es apropiado si no necesitamos verificar si la tarea terminó completamente. Cada tarea expone una propiedad ***Status*** que permite monitorear el estado actual de la tarea durante su ciclo de vida. Si cancelamos una tarea y el método de la tarea retorna, el estatus de la tarea es establecido como ***RanToCompletion***. En otras palabras, la tarea no tiene forma de saber por qué retornó el método. El método pudo haber finalizado en respuesta a una solicitud de cancelación o simplemente pudo haber completado su lógica.

12. Agrega el siguiente código debajo del código que define al botón ***Cancelar Tarea***. Este código define un botón que nos permitirá mostrar el estatus de una tarea.

```
<Button x:Name="ShowStatus" Content="Mostrar estatus"
        Margin="5" Padding="5"
        Click="ShowStatus_Click"/>
```

13. En la clase ***MainWindow***, agrega el siguiente código del manejador del evento clic del botón anterior.

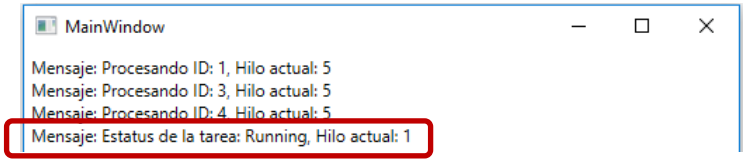
```
private void ShowStatus_Click(object sender, RoutedEventArgs e)
{
    AddMessage($"Estatus de la tarea: {LongRunningTask.Status}");
}
```

    El código permite mostrar el estado de la tarea.

14. Ejecuta la aplicación.

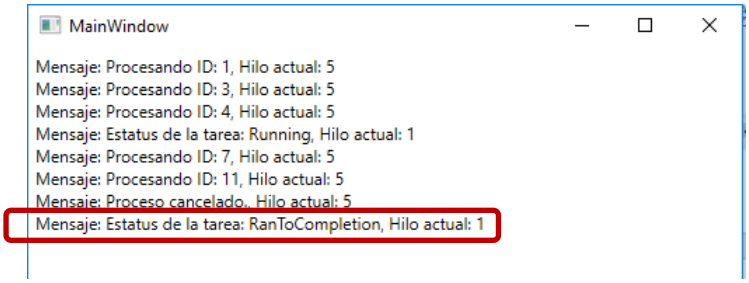
15. Haz clic en el botón ***Iniciar tarea*** y espera a que se muestren unos 3 mensajes.

16. Haz clic en el botón ***Mostrar estatus***. Se mostrará el estatus ***Running***.



17. Haz clic en el botón ***Cancelar tarea***.

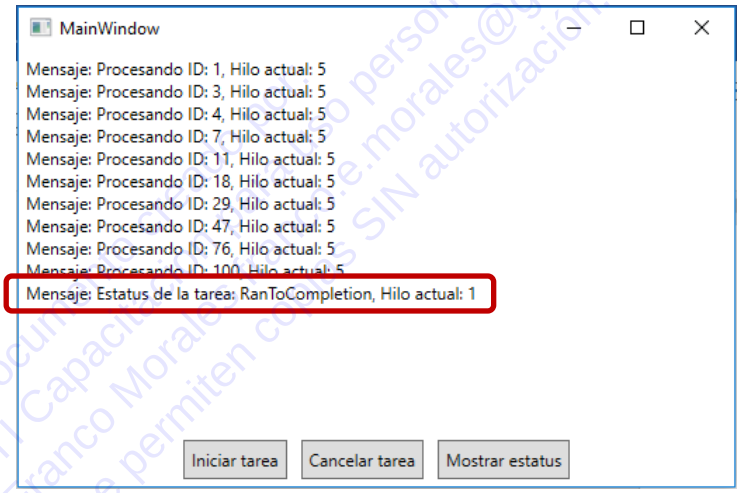
18. Haz clic en el botón ***Mostrar estatus***. Se mostrará el estatus ***RanToCompletion***.



19. Detén la aplicación y vuelve a ejecutarla.

20. Haz clic en el botón ***Iniciar tarea*** y deja que termine la ejecución de la tarea.

21. Haz clic en el botón ***Mostrar estatus***. Se mostrará el estatus ***RanToCompletion***.



Podemos notar que la tarea principal no tiene forma de saber por qué retornó el método. Pudo haber retornado en respuesta a una solicitud de cancelación o simplemente pudo haber completado su lógica.

Si deseamos cancelar una tarea y ser capaces de confirmar que se ha cancelado, es necesario pasar el *Token de Cancelación* como un argumento al constructor de la tarea además de pasarlo al método delegado.

22. Modifica el código que inicia la tarea para pasarle al constructor de la tarea el *Token de Cancelación*.

```
LongRunningTask = Task.Run(() =>
{
    DoLongRunningTask(CT);
```

```
}, CT);
```

En el método de la tarea, debemos verificar el estatus del *Token de Cancelación*. Si se ha solicitado la cancelación de la tarea, debemos disparar una excepción ***OperationCanceledException***.

Para verificar si una cancelación fue solicitada y de ser así, disparar la excepción ***OperationCanceledException***, invocamos al método ***ThrowIfCancellationRequested*** del *Token de Cancelación*.

23. Modifica el código del método *DoLongRunningTask* para invocar al método ***ThrowIfCancellationRequested*** del *Token de cancelación.* El método ***ThrowIfCancellationRequested*** lanzará la excepción ***OperationCanceledException*** en caso de que se haya solicitado la cancelación de la tarea.

```
void DoLongRunningTask(CancellationToken ct)
{
    int[] IDs = { 1, 3, 4, 7, 11, 18, 29, 47, 76, 100 };
    for (int i = 0; i < IDs.Length && !ct.IsCancellationRequested; i++)
    {
        AddMessage($"Procesando ID: {IDs[i]}");
        Thread.Sleep(2000); // Simular un proceso largo.
    }

    if (ct.IsCancellationRequested)
    {
        // Si es necesario, realizar algún proceso de limpieza
        AddMessage("Proceso cancelado.");
        ct.ThrowIfCancellationRequested();
    }
}
```

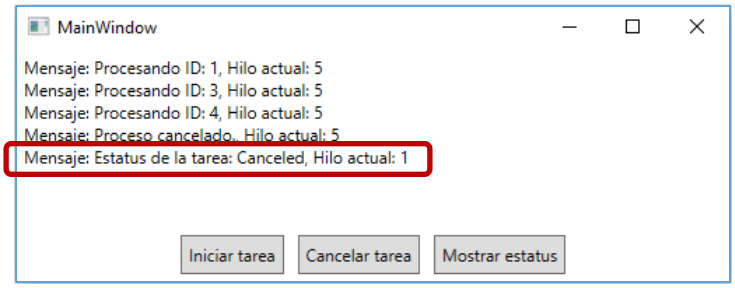
Cuando se produce una excepción *OperationCanceledException*, la biblioteca *Task Parallel* examina el *Token de Cancelación* para verificar si se ha solicitado una cancelación. De ser así, la biblioteca *Task Parallel* maneja la excepción *OperationCanceledException*, establece el estatus de la tarea en ***Canceled*** y lanza una excepción ***TaskCanceledException*** empaquetada en un objeto ***AggregateExeption***. En el código que creó la solitud de cancelación, debemos atrapar la excepción ***TaskCanceledException*** y manejar la cancelación apropiadamente.

24. Presiona las teclas ***CTRL-F5*** para ejecutar la aplicación sin depuración. Esto te permitirá que Visual Studio no detenga la ejecución al generarse la excepción *OperationCanceledException* generada por la llamada al método *ThrowIfCancellationRequested*.

25. Haz clic en el botón ***Iniciar tarea*** y deja que se muestren unos 3 mensajes.

26. Haz clic en el botón ***Cancelar tarea***.

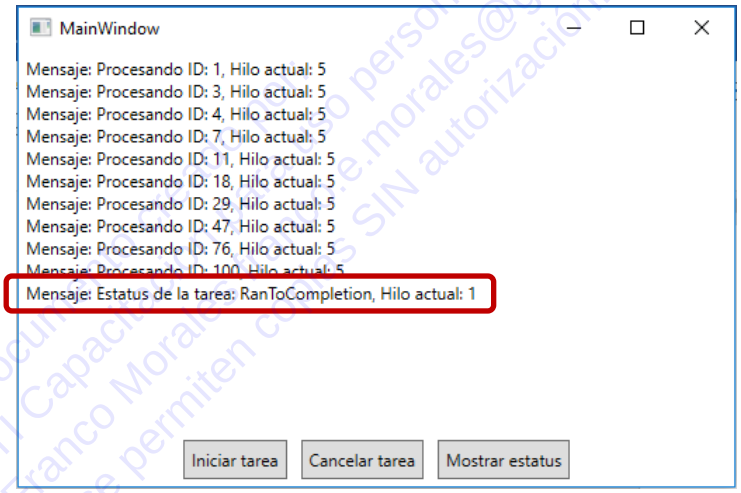
27. Haz clic en el botón ***Mostrar estatus***. Puedes notar que ahora el estatus es ***Canceled***.



28. Detén la aplicación y vuelve a ejecutarla.

29. Haz clic en el botón ***Iniciar tarea*** y deja que termine la ejecución de la tarea.

30. Haz clic en el botón ***Mostrar estatus***. Se mostrará el estatus ***RanToCompletion***.



Ahora la tarea principal ya puede determinar por qué retornó el método. Pudo haber retornado en respuesta a una solicitud de cancelación (estatus *Canceled*) o simplemente pudo haber completado su lógica (estatus *RanToCompletion*).

Para que podamos manejar la excepción ***TaskCanceledException*** en el hilo que lanzó la tarea, debemos esperar a que la tarea finalice. Hacemos esto invocando al método ***Task.Wait*** dentro de un bloque ***try*** y atrapamos una excepción ***AggregateException*** en el bloque *catch* correspondiente.

31. Modifica el código del método ***StartTask_Click*** para que la tarea ***LongRunningTask*** se ejecute dentro de otra tarea. La nueva tarea esperará a que ***LongRunningTask*** termine su ejecución y podrá manejar la excepción *TaskCanceledException.* De esta forma no bloquearemos el hilo de la interfaz de usuario.

```
private void StartTask_Click(object sender, RoutedEventArgs e)
{
    CTS = new CancellationTokenSource();
    CT = CTS.Token;

    Task.Run(() =>
    {
        LongRunningTask = Task.Run(() =>
        {
            DoLongRunningTask(CT);
        }, CT);
    });
}
```

32. Agrega el siguiente bloque *try-catch* para poder esperar la ejecución de la tarea y atrapar la excepción ***AggregateException***.

```
private void StartTask_Click(object sender, RoutedEventArgs e)
{
    CTS = new CancellationTokenSource();
    CT = CTS.Token;

    Task.Run(() =>
    {
        LongRunningTask = Task.Run(() =>
        {
            DoLongRunningTask(CT);
        }, CT);
        try
        {
            LongRunningTask.Wait();
        }
        catch (AggregateException ae)
        {
        }
    });
}
```

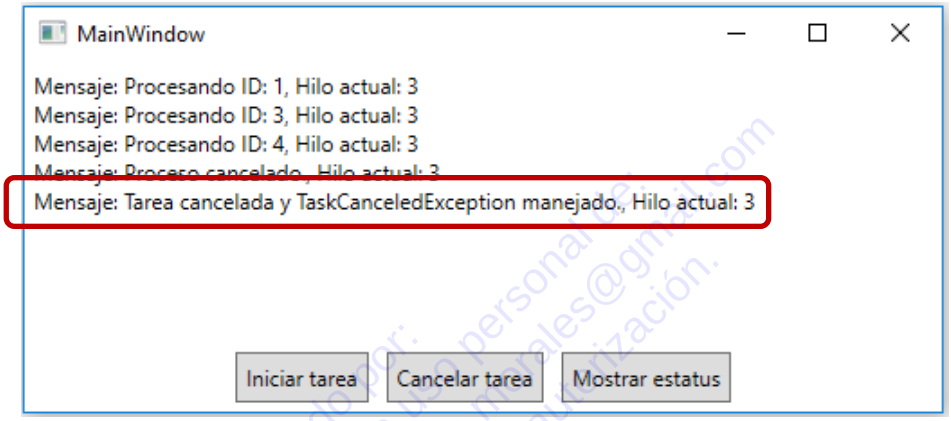

33. Agrega el siguiente código dentro del bloque ***catch*** para mostrar las excepciones generadas en la tarea ***LongRunningTask***.

```
foreach (var Inner in ae.InnerExceptions)
{
    if (Inner is TaskCanceledException)
    {
        AddMessage(
            "Tarea cancelada y TaskCanceledException manejado.");
    }
    else
    {
        // Procesamos excepciones distintas a cancelación.
        AddMessage(Inner.Message);
    }
}
```

34. Presiona las teclas ***CTRL-F5*** para ejecutar la aplicación sin depuración. Esto te permitirá que Visual Studio no detenga la ejecución al generarse la excepción *OperationCanceledException* generada por la llamada al método *ThrowIfCancellationRequested*.

35. Haz clic en el botón ***Iniciar tarea*** y deja que se muestren unos 3 mensajes.

36. Haz clic en el botón ***Cancelar tarea***. Puedes notar que la excepción ***TaskCanceledException*** ha sido manejada.



37. Regresa a Visual Studio y detén la ejecución.

Para obtener más información sobre la cancelación de tareas se recomienda consultar el siguiente enlace:

Task Cancellation
https://go.microsoft.com/fwlink/?LinkID=267837

Resumen

En este laboratorio aprendiste a crear y controlar la ejecución de tareas utilizando delegados, métodos anónimos y expresiones lambda. Conociste la forma de crear tareas que devuelven valores y a cancelar la ejecución de tareas de larga duración.