Apuntes de Fundamentos de Programación UNIDAD 7.

FRANCISCO RÍOS ACOSTA
Instituto Tecnológico de la Laguna
Blvd. Revolución y calzada Cuauhtémoc s/n
Colonia centro
Torreón, Coah; México
Contacto : friosam@prodigy.net.mx

I N D I C E.

7.1 Hacer-Mientras. 3

7.2 Mientras-Hacer. 4

7.3 Hacer-desde. 5

7.4 Ejercicios. 6
Ejercicio 7.4.1 6
Ejercicio 7.4.2 11
Ejercicio 7.4.3 12
Ejercicio 7.4.4 14
Ejercicio 7.4.5 19
Ejercicio 7.4.6 20
Ejercicio 7.4.7 23
Ejercicio 7.4.8 26
Ejercicio 7.4.9 29
Ejercicio 7.4.10 32
Ejercicio 7.4.11 38

7 Estructuras de repetición.

Las computadoras no servirian en gran parte si no se programaran para efectuar operaciones repetitivas. Imaginemos que desearamos obtener la suma de los primeros 10 número naturales. Una solución sería escribir directamente la asignación :

```
sumaPrimNat = 1+2+3+4+5+6+7+8+9+10;
```

Pero que tal si cambiamos el número de naturales a sumar a 100. La solución tal y como la escribimos anteriormente ya no nos satisfacería del todo, debido a que no sería cómodo escribir 1+2+3+4+...+97+98+99+100, menos si quisieramos la suma de los primeros 1000 números. Además no habría una interacción real con el usuario de este programa, a causa de que cada vez que se requiere de cambiar el número de naturales a sumar, el programador debería reescribir la asignación para la variable `sumaPrimNat`.
La solución para este problema es construir una operación repetitiva que vaya sumando cada número natural en la variable `sumaPrimNat`.

```
sumaPrimaNat = sumaPrimNat + num;
num = num + 1;
```

Operación repetitiva.

Esta operación repetitiva requiere de la inicialización de las 2 variables `sumaPrimNat` y `num`, a `0` y a `1` respectivamente.

```
sumaPrimNat = 0;
num = 1;
```

Inicialización.

```
sumaPrimaNat = sumaPrimNat + num;
num = num + 1;
```

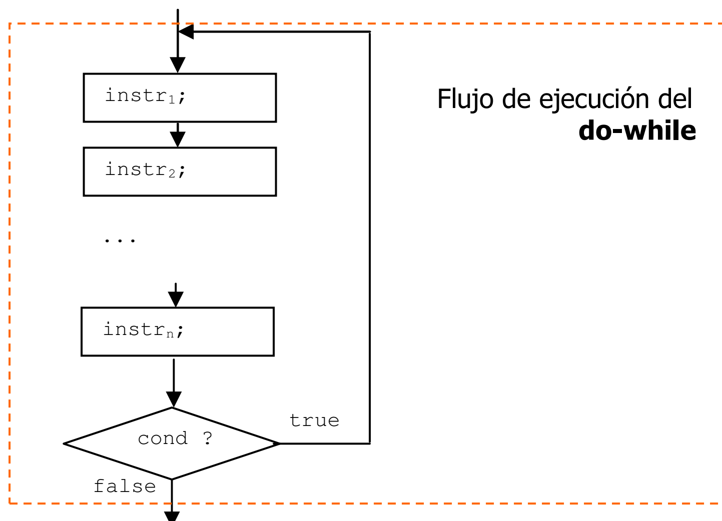
Operación repetitiva deberá realizarse mientras que `num` sea menor que **n.** Siendo **n** el número de naturales a sumar.

Los lenguajes de programación ofrecen sentencias que permiten realizar operaciones repetitivas o de iteración. En esta unidad veremos 3 de ellas para C# :

- do-while
- while
- for

7.1 Hacer-Mientras, do-while.

El **do-while** es una estructura de repetición que se utiliza cuando la operación repetitiva acepte ser realizada al menos una vez. El flujo de ejecución de esta sentencia es :



El grupo de instrucciones `instr1, instr2, ..., instrn` que conforman a la operación repetitiva, son ejecutadas al menos una vez, debido a que la condición es probada hasta el fin del bucle de sentencias. La operación repetitiva es vuelta a efectuar si la condición probada es true, de otra manera el flujo de ejecución sigue con la sentencia encontrada después del **do-while**.
En C# la sentencia **do-while** se escribe de diferentes formas, ya sea que la operación repetitiva tenga una sóla sentencia, ya sea que tenga mas de una sentencia.

```
do
  instr;
while (cond);
```

do-while para una sola sentencia.

```
do
{
  instr1;
  instr2;
  ...
} while (cond);
```

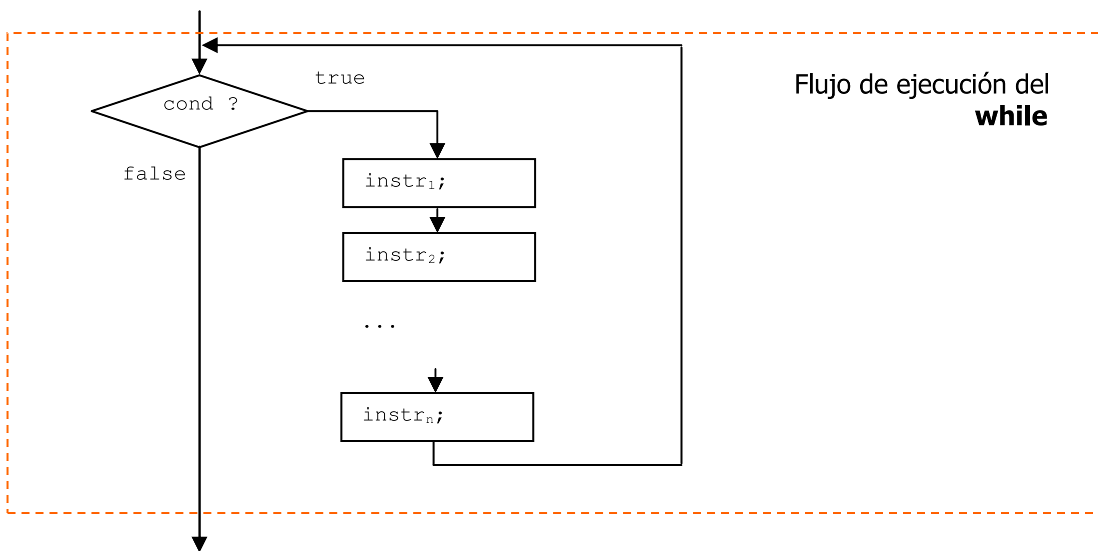
do-while para mas de una sentencia.

Las consideraciones sobre la sintaxis del **do-while** en C# son :

- La condición debe encerrarse entre paréntesis.
- La sentencia `do-while` debe terminar en el caracter (**;**). –después de la condición notemos el **;**-.
- Cuando la operación repetitiva consista de mas de una sentencia, debemos acotarlas con los caracteres llave { y }.

7.2 Mientras-Hacer, while.

Esta sentencia de iteración se caracteriza porque permite controlar que la operación repetitiva no se efectúe a menos que la condición se cumpla. Lo anterior se logra porque la condición es probada al inicio del bucle que contiene a la operación repetitiva, de manera que si en la primera prueba la condición es `false`, la operación repetitiva no se realizará ninguna vez. El flujo de ejecución permite ver mas claramente el comportamiento de la sentencia **while**.



Lo mismo que para el **do-while**, la sentencia **while** en C# se puede escribir de 2 formas : para una sola instrucción en la operación repetitiva y para mas de una sentencia.

```
while (cond)
  instr;
```

while con una sentencia.

```
while(cond)
{
  instr1;
  instr2;
  ...
}
```

while con dos o mas sentencias.

Las consideraciones para el **while** son :

- Usarla cuando la operación repetitiva en ocasiones deba no realizarse -ni una sóla vez-.
- La condición deberá encerrarse entre paréntesis.
- Cuando la operación repetitiva consista de mas de una sentencia, deberá encerrarse entre los caracteres llave { y }.

7.3 Hacer-Desde, for.

La sentencia **for** es ampliamente usada en programas que incluyan operaciones repetitivas, debido a su versatilidad. Consiste de 4 partes :

- Instrucciones de inicio.
- Prueba de condición.
- Cuerpo.
- Instrucciones de paso.

En C# el for se escribe como se indica para una sentencia en su cuerpo, y para mas de una sentencias :

```
for(InstrInicio;Cond;InstrPaso)
  Instr;
```

← **for** para 1 sentencia en su cuerpo.

```
for(InstrInicio;Cond;InstrPaso)
{
  Instr1;
  Instr2;
  ...
}
```

← **for** para 2 o mas sentencias en su cuerpo.

Las consideraciones sobre la sintaxis de la sentencia **for** que podemos tomar en cuenta son :

- La instrucción de inicio, la condición y la instrucción de paso deben ser separadas con el carater (**;**).
- Cuando existan mas de una instrucción de inicio, cada sentencia se debe separar por el caracter (**,**).
- Cuando existan mas de una instrucción de paso, cada sentencia se debe separar por el caracter (**,**).
- En caso que el cuerpo consista de 2 o mas sentencias, éstas deben ser encerradas entre llaves **{** y **}**.
- La instrucción de inicio es opcional, si no se escribe el encabezado debe codificarse así :
  ```
  for(;Cond;InstrPaso)
  ```
- Lo mismo puede decirse de la instrucción de paso la cual es también opcional.

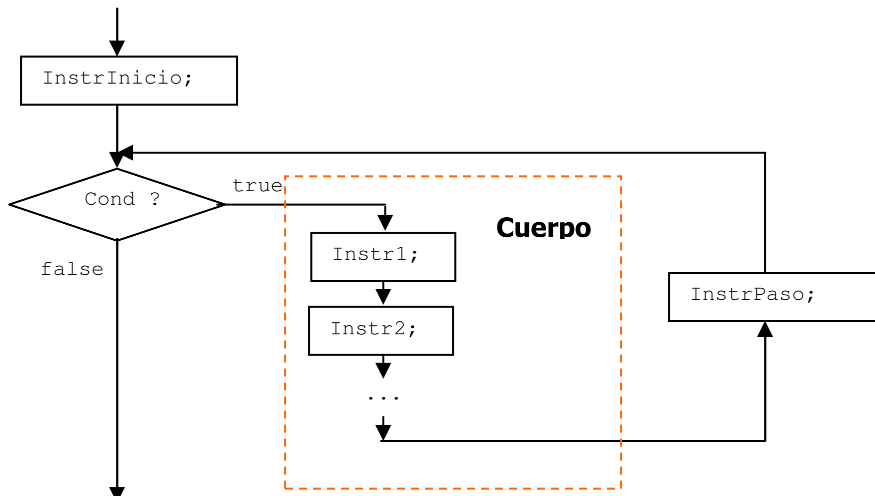
  ```
  for(InstrInicio;Cond;)
  ```

- Para el caso en que no exista ni instrucción de inicio ni de paso, el encabezado deberá escribirse como :

  ```
  for(;Cond;)
  ```

- La condición es una expresión que siempre deberá exisitir –no es opcional-.

El flujo de ejecución de la sentencia **for** es :



7.4 Ejercicios.

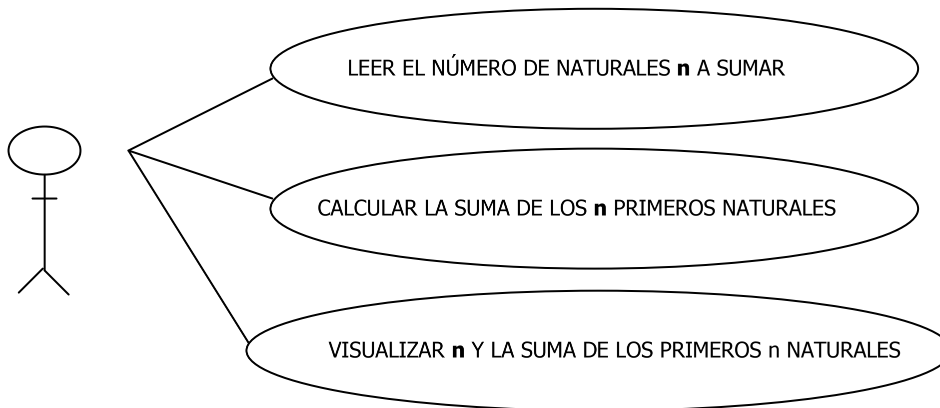
En esta sección resolveremos ejercicios que incluyen operaciones repetitivas codificadas con las sentencias **do-while**, **while** y **for**. Usaremos la metodología propuesta por *R.A. Francisco*, que especifica la utilización de un objeto perteneciente a una clase `Calculador` que será el encargado de llevar a cabo la operación repetitiva.
Los pasos de esta metodología *RAF* se enumeran a continuación :

- Definir la clase `Calculador`. Sus atributos serán los datos que se lean y/o se visualicen.
- Agregar los métodos que permitan la lectura de los atributos de la clase `Calculador` y el acceso –retorno- de dichos atributos.
- Definir el método en la clase `Calculador` que permita efectuar la operación repetitiva.
- Definir el objeto `oCalc` en la clase `Form1` de la aplicación Windows.

Iniciemos con los ejercicios para explicar el uso de la metodología *RAF*, además de la construcción de operaciones repetitivas usando las 3 sentencias de iteración explicadas en las secciones anteriores.

Ejercicio 7.4.1
Escribe una aplicación Windows C# que calcule la suma de los primeros **n** números naturales. **n** es un entero mayor que 0 y se lee. Visualizar el valor de **n** y la suma de los números pedidos.

El diagrama de casos de uso que representa las tareas a realizar en este problema es muy simple.



El diagrama de clases de acuerdo a la metodología RAF se compone de una clase `Calculador`. Los atributos de esta clase son : `n` y la `suma`. Recordemos que todo lo que es leído y lo que es visualizado lo definiremos como un atributo.

CALCULADOR
- _n : int - _suma : int
+ Leer(noNat:int) : void + RetN() : int + RetSuma() : int + EfecSuma() : void

Los métodos son muy claros en su comprensión : `Leer()` recibe el parámetro `noNat` que ha ingresado el usuario –cuántos naturales quiere sumar-, `RetN()` retorna el valor del número de naturales que se suman, `RetSuma()` devuelve el valor de la suma de los primeros `_n` números naturales, y `EfecSuma()` realiza la operación repetitiva que suma a los primeros `_n` números naturales y los deposita en el atributo `_suma`.

Ahora debemos iniciar un nuevo proyecto en C#, e inmediatamente definir la clase `Calculador` además de agregar la definición del objeto `oCalc` según la metodología que estamos empleando. La figura #7.4.1 muestra la interfase gráfica de usuario propuesta.

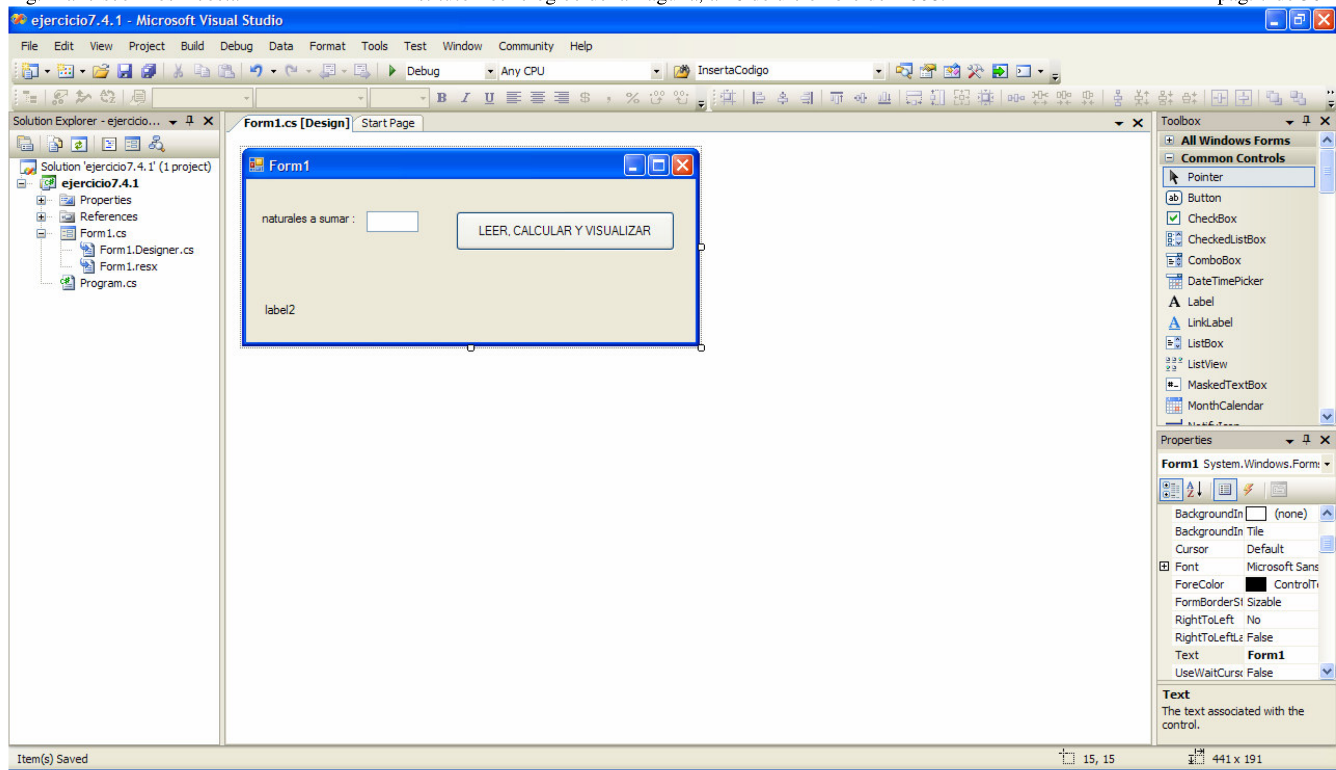


Fig. No. 7.4.1 Interfase gráfica de usuario para el ejercicio 7.4.1.

La definición de la clase `Calculador` la realizamos de acuerdo al diagrama de clases antes mencionado. Los atributos son `_n` y `_suma`. Agregamos pues una nueva clase al proyecto con el menú Project | Add Class, el código en la clase es :

```
class Calculador
{
    private int _n;
    private int _suma;

    public void Leer(int noNat)
    {
       _n = noNat;
    }
    public int RetN()
    {
       return _n;
    }
    public int RetSuma()
    {
       return _suma;
    }
    public void EfecSuma()
    {
       // operación repetitiva
    }
}
```

La operación repetitiva la haremos posteriormente.

Definimos ahora al objeto `oCalc` en la clase `Form1` de nuestra aplicación Windows.

```
public partial class Form1 : Form
{
    Calculador oCalc = new Calculador();
    public Form1()
    {
       InitializeComponent();
    }
}
```

Definición del objeto `oCalc`.

Lo que resta es codificar los mensajes al objeto `oCalc`, que le dan vida a la aplicación. En nuestra interfase gráfica sólo hemos manejado un botón `button1` para hacer las tareas de lectura, cálculo de la suma y la visualización de los resultados. El componente `label2` servirá para visualizar los resultados. Así que tecleemos el método `button1_Click()` de acuerdo al siguiente código :

```
private void button1_Click(object sender, EventArgs e)
{
    oCalc.Leer(Convert.ToInt32(textBox1.Text));
    if (oCalc.RetN() <= 0)
        MessageBox.Show("ERROR ... EL VALOR DE n debe ser mayor que 0");
    else
    {
        oCalc.EfecSuma();
        label2.Text = "La suma de los " + oCalc.RetN().ToString() + " primeros números naturales es = "
                                                                   + oCalc.RetSuma().ToString();
    }
}
```

La primera sentencia del método `button1_Click()` realiza la tarea de leer el número de naturales que el usuario desea sumar. Luego se valida con un **if** de 2 ramas si el valor leído es menor o igual que 0, si ésto es verdadero entonces se envía al usuario un mensaje de error. De lo contrario el flujo de ejecución sigue por la rama falsa del **if**, de manera que se manda el mensaje al objeto `oCalc` para que efectúe la suma. Luego se visualizan los resultados en el componente `label2` empleando los mensajes al objeto `oCalc` para que devuelva el valor de `_n` y el valor del atributo `_suma`.
Tenemos casi todos los métodos menos el que nos atañe explicar en esta unidad : el método `EfecSuma()` que realiza la operación repetitiva usando sentencias **do-while**, **while**, **for**.
La operación repetitiva ya se mencionó en la sección introductoria a la unidad, así que sólo resta escribir el código según se indica a continuación.

```
public void EfecSuma()
{
    int num = 1;
    _suma = 0;
    do
    {
        _suma += num;
        num++;
    } while (num <= _n);
}
```

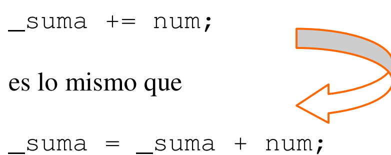
inicialización de la variable local `num` y del atributo `_suma`.

Operación repetitiva que emplea un **do-while** debido a que al menos una vez se realiza la operación repetitiva.

Notemos que :

`_suma += num;`

es lo mismo que

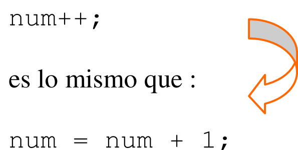


`_suma = _suma + num;`

También recordemos que el operador de incremento ++ aumenta en uno al operando al que se aplica. Asi que :

`num++;`

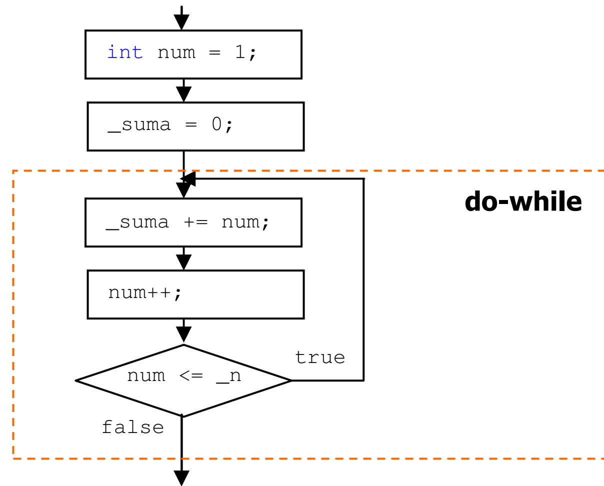
es lo mismo que :



`num = num + 1;`

Utilizamos un **do-while** debido a que el valor del atributo `_n` es siempre mayor que 0. Lo que quiere decir que en el mínimo caso, sumaremos al menos un número natural –cuando `_n` valga 1-. Así que al menos se efectúa la operación repetitiva una vez.
Desde luego que las otras sentencias **while** y **for** podrían emplearse, con la salvedad que deberemos alterar un poco el código.

El flujo de ejecución para el método `EfecSuma()` es :



Para probar el código del método `EfecSuma()` podemos seguir "a mano" ("a lápiz") el flujo de ejecución de las sentencias de dicho método, dando valores a las variables involucradas. Lo que se hace es crear una tabla donde las columnas sean las variables y/o atributos según lo indicamos enseguida.

_n	num	_suma	comentarios
1	1		Suponemos que el valor leído para _n es 1. Asi lo hacemos ya que es el valor mínimo que puede tener. O sea, seleccionamos para _n un valor en la frontera. El valor de num se inicializa a 1. El valor del atributo _suma es inicializado a 0. generalmente un almacén de sumas se inicializa a 0. En **_suma** almacenaremos precisamente la suma de los números naturales, los cuales son generados por el dato entero **num**. `int num = 1;` ← `_suma = 0;` Entramos por primera vez al **do-while**
		0 + 1 = 1	`_suma += num;` ← Es sumado el primer número natural 1. El valor ahora de _suma es 1.
	2		`num++;` ← Es incrementado el valor del dato num. El valor de num ahora es 2. `while(num <= _n);` Es probada la condición del **do-while**, 2 <= 1 ? false El resultado de la prueba es false. Se termina la operación repetitiva del **do-while** ya que el ciclo se termina cuando la condición al probarse retorna un false. **TERMINA EL MÉTODO `EfecSuma()`.** Observemos que el valor final del atributo `_suma` es = 1, es decir se han sumado los `_n`=1 primeros números naturales.

Ejecuta la aplicación para que observes los diferentes resultados. Puedes usar el depurador del C# para que sigas los valores en las variables y atributos. La figura #7.4.2 muestra la corrida para un valor de `_n` = 100.

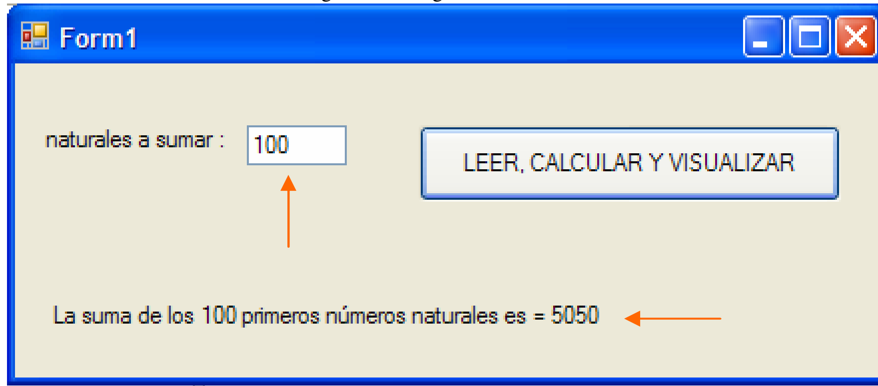


Fig. No. 7.4.2 Resultados para una lectura de n = 100.

A continuación visualizamos la prueba "a mano" para un valor de n = 2.

_n	num	_suma	comentarios

2	1		Suponemos que el valor leído para _n es 2. El valor de num se inicializa a 1. El valor del atributo _suma es inicializado a 0.
			
			`int num = 1;`
			`_suma = 0;`
			
			Entramos por primera vez al **do-while**

		0 + 1 = 1	`_suma += num;` Es sumado el primer número natural 1. El valor ahora de _suma es 1.

	2		`num++;`
			
			Es incrementado el valor del dato num. El valor de num ahora es 2.
			
			`while(num <= _n);`
			
			Es probada la condición del **do-while**,
			
			2 <= 2 ? true
			
			El resultado de la prueba es true. La operación repetitiva del **do-while** se vuelve a iterar.
			
			Entramos por segunda vez al **do-while**

		1 + 2 = 3	`_suma += num;` Es sumado el segundo número natural 2. El valor ahora de _suma es 3.

	3		`num++;`
			
			Es incrementado el valor del dato num. El valor de num ahora es 3.
			
			`while(num <= _n);`
			
			Es probada la condición del **do-while**,
			
			3 <= 2 ? false
			
			El resultado de la prueba es false. La operación repetitiva del **do-while** SE TERMINA.
			
			Observemos que el valor final del atributo _suma es = 3, es decir se han sumado los _n=2 primeros números naturales.

PREGUNTA.

¿Qué otro operador relacional puedes usar en la condición del **do-while** en vez del <=, de manera que todo siga funcionando igual sin cambiar ninguna otra parte del código?.

Ejercicio 7.4.2

Cambia el método **EfecSuma()** de forma que utilice un **while** en lugar del **do-while.**

Recordemos el método `EfecSuma()` :

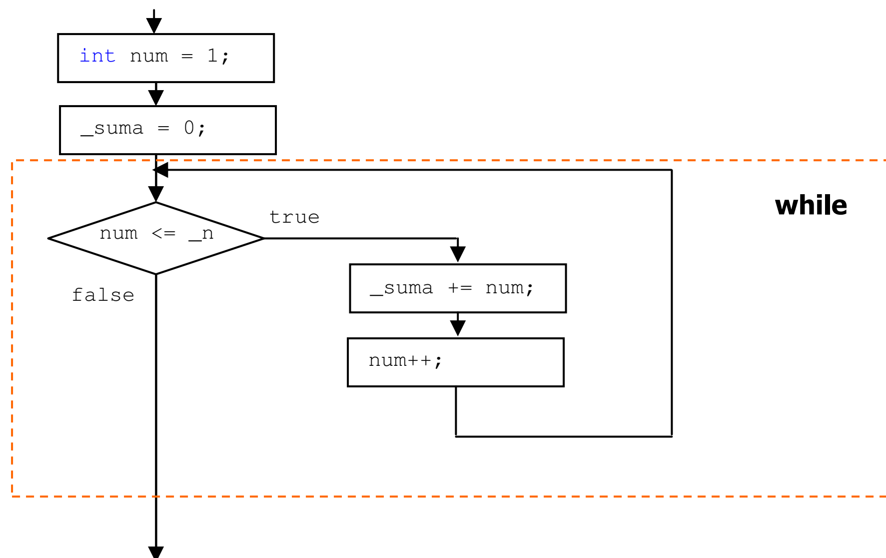
```
public void EfecSuma()
{
    int num = 1;
    _suma = 0;
    do
    {
        _suma += num;
        num++;
    } while (num <= _n);
}
```

Cambiar el **do-while** por un **while.**

Realmente el cambio en este caso sólo es de forma ya que la condición del **while** seguirá igual que el **do-while** anterior.

```
public void EfecSuma()
{
    int num = 1;
    _suma = 0;
    while (num <= _n)
    {
        _suma += num;
        num++;
    }
}
```

El flujo de ejecución para el método `EfecSuma()` es ahora :



Notemos que mientras la condición es verdadera la operación repetitiva seguirá efectuandose. La inicialización no se ha modificado ni las sentencias de la operación repetitiva.

A continuación mostramos la corrida "a mano" para `_n = 2`.

_n	**num**	**_suma**	**comentarios**

2	1		Suponemos que el valor leído para _n es 2. El valor de num se inicializa a 1. El valor del atributo _suma es inicializado a 0.
			
			`int num = 1;`
			`_suma = 0;`

			Se prueba la condición `while(num <= _n)` 1 <= 2 ? true El resultado de la prueba es true. La operación repetitiva del **while** se itera. Entramos por primera vez al **while**
		0 + 1 = 1	`_suma += num;`
			Es sumado el primer número natural 1. El valor ahora de _suma es 1.
	2		`num++;`
			Es incrementado el valor del dato num. El valor de num ahora es 2. El flujo de ejecución sigue ahora con la prueba de la condición del **while**. Se prueba la condición `while(num <= _n)` 2 <= 2 ? true El resultado de la prueba es true. La operación repetitiva del **while** se itera de nuevo. Entramos por segunda vez al **while**
		1 + 2 = 3	`_suma += num;`
			Es sumado el segundo número natural 2. El valor ahora de _suma es 3.
	3		`num++;`
			Es incrementado el valor del dato num. El valor de num ahora es 3. El flujo de ejecución sigue ahora con la prueba de la condición del **while**. Se prueba la condición `while(num <= _n)` 3 <= 2 ? false El resultado de la prueba es false. La operación repetitiva del **while** SE TERMINA. Observemos que el valor final del atributo `_suma` es = 3, es decir se han sumado los `_n`=2 primeros números naturales.

Ejercicio 7.4.3
Cambia el método **EfecSuma()** de forma que utilice un **for** en lugar del **do-while** (en lugar del **while**).

La sustitución es simple, sólo debemos recordar que cuando tenemos mas de una sentencia en las instrucciones de inicio, debemos separar a cada instrucción con el caracter (**,**). El código del método `EfecSuma()` ahora sería escrito asi empleando una sentencia **for** :

```
public void EfecSuma()
{
     int num;
     for(_suma=0,num=1;num<=_n;num++)
          _suma+=num;
}
```

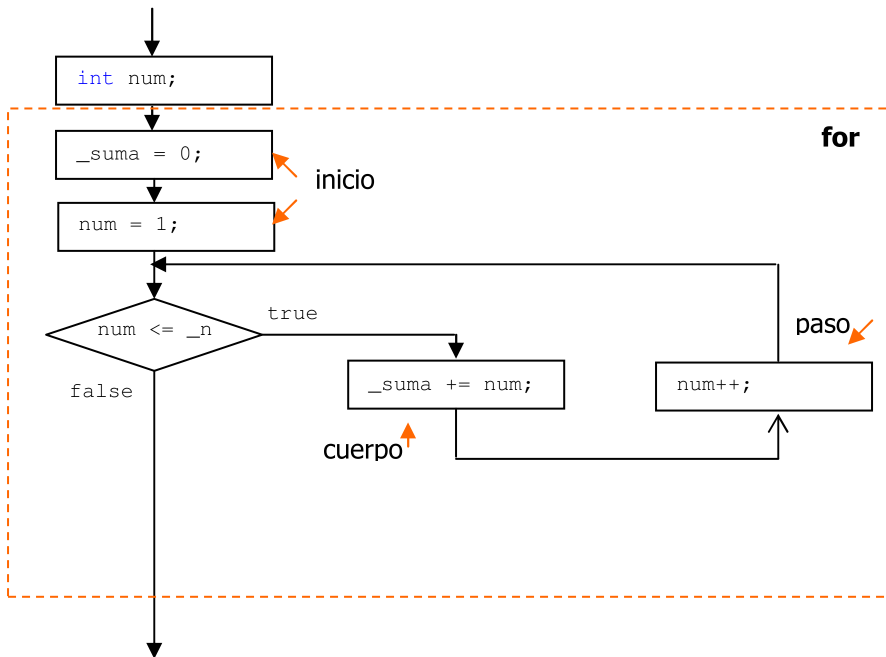
Observemos que ahora la inicialización es realizada dentro de la parte del **for** que acepta a las sentencias de inicio. Como estas son 2, entonces las separamos con un caracter (**,**).

El cuerpo del **for** consiste de sólo una sentencia la cual es la encargada de realizar la operación de suma del número natural correspondiente.

`_suma += num;`

La instrucción de paso `num++`, puede trasladarse al cuerpo del **for** sin afectar el funcionamiento del código.

Ahora el flujo de ejecución del método `EfecSuma()` es :



Veamos la ejecución "en lápiz" del método `EfecSuma()` para un valor de `_n` leído = 2.

_n	**num**	**_suma**	**comentarios**
2	1		Suponemos que el valor leído para `_n` es 2. Declaramos la variable `num` ya que dentro del **for** no nos lo permite, consulta con tu profesor los detalles de una declaración dentro del **for**. `int num;` ←

			Entramos al **for** por vez primera, por lo tanto son ejecutadas las instrucciones de inicio. Estas sólo son ejecutadas una sola ocasión y es cuando se entra por primera vez al **for**.
			
			`_suma = 0;`
			`num = 1;` ←
			
			Entramos por primera vez a la prueba de la condición del **for**.
			
			`num <= _n`
			
			`1 <= 2        true`
			
			El resultado de la prueba es `true`. La operación repetitiva en el cuerpo del **for** se ejecuta.

Son efectuadas las sentencias del cuerpo del **for**.

0 + 1 = 1 _suma += num;

Es sumado el primer número natural 1. El valor ahora de _suma es 1.

Se efectúan las sentencias de paso del **for**.

2 num++;

Es incrementado el valor del dato num. El valor de num ahora es 2.

El flujo de ejecución sigue ahora con la prueba de la condición del **for**.

Se prueba la condición

num <= _n

2 <= 2 ? true

El resultado de la prueba es true. El cuerpo del **for** se itera de nuevo.

Entramos por segunda vez al **for**

1 + 2 = 3 _suma += num;

Es sumado el segundo número natural 2. El valor ahora de _suma es 3.

Son efectuadas las sentencias de paso del **for**.

3 num++;

Es incrementado el valor del dato num. El valor de num ahora es 3.

El flujo de ejecución sigue ahora con la prueba de la condición del **for**.

Se prueba la condición

num <= _n

3 <= 2 ? false

El resultado de la prueba es false. La operación repetitiva del **for** SE TERMINA.

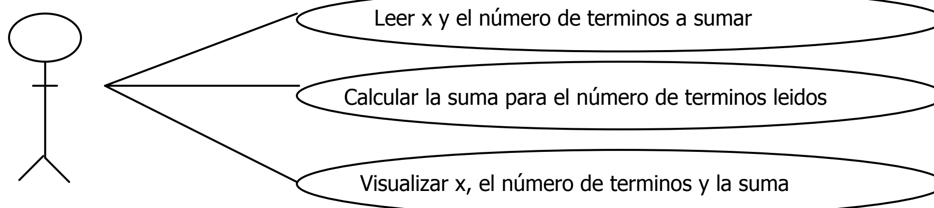
Observemos que el valor final del atributo _suma es = 3, es decir se han sumado los _n=2 primeros números naturales.

Ejercicio 7.4.4
Escribe una aplicación Windows C# que realice la suma :

suma = $1 + \sqrt{x} + x + \sqrt{x^3} + x^2 + \sqrt{x^5} + \ldots$

El valor de x es un real positivo y se lee. El número de términos a sumar también es leido y tiene valores en el intervalo 0 ≤ noterm ≥ 12.
Visualizar el resultado de la suma, el valor de x y el número de terminos sumado.

El diagrama de casos de uso es similar al del ejercicio 7.4.1 :



Lo mismo podemos decir del diagrama de clases. La clase `Calculador` tiene los atributos : `_x`, `_noTerm` y `_suma` que representan al valor de x leído, al número de terminos a sumar y el resultado de la suma, respectivamente. Los atributos se abstraen de acuerdo a la metodología *RAF*, recordemos que todo lo que se lee y/o visualiza será un atributo en la clase `Calculador`.

CALCULADOR
- _x : double - _noTerm : int - _suma : double
+ Leer(x:double, noTerm:int) : void + RetX() : double + RetNoTerm() : int + RetSuma() : double + EfecSuma() : void

La lectura de los valores ingresados por el usuario para el valor del real x y del número de términos a sumar, la haremos con un mismo método `Leer()`. Además tenemos 3 métodos que retornan el valor de x leído, el número de términos a sumar leido y el valor del resulatdo de la suma, `RetX()`, `RetNoTerm()`, y `RetSuma()`, respectivamente. El método `EfecSuma()` realiza la operación repetitiva y deposita el valor de la suma en el atributo `_suma`.

La interfase gráfica –figura #7.4.3 es simple : 2 componentes TextBox para permitir el ingreso del valor de x y del valor del número de términos a sumar. Un componente Button que se encargue de realizar la acción de leer, de calcular la suma y de visualizar los datos leídos además del resultado de la suma. La visualización la haremos en un componente TextBox sólo para tener un poco de variedad. debemos hacer la propiedad `Multiline` = `true`. Desde luego que podemos hacer el ejercicio de visualizar los resultados en un componente Label, o en 3 componentes Label, según cada uno de nosotros asi lo desee.

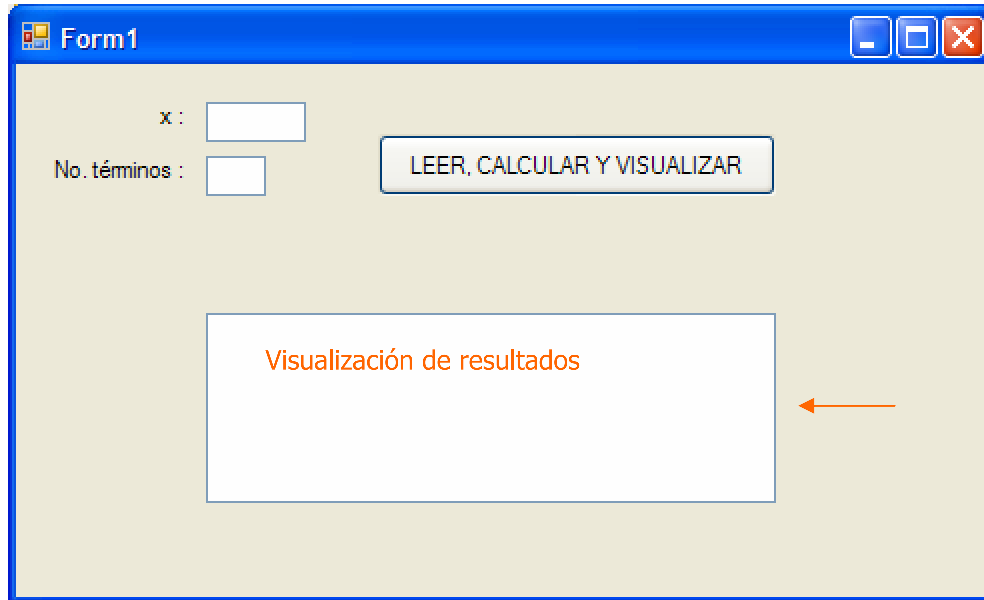


Fig. No. 7.4.3 Interfase gráfica usando un TextBox para visualizar los resultados.

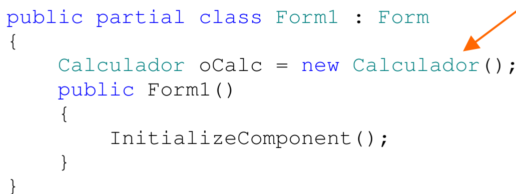
Definamos la clase `Calculador` la cual previamente debemos agregarla al proyecto.

```
class Calculador
{
    private double _x;
    private int _noTerm;
    private double _suma;

    public void Leer(double x, int noTerm)
    {
```

```
        _x = x;
        _noTerm = noTerm;
    }
    public double RetX()
    {
        return _x;
    }
    public int RetNoTerm()
    {
        return _noTerm;
    }
    public double RetSuma()
    {
        return _suma;
    }
    public void EfecSuma()
    {
        // operación repetitiva para calcular la suma
    }
}
```

También debemos añadir la definición del objeto `oCalc` en la clase `Form1` de nuestra aplicación Windows.

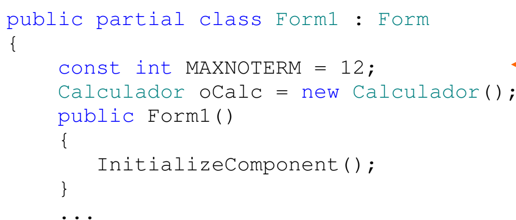


```
public partial class Form1 : Form
{
    Calculador oCalc = new Calculador();
    public Form1()
    {
        InitializeComponent();
    }
}
```

El paso siguiente es agregar el código para el método `button1_Click()` el cual es el encargado de leer los valores de **x** y del número de términos a sumar, de efectuar la suma y de visualizar los resultados.

```
private void button1_Click(object sender, EventArgs e)
{
     oCalc.Leer(Convert.ToDouble(textBox1.Text), Convert.ToInt32(textBox2.Text));
     if (oCalc.RetX() <= 0)
         MessageBox.Show("ERROR ... EL VALOR DE X DEBE SER MAYOR QUE 0");
     else if (oCalc.RetNoTerm() < 0 || oCalc.RetNoTerm() > MAXNOTERM)
         MessageBox.Show("ERROR ... EL VALOR DEL NÚMERO DE TERMINOS A SUMAR DEBE SER MAYOR O IGUAL QUE
                                              0 Y MENOR O IGUAL QUE " + MAXNOTERM.ToString());
     else
     {
         // Efectuar la suma y visualizar los resultados.
     }
}
```

Con el primer mensaje al objeto `oCalc.Leer()` se implementa la lectura del valor de `x` y el número de términos a sumar. Se emplea la clase `Convert` para efectuar las conversiones a `double` e `int` de los valores `x` y `noTerm`, respectivamente. Después se utiliza un **if** para validar que el valor de `x` leído sea positivo, si no es así, entonces se despliega un mensaje de error al usuario y no se sigue con el cálculo y la visualización de resultados. Si el valor de `x` es válido, entonces se prueba el valor del número de términos a sumar, donde se utiliza otro **if**, además que se emplea una constante MAXNOTERM que tiene el valor máximo de número de terminos a sumar. Es necesario agregar la definición de dicha constante dentro de la clase `Form1`, justo antes de la definición del objeto `oCalc`.



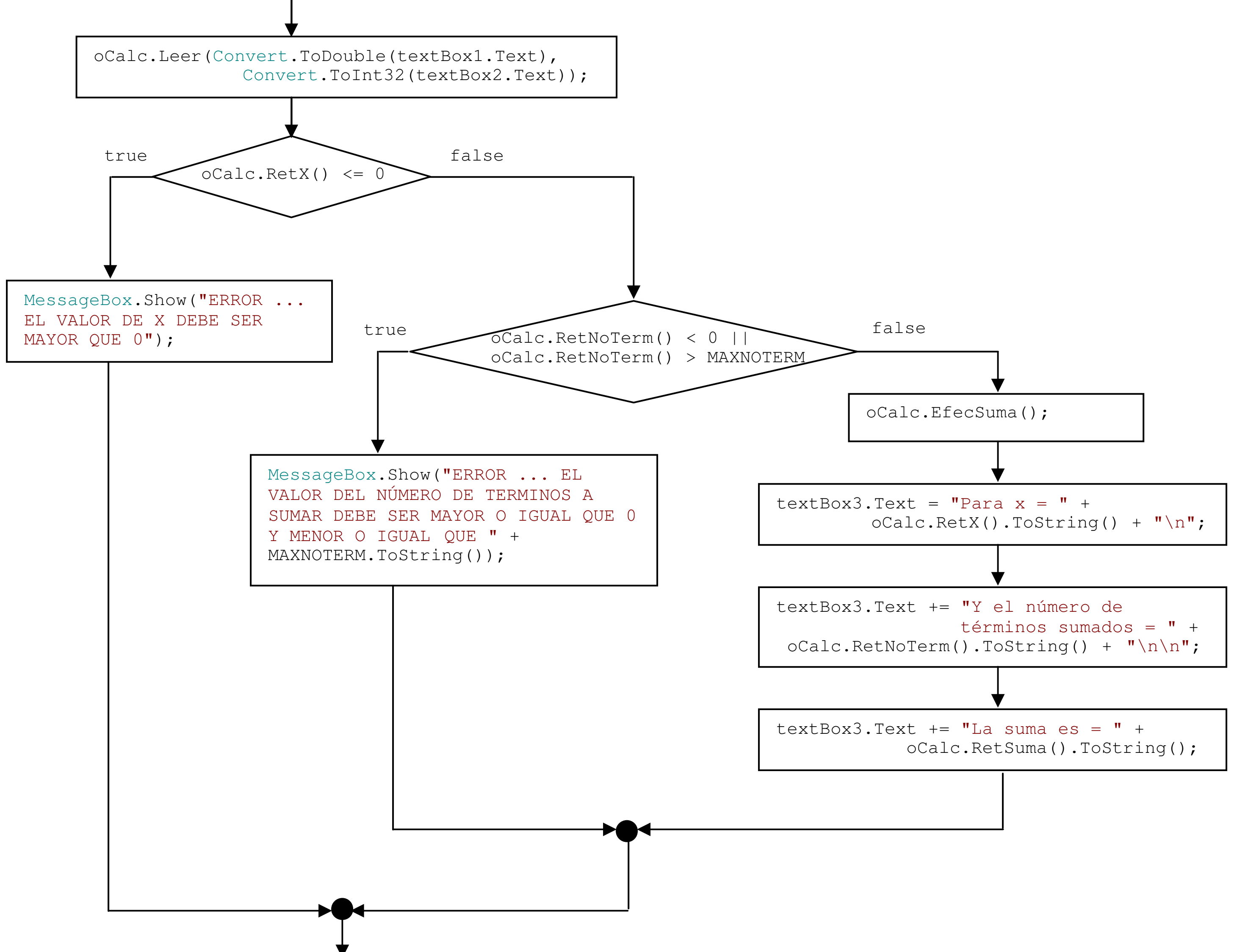
```
public partial class Form1 : Form
{
    const int MAXNOTERM = 12;
    Calculador oCalc = new Calculador();
    public Form1()
    {
       InitializeComponent();
    }
    ...
```

Constante MAXNOTERM cuyo valor es 12, según lo establecido en el enunciado del problema.

El código del método `button1_Click()` lo concluimos añadiendo los mensajes necesarios para calcular la suma y para visualizar los resultados.

```
private void button1_Click(object sender, EventArgs e)
{
    oCalc.Leer(Convert.ToDouble(textBox1.Text), Convert.ToInt32(textBox2.Text));
    if (oCalc.RetX() <= 0)
        MessageBox.Show("ERROR ... EL VALOR DE X DEBE SER MAYOR QUE 0");
    else if (oCalc.RetNoTerm() < 0 || oCalc.RetNoTerm() > MAXNOTERM)
        MessageBox.Show("ERROR ... EL VALOR DEL NÚMERO DE TERMINOS A SUMAR DEBE SER MAYOR O IGUAL QUE 0
                                                 Y MENOR O IGUAL QUE " + MAXNOTERM.ToString());
    else
    {
        oCalc.EfecSuma();
        textBox3.Text = "Para x = " + oCalc.RetX().ToString() + "\r\n";
        textBox3.Text += "Y el número de términos sumados = " + oCalc.RetNoTerm().ToString() +
                                                                                "\r\n\r\n";
        textBox3.Text += "La suma es = " + oCalc.RetSuma().ToString();
    }
}
```

El objeto `oCalc` efectúa la suma cuando recibe el mensaje `oCalc.EfecSuma()`. La visualización de los resultados la hemos hecho sobre la propiedad Text de un componente TextBox con propiedad name = `textBox3`. Notemos que usamos los caracteres no visibles `\r` (carriage return) y `\n` (new line) para "saltar" una nueva línea, además de empezar al extremo izquierdo de la superficie de escritura del componente `textBox3`. Observemos que hemos usado el operador += para realizar la concatenación de las cadenas que queremos desplegar en el componente `textBox3`, de manera que no se pierda la anterior información exhibida. El flujo de ejecución del método `button1_Click()` es :



Sólo resta escribir el código para el método `EfecSuma()` en la clase `Calculador`. Antes de escribirlo debemos efectuar un análisis de la operación repetitiva. Revisando el enunciado podemos reescribir la suma como :

$$suma = 1 + \sqrt{x} + x + \sqrt{x^3} + x^2 + \sqrt{x^5} + \ldots$$

$$suma = x^0 + x^{1/2} + x^{2/2} + x^{3/2} + x^{4/2} + x^{5/2} + \ldots$$

De forma que podemos ver que en cada término que es sumado el exponente se incrementa en ½, empezando para el primer término en `0`. Ahora ya podemos escribir la operación repetitiva :

```
suma = suma + x^exp;
exp = exp + 1/2;
cont = cont + 1;
```

OPERACIÓN REPETITIVA

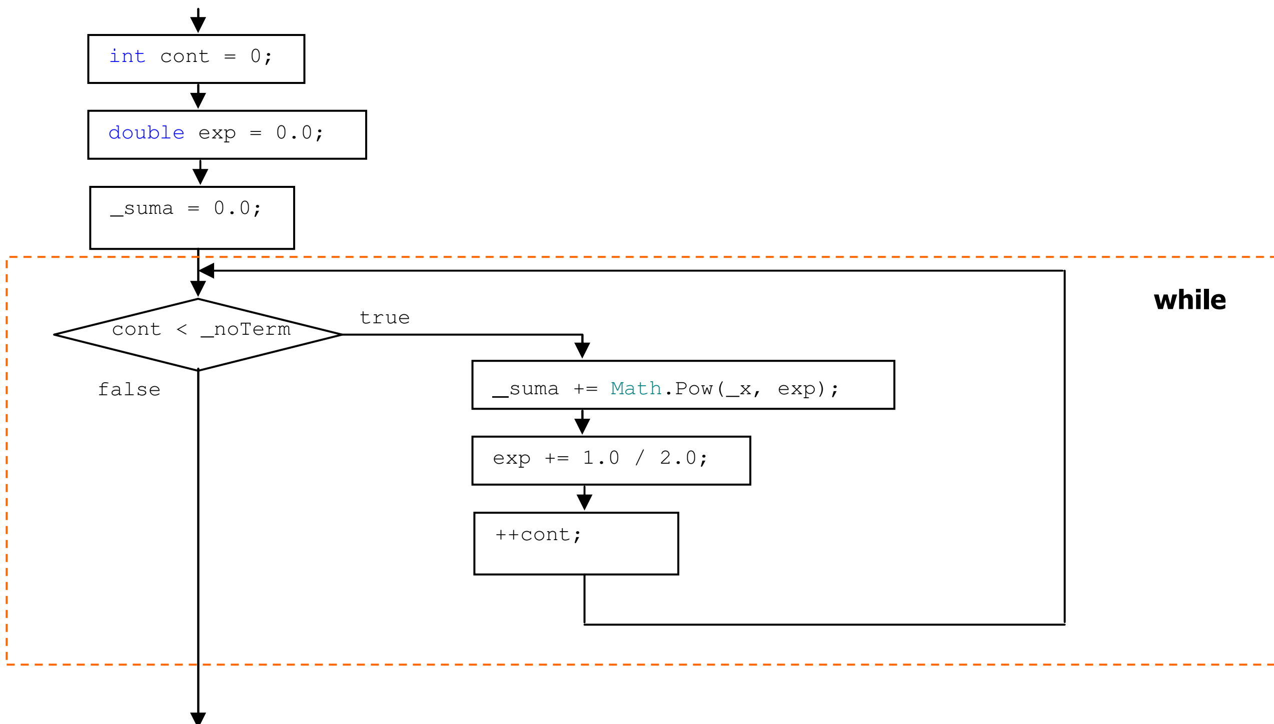
Necesitamos un contador `cont`, con el fin de llevar la cuenta del número de términos sumados. Cada vez que se efectúa la operación de suma, el contador `cont` lo incrementamos para saber cuántos términos se han sumado. De manera que la operación repetitiva la realizamos `MIENTRAS cont < _noTerm`, ya que cuando `cont == _noTerm` la condición nos dice que ya se sumaron los términos que se deseaba.
El método `EfecSuma()` entones constará del código según se indica a continuación :

```
public void EfecSuma()
{
    int cont = 0;
    double exp = 0.0;
    _suma = 0;
    while (cont < _noTerm)
    {
        _suma += Math.Pow(_x, exp);
        exp += 1.0 / 2.0;
        ++cont;
    }
}
```

Inicialización

Operación repetitiva

Hemos hecho uso de la clase `Math` para realizar la potencia de `_x` al exponente `exp`. El flujo de ejecución del método `EfecSuma()` es el que se muestra a continuación.



La ejecución de la aplicación es vista en la figura #7.4.4 para x = 4 y número de términos = 2.

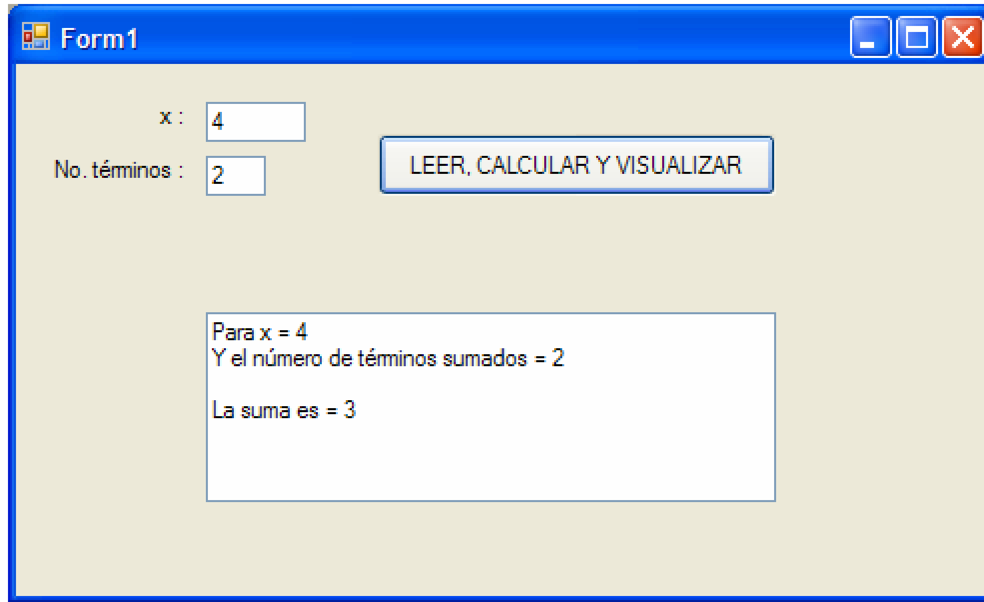


Fig. No. 7.4.4 Ejecución para `x = 4`, `noTerm = 2`.

Pregunta.
Realiza la corrida "a lápiz" del método `EfecSuma()` para valoes de :

- x = 4.0, noTerm = 0.
- x = 4.0, noTerm = 3.

Ejercicio 7.4.5
Cambia el **while** en el método `EfecSuma()` por un **for**.

Existen varias maneras de realizar el cambio, aquí vemos a una de ellas.

Instrucciones de inicio — Instrucciones de paso

```
public void EfecSuma()
{
    int cont;
    double exp;
    for (cont = 0, exp = 0.0, _suma = 0.0; cont < _noTerm; exp += 1.0 / 2.0, ++cont)
        _suma += Math.Pow(_x, exp);
}
```

Seleccionamos esta forma debido a que empleamos varias instrucciones tanto en el inicio como en las de paso. Desde luego que podemos poner las instrucciones de paso dentro del cuerpo del **for**, y el resultado será el mismo : correcto.

```
public void EfecSuma()
{
    int cont;
    double exp;
    for (cont = 0, exp = 0.0, _suma = 0.0; cont < _noTerm;)
    {
        _suma += Math.Pow(_x, exp);
        exp += 1.0 / 2.0;
        ++cont;
    }
}
```

Las sentencias de paso se han cambiado al cuerpo del **for**.

Pregunta.
Cambia el **while** en el método `EfecSuma()` por un **do-while**.

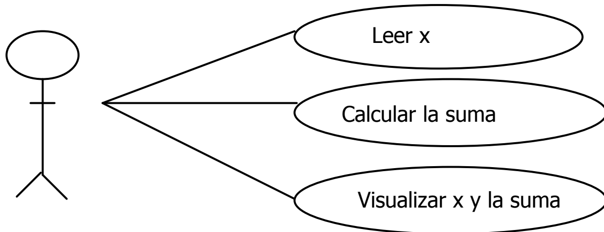
Ejercicio 7.4.6

Escribe una aplicación Windows C# que efectúe la siguiente suma :

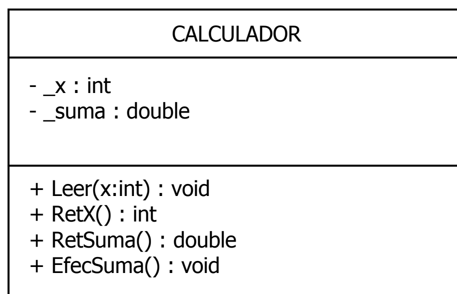
$\texttt{suma} = x + (x-1)x + (x-2)x^2 + (x-3)x^3 + \ldots + 0$

El valor de x es un entero mayor que 0, y se lee. Visualiza el resultado de la suma y el valor de x leído.

El diagrama de casos de uso vuelve a ser muy similar a los anteriores :



El diagrama de clases :



La clase `Calculador` sólo tiene 2 atributos : `_x` y `_suma` de acuerdo a la metodología *RAF*. Una vez que hayamos creado un nuevo proyecto para resolver este ejercicio, creamos la interfase gráfica de acuerdo a la figura #7.4.5.

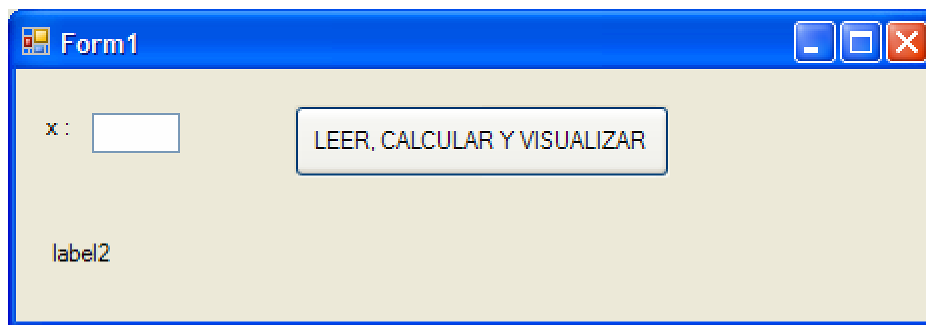


Fig. No. 7.4.5 Interfase gráfica ejercicio 7.4.6.

Agregamos la clase `Calculador` con el código :

```
class Calculador
{
    private int _x;
    private double _suma;

    public void Leer(int x)
    {
        _x = x;
    }
    public int RetX()
    {
        return _x;
    }
    public double RetSuma()
    {
```

```
        return _suma;
    }
    public void EfecSuma()
    {
        // operacion repetitiva
    }
}
```

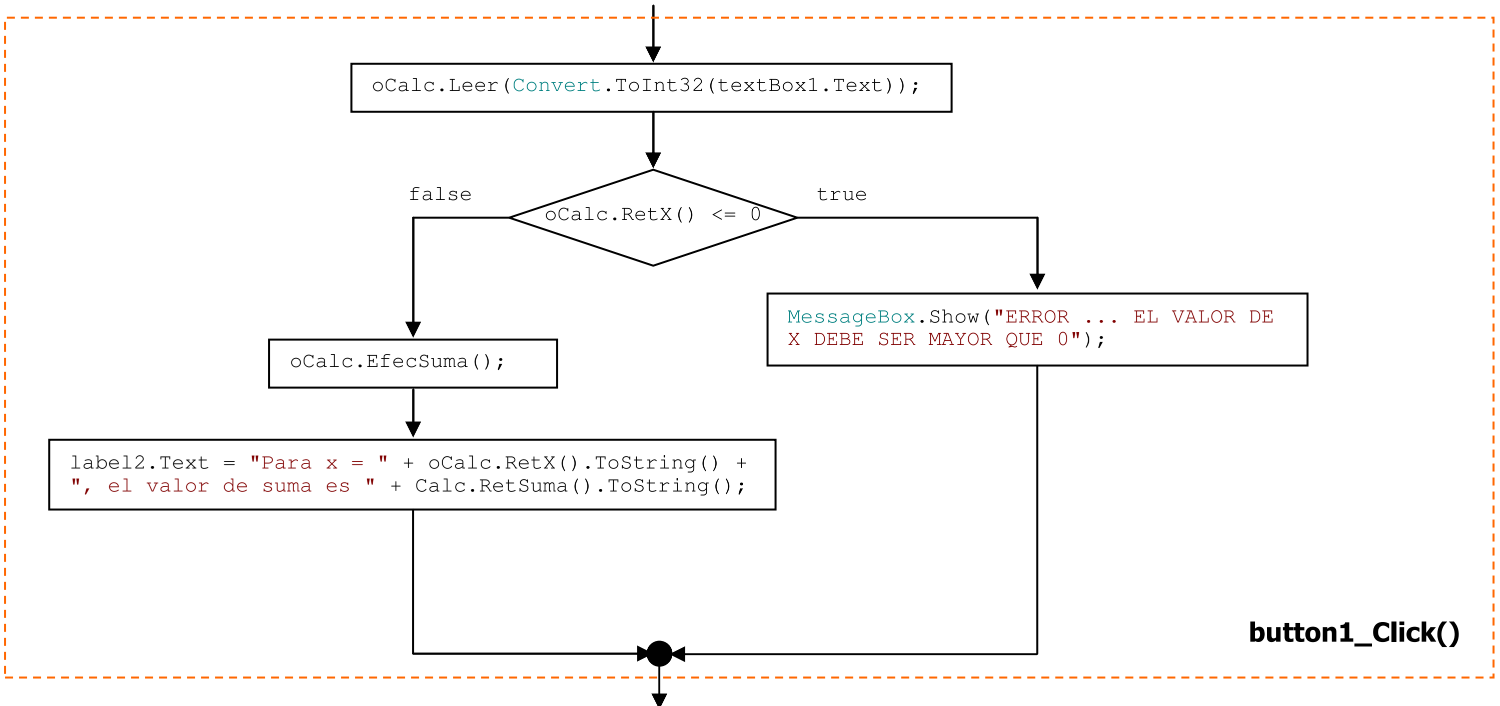
Definimos al objeto `oCalc` en la clase `Form1` de la aplicación Windows :

```
public partial class Form1 : Form
{
    Calculador oCalc = new Calculador();
    public Form1()
    {
        InitializeComponent();
    }
}
```

Las tareas descritas en el diagrama de casos de uso las implementamos en el método `button1_Click()` cuyo código es mostrado :

```
private void button1_Click(object sender, EventArgs e)
{
     oCalc.Leer(Convert.ToInt32(textBox1.Text));
     if (oCalc.RetX() <= 0)
         MessageBox.Show("ERROR ... EL VALOR DE X DEBE SER MAYOR QUE 0");
     else
     {
         oCalc.EfecSuma();
         label2.Text = "Para x = " + oCalc.RetX().ToString() + ", el valor de suma es " +
                                                                 oCalc.RetSuma().ToString();
     }
}
```

Los métodos `Leer()`, `RetX()`, `RetSuma()` ya estan escritos y definidos en la clase `Calculador`. El **if** de 2 ramas nos permite validar el valor de `x` de tal manera que sólo hagamos la operación repetitiva para valores de `x` mayores que 0. El flujo de ejecución del método `button1_Click()` es :



Para terminar el ejercicio resta escribir el código del método `EfecSuma()` en la clase `Calculador`. Debemos tomar en cuenta las consideraciones siguientes :

$$suma = x + (x-1)x + (x-2)x^2 + (x-3)x^3 + ... + 0$$

- El primer término que se suma tiene la forma : $(x-0)x^0$ que es lo mismo si lo escribimos como se encuentra en el enunciado, `x`.
- De lo anterior podemos concluir que cada vez que efectuamos la operación repetitiva, el término a sumar tiene la forma $(x - n)\ x^n$, donde `n = 0,1, ..., x`.
- La operación repetitiva debe terminar cuando `n == x`. De manera que la resta `x - n` sea `0`.
- Como el valor de `x` es mayor que 0, entonces la operación repetitiva se efectúa al menos una vez, por consiguiente usaremos un **do-while**.

```
public void EfecSuma()
{
    int n = 0;
    _suma = 0.0;
    do
    {
        _suma += (_x - n) * Math.Pow((double)_x, (double)n);
        ++n;
    } while (n <= _x);
}
```

Hemos empleado moldes (`double`) para efectuar la conversión explícita de los enteros `_x` y `n` a tipo `double`, ya que los parámetros para el método `Pow` deben ser de tipo `double`.
Realiza una prueba "a lápiz" del método `EfecSuma()` para probar su funcionamiento. La figura #7.4.6 muestra la ejecución de la aplicación para un valor de `_x` leído = `4`.

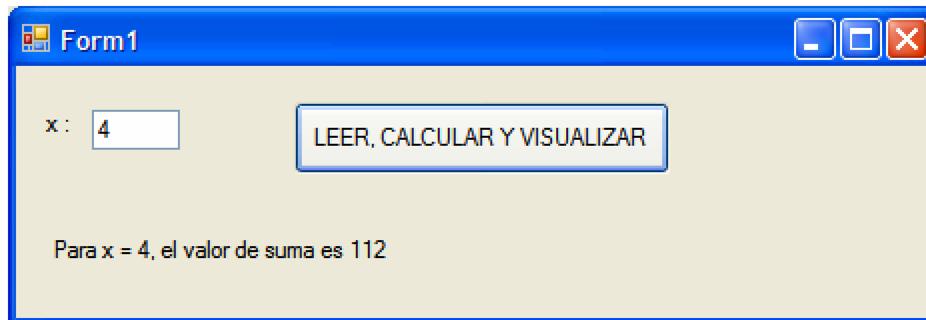


Fig. No. 7.4.6 Ejecución para `_x = 4`.

Pregunta.
Cambia el **do-while** en el método `EfecSuma()` por un **while**.

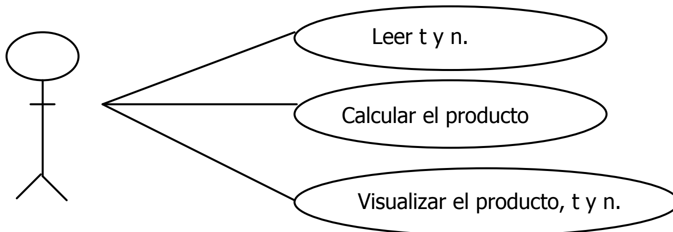
Pregunta.
Cambia el **do-while** en el método `EfecSuma()` por un **for**.

Pregunta.
Visualiza los resultados en un componente TextBox.

Pregunta.
Visualiza los resultados en 2 componentes Label.

Ejercicio 7.4.7
Escribe una aplicación Windows C# que efectúe el producto :
$prod = t^{n} * t^{n-1} * t^{n-2} * \ldots * 1$
El valor de t es un real mayor que 0, y se lee.
El valor de n es un entero en el intervalo 0<=n<=8.
Visualiza el resultado del producto, el valor de t y de n leidos.

El diagrama de casos de uso es :



Siguiendo la metodología *RAF*, el diagrama de clases –clase `Calculador`- es :

CALCULADOR
- _t : double - _n : int - _prod : double
+ Leer(t:double, n:int) : void + RetT() : double + RetN() : int + RetProd() : double + EfecProd() : void

Ya que creamos el nuevo proyecto C#, construyamos la interfase gráfica según se ve en la figura #7.4.7.

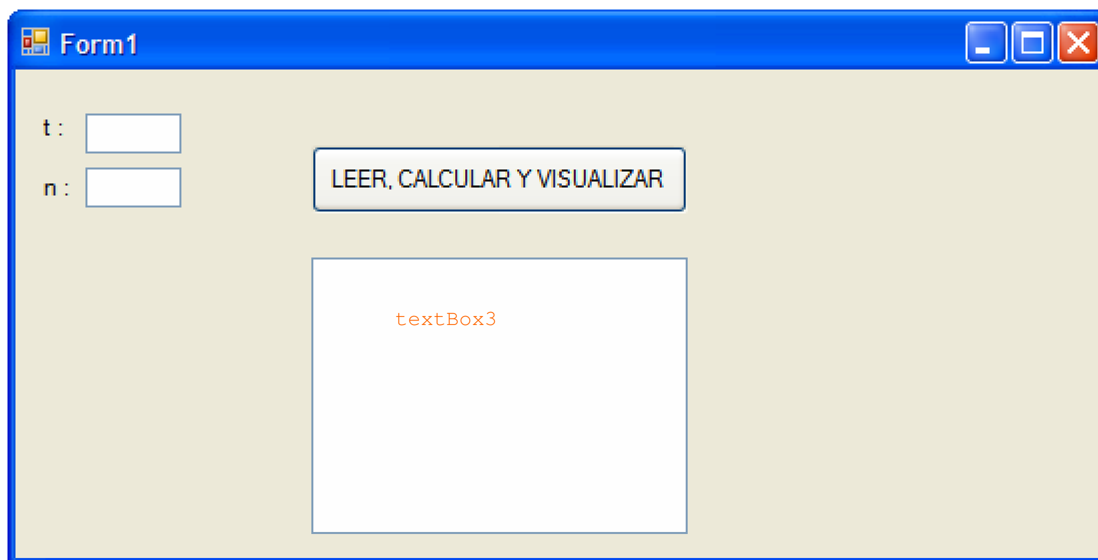


Fig. No. 7.4.7 Interfase gráfica ejercicio 7.4.7.

La propiedad Multiline del componente `textBox3` la debemos poner en `true`. Lo siguiente es agregar la clase Calculador con el código indicado :

```
class Calculador
{
    private double _t;
    private int _n;
    private double _prod;

    public void Leer(double t, int n)
    {
        _t = t;
        _n = n;
    }
    public double RetT()
    {
       return _t;
    }
    public int RetN()
    {
       return _n;
    }
    public double RetProd()
    {
       return _prod;
    }
    public void EfecProd()
    {
       // operación repetitiva
    }
}
```

El objeto `oCalc` lo definimos en la clase `Form1` :

```
public partial class Form1 : Form
{
    Calculador oCalc = new Calculador();
    public Form1()
    {
       InitializeComponent();
    }
}
```

Lo último es agregar el código en el botón LEER, CALCULAR Y VISUALIZAR que permite realizar las tareas definidas en el diagrama de casos de uso. El método `button1_Click()` tiene el código :

```
private void button1_Click(object sender, EventArgs e)
{
    oCalc.Leer(Convert.ToDouble(textBox1.Text), Convert.ToInt32(textBox2.Text));
    if (oCalc.RetT() <= 0.0)
        MessageBox.Show("ERROR ... EL VALOR DE t DEBE SER MAYOR QUE 0.0");
    else if (oCalc.RetN() < 0 || oCalc.RetN() > MAXVALORN)
        MessageBox.Show("ERROR ... EL VALOR DE N DEBE SER >=0 PERO <= QUE " + MAXVALORN.ToString());
    else
    {
        oCalc.EfecProd();
        textBox3.Text = "Para t = " + oCalc.RetT().ToString() + "\r\n";
        textBox3.Text += "Y para n = " + oCalc.RetN().ToString() + "\r\n\r\n";
        textBox3.Text += "El producto prod es = " + oCalc.RetProd().ToString();
    }
}
```

Hemos utilizado una constante `MAXVALORN` que aún no la hemos definido en la clase `Form1`, así que hagámoslo :

```
public partial class Form1 : Form
{
    const int MAXVALORN = 8;
    Calculador oCalc = new Calculador();
    ...
```

Notemos que empleamos **if** anidados para validar el valor de `t` y de `n` ingresados por el usuario. Si sus valores son validos, entonces podemos efectuar el producto y visualizar los resultados. Los resultados son vaciados en la proiedad Text del componente `textBox3`, tal y como lo hicimos en el ejercicio 7.4.4.

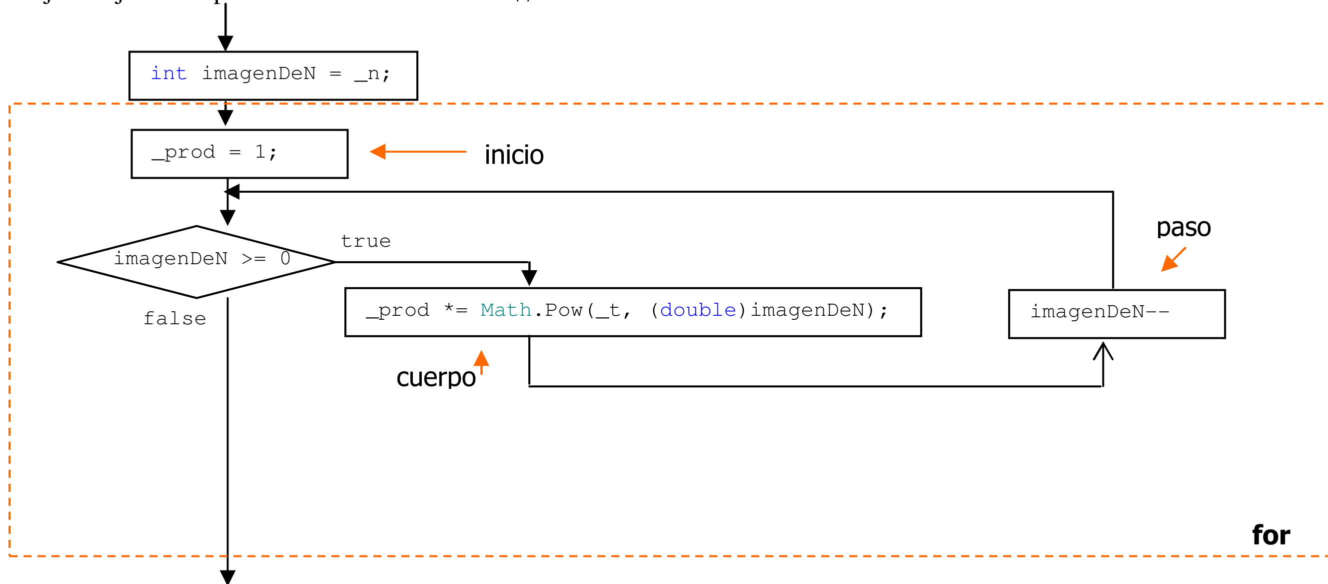
Nos falta codificar la operación repetitiva dentro del método `EfecProd()` de la clase `Calculador`. Para hacerlo enumeramos las consideraciones siguientes :

- La operación repetitiva termina cuando el exponente del término `t` es igual a 0.
- En cada iteración de la operación `prod`, el valor del exponente se decrementa en 1.
- Dado que se tiene que visualizar el valor de `_n`, debemos utilizar un dato imagen a `_n` de manera que decrementemos a este dato imagen y no a `_n`. De lo contrario, perderíamos el valor de `_n`.
- Emplearemos un **for** para efectuar la operación repetitiva.
- Un almacén de productos generalmente se inicializa al valor de 1, o a un valor diferente de 0. Por eso es que el valor del atributo `_prod` se inicializa a 1.

Tecleemos el método `EfecProd()` de acuerdo al código listado a continuación :

```
public void EfecProd()
{
      int imagenDeN = _n;
      for (_prod = 1; imagenDeN >= 0; imagenDeN--)
            _prod *= Math.Pow(_t, (double)imagenDeN);
}
```

El flujo de ejecución para el método `EfecProd()` es :



La ejecución para valores de `t = 2` y `n = 2` se muestra en la figura #7.4.8.

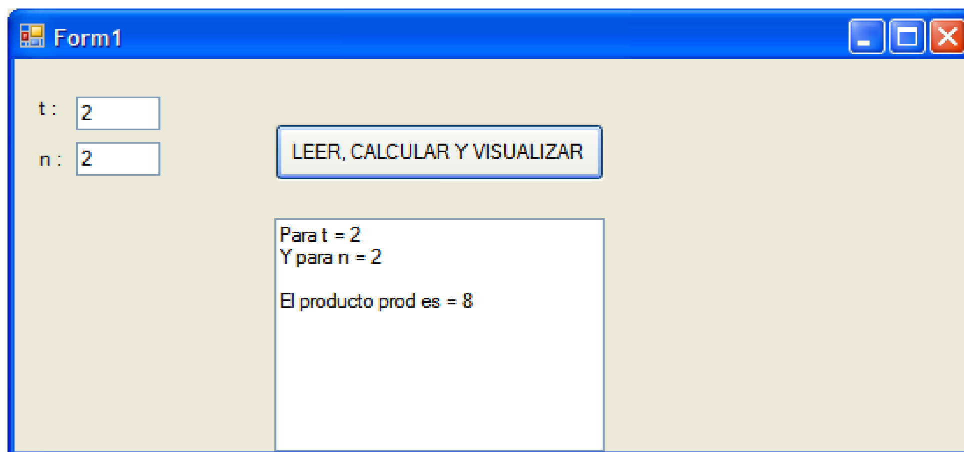


Fig. No. 7.4.8 Aplicación en ejecución.

Pregunta.
Cambia el **for** en el método `EfecProd()` por un **do-while**.

Pregunta.
Cambia el **for** en el método `EfecProd()` por un **while**.

Pregunta.
Visualiza los resultados en un componente Label.

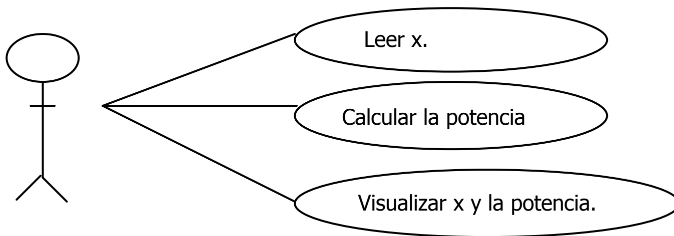
Pregunta.
Visualiza los resultados en 3 componentes Label.

Ejercicio 7.4.8
Escribe una aplicación Windows C# que efectúe la potencia :

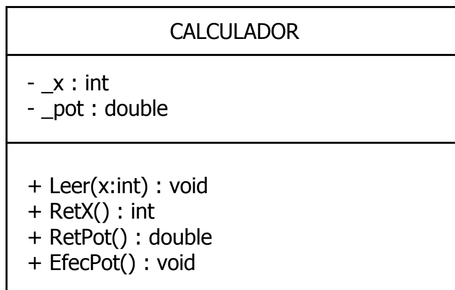
$$pot = ((((\Pi^{x})^{x-1})^{x-2})\cdots)^{1}$$

Π es la constante real 3.1416
El valor de x es un entero positivo y se lee.
Visualizar el resultado de la potencia y el valor de x.

El diagrama de casos de uso es similar a los anteriores :



Lo mismo podemos decir del diagrama de la clase `Calculador` :



Antes de construir la interfase gráfica, vamnos a crear el nuevo proyecto y agregar la clase `Calculador`.

```
class Calculador
{
    private int _x;
    private double _pot;

    public void Leer(int x)
    {
       _x = x;
    }
    public int RetX()
    {
       return _x;
    }
```

Recordemos la metodología : todo lo que se lee y/o se visualiza lo hacemos atributo de la clase `Calculador`.

```
    public double RetPot()
    {
       return _pot;
    }
    public void EfecPot()
    {
       // operación repetitiva
    }
}
```

La definición del objeto `oCalc` en la clase `Form1` :

```
public partial class Form1 : Form
{
    Calculador oCalc = new Calculador();    <------   Definición del objeto calculador oCalc.
    public Form1()
    {
       InitializeComponent();
    }
}
```

La interfase gráfica la construimos según la figura #7.4.9.

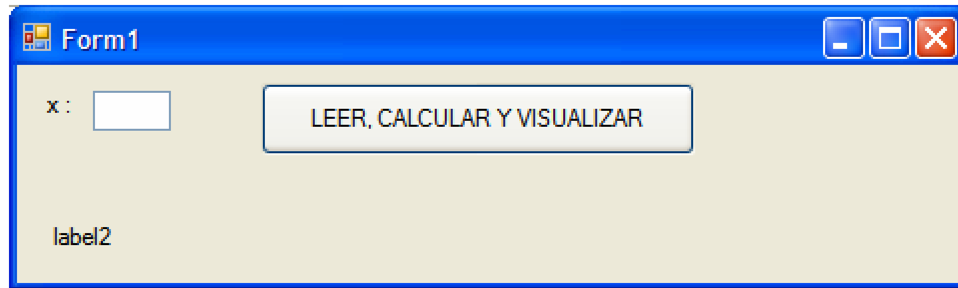
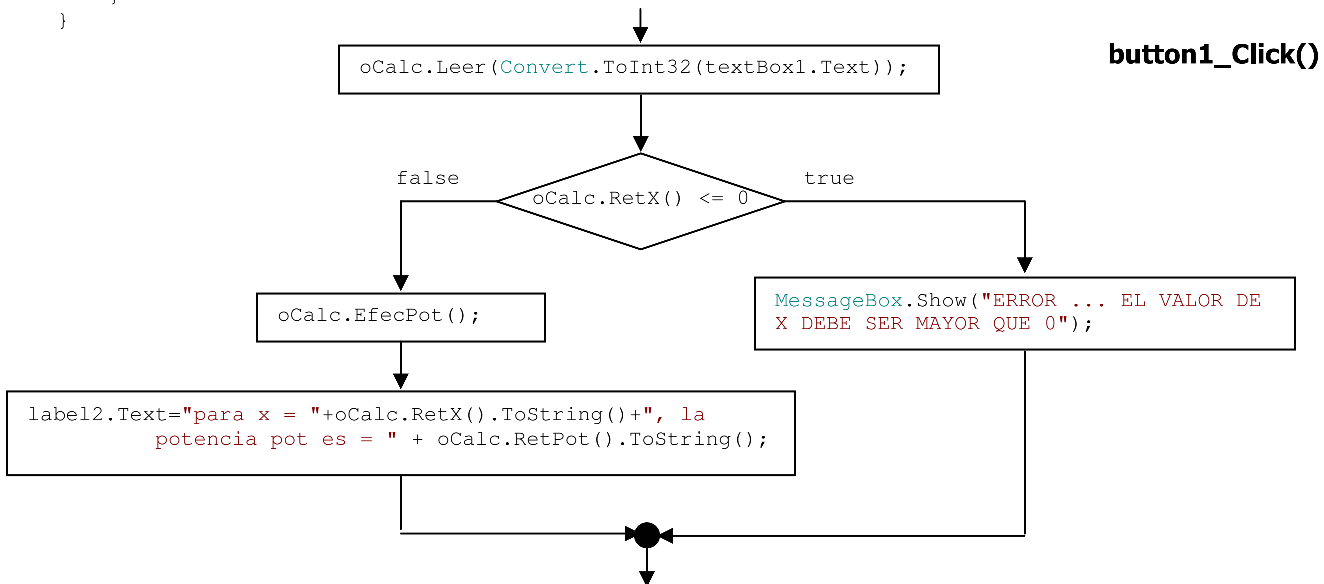


Fig. No. 7.4.9 Interfase gráfica ejercicio 7.4.8.

La lectura, cálculo de la potencia y la visualización de resultados, la escribimos en el método `button1_Click()` :

```
private void button1_Click(object sender, EventArgs e)
{
    oCalc.Leer(Convert.ToInt32(textBox1.Text));
    if (oCalc.RetX() <= 0)
        MessageBox.Show("ERROR ... EL VALOR DE X DEBE SER MAYOR QUE 0");
    else
    {
        oCalc.EfecPot();
        label2.Text="para x = "+oCalc.RetX().ToString()+", la potencia pot es = " +
                                                                oCalc.RetPot().ToString();
    }
}
```



Para concluir el ejercicio tenemos que escribir el código del método `EfecPot()` en la clase `Calculador`. Utilizaremos una sentencia **while** para construir la operación repetitiva.

```
public void EfecPot()
{
     int imagenX = _x;
     _pot = Math.PI;
     while (imagenX > 0)
     {
         _pot = Math.Pow(_pot, imagenX);
         imagenX--;
     }
}
```

Consideraciones sobre el código del método `EfecPot()` :

- Se ha definido una variable `imagenX` para guardar el valor de `_x` de manera que no perdamos su valor, para posteriormente visualizarlo.
- Hemos incializado el almacén de potencias `_pot` al valor de la constante `PI`, definida en la clase `Math`.
- La operación repetitiva se efectúa mientras el valor de la imagen `imagenX` es mayor que `0`, es decir se efectúa la potencia hasta que `imagenX` sea igual a `1`, inclusive.

La ejecución para valor de `x = 2` se ve en la figura #7.4.10.

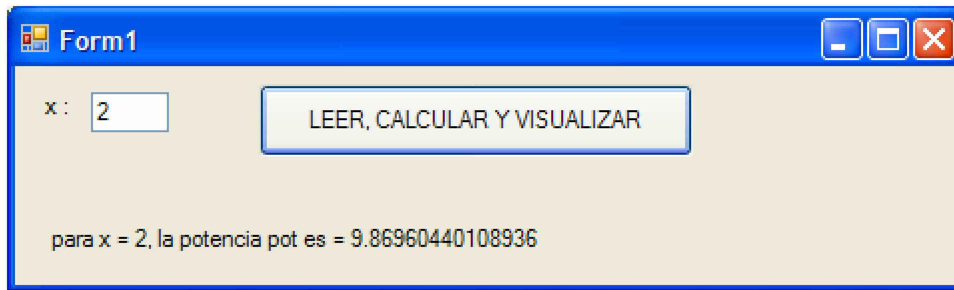


Fig. No. 7.4.10 Resultados de la potencia de `PI` para `x = 2`.

Veamos en la siguiente tabla la corrida "a lápiz" del método `EfecPot()` para `x=2`.

_x	imagenX	_pot	comentarios
2		3.1416	Suponemos que el valor leído para _x es 2. El valor de imagenX se inicializa a _x. El valor del atributo _pot es inicializado a 3.1416.
	2		`int imagenX = _x;` ← `_pot = 3.1416;   // valor de la constante PI`

			Se prueba la condición
			 `while(imagenX > 0)`
			 `2 >= 0 ?   true`
			 El resultado de la prueba es true. La operación repetitiva del **while** se itera.
			 Entramos por primera vez al **while**

		$(3.1416)^2$	`_pot = Math.Pow(_pot, imagenX);` ← Es potenciado _pot a la 2.
	1		`imagenX--;` ← Es decrementado el valor de imagenX. El valor de imagenX ahora es 1.

El flujo de ejecución sigue ahora con la prueba de la condición del **while**.

Se prueba la condición

```
while(imagenX > 0)
```

1 > 0 ? true

El resultado de la prueba es true. La operación repetitiva del **while** se itera de nuevo.

Entramos por segunda vez al **while**

$((3.1416)^2)^1$

```
_pot = Math.Pow(_pot, imagenX);
```

Es potenciado _pot a la 1.

0

```
imagenX--;
```

Es decrementado el valor de imagenX. El valor de imagenX ahora es 0.

El flujo de ejecución sigue ahora con la prueba de la condición del **while**.

Se prueba la condición

```
while(imagenX > 0)
```

0 > 0 ? false

El resultado de la prueba es false. La operación repetitiva del **while** SE TERMINA.

Pregunta.
Cambia el **while** en el método `EfecPot()` por un **do-while**.

Pregunta.
Cambia el **while** en el método `EfecPot()` por un **for**.

Pregunta.
Visualiza los resultados en un componente `TextBox`.

Ejercicio 7.4.9
Escribe una aplicación Windows C# que efectúe la suma de raices :

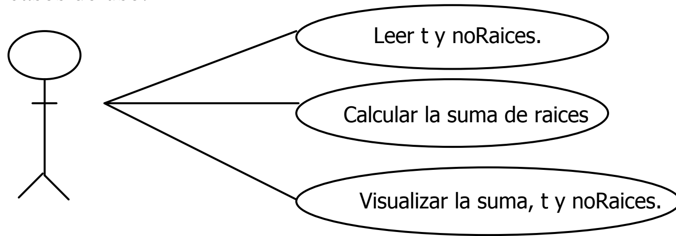
$$raices = \sqrt{t + \sqrt{t + \ldots + \sqrt{t}}}$$

El valor de t es un real positivo y se lee.
El número de raices a sumar se lee, y tiene el intervalo 1<=noRaices<=10.
Visualizar el resultado de la suma de las raices, el valor de t y el valor del número de raices a sumar.

El diagrama de clases consiste como hasta ahora de una clase `Calculador`. Sus atributos y métodos son los indicados.

CALCULADOR
- _t : double - _noRaices : int - _raices : double
+ Leer(t:double, noRaices:int) : void + RetT() : double + RetNoRaices() : int + RetRaices() : double + EfecSumaRaices() : void

Los métodos indicados en la clase `Calculador` son los requeridos para conseguir las tareas denotadas en el diagrama de casos de uso.



La interfase gráfica es muy similar a la del ejercicio 7.4.7, se leen 2 valores en sus respectivos componente TextBox y la visualización la hacemos sobre otro componente TextBox con la propiedad `Multiline = true`.

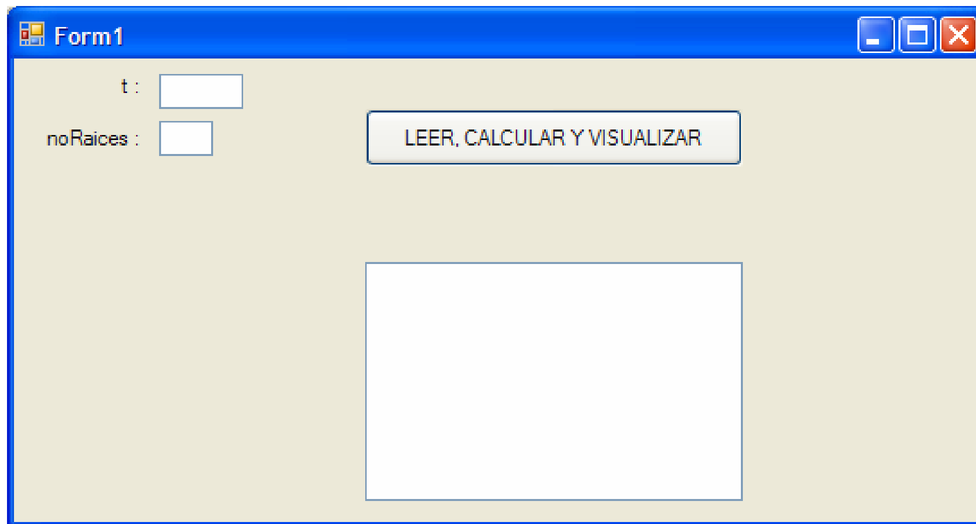


Fig. No. 7.4.11 Interfase gráfica para el cálculo de raices de un valor `t`.

Tecleemos la clase `Calculador` de acuerdo al código mostrado, antes deberemos agregar la clase al proyecto.

```
class Calculador
    {
        private double _t;
        private int _noRaices;
        private double _raices;

        public void Leer(double t, int noRaices)
        {
            _t = t;
            _noRaices = noRaices;
        }
        public double RetT()
        {
            return _t;
        }
        public int RetNoRaices()
        {
            return _noRaices;
        }
        public double RetRaices()
        {
            return _raices;
        }
        public void EfecSumaRaices()
        {
            // operación repetitiva
        }
    }
```

Luego definimos el objeto `oCalc` en la clase `Form1`.

```
public partial class Form1 : Form
{
    Calculador oCalc = new Calculador();
    public Form1()
    {
        InitializeComponent();
    }
}
```

Hagamos la lectura de `t` y del número de raices a sumar, además de efectuar la suma de las raíces y la visualización de los resultados. estas 3 tareas las realiza el método `button1_Click()` de la clase `Form1`.

```
private void button1_Click(object sender, EventArgs e)
{
     oCalc.Leer(Convert.ToDouble(textBox1.Text), Convert.ToInt32(textBox2.Text));
     if (oCalc.RetX() <= 0.0)
         MessageBox.Show("ERROR .. EL VALOR DE t DEBE SER MAYOR QUE 0");
     else if (oCalc.RetNoRaices() <= 0 || oCalc.RetNoRaices() > MAXNORAICES)
         MessageBox.Show("ERROR ... VALOR DEL NÚMERO DE RAICES FUERA DE RANGO");
     else
     {
         oCalc.EfecSumaRaices();
         textBox3.Text = "Para t = " + oCalc.RetT().ToString() + "\r\n";
         textBox3.Text += " Y para número de raices a sumar = " + oCalc.RetNoRaices().ToString() +
                                                                                    "\r\n\r\n";
         textBox3.Text += "La suma es = " + oCalc.RetRaices().ToString();
     }
}
```

Definamos la constante entera `MAXNORAICES` tal y como lo hemos hecho en otros ejercicios en la clase `Form1`.

```
public partial class Form1 : Form
{
    const int MAXNORAICES = 10;
    Calculador oCalc = new Calculador();
    ...
```

Para escribir el código del método `EfecSumaRaices()` en la clase `Calculador`, es bueno hacer el análisis de cómo es que vamos a sumar las raíces. Veamos los casos mas simples :

noRaices leído	Operación
1	$raices = \sqrt{t}$
2	$raices = \sqrt{t + \sqrt{t}}$
3	$raices = \sqrt{t + \sqrt{t + \sqrt{t}}}$

Observemos los términos marcado en naranja y en verde agua. Notemos que el término señalado en naranja representa el valor de `raices` cuando `noRaices` es 1. El señalado en verde agua es el valor de `raices` cuando `noRaices` es 2.

De acuerdo a las observaciones hechas en el recuadro de la derecha de la tabla, podemos reescribir la tabla :

noRaices leído	Operación
1	$raices = \sqrt{t + raices(0)}$
2	$raices = \sqrt{t + raices(1)}$
3	$raices = \sqrt{t + raices(2)}$

Donde `raices(0)` es = `0`, es decir, vamos a inicializar `raices` al valor de `0`. Escribamos el código del método `EfecSumaRaices()` según se indica :

```
public void EfecSumaRaices()
{
      _raices = 0;
      int cont=0;
      do
      {
          _raices = Math.Sqrt(_t + _raices);
          ++cont;
      } while (cont < _noRaices);
}
```

La figura #7.4.12 muestra la ejecución de la aplicación Windows, para valores de `t = 9`, y el número de raíces leido `noRaices = 4`.

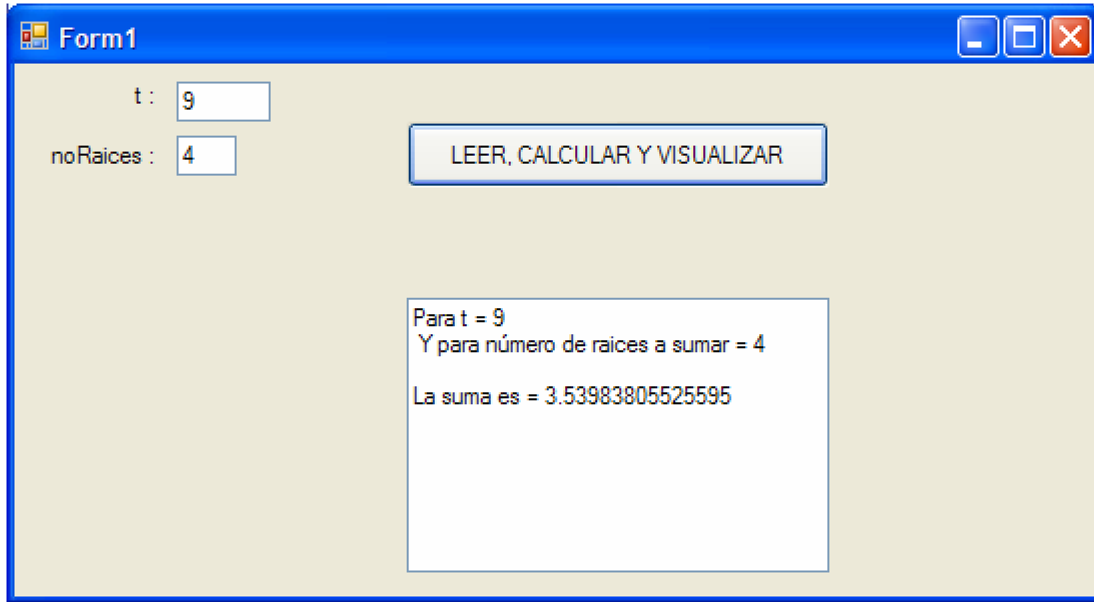


Fig. No. 7.4.12 Ejecución de la aplicación para `t=9`, `noRaices=4`.

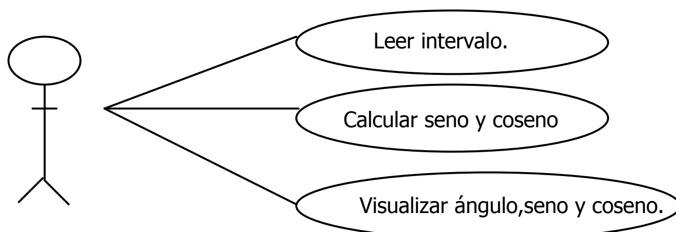
Pregunta.
Cambia el **do-while** en el método `EfecSumaRaices()` por un **while.**

Pregunta.
Cambia el **do-while** en el método `EfecSumaRaices()` por un **for.**

Pregunta.
Visualiza los resultados en un componente `Label`.

Ejercicio 7.4.10
Escribe una aplicación Windows C# que calcule y visualice las funciones seno y coseno de los ángulos desde 0° a 45º. El usuario de la aplicación ingresa en un TextBox el intervalo de ángulos para el cálculo, por ejemplo si ingresa un 1, el programa deberá visualizar el seno y coseno de los ángulos 0,1,2,3,...,45. Si ingresa un 2 se visualizan el seno y coseno de los ángulos 0,2,4,6,8,...,44. Si ingresa un 3 serán los ángulos 0,3,6,9,12,...,45.

El diagrama de casos de uso es :



Si seguimos la metodología *RAF* el diagrama de clases será la clase `Calculador`, con atributos `_intervalo`, `_seno`, `_coseno`, y los métodos que se muestran.

CALCULADOR
- _intervalo : double - _seno : double - _coseno : double
+ Leer(intervalo:double) : void + RetIntervalo() : double + RetSeno() : double + RetCoseno() : double + CalcSenCos(ang:double) : void

El método `CalcSenCos()` recibe un parámetro que contiene el valor del ángulo en grados `ang`, al que se requiere obtener las funciones seno y coseno.

Utilizaremos un componente TextBox con la propiedad `Multiline=true`, para visualizar el ángulo, el seno y el coseno, según se muestra en la interfase gráfica de la figura #7.4.13.

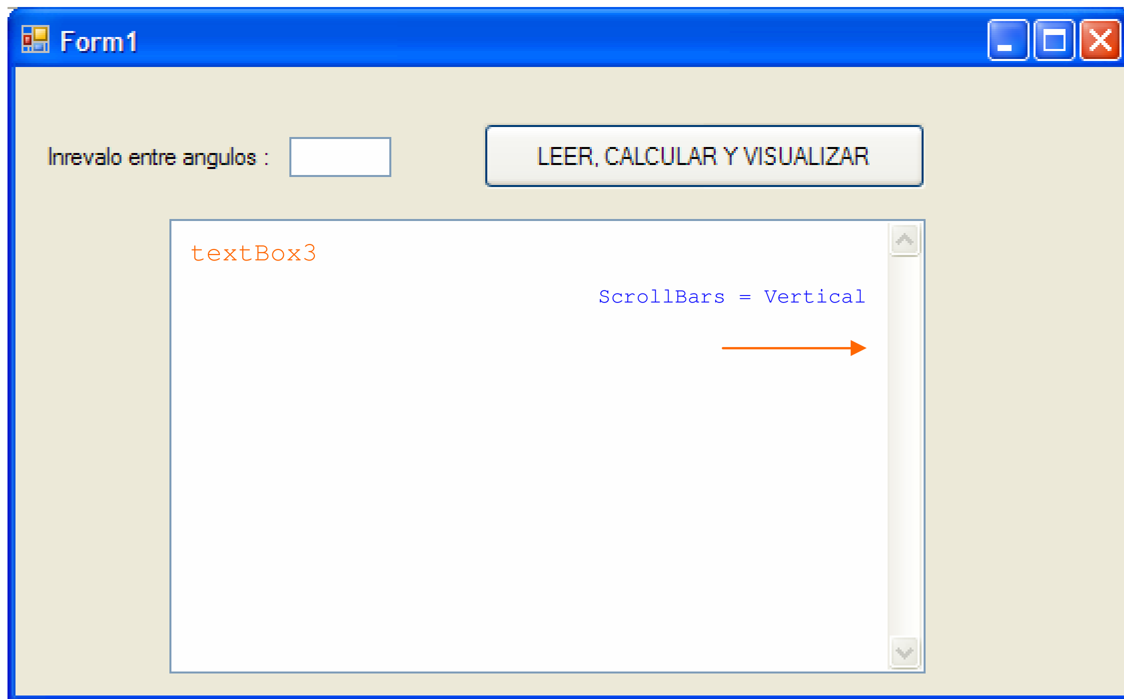


Fig. No. 7.4.13 Interfase gráfica, `textBox3` con su propiedad `ScrollBars=Vertical`.

También modifica la propiedad `Font` del componente `textBox3` a Courier New para tener caracteres con igual ancho, de manera que podamos formatear la salida. Agreguemos la clase `Calculador` al nuevo proyecto.

```
class Calculador
{
    private double _intervalo;
    private double _seno;
    private double _coseno;

    public void Leer(double Intervalo)
    {
       _intervalo = Intervalo;
    }
    public double RetIntervalo()
    {
       return _intervalo;
```

```
    }
    public double RetSeno()
    {
        return _seno;
    }
    public double RetCoseno()
    {
        return _coseno;
    }
    public void CalcSenCos(double ang)
    {
        _seno = Math.Sin(ang * Math.PI / 180.0);
        _coseno = Math.Cos(ang * Math.PI / 180.0);
    }
}
```

Para el cálculo del seno y del coseno, el argumento a los métodos `Sin` y `Cos` debe estar en radianes.
Así que debemos efectuar la conversión de acuerdo a : `180° = Π radianes`.

Definimos el objeto `oCalc` en la clase `Form1`. De una vez agregamos las constantes `MINANG` y `MAXANG` que representan los valores mínimo y máximo –inferior y superior- para los ángulos a los cuales se les calcula el seno y el coseno.

```
public partial class Form1 : Form
{
    const double MINANG = 0.0;
    const double MAXANG = 45.0;
    Calculador oCalc = new Calculador();
    ...
```

Los cálculos serán desde 0° a 45º.

Ahora sigamos con el método `button1_Click()` del botón con leyenda LEER, CALCULAR Y VISUALIZAR, en la clase `Form1`. El código deberá validar que el intervalo sea mayor a 0 y menor o igual que la constante `double MAXANG`.

```
private void button1_Click(object sender, EventArgs e)
{
    oCalc.Leer(Convert.ToDouble(textBox1.Text));
    if (oCalc.RetIntervalo() <= 0.0 || oCalc.RetIntervalo() > MAXANG)
        MessageBox.Show("ERROR ... EL VALOR DEL INTERVALO ESTA FUERA DE RANGO.");
    else
    {
        textBox2.Text = "ANGULO      SENO          COSENO\r\n";
        textBox2.Text += "--------------------------------\r\n";
    }
}
```

La aplicación en ejcución hasta este código es el de la figura #7.4.14.

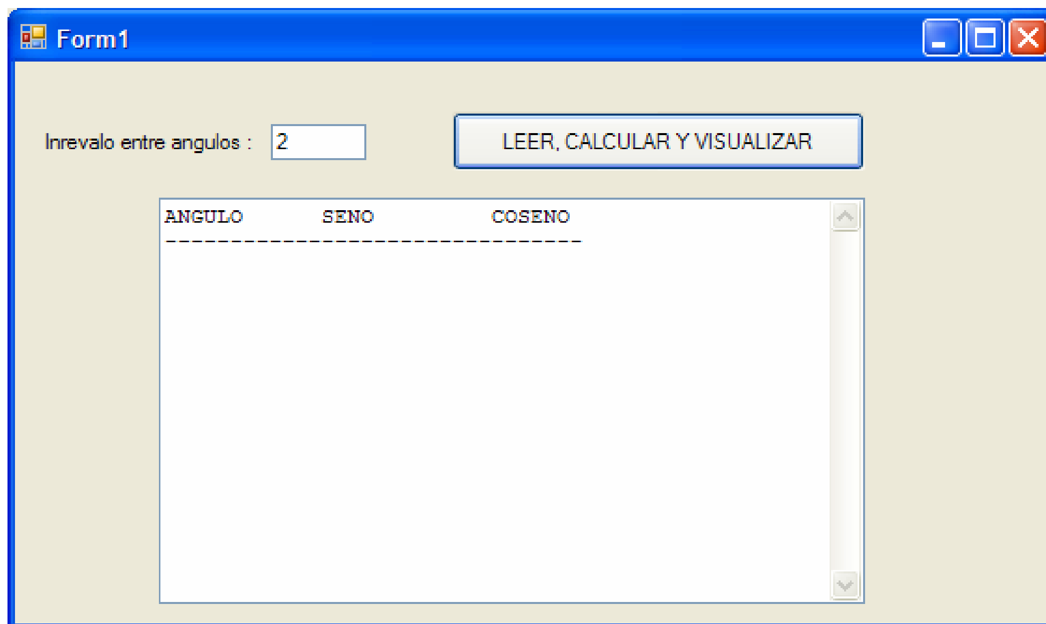


Fig. No. 7.4.14 Visualización del encabezado en el componente `textBox3`.

Después de observar la ejecución en la figura #7.4.14 ya podemos darnos una idea de lo que vamos a hacer para terminar de resolver nuestro problema. Visualizaremos en cada renglón la triada : ángulo (en grados), seno y coseno. Bien, agreguemos el código al método `button1_Click()` que realiza la visualización de cada triada según el intervalo leído.

```
private void button1_Click(object sender, EventArgs e)
{
    oCalc.Leer(Convert.ToDouble(textBox1.Text));
    if (oCalc.RetIntervalo() <= 0.0 || oCalc.RetIntervalo() > MAXANG)
        MessageBox.Show("ERROR ... EL VALOR DEL INTERVALO ESTA FUERA DE RANGO.");
    else
    {
        textBox2.Text = "ANGULO      SENO         COSENO\r\n";
        textBox2.Text += "--------------------------------\r\n";
        for (double ang = MINANG; ang <= MAXANG; ang += oCalc.RetIntervalo())
        {
            oCalc.CalcSenCos(ang);
            textBox2.Text += ang.ToString() + "        "+oCalc.RetSeno().ToString()+"        " +
                                                       oCalc.RetCoseno().ToString()+"\r\n";
        }
    }
}
```

El **for** está gobernado por la variable `ang` que es inicializada a la constante `MINANG` y que a su vez termina el ciclo repetitivo cuando `ang` es mayor que la constante `MAXANG`. En cada operación de iteración, la variable `ang` se incrementa las veces que dicta el valor del atributo `_intervalo` del objeto `oCalc` y que previamente habia sido leido.
Notemos que :

```
ang += oCalc.RetIntervalo()
```

es lo mismo que :

```
ang = ang + oCalc.RetIntervalo()
```

Dentro del cuerpo del **for** primero es calculado el valor del seno y del coseno para el valor del ángulo `ang` y entonces es visualizada la triada ángulo, seno y coseno en el componente `textBox2`. La figura #7.4.15 muestra la ejecución de la aplicación para un intervalo de 5.

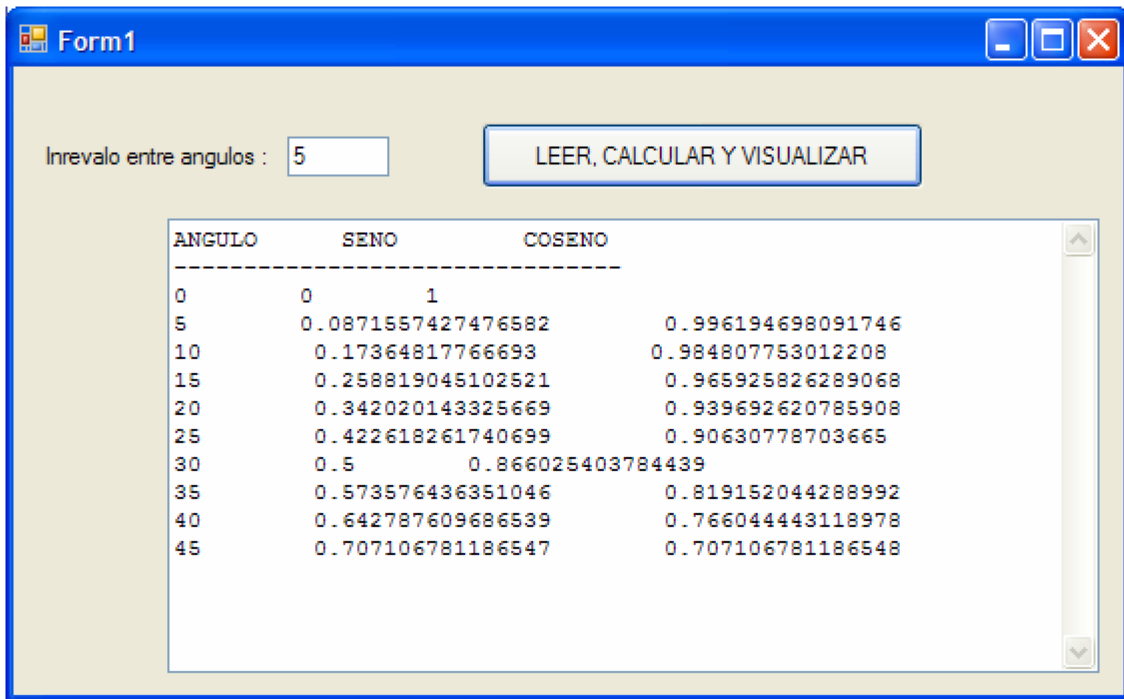


Fig. No. 7.4.15 Resultados para valor de intervalo = 5.

Notemos en la figura #7.4.15 que la salida no está formateada. "Formatear" significa en programación, darle forma a los datos de salida de manera que tengan una apariencia para su lectura por parte del usuario, legible, comprensible y ortogonal. Por ejemplo, una mejora a la salida del ángulo sería utilizar siempre 2 cifras en su visualización. De acuerdo a ésto, entonces el ángulo de 0o y 5º podríamos visualizarlos :

```
00   <----
05              Insertando un 0 –a la izquierda- para los valores de una cifra.
10
15
...
       ó
 0
 5   <----      Insertando un blanco –a la izquierda- para los valores de una cifra.
10
15
...
```

En este caso, nos inclinaremos por insertar un blanco a la izquierda si el valor del ángulo tiene una cifra. Emplearemos el operador ternario **? :** que tiene la forma **:**

```
cond ? valor1 : valor2
```

El operador ternario **? :** prueba la condición que antecede al caracter **?** y si es true retorna el `valor1`, si es false retorna el `valor2`. Entonces el operador **? :** siempre retorna un valor de manera que se puede incrustar en expresiones de asignación, cuando usamos la sentencia **return**, en cualquier operación donde se incluya un operando. Nosotros lo emplearemos según se indica en el código dentro de la rama `false` del **if** en el método `button1_Click()` :

```
private void button1_Click(object sender, EventArgs e)
{
    oCalc.Leer(Convert.ToDouble(textBox1.Text));
    if (oCalc.RetIntervalo() <= 0.0 || oCalc.RetIntervalo() > MAXANG)
        MessageBox.Show("ERROR ... EL VALOR DEL INTERVALO ESTA FUERA DE RANGO.");
    else
    {
        textBox2.Text = "ANGULO      SENO         COSENO\r\n";
        textBox2.Text += "---------------------------------\r\n";
        for (double ang = MINANG; ang <= MAXANG; ang += oCalc.RetIntervalo())
        {
             oCalc.CalcSenCos(ang);
             textBox2.Text += (ang<10 ? " "+ang.ToString() : ang.ToString()) + "        " +
                            oCalc.RetSeno().ToString()+"        " + oCalc.RetCoseno().ToString()+"\r\n";
        }
    }
}
```

Hemos introducido al operador **?:** dentro de la visualización empleando los paréntesis `( )` para dentro de ellos especificar el retorno de la cadena a visualizar para el dato `ang`, con un blanco a su izquierda o sin él.

```
textBox2.Text += (ang<10 ? " "+ang.ToString() : ang.ToString()) +
```

Notemos que hacemos la prueba si el valor de `ang` es menor a 10, si asi es entonces el ángulo tiene una sóla cifra por lo que debemos añadir un blanco a su izquierda. Ahora tenemos un mejor formateo en la salida según lo vemos en la figura #7.4.16.

Observamos también en la figura #7.4.16 que el valor de los senos y de los cosenos para los ángulos listados, no tienen un buen formato en su salida. La diferencia entre las maneras en que son visualizados estriba en el número de cifras enteras y el número de cifras decimales que se exhiben, ya que en algunos las cifras decimales no existen. en otros valores la cifra decimal es sólo una, y en la mayoría de los senos y cosenos las cifras decimales son 16.

Lo primero que debe hacerse es fijar el criterio de exhibición o de formato de la salida. En este caso, nos inclinamos por tener siempre 1 cifra entera, el punto decimal y 4 cifras decimales. ¿Cómo logramos este criterio en el formato de salida?.

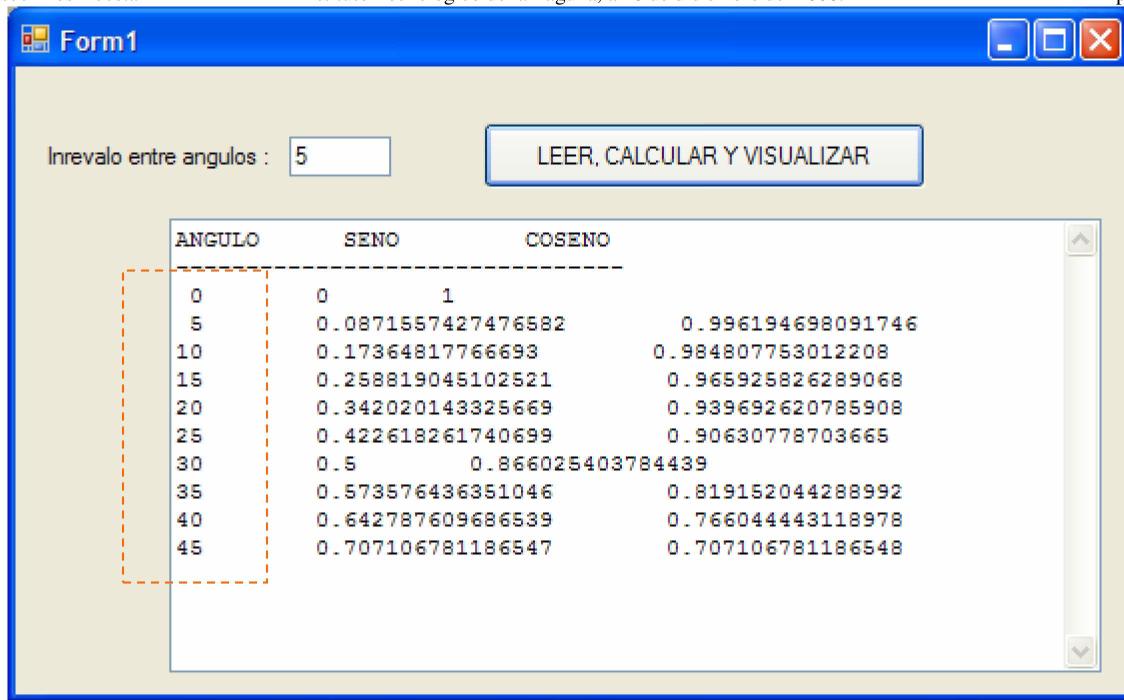


Fig. No. 7.4.16 Formateo de la salida para el valor del ángulo `ang`.

Usaremos el método `ToString(string formato)` para realizar el formato de la salida. La cadena de formato que se incluye como parámtero al método `ToString()`, indica la manera en que se exhibe el dato al que se le aplica dicho método.
Nosotros hemos utilizado el formato `“0.0000”` que indica que todos los lugares en que no haya cifra significativa, deberán rellenarse con un `0`. Utiliza como ejercicio el formato `“#.####”` para que observes lo que sucede.

La figura #7.4.17 muestra el formato de salida para los valores seno y coseno del ángulo `ang`.

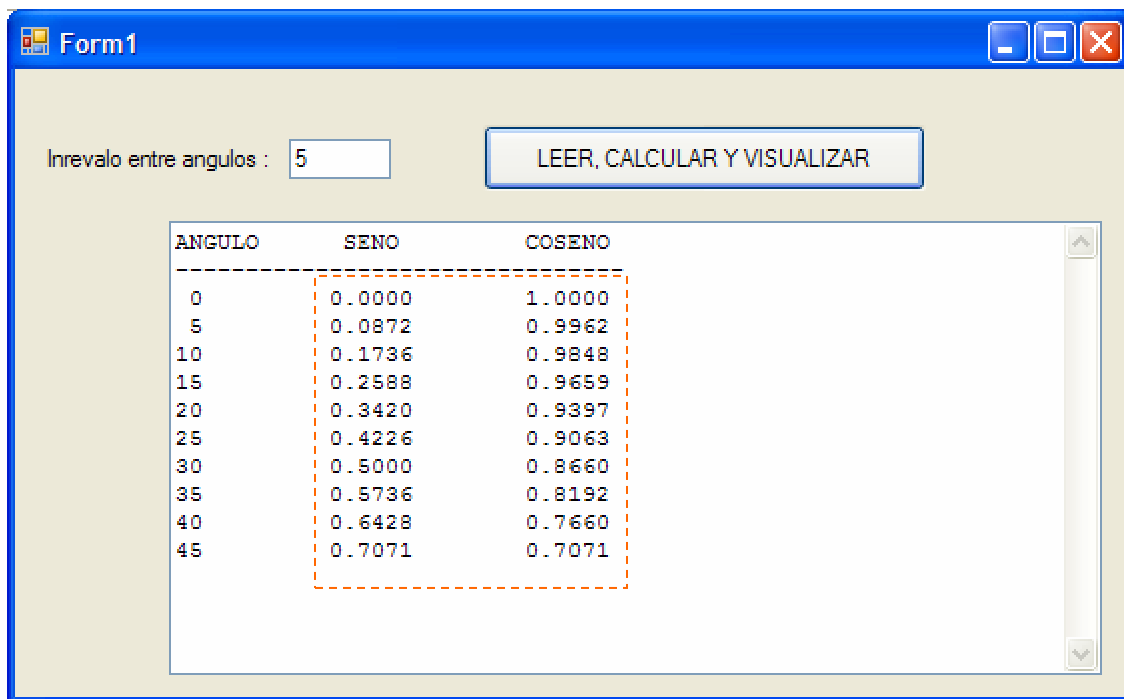


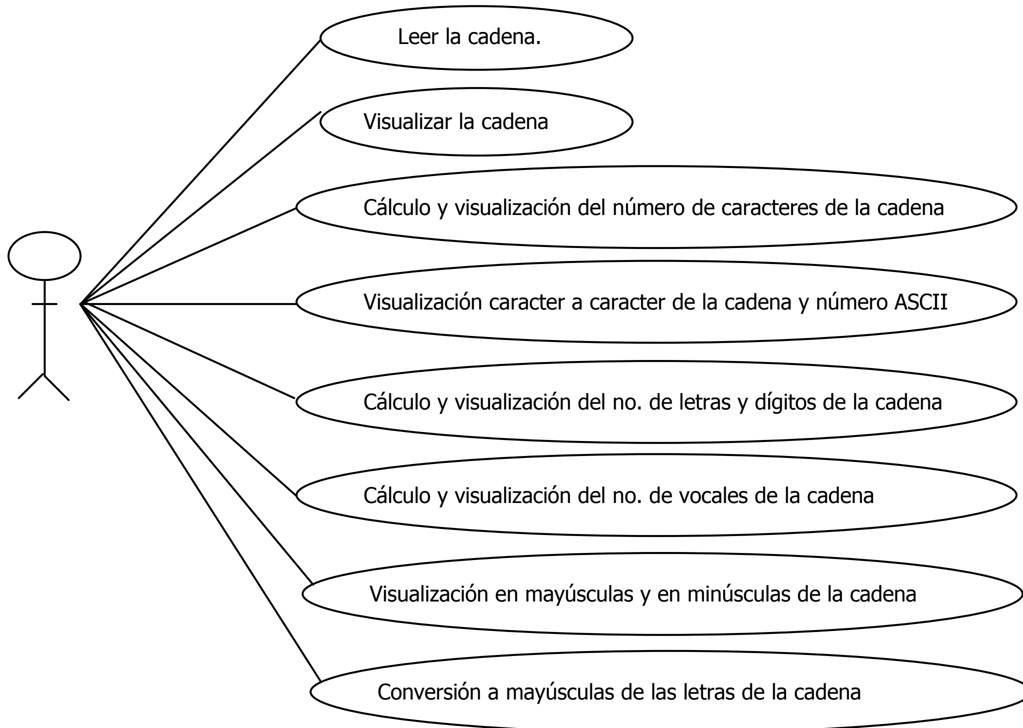
Fig. No. 7.4.17 Formateo de la salida para el seno y el coseno del ángulo.

El código que debemos cambiar en el método `button1_Click()` con el fin de lograr el formato en los valores de salida del seno y del coseno, es el mostrado a continuación :

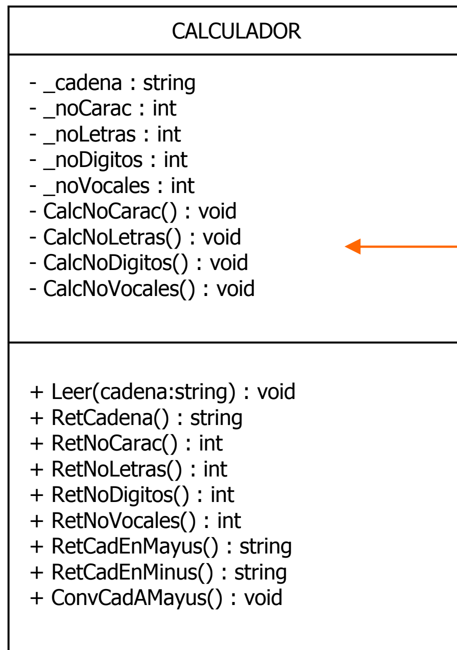
```
private void button1_Click(object sender, EventArgs e)
{
    oCalc.Leer(Convert.ToDouble(textBox1.Text));
    if (oCalc.RetIntervalo() <= 0.0 || oCalc.RetIntervalo() > MAXANG)
        MessageBox.Show("ERROR ... EL VALOR DEL INTERVALO ESTA FUERA DE RANGO.");
    else
    {
        textBox2.Text = "ANGULO      SENO         COSENO\r\n";
        textBox2.Text += "--------------------------------\r\n";
        for (double ang = MINANG; ang <= MAXANG; ang += oCalc.RetIntervalo())
        {
             oCalc.CalcSenCos(ang);
             textBox2.Text += (ang<10 ? " "+ang.ToString() : ang.ToString()) + "         " +
                                                 oCalc.RetSeno().ToString("0.0000")+"         "+
                                                 oCalc.RetCoseno().ToString("0.0000") + "\r\n";
        }
    }
}
```

Ejercicio 7.4.11
Escribe una aplicación Windows C# que lea una cadena de caracteres y que calcule y visualice el número de caracteres que contiene, visualice la cadena caracter a caracter separados por el caracter nueva línea y el número de código ASCII que le corresponda, el número de letras y de dígitos que contiene, el número de vocales que contiene. Además deberá visualizar la cadena en mayúsculas y en minúsculas. También deberá convertir sus letras que contenga a mayúsculas.

Este ejercicio tiene como fin manejar algunos métodos de la clase `string` y de la clase `Char`. Algunas de las operaciones constarán de iteraciones y otras no. Seguiremos utilizando al objeto `oCalc` con el fin de concentrarnos solamente en el manejo de los métodos y de no perder de vista el uso de objetos y clases para resolver un problema. Iniciemos con el diagrama de casos de uso :



El diagrama de clases sigue siendo la clase `Calculador` con atributos y métodos según se va explicando.



Notemos que ahora hemos definido 4 métodos privados : `CalcNoCarac()`, `CalcNoLetras()`, `CalcNoDigitos()` y `CalcNoVocales()`. Estos métodos no podrán ser accedidos en mensajes fuera de la clase `Calculador`. La definición de métodos `private` es posible hacerlo cuando dichos métodos sólo sean usados dentro de los métodos de la clase –en este caso la clase `Calculador`-.

Hagamos la creación de un nuevo proyecto y agreguemos la clase `Calculador` con los atributos y métodos especificados en el diagrama de clases.

```
class Calculador
{
    private string _cadena;
    private int _noCarac;
    private int _noLetras;
    private int _noDigitos;
    private int _noVocales;

    private void CalcNoCarac()
    {
       // por codificar
    }
    private void CalcNoLetras()
    {
       // por codificar
    }
    private void CalcNoDigitos()
    {
       // por codificar
    }
    private void CalcNoVocales()
    {
       // por codificar
    }
    public void Leer(string cadena)
    {
       // por codificar
    }
    public string RetCadena()
    {
        return _cadena;
    }
    public int RetNoCarac()
    {
        return _noCarac;
    }
    public int RetNoLetras()
    {
        return _noLetras;
    }
    public int RetNoDigitos()
```

```
    {
       return _noDigitos;
    }
    public int RetNoVocales()
    {
       return _noVocales;
    }
    public string RetCadEnMayus()
    {
       // por codificar
       return "";    // solo para que no reporte error al ejecutar
    }
    public string RetCadEnMinus()
    {
       // por codificar
       return "";     // solo para que no reporte error al ejecutar
    }
    public void ConvMayMinMinMay()
    {
      // por codificar
    }
}
```

No incluirlas. Al final no se utilizarán.

Definimos el objeto `oCalc` en la clase `Form1`.

```
public partial class Form1 : Form
{
    Calculador oCalc = new Calculador();
    public Form1()
    {
       InitializeComponent();
    }
}
```

La interfase gráfica con la que trabajaremos inicialmente es la vista en la figura #7.4.18, donde se ve el componente `textBox1` que servirá para ingresar la cadena. El botón `button1` hará la función de leer y visualizar la cadena de trabajo. Usaremos un componente `textBox2` para visualizar algunos resultados los cuales habremos de mencionar conforme vayamos construyendo el código de la aplicación.

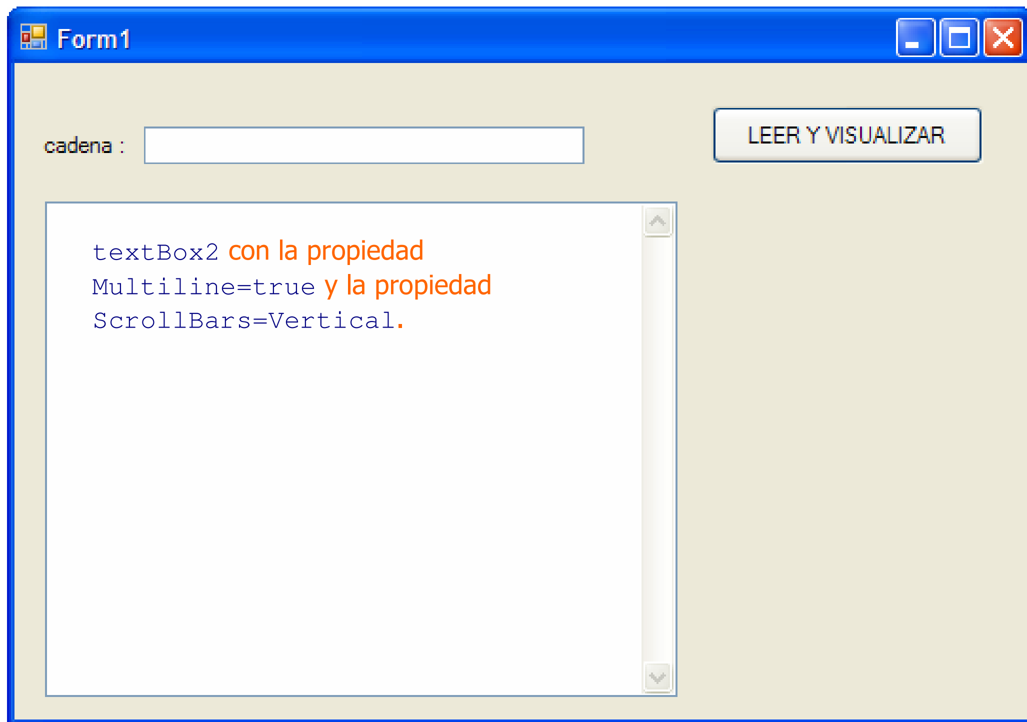


Fig. No. 7.4.18 Interfase gráfica inicial.

Iniciamos con las tareas mas simples : lectura y visualización de la cadena. El método `button1_Click()` es el encargado de realizar dichas tareas.

```
private void button1_Click(object sender, EventArgs e)
{
    oCalc.Leer(textBox1.Text);
    textBox2.Text = "La cadena tecleada es : " + oCalc.RetCadena();
}
```

Desde luego que debemos escribir el código para el método `Leer()` de la clase `Calculador`.

```
public void Leer(string cadena)
{
    _cadena = cadena;
}
```

La figura #7.4.19 muestra la ejecución de la aplicación hasta este estado del código.

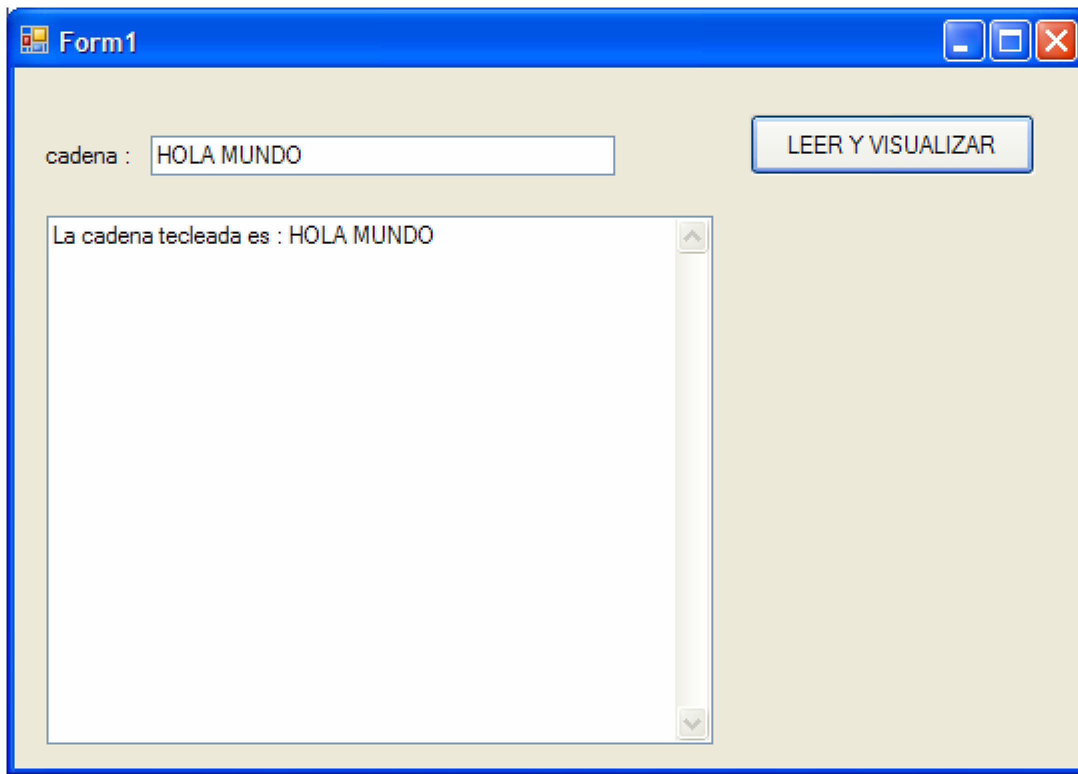


Fig. No. 7.4.19 Lectura y visualización de la cadena HOLA MUNDO.

La tarea siguiente en el diagrama de casos de uso es el cálculo y visualización del número de caracteres que contiene la cadena leida. Para ello vamos a utilizar la propiedad `Length` descrita en la clase `string`, que precisamente retorna el número de caracteres de la cadena a la cual se le aplica la propiedad. El cálculo del número de caracteres lo agregaremos dentro del método `Leer()` de la clase `Calculador`.

```
public void Leer(string cadena)
{
     _cadena = cadena;
     CalcNoCarac();
}
```

Notemos que el cálculo lo hacemos mediante la llamada al método **private** `CalcNoCarac()` de la clase `Calculador`.

```
private void CalcNoCarac()
{
    _noCarac = _cadena.Length;
}
```

El método `CalcNoCarac()` tiene el código que se muestra. La propiedad `Length` se aplica al atributo `_cadena`, valor que lo asigna al atributo `_noCarac`.

Vamos a añadir la visualización del número de caracteres de la cadena en el mismo `button1`, así que agreguemos el código que efectúa dicha cuestión en el método `button1_Click()` :

```
private void button1_Click(object sender, EventArgs e)
{
    oCalc.Leer(textBox1.Text);
    textBox2.Text = "La cadena tecleada es : " + oCalc.RetCadena();
    textBox2.Text += "\r\n------------------------------------------\r\n";
    textBox2.Text += "# de caracteres es : " + oCalc.RetNoCarac().ToString();
}
```

El método `RetNoCarac()` ya estaba codificado.

La corrida la observamos en la figura #7.4.20.

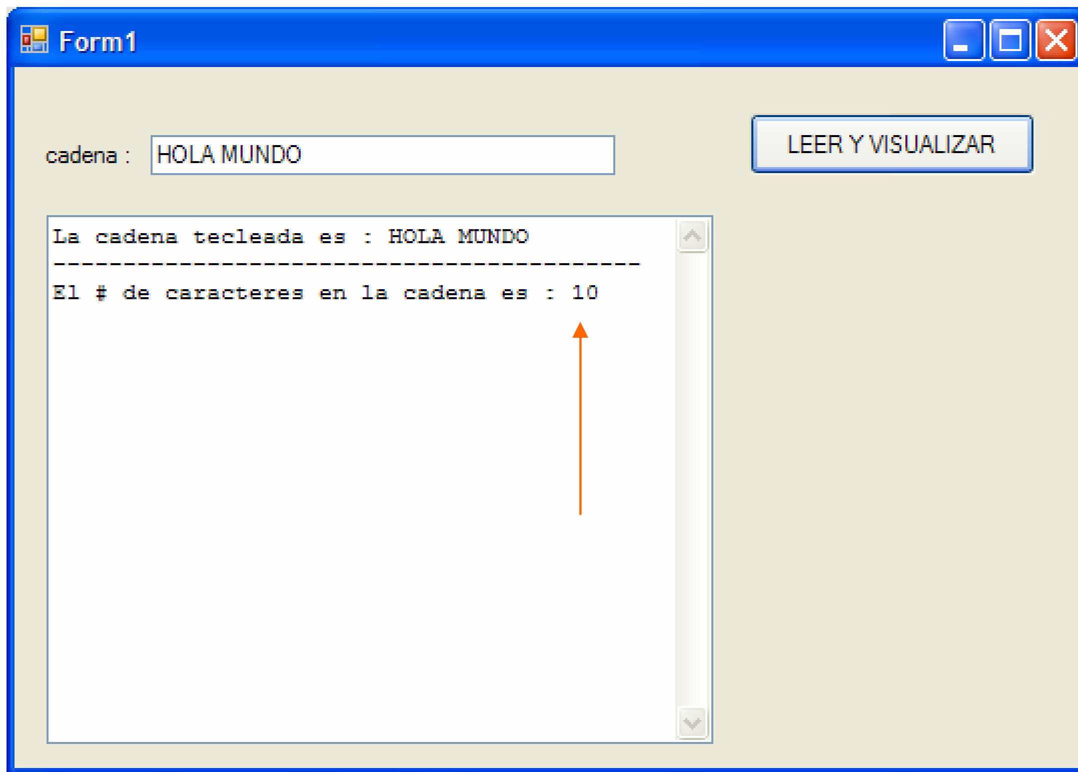


Fig. No. 7.4.20 Visualización del # de caracteres en la cadena leida.

Sigamos con la visualización caracter a caracter de la cadena leida junto a su respectivo número de código ASCII. Utilizaremos el mismo `textBox2` para efectuar en él la visualización de esta tarea. Para ejecutar esta acción añadimos un nuevo botón `button2` y el código para su método `button2_Click()` es :

```
private void button2_Click(object sender, EventArgs e)
{
    textBox2.Text += "\r\n------------------------------------------\r\n";
    textBox2.Text+=oCalc.RetCadena()+" - caracter a caracter ,  ASCII : \r\n";
    for (int i = 0; i < oCalc.RetNoCarac(); i++)
        textBox2.Text += oCalc.RetCadena().Substring(i, 1) + "      " +
                  Convert.ToInt32(Convert.ToChar(oCalc.RetCadena().Substring(i, 1))).ToString()+"\r\n";
}
```

La condición en el **for** permite la iteración desde el primer caracter en la cadena –índice 0- hasta el último caracter de ella `_noCarac-1`. Por ejemplo, si la cadena es "HOLA" la cadena tendrá 4 caracteres : el caracter 0 es 'H', el caracter 1 es 'O', el 2 es el 'L' y el caracter 3 es la 'A'. Por eso, hemos utilizado el operador relacional < en la condición :

```
for (int i = 0; i < oCalc.RetNoCarac(); i++)
```

En la operación repetitiva –cuerpo del **for**-, para acceder al i-ésimo caracter en la cadena leida empleamos el método `Substring()` de la clase **string**. `Substring()` permite acceder a una parte (subcadena) de los caracteres de la cadena y retorna a dicha subcadena. Este método recibe 2 parámetros : el primero indica el índice del caracter –número de caracter- a partir del cual queremos la subcadena, y el segundo es otro entero que indica cuántos caracteres queremos en la subcadena. Por ejemplo, si la cadena es "HOLA" aplicando `Substring(1,2)` tendremos el retorno de "OL". La manera en que accedemos a cada caracter en la cadena y lo visualizamos, es precisamente utilizando el método `Substring()` sobre la cadena que retorna el mensaje `oCalc.RetCadena()`.

```
textBox2.Text += oCalc.RetCadena().Substring(i, 1) + "        " +
                 Convert.ToInt32(Convert.ToChar(oCalc.RetCadena().Substring(i, 1))).ToString()+"\r\n";
```

La figura #7.4.21 mestra la visualización caracter a caracter de la cadena leida "HOLA MUNDO", y el código ASCII correspondiente a cada caracter visualizado –por renglón-.

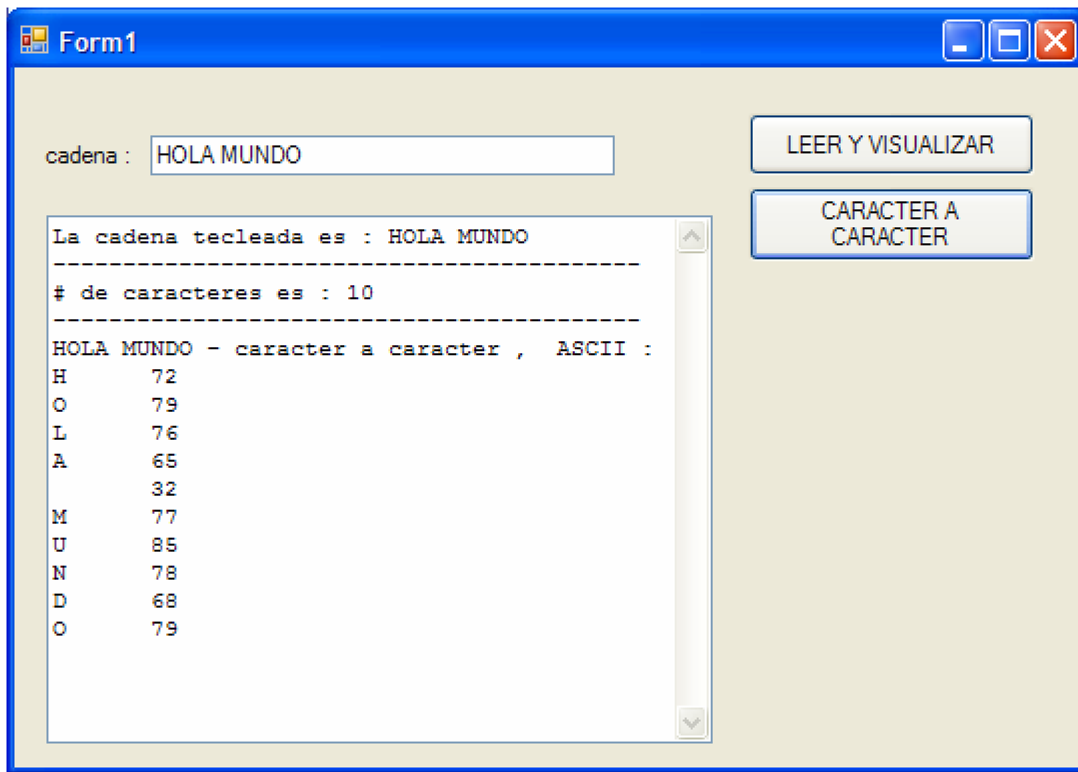


Fig. No. 7.4.21 Visualización caracter a caracter y su código ASCII.

El código ASCII es un número entero por medio del cual, los programas de computadora conocen a los caracteres. Lo anterior significa, que la computadora no es como el humano que reconoce a los caracteres de manera gráfica, sino que los reconoce como un número. La forma en que visualizamos al código ASCII de cada caracter que visualizamos y que forma parte de la cadena, es convirtiendo la subcadena retornada por el mensaje compuesto a tipo `char` :

```
oCalc.RetCadena().Substring(i, 1)
```

Retorna una cadena –subcadena-. Una subcadena es también una cadena.

La conversión a caracter –`char`- la hacemos por medio de la clase Convert con el método `ToChar` según se observa en el código que forma la parte del cuerpo del **for** :

```
Convert.ToChar(oCalc.RetCadena().Substring(i, 1))
```

Aquí ya tenemos el retorno de un `char`, ya que la subcadena siempre tendrá un sólo caracter, de manera que la conversión es válida. La conversión a `char` no sería correcta si la subcadena tuviera 2 o mas caracteres. Bien, ahora ya tenemos el caracter al cual, si le aplicamos una conversión a entero –ya que la computadora conoce a un caracter por su número entero ASCII- tendremos precisamente el código ASCII que le corresponde o como es conocido por la computadora.

Por eso, es que hemos aplicado el método `ToInt32` de la clase `Convert` al caracter que retorna la conversión descrita en el párrafo anterior.

```
Convert.ToInt32(Convert.ToChar(oCalc.RetCadena().Substring(i, 1)))
```

Por último, dado que la propiedad Text del componente `textBox2` es de tipo **string**, se convierte el entero –ASCII- a **string** por medio del método `ToString()`.

```
Convert.ToInt32(Convert.ToChar(oCalc.RetCadena().Substring(i, 1))).ToString()
```

La sentencia final en el cuerpo del **for** que se itera en cada ciclo es :

```
private void button2_Click(object sender, EventArgs e)
{
    textBox2.Text += "\r\n-------------------------------------------\r\n";
    textBox2.Text+=oCalc.RetCadena()+" - caracter a caracter ,  ASCII : \r\n";
    for (int i = 0; i < oCalc.RetNoCarac(); i++)
        textBox2.Text += oCalc.RetCadena().Substring(i, 1) + "       " +
                 Convert.ToInt32(Convert.ToChar(oCalc.RetCadena().Substring(i, 1))).ToString()+"\r\n";
}
```

Veamos el proceso de conversión que se empleó; si la cadena tecleada es "HOLA" tenemos que cuando entramos por primera vez al **for**, la `i` es = 0. Podemos ver a la cadena de la forma siguiente :

índice de caracter –# de caracter-	**caracter**
0	'H'
1	'O'
2	'L'
3	'A'

Entonces el mensaje :

`oCalc.RetCadena().Substring(i, 1)` retorna la cadena "H" ya que i=0 Substring(0,1) es = "H".

Notemos que hemos escrito la constante cadena como "H" entre comillas, y así debe de ser ya que una constante cadena se encierra entre comillas.

Luego esta cadena "H" la convertimos a caracter por medio del método `ToChar()` de la clase `Convert` :

```
Convert.ToChar(oCalc.RetCadena().Substring(i, 1))
```

Por pasos, podemos apreciar la sentencia anterior como :

`Convert.ToChar("H")` cuyo resultado es el retorno de un caracter 'H'.

'H' entre apóstrofes ya que es una constante tipo char.

Luego que se tiene ya un tipo `char` –un caracter- como la computadora conoce a un determinado caracter como un entero –código ASCII-, seguimos con la conversión a entero usando el método `ToInt32` de la clase `Convert`.

```
Convert.ToInt32(Convert.ToChar(oCalc.RetCadena().Substring(i, 1)))
```

Sentencia que podemos escribir como :

```
Convert.ToInt32('H')
```

Conversión que retorna el entero 72 que podemos comprobar en la figura 7.4.21 cuando se visualiza el caracter 'H'

Volviendo al diagrama de casos de uso, la tarea que procede ahora es el cálculo y visualización del no. de letras y dígitos de la cadena. Para saber si un caracter en la cadena es una letra o es un dígito, tenemos precisamente que acceder a cada uno de los caracteres y aplicarles un método de la clase `Char` sea `IsLetter()` para conocer si es letra, sea `IsDigit()` para saber si es dígito. Recordemos el código del método `Leer()` en la clase `Calculador` :

```
public void Leer(string cadena)
{
     _cadena = cadena;
     CalcNoCarac();
}
```

Agreguémosle la llamada a los métodos privados `CalcNoLetras()` y `CalcNoDigitos()`, para luego definirlos en la misma clase `Calculador`.

```
public void Leer(string cadena)
{
     _cadena = cadena;
     CalcNoCarac();
     CalcNoLetras();
     CalcNoDigitos();
}
```

Vamos a seguir con la definición del método `CalcNoLetras()` en el que accedemos a cada caracter en la cadena usando el método `Substring()`, para luego probar si es letra incrementando el contador de letras. El método `CalcNoDigitos()` es similar a `CalcNoLetras()` sólo cambia el atributo que se incrementa `_noDigitos` por `_noLetras` según corresponde en el método.

```
private void CalcNoLetras()
{
     _noLetras=0;
     for(int i=0;i<_noCarac;i++)
     {
         char c=Convert.ToChar(_cadena.Substring(i,1));
         _noLetras = Char.IsLetter(c) ? _noLetras+1 : _noLetras;
     }
}

private void CalcNoDigitos()
{
     _noDigitos=0;
     for(int i=0;i<_noCarac;i++)
     {
         char c=Convert.ToChar(_cadena.Substring(i,1));
         _noDigitos = Char.IsDigit(c) ? _noDigitos+1 : _noDigitos;
     }
}
```

Se han separado en dos sentencias la obtención del caracter i-ésimo y el incremento del atributo `_noLetras` o `_noDigitos` según corresponda. El incremento se realiza usando el operador ternario **? :**

Modifiquemos la interfase gráfica tal y como se muestra en la figura #7.4.22 añadiendo un nuevo botón `button3` que visualice el número de letras y de dígitos en la cadena leida. En dicha figura también se muestra la ejecución para la cadena leida "FZ10 TRI", y los resultados que visualiza el botón `button3`. El código del método `button3_Click()` es :

```
private void button3_Click(object sender, EventArgs e)
{
    textBox2.Text += "-----------------------------------------\r\n";
    textBox2.Text += oCalc.RetCadena() + " TIENE : "+oCalc.RetNoLetras().ToString()+" LETRAS.\r\n";
    textBox2.Text += "Y TIENE : " + oCalc.RetNoDigitos().ToString() + " DIGITOS.\r\n";
}
```

El método `button3_Click()` lo definimos en la clase `Form1`, y hace uso de 2 métodos en 2 mensajes al objeto `oCalc`, previamente definidos en la clase `Calculador` :

`oCalc.RetNoLetras()` y

`oCalc.RetNoDigitos()`

```
public int RetNoLetras()
{
    return _noLetras;
}
public int RetNoDigitos()
{
    return _noDigitos;
}
```

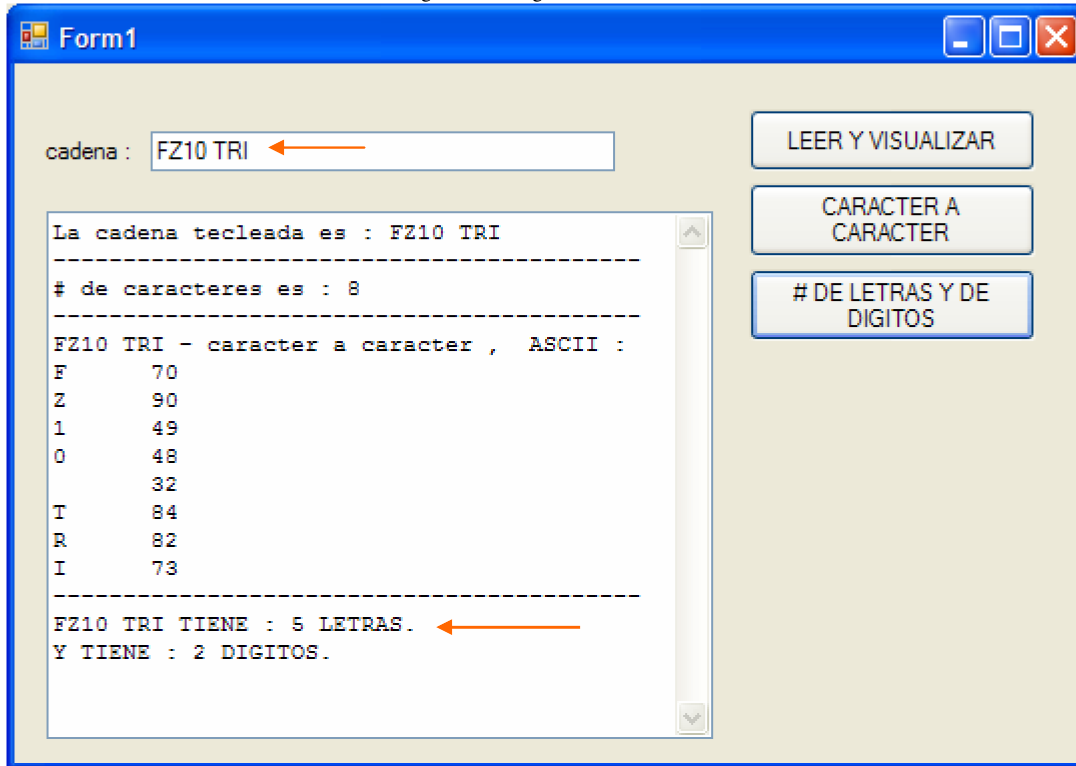


Fig. No. 7.4.22 Nuevo botón `button3` y los resultados para la cadena "FZ10 TRI".

La tarea en el diagrama de casos de uso siguiente es la de calcular el número de vocales que contiene la cadena leida. El número de vocales es un atributo que ya hemos añadido previamente. Lo que necesitamos ahora es añadir la llamada al método privado `CalcNoVocales()` dentro del método `Leer()` :

```
public void Leer(string cadena)
{
      _cadena = cadena;
      CalcNoCarac();
      CalcNoLetras();
      CalcNoDigitos();
      CalcNoVocales();
}
```

Llamada al método `CalcNoVocales()` incluida en el bloque del método `Leer()` en la clase `Calculador`.

La definición del método `CalcNoVocales()` tiene el código :

```
private void CalcNoVocales()
{
      _noVocales = 0;
      for (int i = 0; i < _noCarac; i++)
          _noVocales = ("AEIOUaeiou").IndexOf(_cadena.Substring(i, 1))>=0 ? _noVocales + 1 : _noVocales;
}
```

El **for** compara el valor de la variable `i` con el número de caracteres en la cadena `_noCarac` que ha sido calculado previamente durante la lectura de la cadena usando el método `CalcNoCarac()`.

```
for (int i = 0; i < _noCarac; i++)
```

Es decir, con el **for** recorremos todos los caracteres del atributo `_cadena` que contiene la cadena leida. Luego dentro del cuerpo del **for**, incrementamos el atributo `_noVocales` sólo si el i-ésimo caracter en el atributo `_cadena` es una vocal.

```
_noVocales = ("AEIOUaeiou").IndexOf(_cadena.Substring(i, 1))>= 0 ? _noVocales + 1 : _noVocales;
```

Notemos que hemos vuelto a usar el operador ternario **? :** en el método `CalcNoVocales()` dentro del **for** para incrementar el número de vocales `_noVocales` si así fuere el caso. Vayamos paso a paso en la sentencia que contiene al operador ternario mencionado, poniendo como ejemplo la cadena "FZ10 TRI" cuyo resultado del número de vocales es `1` y que se muestra en la figura #7.4.23.

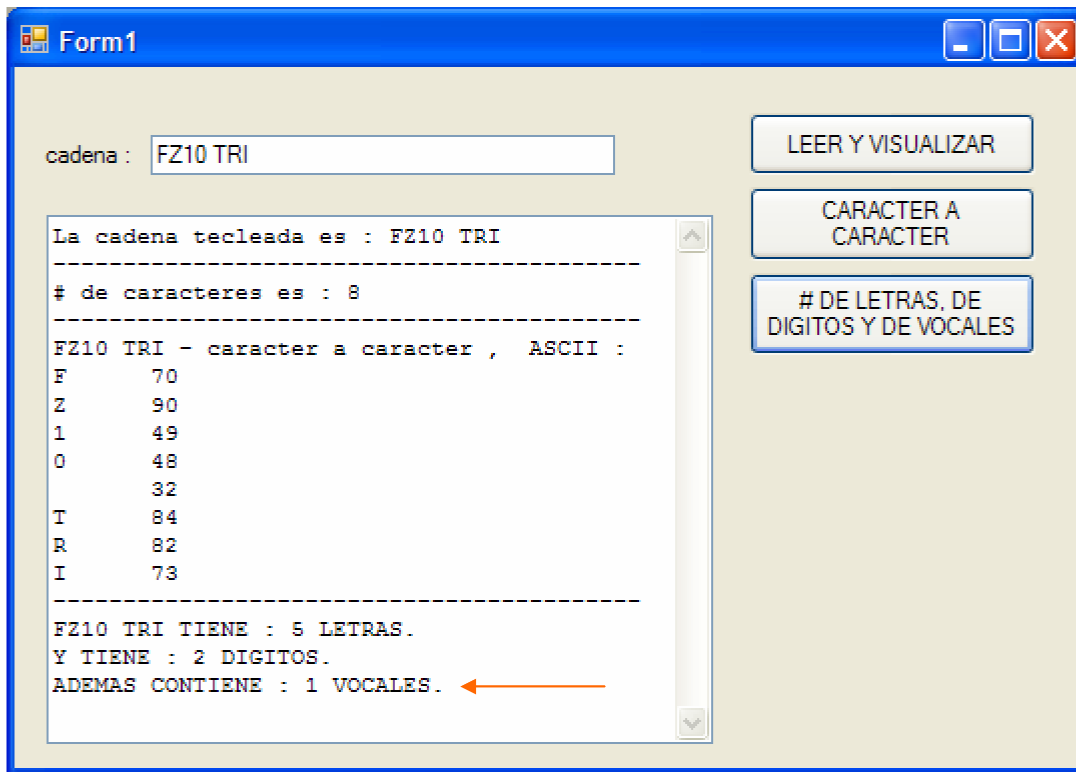


Fig. No. 7.4.23 No. de vocales = 1, para la cadena "FZ10 TRI".

Por ejemplo para `i = 0`, el valor de `_cadena.Substring(0, 1)` es la subcadena "F".

De manera que en el segmento de código :

```
for (int i = 0; i < _noCarac; i++)
     _noVocales = ("AEIOUaeiou").IndexOf(_cadena.Substring(i, 1))>=0 ? _noVocales + 1 : _noVocales;
```

Podemos sutituir por :

```
for (int i = 0; i < _noCarac; i++)
    _noVocales = ("AEIOUaeiou").IndexOf("F")>=0 ? _noVocales + 1 : _noVocales;
```

El método `IndexOf()` de la clase **`string`** precisamente se aplica a una cadena constante "AEIOUaeiou" la que debemos encerrar entre paréntesis. El método `IndexOf()` está sobrecargado para manejo de parámetros de 9 maneras posibles, y uno de ellos es el que utilizamos, el parámetro tipo **`string`** –en este caso la cadena "F"-. La función que realiza el método `IndexOf()` es buscar la cadena pasada como parámetro en la cadena objeto del mensaje, en nuestro caso busca a `"F"` dentro de la cadena `("AEIOUaeiou")`. Si la encuentra retorna el índice –número de caracter dentro de la cadena- donde inicia la subcadena buscada, este índice es un entero mayor o igual que 0 hasta `_noCarac-1` . Si no la encuentra entonces retorna un entero negativo. Por eso es que en el operador ternario la condición prueba que el entero retornado por `IndexOf()` sea mayor o igual que 0, ya que si es `true` obviamente es una vocal. En nuestro ejemplo la condición retorna un `false` para la subcadena `"F"`.

```
for (int i = 0; i < _noCarac; i++)
    _noVocales = ("AEIOUaeiou").IndexOf("F")>=0 ? _noVocales + 1 : _noVocales;
```

`false`, por lo que se asigna el mismo valor `_noVocales` al atributo `_noVocales`.

En la figura #7.4.23 el número de vocales visualizado es 1 debido a que la cadena ingresada "FZ10 TRI" tiene una **I** al final de ella. La posición de la I en la cadena es el caracter 7, asi que el mensaje :

```
_cadena.Substring(7, 1)
```

Retorna la cadena "I", que luego es buscada por el método IndexOf() aplicado al objeto tipo string `("AEIOUaeiou")` que en este caso si encuentra la "I" en dicha cadena, por lo que la expresión :

`("AEIOUaeiou").IndexOf("F")>=0` ← retorna el entero 7 o sea un entero mayor o igual que 0.

Debido a que la condición en el operador ternario al ser evaluada retorna un `true`, el atributo `_noVocales` recibe la asignación del incremento `_noVocales + 1`

```
for (int i = 0; i < _noCarac; i++)
    _noVocales = ("AEIOUaeiou").IndexOf("F")>=0 ? _noVocales + 1 : _noVocales;
```

`true`, por lo que se asigna el incremento `_noVocales + 1` al atributo `_noVocales`.

La siguiente tarea en el diagrama de casos de uso es la visualización de la cadena tanto en mayúsculas y en minúsculas, sin afectar o cambiar su valor –el de la cadena-. Para efectuar esta función vamos a añadir un nuevo botón `button4` en la interfase gráfica según se ve en la figura #7.4.24, con el siguiente código :

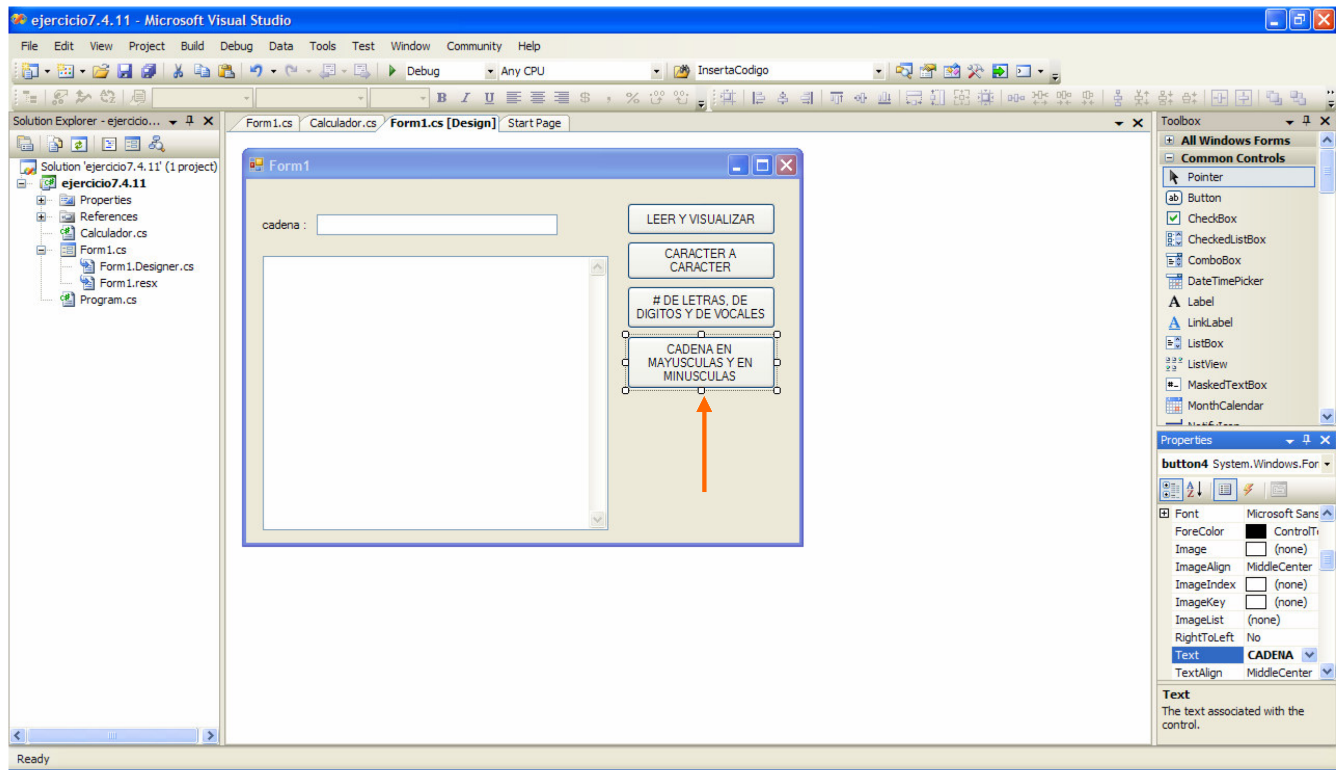


Fig. No. 7.4.24 Nuevo botón `button4` en la interfase gráfica.

```
private void button4_Click(object sender, EventArgs e)
{
    textBox2.Text += "\r\n-------------------------------------------\r\n";
    textBox2.Text += oCalc.RetCadena() + " ANTES DE LA CONVERSION.\r\n";
    textBox2.Text += oCalc.RetCadena().ToUpper() + " EN MAYUSCULAS.\r\n";
    textBox2.Text += oCalc.RetCadena() + " ANTES DE LA CONVERSION.\r\n";
    textBox2.Text += oCalc.RetCadena().ToLower() + " en minúsculas.\r\n";
}
```

Hechando un vistazo al código anterior, notemos que se están utilizando 2 métodos sobre objetos de tipo **`string`** :

ToUpper() y ToLower() para convertir una cadena a mayúsculas y a minúsculas respectivamente. Mencionaremos algo importante en el sentido de que la cadena sobre la cual se envían los mensajes que contienen a estos métodos, NO SE MODIFICA. En nuestro caso la aplicación no afecta al atributo _cadena el cual es retornado por el mensaje que hace uso del método RetCadena().

```
textBox2.Text += oCalc.RetCadena().ToUpper() + " EN MAYUSCULAS.\r\n";
```

mensaje que retorna al atributo _cadena

ToUpper() retorna al atributo _cadena pero con sus letras en mayúsculas.

Algo similar sucede para la conversión a minúsculas :

```
textBox2.Text += oCalc.RetCadena().ToLower() + " en minúsculas.\r\n";
```

mensaje que retorna al atributo _cadena

ToLower() retorna al atributo _cadena pero con sus letras en minúsculas.

Un ejemplo del comportamiento de la aplicación para cuando se lee la cadena "Hola Mundo" es el mostrado en la figura 7.4.24.

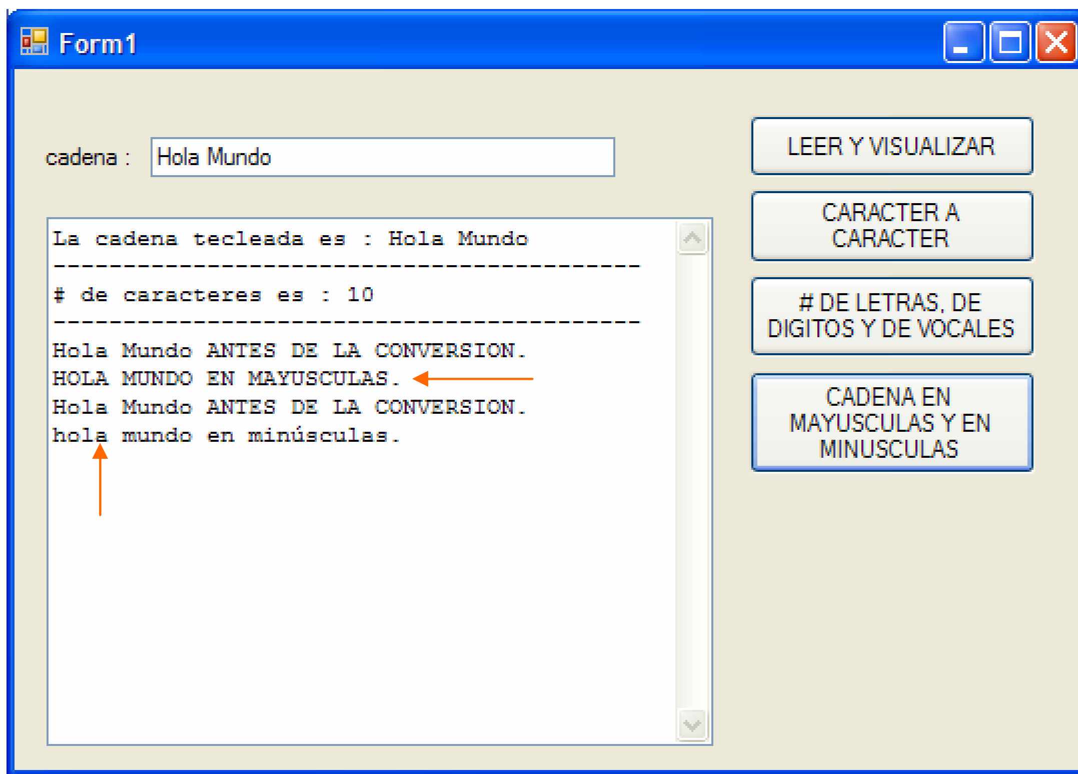


Fig. No. 7.4.24 Visualización de la cadena leida "Hola Mundo" en mayúsculas y en minúsculas.

Por último, tenemos la tarea de convertir a mayúsculas las letras minúsculas y a minúsculas las letras mayúsculas que forman parte de la cadena. Por ejemplo, si tenemos la cadena leida "Hola Mundo" después de la conversión la cadena tendrá el valor de "hOLA mUNDO". Notemos que en este caso, si existen letras en la cadena ingresada ésta SI SERÁ MODIFICADA.
El algoritmo que escribiremos tendrá que recorrer cada caracter de la cadena y si es letra deberá efectuar la conversión mayúscula-minúscula o bien minúscula-mayúscula.
Por lo pronto tenemos que agregar un nuevo botón button5, que realice la acción de la conversión. Una vez que lo hagamos, escribimos el código del método button5_Click() que mostramos a continuación :

```
private void button5_Click(object sender, EventArgs e)
{
      textBox2.Text += "\r\n-----------------------------------------\r\n";
      textBox2.Text += oCalc.RetCadena() + " ANTES DE LA CONVERSION.\r\n";
      oCalc.ConvMayMinMinMay();
      textBox2.Text += oCalc.RetCadena() + " DESPUES DE HECHA LA CONVERSION.\r\n";
}
```

El método que se encarga de efectuar la conversión es `ConvMayMinMinMay()`. El mensaje con este método es enviado al objeto `oCalc` de manera que la conversión tenga lugar.

```
oCalc.ConvMayMinMinMay();
```

Falta definir el método `ConvMayMinMinMay()` en la clase `Calculador`, así que agreguemos el código para este método :

```
public void ConvMayMinMinMay()
{
    for (int i = 0; i < _noCarac; i++)
    {
        char c = Convert.ToChar(_cadena.Substring(i, 1));
        if (Char.IsLetter(c))
        {
            c = Char.IsUpper(c) ? Char.ToLower(c) : Char.ToUpper(c);
            _cadena=_cadena.Remove(i, 1);
            _cadena=_cadena.Insert(i, c.ToString());
        }
    }
}
```

La ejecución de la aplicación se muestra en la figura #7.4.25.

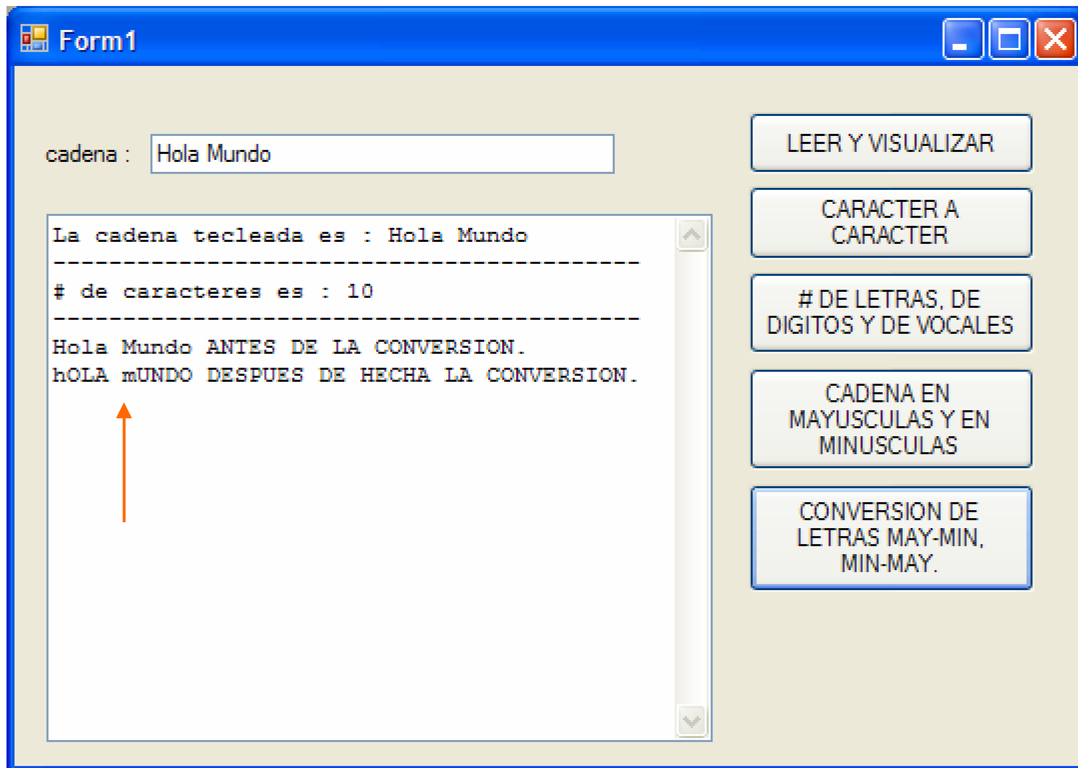


Fig. No. 7.4.25 Conversión mayúsculas-minúsculas, minúsculas-mayúsculas.

Dejamos de ejercicio al alumno-maestro la discusión sobre el código del método `ConvMayMinMinMay()`.