Apuntes de Fundamentos de Programación.

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.

5 Implementación de la clase.

La definición de una clase en C# -su codificación o implementación- es por medio de la palabra reservada **class** seguida del identificador o nombre de la clase y tanto sus atributos como sus métodos, encerrados –definidos- entre llaves. Por ejemplo la clase `Alumno` :

```
class Alumno
{
    // definición de atributos

    // definición de métodos
}
```

Tanto en C# como en otros lenguajes muy usados por los programadores hoy en día, cada clase reside en un archivo donde es definida. Si tenemos 3 clases digamos `Alumno`, `Profesor`, `Grupo`, entonces tendremos 3 archivos uno por cada clase.
En la sección 4.6 explicamos como una clase es agregada a un proyecto –aplicación-, de manera que seguiremos con otros conceptos tales como :

- Modificadores de acceso para atributos y métodos.
- Definición de atributos y métodos de una clase.
- Parámetros de entrada y de salida.
- Constructores.
- Destructores.

5.1 Modificadores de acceso - public y private -.

El uso de los modificadores de acceso constituyen una cuestión fundamental en la programación orientada a objetos. Estos permiten la buena o la mala programación orientada a objetos.

En C# existen 3 tipos de modificadores de acceso :

- `private`
- `public`
- `protected`

Un beneficio fundamental en la orientación a objetos es el ocultamiento de datos –atributos-, que consiste en que sólo los métodos definidos en la clase, son los únicos que pueden acceder a los atributos de los objetos pertenecientes a una clase.
La implementación del concepto de ocultamiento de datos es efectuada haciendo a los atributos privados y a los métodos, públicos. En otras palabras, aplicar el modificador de acceso `private` a cada atributo y el modificador de acceso `public` a cada método.
El modificador de acceso `public` permite desde cualquier parte del programa, el uso del método o atributo de la clase al que se le aplique. Por el contrario, el modificador de acceso `private` no permite el uso del atributo o método de la clase, solamente es visible a los métodos de la clase.
Digamos que la clase `Alumno` tiene 3 atributos : `noControl`, `nombre` y `calif`. Además tiene 2 métodos : `AsignaNoControl()` y `RetNoControl()`, donde el primero de ellos recibe un parámetro necesario para asignarlo al atributo `_noControl`. La clase `Alumno` la escribiríamos de la siguiente manera :

```
class Alumno
{
  private string _noControl;
  private string _nombre;
  private int _calif;

  public void AsignaNoControl(string valor)
  {
    _noControl = valor;
  }

  public string RetNoControl()
  {
    return _noControl;
  }

}
```

Atributos

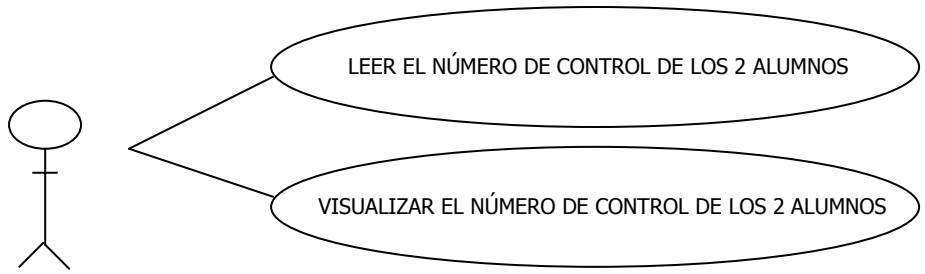
Métodos

No hemos escrito los métodos para asignar y retornar los atributos `_nombre` y `_calif` sólo por no hacer mas abultada la clase `Alumno`, después los agregaremos.

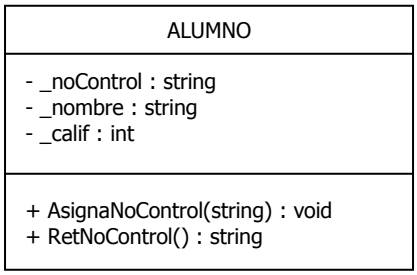
Notemos que los 3 atributos han sido declarados antecediéndolos de la palabra reservada `private`, que indica que el atributo no puede ser accedido desde cualquier parte de la aplicación. El atributo así declarado, sólo podrá ser accedido por los métodos de la clase `Alumno`, en este caso por lo tanto, el atributo `_noControl` sólo puede ser accesado por los métodos `AsignaNoControl()` y `RetNoControl()`. Estos métodos han sido declarados con el modificador de acceso `public`, de manera que ellos si pueden ser visibles o accedidos, desde o por otras partes del programa. Lo que significa que si queremos modificar el atributo `_noControl` de un alumno determinado, deberíamos de llamar en un mensaje al alumno con el método `AsignaNoControl()` aplicado a dicho alumno.

Hagamos una aplicación Windows para fortalecer la explicación de los modificadores de acceso : `private` y `public`. Lo que esperamos que efectúe dicha aplicación –programa-, es la lectura y visualización del número de control para 2 alumnos. Ambos alumnos son objetos que pertenecen a la clase `Alumno`.

Podemos decir fácilmente que el diagrama de casos de uso que representa las tareas a efectuar en este programa es :



El Diagrama de clases es simple ya que sólo tenemos una clase involucrada en la solución del problema : la clase `Alumno`.



En el diagrama de clases recordemos que el **–** indica que el atributo o método es privado. El símbolo **+** indica que el atributo o método es publico. Observemos que el método `AsignaNoControl()` tiene un parámetro `string`.

Ahora sí vayamos al ambiente de desarrollo Visual Studio C#, creamos una nueva aplicación y en la ventana con la forma `Form1.cs[Design]` agregamos 6 componentes `Label`, 2 componentes `TextBox` y 4 componentes `Button`. Modificamos las propiedades según la tabla siguiente :

objeto	propiedad	valor
label1	Text	ALUMNO 1
label2	Text	ALUMNO 2
label3	Text	No. control
label4	Text	No. control
label5	Text	label5
label6	Text	label6
textBox1	Text	
textBox2	Text	
button1	Text	LEER ALUMNO 1
button2	Text	LEER ALUMNO 2
button3	Text	VISUA ALUMNO 1
button4	Text	VISUA ALUMNO 2

La interfase gráfica Form1.cs[design] deberá quedar según se muestra en la figura #5.1.1. Notemos que las etiquetas label5 y label6 tendrán la utilidad de visualizar los números de control leídos.

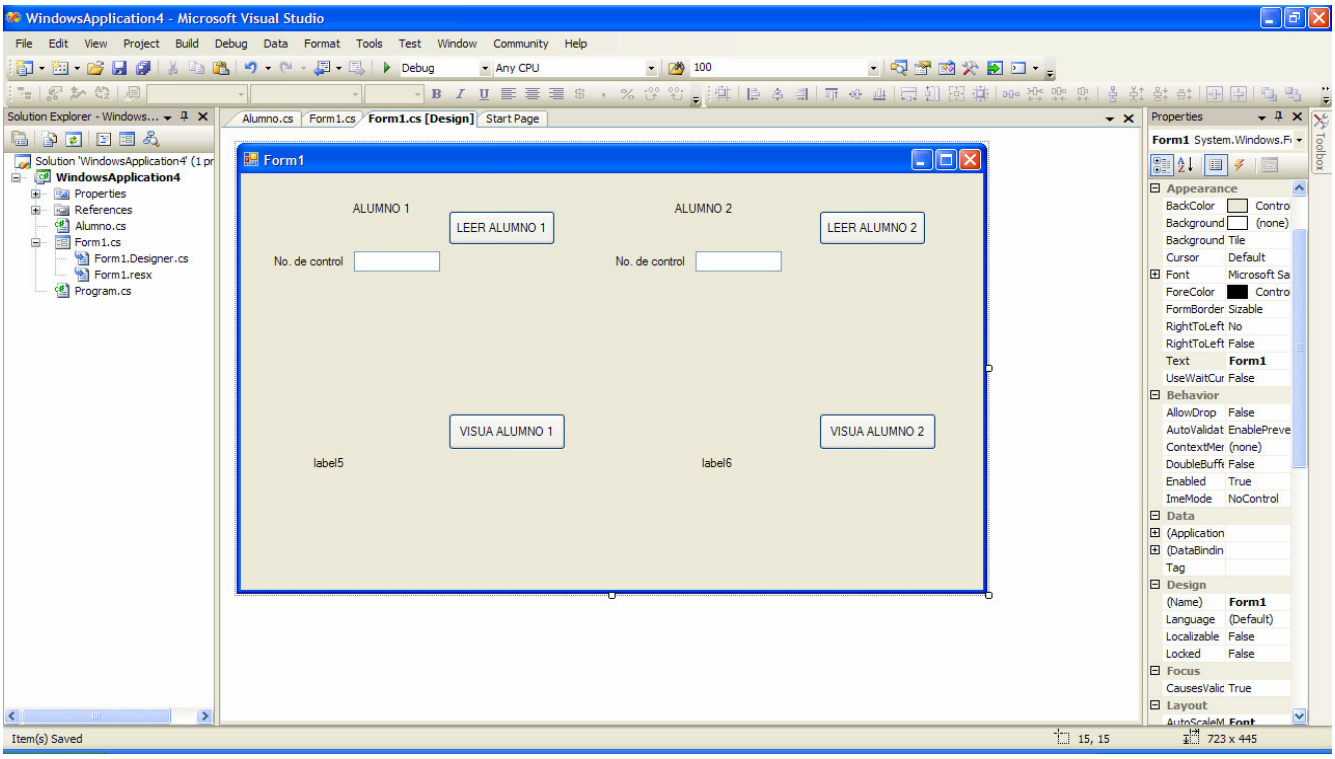


Fig. No. 5.1.1 Interfase gráfica de usuario Form1.cs[Design].

Agreguemos la clase `Alumno` usando la opción del menú *Project | Add Class*, tecleando en el diálogo el nombre de la clase Alumno.cs. Después añadimos el código donde se definen los atributos y métodos según se ha indicado al inicio de esta sección. La figura #5.1.2 muestra a la clase Alumno.cs.

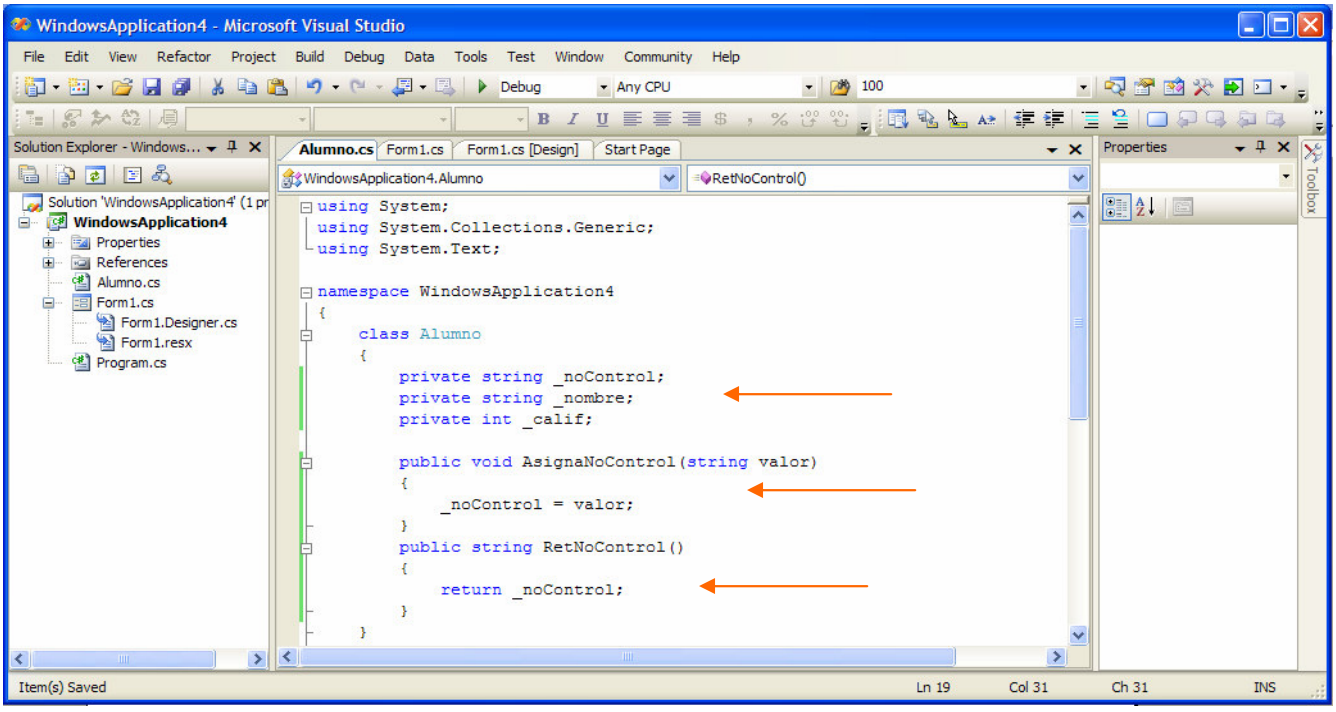


Fig. No. 5.1.2 Clase `Alumno`.

Las clases por si mismas no sirven para efectos de programar en cambio, los objetos son los que reciben o envían mensajes que son los ingredientes principales en la programación orientada a objetos. Es simple abstraer que debemos trabajar con 2 objetos de entrada pertenecientes a la clase `Alumno`, ya que ellos son los que recibirán el mensaje de asignación y de visualización de su atributo `_noControl`. Llamemos a los objetos usando los identificadores `oAlu1` y `oAlu2`.

¿Dónde deben ser definidos dentro de nuestra aplicación?. Recordemos que el código deberá insertarse dentro del archivo Form1.cs. Los objetos y datos sean variables o sean constantes de un tipo predefinido, son declarados como atributos de la clase `Form1` definida en dicho archivo Form1.cs. Agreguemos loa definición de los objetos `oAlu1` y `oAlu2` según se indica en la figura 5.1.3.

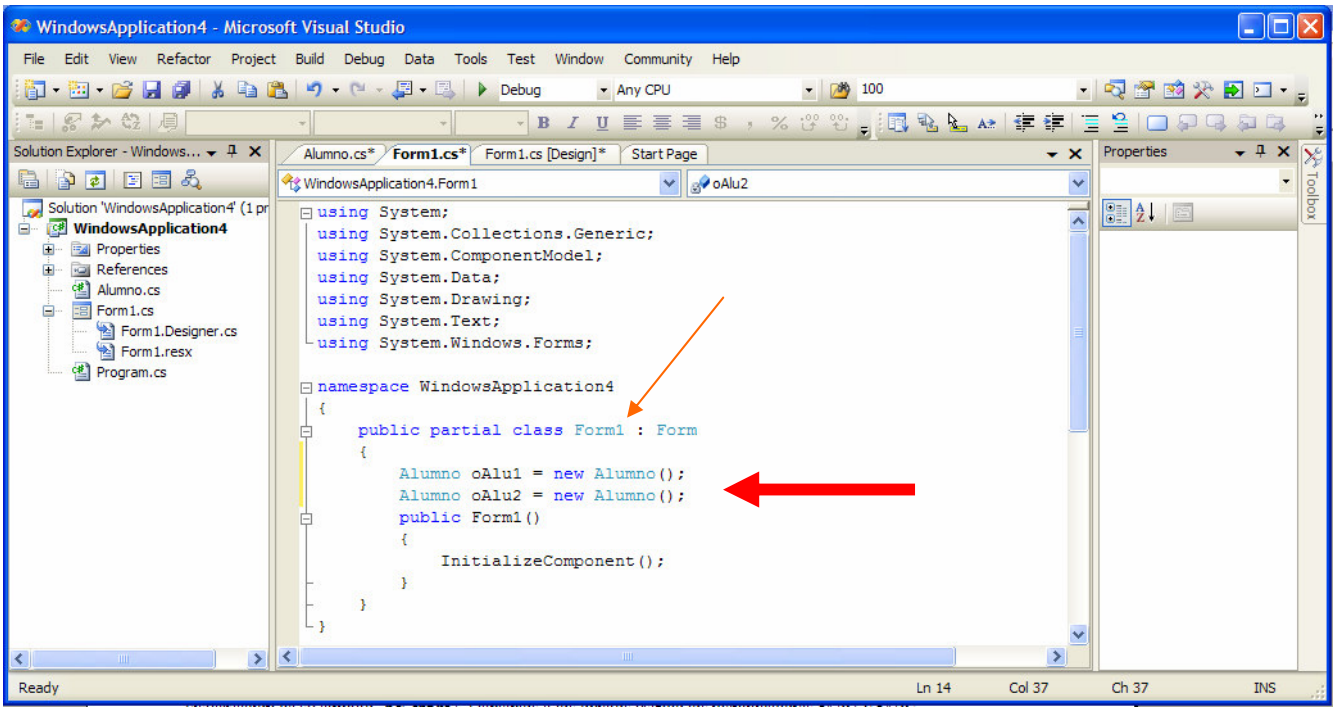


Fig. No. 5.1.3 Definición de los objetos `oAlu1` y `oAlu2` en la clase `Form1`.

Ejecutemos la aplicación sólo con el fin de conocer si no hemos cometido algún error de dedo, figura #5.1.4.

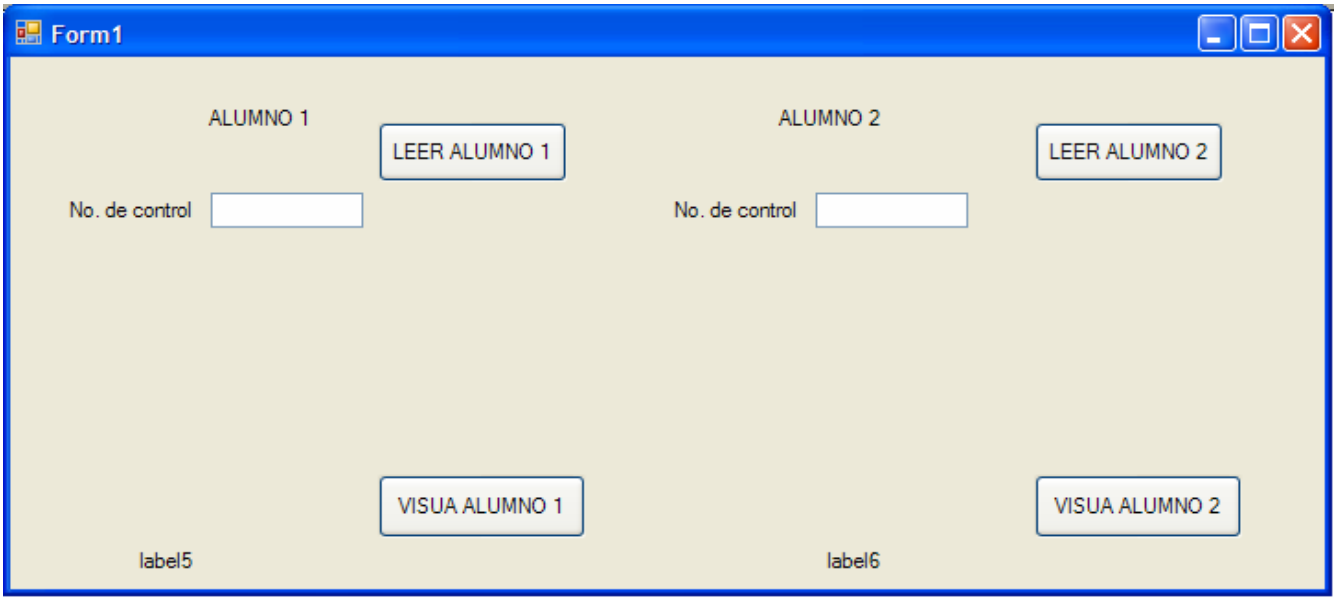


Fig. No. 5.1.4 Nuestra aplicación en ejecución.

Seguimos con la implementación del caso de uso : LEER EL NÚMERO DE CONTROL DE LOS 2 ALUMNOS. La efectuaremos insertando código en los botones `button1` y `button2` con leyendas LEER ALUMNO 1 y LEER ALUMNO 2 respectivamente. Lo que deseamos es que cuando hagamos click en dichos botones, tomemos el número de control tecleado por el usuario en el `textBox1` o en el `textBox2` según corresponda, y por medio del mensaje `AsignaNoControl()` al objeto sea el `oAlu1`, sea el `oAlu2`, accedamos al atributo `_noControl` y le asignemos el valor de lo tecleado en el `TextBox` correspondiente. Recordemos que para acceder a lo tecleado en un `TextBox` deberemos acceder a la propiedad `Text` del componente.

```
oAlu1.AsignaNoControl(textBox1.Text);     // mensaje al objeto oAlu1

oAlu2.AsignaNoControl(textBox2.Text);     // mensaje al objeto oAlu2
```

Notemos que cada mensaje termina con el caracter (;). En C# todas las sentencias -un mensaje es una sentencia- terminan con (;). Agreguemos estos mensajes en el evento Click de cada botón `button1` y `button2`, figura #5.1.5. Observemos que los métodos Click de cada botón residen dentro de la clase `Form1`.

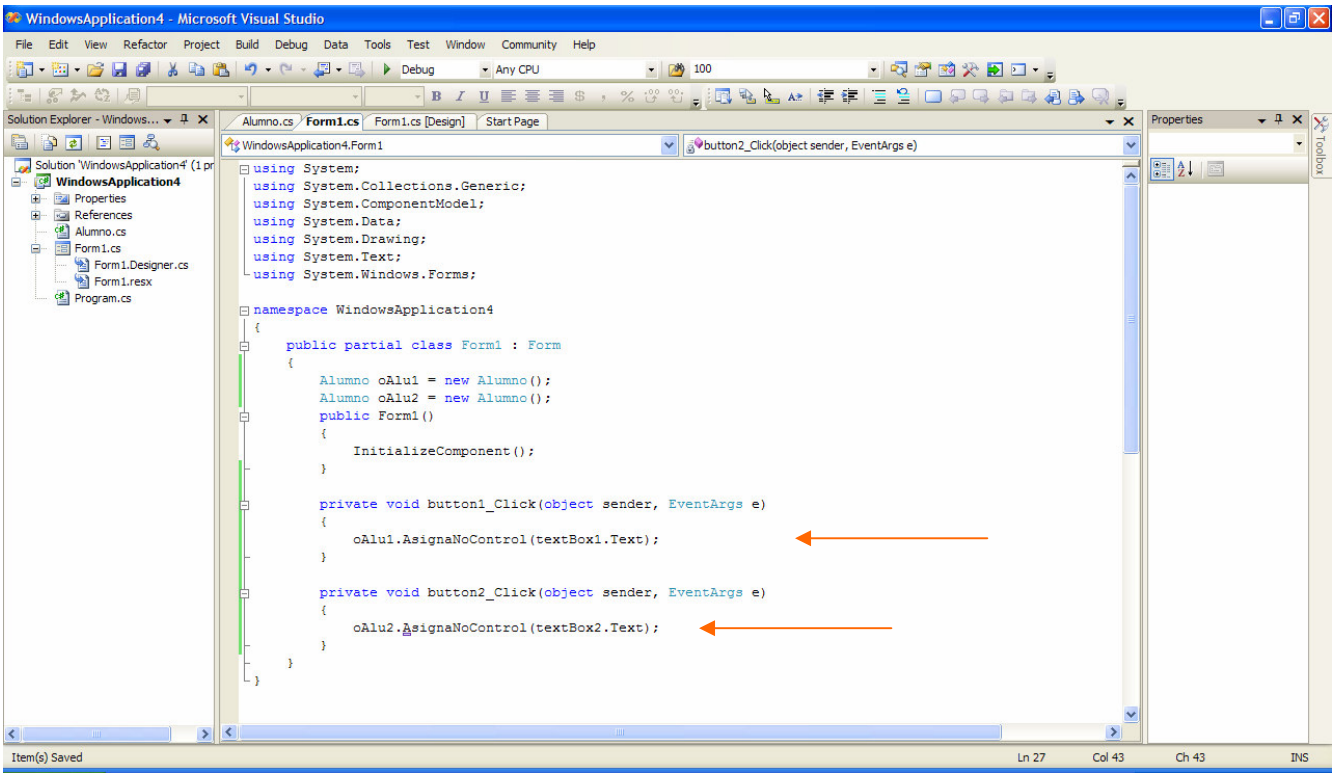


Fig. No. 5.1.5 Inclusión de los mensajes para leer el número de control de un alumno.

Si ejecutamos la aplicación en este momento, si leemos lo tecleado en los componentes `TextBox` además de asignarlo a los atributos `_noControl` del alumno, pero no hay manera de visualizar debido a que no hemos añadido los mensajes para visualizar los números de control de los alumnos.

El segundo caso de uso : VISUALIZAR EL NÚMERO DE CONTROL DE LOS 2 ALUMNOS es el que deberemos codificar para poder realizar la visualización del número de control de los alumnos. ¿Qué es lo que debemos hacer?, pues acceder al atributo `_noControl` del alumno mediante el uso de un método, en este caso `RetNoControl()` que como vemos retorna con la sentencia `return` al atributo `_noControl`. Este valor lo debemos asignar a la propiedad `Text` del componente Label correspondiente sea `label5` para el `oAlu1`, sea `label6` para el `oAlu2`. Dicha asignación la realizamos mediante un mensaje al objeto correspondiente que involucre al método `RetNoControl()`.

```
label5.Text = oAlu1.RetNoControl();     // visualización del número de control del alumno oAlu1

label6.Text = oAlu2.RetNoControl();     // visualización del número de control del alumno oAlu2
```

Estos mensajes debemos escribirlos en el evento Click de los botones `button3` y `button4` según es mostrado en la figura 5.1.6.

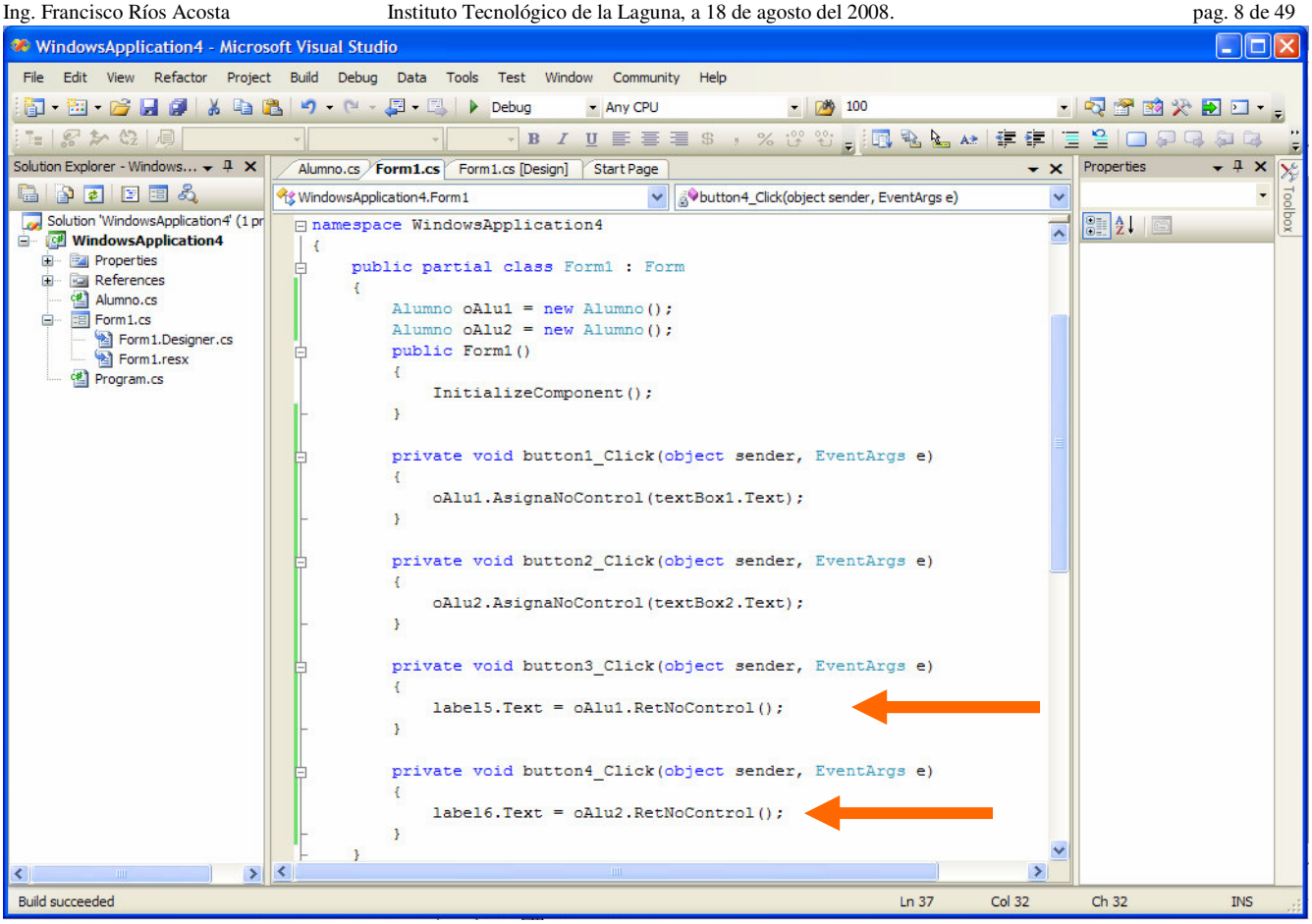


Fig. No. 5.1.6 Inclusión de los mensajes para visualizar el número de control de un alumno.

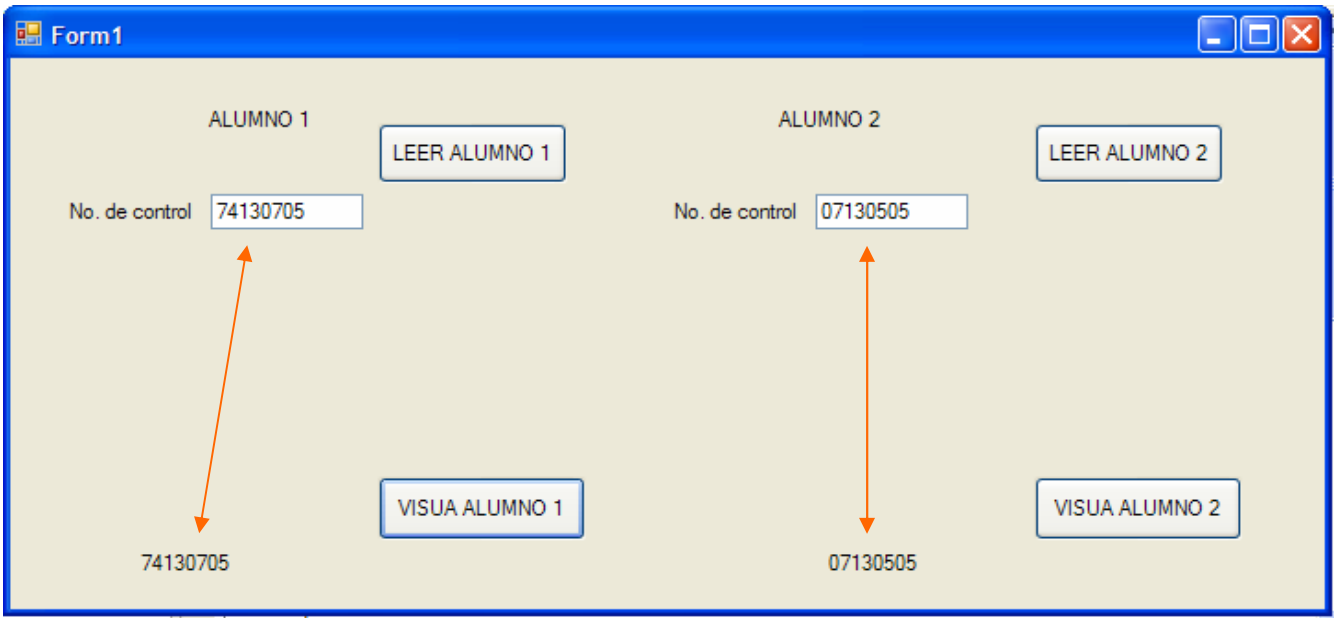


Fig. No. 5.1.7 Ejecución de la aplicación vista después de hacer click en los botones de lectura y de visualización.

El ejercicio hecho de la forma en que lo hemos presentado, representa el ocultamiento típico de la programación orientada a objetos, donde los atributos son definidos privados y la única forma de accederlos es mediante un método de la clase llamado dentro de un mensaje al objeto.

Para comprobar que los atributos de los objetos `oAlu1` y `oAlu2` pertenecientes a la clase `Alumno`, no son visibles desde cualquier parte del programa sino sólo son visibles a los métodos de la clase, hagamos el intento de acceder al atributo `_noControl` con un mensaje involucrando al alumno `oAlu1` :

```
oAlu1._noControl = textBox1.Text;
```

En el mensaje anterior tratamos de asignar al atributo `_noControl` del alumno `oAlu1` el valor ingresado en el componente `textBox1`. Sustituyamos el mensaje `oAlu1.AsignaNoControl()` que se encuentra en el evento Click del botón `button1`, por el mensaje donde accedemos al atributo `_noControl` de manera directa. La figura #5.1.8 muestra al mensaje previamente escrito como un comentario –no es tomado en cuenta por la compilación-, y el nuevo mensaje que lo sustituye.

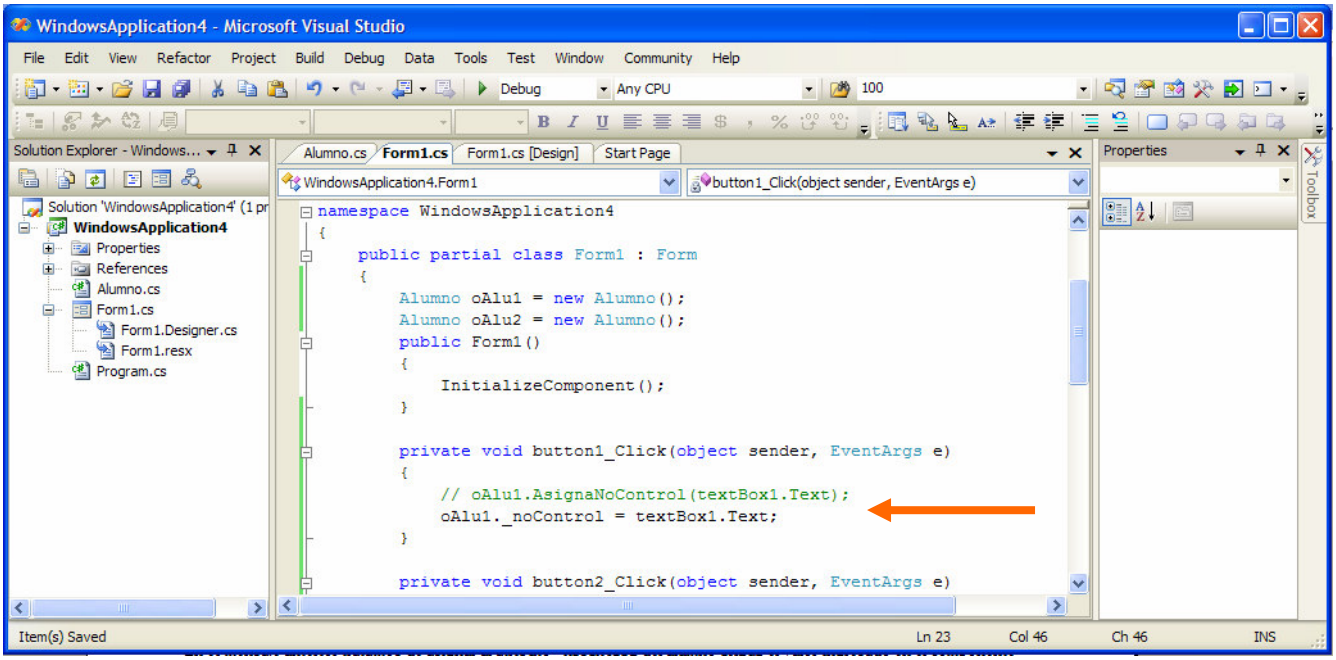


Fig. No. 5.1.8 Acceso directo al atributo `_noControl` del objeto `oAlu1`.

Si compilamos y tratamos de ejecutar la aplicación obtenemos el error de el atributo `_noControl` no es accesible debido a su nivel de protección especificado por el modificador de acceso `private`, figura #5.1.9

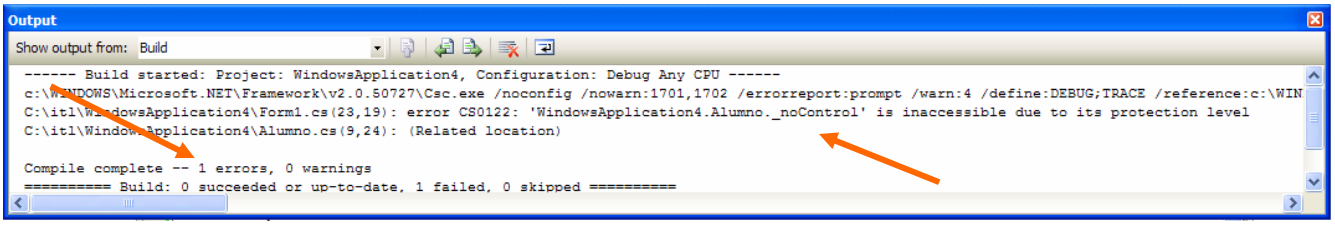


Fig. No. 5.1.9 ERROR al tratar de accesar al atributo `_noControl` del objeto `oAlu1`.

Ejercicio propuesto :

- Cambia el modificador de acceso del método AsignaNoControl() o del método RetNoControl() de public a private de manera que observes lo que sucede.

5.2 Encapsulamiento de la clase.

Tal y como se ha venido explicando en las secciones anteriores, la clase encapsula tanto a los atributos como a los métodos en una sola entidad. El ocultamiento de datos lo realizamos haciendo a los atributos privados y a los métodos públicos. Sigamos con la aplicación de la sección anterior para complementarla agregando las definiciones de los atributos `_nombre` y `_calif`, además de los métodos `AsignaNombre()`, `AsignaCalif()`, `RetNombre()` y `RetCalif()`, necesarios para la lectura y

visualización. A continuación se muestra el código para la clase `Alumno` con las adiciones de atributos y métodos mencionados.

```
class Alumno
{
        private string _noControl;
        private string _nombre;
        private int _calif;

        public void AsignaNoControl(string valor)
        {
            _noControl = valor;
        }
        public string RetNoControl()
        {
            return _noControl;
        }
        public void AsignaNombre(string valor)
        {
            _nombre = valor;
        }
        public string RetNombre()
        {
            return _nombre;
        }
        public void AsignaCalif(int valor)
        {
            _calif = valor;
        }
        public int RetCalif()
        {
            return _calif;
        }
}
```

Notemos que el método `AsignaCalif()` recibe un parámetro de tipo **int**, y que el método `RetCalif()` retorna un tipo **int**. Lo anterior es debido a que el atributo `_calif` de los objetos de la clase `Alumno` es de tipo entero **int**.
Debemos modificar la interfase gráfica `Form1.cs[Design]` de manera que permita el ingreso de los valores para el nombre y para la calificación de los alumnos, además de los componentes Label para realizar la visualización de los atributos `_nombre` y `_calif` de los objetos `oAlu1` y `oAlu2`. Entonces tenemos que añadir otros 8 componentes Label y 4 componentes TextBox.

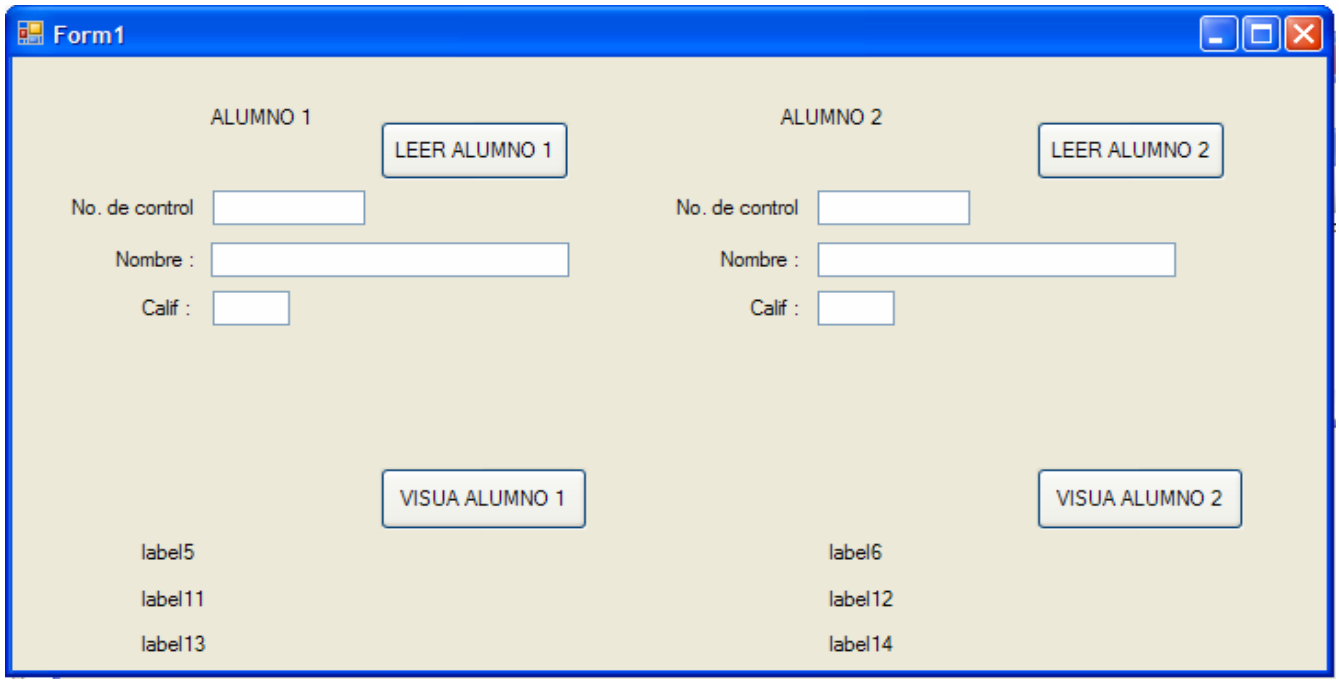


Fig. No. 5.2.1 Interfase gráfica para leer y visualizar los 3 atributos de los 2 alumnos `oAlu1`, `oAlu2`.

Modifica los componentes añadidos según la figura #5.2.1 y de acuerdo a la tabla mostrada a continuación.

objeto	propiedad	valor
label7	Text	Nombre :
label8	Text	Nombre :
label9	Text	Calif :
label10	Text	Calif :
label11	Text	label11
label12	Text	label12
label13	Text	label13
label14	Text	label14
textBox3	Text	
textBox4	Text	
textBox5	Text	
textBox6	Text	

El siguiente paso es agregar los mensajes a los eventos Click de los botones usados para la lectura y para la visualización de los objetos `oAlu1` y `oAlu2`. El código en el archivo Form1.cs que contiene la clase Form1 donde se incluye la inserción de los mensajes para lectura y visualización se muestra a continuación.

```
namespace WindowsApplication4
{
    public partial class Form1 : Form
    {
        Alumno oAlu1 = new Alumno();
        Alumno oAlu2 = new Alumno();
        public Form1()
        {
            InitializeComponent();
        }

        private void button1_Click(object sender, EventArgs e)
        {
             oAlu1.AsignaNoControl(textBox1.Text);
             oAlu1.AsignaNombre(textBox3.Text);
             oAlu1.AsignaCalif(Convert.ToInt32(textBox5.Text));
        }

        private void button2_Click(object sender, EventArgs e)
        {
            oAlu2.AsignaNoControl(textBox2.Text);
            oAlu2.AsignaNombre(textBox4.Text);
            oAlu2.AsignaCalif(Convert.ToInt32(textBox6.Text));
        }

        private void button3_Click(object sender, EventArgs e)
        {
            label5.Text = oAlu1.RetNoControl();
            label11.Text = oAlu1.RetNombre();
            label13.Text = Convert.ToString(oAlu1.RetCalif());
        }

        private void button4_Click(object sender, EventArgs e)
        {
            label6.Text = oAlu2.RetNoControl();
            label12.Text = oAlu2.RetNombre();
            label14.Text = Convert.ToString(oAlu2.RetCalif());
        }
    }
}
```

Notemos que cuando llamamos a los mensajes :

```
oAlu1.AsignaCalif(Convert.ToInt32(textBox5.Text));

oAlu2.AsignaCalif(Convert.ToInt32(textBox6.Text));
```

Usamos una clase predefinida –incluida en el C#- llamada `Convert`. Esta clase contiene métodos que podemos llamar para efectuar conversiones explícitas de tipo. En este caso queremos convertir la propiedad `Text` de los componentes `textBox5` y

`textBox6` a tipo entero **int**. Lo anterior debe hacerse debido a que el parámetro que acepta el método `AsignaCalif()` es de tipo **int**.

Algo similar efectuamos en los botones de visualizar, cuando convertimos el entero que retorna el método `RetCalif()` a un tipo **string**. Debemos de hacerlo, debido a que la propiedad `Text` de un componente `Label` es de tipo **string**.

La aplicación en ejecución para determinados datos de entrada se muestra en la figura #5.2.2.

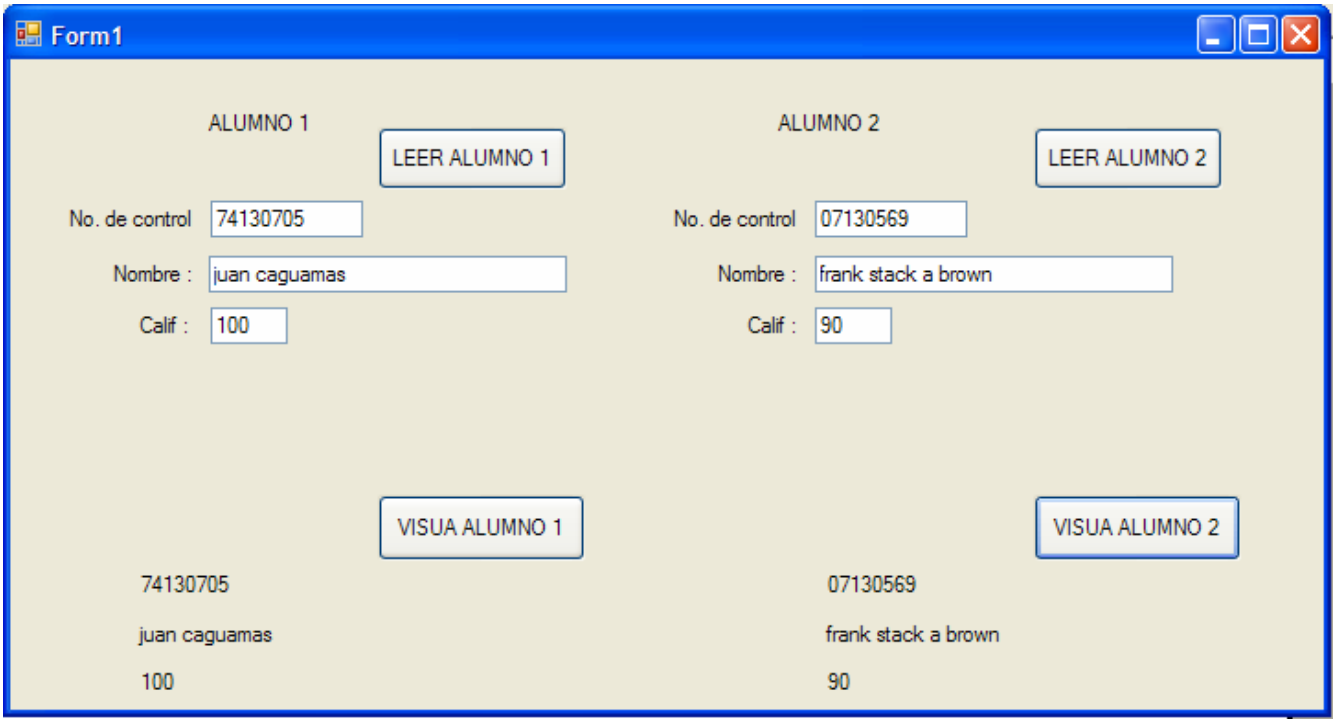
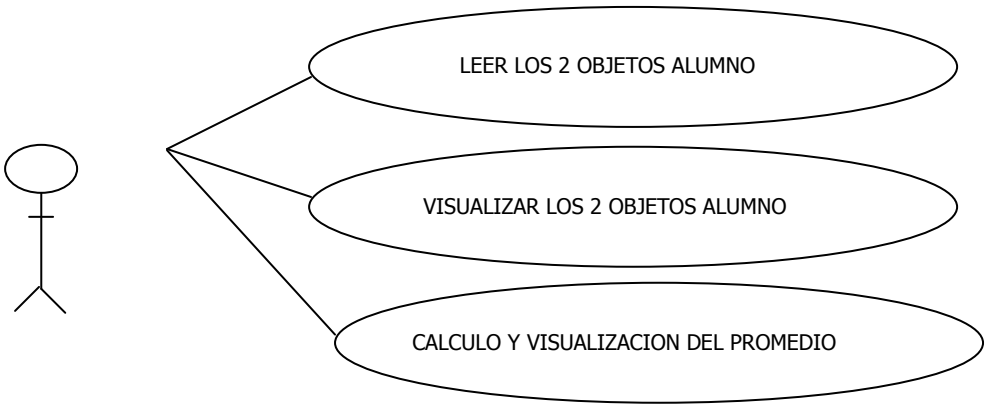


Fig. No. 5.2.2 Aplicación Windows C# con la lectura y visualización de los 3 atributos de los alumnos `oAlu1`, `oAlu2`.

Cálculo del promedio de las calificaciones de los alumnos.- Para completar la exposición de esta sección, calcularemos el promedio de las calificaciones de los 2 alumnos. Dado que la clase encapsula tanto a los atributos como a los métodos, la calificación de cada alumno para sumarlas y luego dividir la suma entre 2 con el fin de obtener el promedio, deberemos accederlas por medio de un método.

El método para acceder el valor de la calificación de un alumno ya lo hemos definido en la clase Alumno : `RetCalif()`. La llamada a este método nos retorna el valor de la calificación del alumno al cual se le envía el mensaje. Otra cuestión que tenemos que tomar en cuenta, es el hecho de agregar el botón a la aplicación que efectúe el cálculo y la visualización del promedio. La visualización del promedio tendremos que realizarla sobre un componente Label que también debemos de añadir a la interfase gráfica. El diagrama de casos de uso se modifica según se indica :



La interfase gráfica con los componentes Label y Button nuevos incluidos se muestra en la figura #5.2.3.

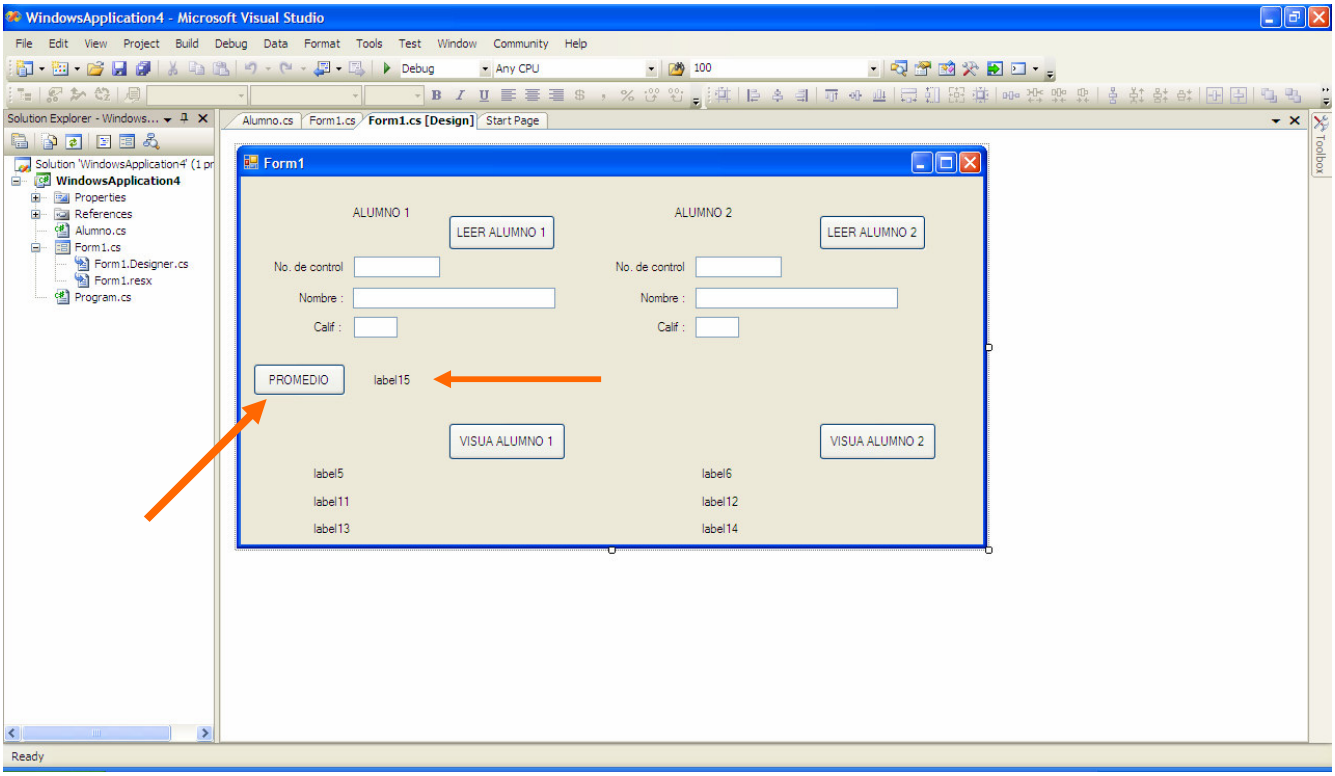


Fig. No. 5.2.3 Cambios en la interfase gráfica para cálculo del promedio.

Los mensajes requeridos para acceder a las 2 calificaciones son :

```
oAlu1.RetCalif()

oAlu2.RetCalif()
```

Cada uno de los mensajes retorna un entero que precisamente es la calificación del objeto que recibe el mensaje. Debemos sumar estos 2 mensajes y su resultado dividirlo entre 2 para lograr el cálculo del promedio.

```
float promedio = (oAlu1.RetCalif() + oAlu2.RetCalif()) / 2.0F;
```

Notemos que debemos agrupar la suma entre paréntesis de manera que primero se efectúe la suma y luego la división. ¿Qué pasa si no usamos la constante literal 2.0F y en su lugar usamos la constante literal entera 2?.

Incluyamos el cálculo del promedio en el evento Click del botón button5 según mostramos a continuación :

```
private void button5_Click(object sender, EventArgs e)
{
    float prom = (oAlu1.RetCalif() + oAlu2.RetCalif()) / 2.0F;
    label15.Text = "El promedio es igual a " + Convert.ToString(prom);
}
```

Notemos que hemos tenido que aplicar un molde (cast) sobre el valor del promedio representado por la variable `prom`, de manera que podamos concatenar su valor con la constante literal cadena "`El promedio es igual a `".

Otra forma de visualizar el promedio es aplicar el método `ToString()` a la variable `prom` :

```
label15.Text = "El promedio es igual a " + prom.ToString();
```

Lo anterior es posible hacerlo debido a que C# considera a los datos **int**, **float**, **double** como objetos que aceptan ciertos métodos entre los que se encuentra el método `ToString()`.

La figura #5.2.4 muestra la ejecución de la aplicación después de haber hecho la lectura, la visualización de los datos de los objetos `oAlu1` y `oAlu2`, además del cálculo y visualización del promedio de las calificaciones de los 2 objetos.

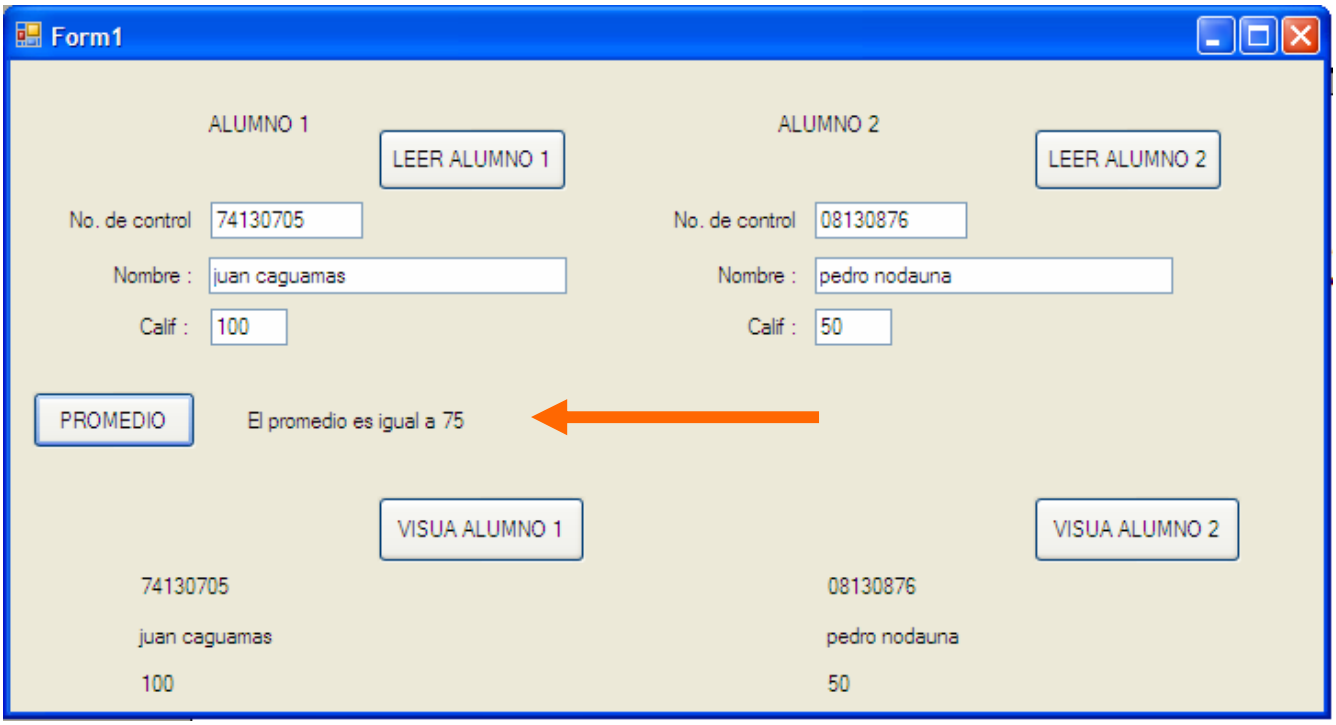


Fig. No. 5.2.4 Ejecución de la aplicación con la adición de la tarea del cálculo del promedio.

5.3 El método como elemento de la comunicación.

En la sección anterior vimos como se efectúa el encapsulamiento de atributos y métodos, combinándolo con el concepto de ocultamiento, de manera que sólo se puede acceder externamente a los atributos de un objeto mediante el uso de métodos que involucren los correspondientes mensajes a los objetos dueños de esos atributos.

Por consiguiente, hemos visto como el método es empleado en los mensajes precisamente para comunicarnos con los objetos. La programación orientada a objetos como lo explicamos en la unidad I secciones 1.4.3 y 1.4.4 consiste de la escritura de sentencias las cuales en realidad son mensajes para o entre objetos. Un objeto implícito en nuestros programas es precisamente la aplicación. Dentro de nuestra aplicación representada por la clase `Form1`, nosotros como programadores insertamos mensajes a los objetos que previamente declaramos y definimos.

Por ejemplo podemos citar los mensajes en la aplicación de la lectura y visualización de los datos de 2 alumnos, además del cálculo del promedio de sus calificaciones, a los objetos `oAlu1` y `oAlu2`. Los mensajes para la asignación de valores a los atributos `noControl`, `nombre` y `calif` del objeto `oAlu1` son :

```
oAlu1.AsignaNoControl(textBox1.Text);
oAlu1.AsignaNombre(textBox3.Text);
oAlu1.AsignaCalif(Convert.ToInt32(textBox5.Text));
```

Y los mensajes de asignación para los atributos del objeto `oAlu2` :

```
oAlu2.AsignaNoControl(textBox2.Text);
oAlu2.AsignaNombre(textBox4.Text);
oAlu2.AsignaCalif(Convert.ToInt32(textBox6.Text));
```

En este caso nos hemos comunicado con los 2 objetos para realizar la lectura y asignación de sus atributos los cuales sólo podemos acceder utilizando los métodos de la clase. Este acceso lo logramos mediante el uso de mensajes que utilicen los métodos para asignación de valores a los atributos correspondientes.

Para la visualización de los atributos, igualmente utilizamos métodos que para este caso, retornan a los correspondientes atributos de manera que puedan ser asignados a los componentes Label en su propiedad Text, los cuales son los encargados de realizar la visualización.

Para el objeto `oAlu1` :

```
label5.Text = oAlu1.RetNoControl();
label11.Text = oAlu1.RetNombre();
label13.Text = Convert.ToString(oAlu1.RetCalif());
```

Para el objeto `oAlu2` :

```
label6.Text = oAlu2.RetNoControl();
label12.Text = oAlu2.RetNombre();
label14.Text = Convert.ToString(oAlu2.RetCalif());
```

Resumiendo entonces, el método es el elemento que nos permite comunicarnos con los objetos mediante la utilización de mensajes, los cuales involucran tanto al objeto como al método.

5.3.1 Sintaxis.

La sintaxis para la construcción de un método varia según si es :

- Llamada al método.
- Definición del método.

Llamada. La llamada al método generalmente es efectuada utilizando un mensaje. En el mensaje se concatena al identificador del objeto seguido del caracter punto (.) , seguido del nombre del método. El método puede tener o no, parámetros. Por ejemplo, el mensaje `oAlu1.RetNoControl()` hace una llamada al método `RetNoControl()` que retorna el número de control del alumno `oAlu1`.

```
oAlu1.RetNoControl()
```

objeto método

En el mensaje es donde realmente se hace la llamada al método `RetNoControl()`. Este método no recibe parámetros de manera que los paréntesis () deben agregarse aunque no se pasen parámetros al método.

Los métodos pueden recibir parámetros para utilizarlos dentro de su cuerpo –sentencias-, en asignaciones, cálculos, o cualquier acción necesaria. El valor de estos parámetros se envía al momento de su llamada, precisamente en el mensaje. por ejemplo, hemos utilizado los métodos `AsignaNoControl(), AsignaNombre y AsignaCalif()` para leer o asignar el valor que se pasa de parámtero, al atributo correspondiente.

```
oAlu1.AsignaNoControl(textBox1.Text);
```

Parámetro de tipo string que será utilizado para asignarlo al atributo `noControl` del alumno **oAlu1**.

```
oAlu1.AsignaNombre(textBox3.Text);
```

Parámetro de tipo string que será utilizado para asignarlo al atributo `nombre` del alumno **oAlu1**.

```
oAlu1.AsignaCalif(Convert.ToInt32(textBox5.Text));
```

Parámetro de tipo string que se convierte a int por medio del uso de la clase `Convert` y su método `ToInt32`, que será utilizado para asignarlo al atributo `noControl` del alumno **oAlu1**.

Desde luego que podemos utilizar un sólo método que reciba a los 3 valores por medio del uso de 3 parámetros. Digamos que llamamos al método `Leer()`, entonces la llamada será el mensaje al objeto `oAlu1` o bien al objeto `oAlu2` según el botón que presionemos.

Para el alumno `oAlu1` tendríamos el mensaje :

```
oAlu1.Leer(textBox1.Text, textBox3.Text, Convert.ToInt32(textBox5.Text));
```

Para el alumno `oAlu2` :

3 parámetros : **string**, **string**, **int**

```
oAlu2.Leer(textBox2.Text, textBox4.Text, Convert.ToInt32(textBox6.Text));
```

Definición. La definición de un método consiste de indicar el código –sentencias- que se van a ejecutar cuando el método sea llamado en el mensaje. El código en el método utilizará los valores de los parámetros que recibe para efectuar las acciones sobre atributos y/o cálculos con ellos.

En el ejercicio de la sección anterior, hemos definido varios métodos sin y con parámetros. Veamos la sintaxis para definir un método en C# :

```
modificador_de_acceso  tipo_de_retorno  nombre_del_método ( declaración_de_parámetros )
{
   // sentencias del método
}
```

modificador_de_acceso. Puede ser `public`, `private` o `protected`. Recordemos que generalmente los métodos de una clase son del tipo `public`, para que puedan ser accedidos desde cualquier parte del programa fuera de la clase. Algunos métodos son declarados `private` cuando sólo son usados por los métodos de la clase donde ellos son definidos. Cuando un método es declarado de tipo `private` no puede ser accedido por ninguna sentencia fuera de la clase. El modificador de acceso `protected` es utilizado cuando se trabaja con herencia, cuestión que en nuestro curso no lo veremos pues está fuera de su alcance.

tipo_de_retorno. Se refiere a que un método puede retornar un valor mediante el uso de la sentencia `return`. El valor que retorna dicha sentencia tiene un tipo de dato, y es precisamente este tipo de dato el que debemos especificar como `tipo_de_retorno` en el encabezado de la definición del método. Cuando el método no retorna un valor el tipo_de_retorno es igual al tipo `void`. `void` es un tipo de dato que no tiene ningún valor –no tiene rango de valores, no tiene ni un solo valor-. Estos métodos que no retornan nada, cumplen con efectuar una determinada tarea y cuando acaban retornan el control de ejecución al programa principal.

nombre_del_método. Es el identificador del método, sigue las mismas reglas que las establecidas para construir identificadores de variables y constantes en C#.

declaración_de_parámetros. Es una lista donde se declara el tipo y el identificador de cada uno de los parámetros. Cuando es mas de un parámtero lo que recibe el método, entonces se deben separar las declaraciones de cada parámetro utilizando el caracter (,).

5.3.2 Concepto de parámetro.

Los parámetros son utilizados por un programador para pasarle información a un método, de manera que éste pueda utilizar los valores de los parámetros para efectuar asignaciones a atributos, hacer cálculos combinando atributos con parámetros, entre otras.

Por ejemplo, el método `AsignaNoControl()` recibe un valor –la propiedad `Text` del `textBox1`- para asignarlo al atributo `_noControl`. El mensaje lo mostramos enseguida :

```
oAlu1.AsignaNoControl(textBox1.Text);
```

La llamada del método en el mensaje recibe el valor que se debe asignar al atributo `_noControl`. La definición del método recibe al valor por medio de un parámetro según lo vemos en el código del método :

```
public void AsignaNoControl(string valor)
{
     _noControl = valor;
}
```

El parámetro **string** `valor` declarado entre los paréntesis, recibe lo enviado por la propiedad `Text` del `textBox1`. El método toma el parámetro con identificador `valor` y lo asigna al atributo `_noControl` del objeto que hizo o recibió el mensaje donde se involucra al método. El lenguaje en nuestro caso el C#, "sabe" quién fue el objeto que hizo la llamada –el mensaje- al método.

Pasaje por valor.- Cuando un parámetro se utiliza sólo para asignarlo a un atributo, o sólo interviene como operando para efectuar algún cálculo o alguna decisión, se dice que el pasaje del parámetro es POR VALOR. Un parámetro es pasado por valor desde un mensaje –llamada al método- a la definición del método, cuando no requerimos cambiarlo –modificarlo-.

Veamos de nuevo el método `Leer()` perteneciente a la clase `Alumno`, y que hemos mencionado anteriormente que sustituye a los 3 mensajes :

```
oAlu1.AsignaNoControl(textBox1.Text);

oAlu1.AsignaNombre(textBox3.Text);

oAlu1.AsignaCalif(Convert.ToInt32(textBox5.Text));
```

```
oAlu1.Leer(textBox1.Text, textBox3.Text, Convert.ToInt32(textBox5.Text));
```

Observemos que el método `Leer()` recibe 3 parámetros que corresponden al valor que el usuario ingresa en los componentes TextBox para que sean asignados a los atributos del alumno `oAlu1` : `_noControl`, `_nombre`, `_calif`. El pasaje de estos 3 parámetros es por valor a causa de que sólo se utilizan para asignarlos a los respectivos atributos dentro del método `Leer()`. El método `Leer()` realiza la misma función –transformación- sobre los atributos del objeto `oAlu1` que los 3 mensajes `AsignaNoControl()`, `AsignaNombre()` y `AsignaCalif()`. El primero recibe 3 parámetros y los 3 últimos reciben sólo un parámetro.

Vayamos a la aplicación Windows C# que escribimos en la sección 5.2 y hagamos la sustitución en los eventos Click de los botones `button1` y `button2` que son los encargados de realizar la tarea de leer los datos para el alumno correspondiente.

```
private void button1_Click(object sender, EventArgs e)
{
   /*  oAlu1.AsignaNoControl(textBox1.Text);
       oAlu1.AsignaNombre(textBox3.Text);
       oAlu1.AsignaCalif(Convert.ToInt32(textBox5.Text));  */
       oAlu1.Leer(textBox1.Text, textBox3.Text, Convert.ToInt32(textBox5.Text));
}

private void button2_Click(object sender, EventArgs e)
{
   /* oAlu2.AsignaNoControl(textBox2.Text);
      oAlu2.AsignaNombre(textBox4.Text);
      oAlu2.AsignaCalif(Convert.ToInt32(textBox6.Text)); */
      oAlu2.Leer(textBox2.Text, textBox4.Text, Convert.ToInt32(textBox6.Text));
}
```

Los 3 mensajes que inicialmente teníamos los hemos encerrado entre los caracteres /* y */ , de manera que el compilador los tomará como comentarios y no los reconoce como parte del programa ejecutable. Entonces el método `Leer()` recibirá estos 3 valores y los asignará a los 3 atributos del objeto según corresponda.

La definición del método `Leer()` tenemos que añadirla a la clase `Alumno` según lo mostramos en la figura #5.3.1. Ejecutemos la aplicación Windows C# para observar que todo sigue igual figura #5.3.2, la lectura se realiza de buena manera efectuando la llamada al método `Leer()` usando el mensaje apropiado con sus 3 parámetros cuyo valor es recibido en la definición del método `Leer()`. Los 3 parámetros son pasados por valor, es decir, sólo se necesita el valor de cada parámtero para asignarlo al atributo correspondiente, ninguno de ellos se modifica dentro de la definición del método.

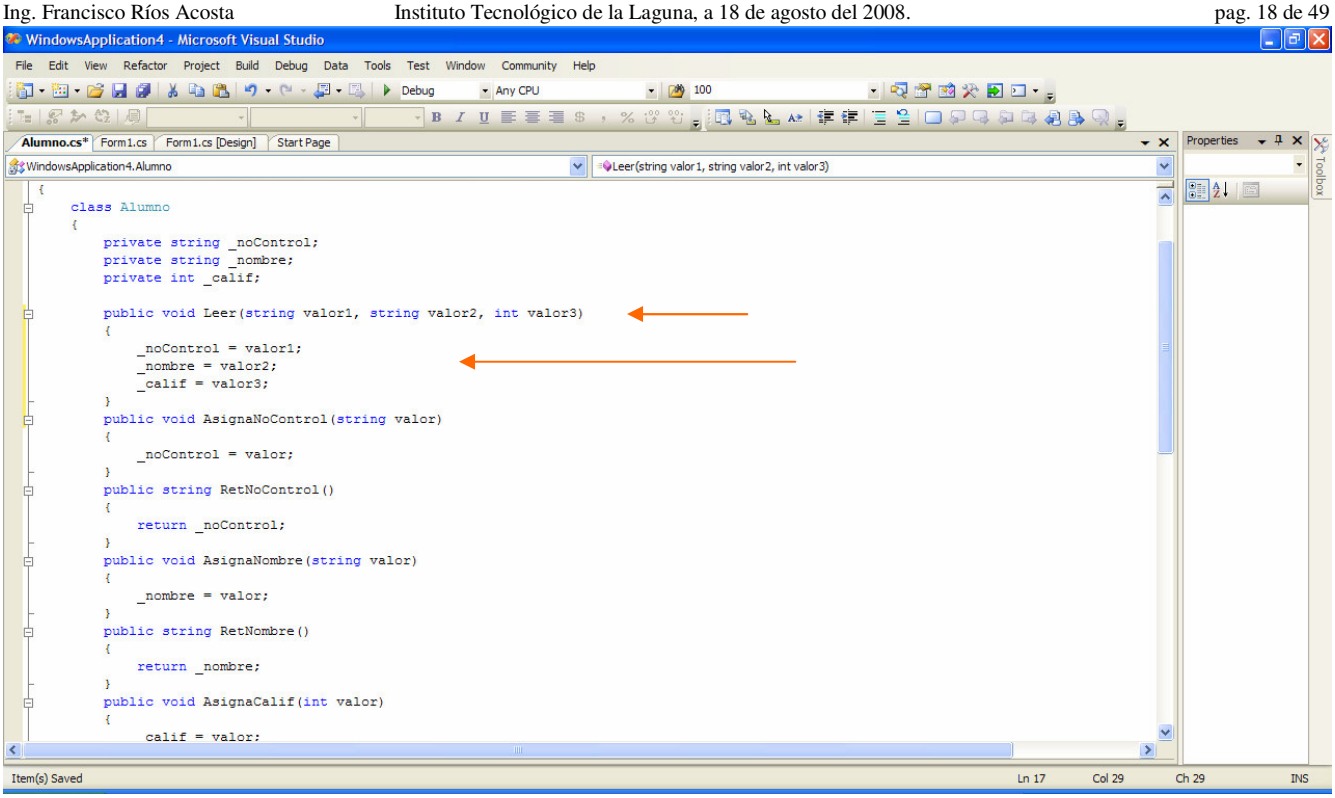


Fig. No. 5.3.1 Definición del método `Leer()` en la clase `Alumno`.

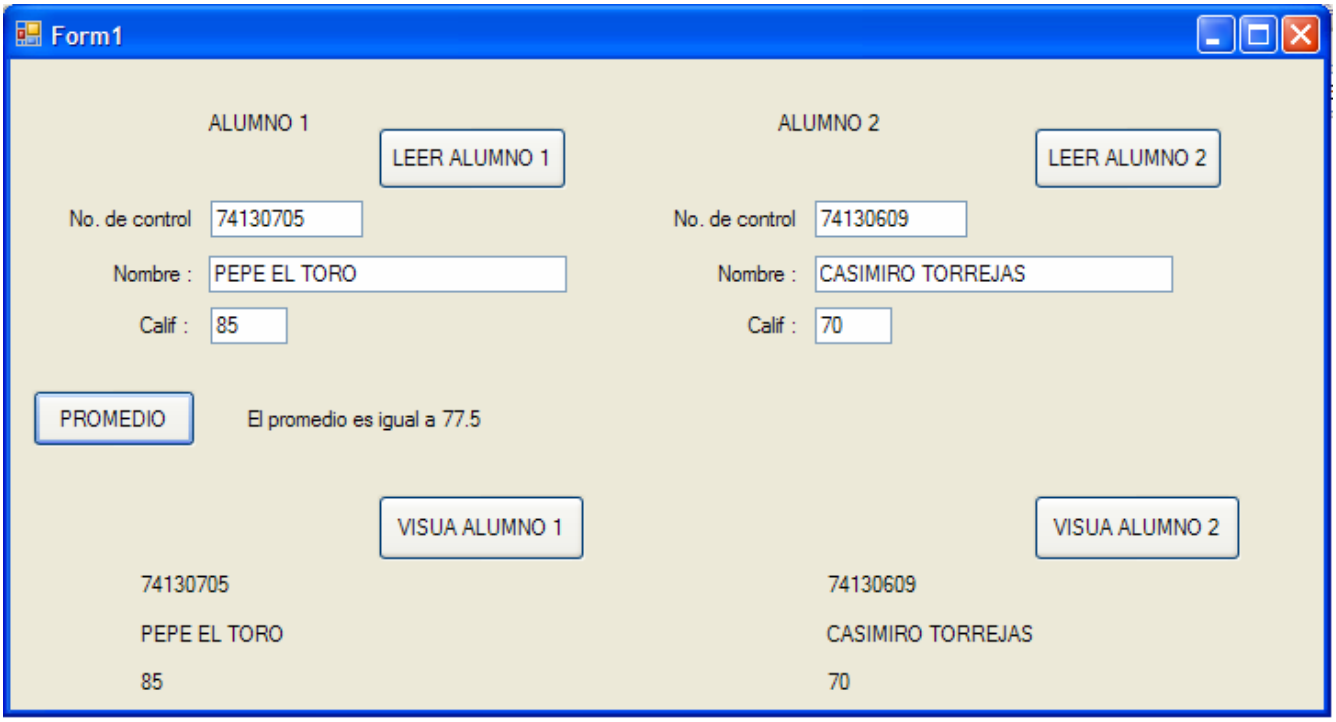
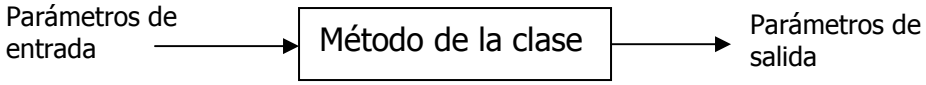


Fig. No. 5.3.2 Ejecución de la aplicación Windows C#.

Los parámetros enviados a un método podemos llamarlos parámetros de entrada al método. Un método realmente es un programa o sea, un conjunto de sentencias cuyos parámetros que recibe podemos comprenderlos como que son datos de entrada que son utilizados por las sentencias del cuerpo del método.

De acuerdo a ésto, podemos “ver” a un método como un programa que recibe datos –parámetros- de entrada :



5.3.3 Parámetros de entrada y de salida.

Ya hemos visto en la sección anterior que los parámetros de entrada son recibidos por un método con el fin de aprovechar su valor para usarlo en cálculos, asignar atributos, entre otras cuestiones. Un parámetro de entrada es pasado por valor al método, o sea que no se modifica su valor dentro del método. Si modificamos el valor del parámetro dentro del método, el cambio no surte efecto debido a que los parámteros que son pasados por valor, sólo existen en el ámbito del método. El ámbito del método es el bloque de sentencias encerrado entre los caracteres llave. Por ejemplo, veamos el método `AsignaNoControl()` que recibe un valor –la propiedad Text del componente TextBox donde es ingresado el número de control del alumno-, y que lo almacena en la variable de tipo **string** con identificador `valor`.

```
public void AsignaNoControl(string valor)
{
     _noControl = valor;
}
```

parámetro de entrada que recibe a la propiedad Text del componente TextBox correspondiente

El parámetro `valor` se asigna al atributo `_noControl` del alumno que efectúa la llamada al método mediante el mensaje, `oAlu1.AsignaNoControl()` u `oAlu2.AsignaNoControl()`. El parámetro `valor` sólo existe en el cuerpo del método o sea, dentro del bloque de código limitado por los caracteres { y } del método.

```
public void AsignaNoControl(string valor)
{
     _noControl = valor;
}
```

parámetro valor sólo existe dentro del bloque de código del método.

Podríamos pensar que si modificamos al parámetro `valor`, estaríamos modificando a la propiedad Text del TextBox involucrado en la llamada al método. NO ES ASI, ya que dicha propiedad se pasó por valor, y habíamos establecido que el pasaje por valor se realiza cuando no queremos que se modifique el parámetro enviado al método. Hagamos la prueba agregando cualquier asignación al parámetro `valor` y observemos si la propiedad Text se modifica.

```
public void AsignaNoControl(string valor)
{
     _noControl = valor;
     valor = "CAMBIO EN EL METODO";
}
```

Quitamos los comentarios a los 3 mensajes :

```
oAlu1.AsignaNoControl(textBox1.Text);
oAlu1.AsignaNombre(textBox3.Text);
oAlu1.AsignaCalif(Convert.ToInt32(textBox5.Text));
```

Y ponemos como comentario al mensaje :

```
//  oAlu1.Leer(textBox1.Text, textBox3.Text, Convert.ToInt32(textBox5.Text));
```

De manera que los eventos Click de los botones `button1` y `button2` de la aplicación quedan con el código que se escribe a continuación :

```
private void button1_Click(object sender, EventArgs e)
{
    oAlu1.AsignaNoControl(textBox1.Text);
    oAlu1.AsignaNombre(textBox3.Text);
    oAlu1.AsignaCalif(Convert.ToInt32(textBox5.Text));
 //  oAlu1.Leer(textBox1.Text, textBox3.Text, Convert.ToInt32(textBox5.Text));
}
```

```
private void button2_Click(object sender, EventArgs e)
{
    oAlu2.AsignaNoControl(textBox2.Text);
    oAlu2.AsignaNombre(textBox4.Text);
    oAlu2.AsignaCalif(Convert.ToInt32(textBox6.Text));
 //  oAlu2.Leer(textBox2.Text, textBox4.Text, Convert.ToInt32(textBox6.Text));
}
```

Una vez que ejecutemos la aplicación leamos los datos de los 2 alumnos y observemos que cuando hacemos click en los botones para lectura de los datos, la propiedad Text del `textBox1` y del `textBox2` quedan sin cambio alguno.

Parámteros de salida.- En ocasiones requerimos de modificar el valor de un parámetro que es pasado al método. este cambio es hecho dentro del método es decir, en el código del método. El parámetro toma las 2 formas : es de entrada y a la vez, es de salida. Su valor puede ser utilizado para efectuar cálculos o asignaciones, y además dentro del método podemos cambiar su valor y retornarlo al programa –segmento de programa- en donde se efectuó el llamado al mensaje que contiene al método. Veamos un ejercicio donde involucremos llamadas a métodos que reciban parámetros que se desea modificar dentro del cuerpo del método.

***Ejercicio.*- Deseamos leer el número de control, el nombre y 3 calificaciones parciales de un alumno. Cuando se realice la lectura, debemos calcular y visualizar el promedio de las calificaciones del alumno leído. Utiliza un parámetro de salida para almacenar el promedio.**

La interfase será muy parecida a la anterior aplicación, con la diferencia de que ahora tenemos la lectura de un sólo alumno. También existe otra diferencia : ahora son 3 calificaciones que leer y antes era sólo una. La interfase gráfica se presenta en la figura 5.3.3. El componente label6 será utilizado para visualizar el promedio de las calificaciones del alumno leido.

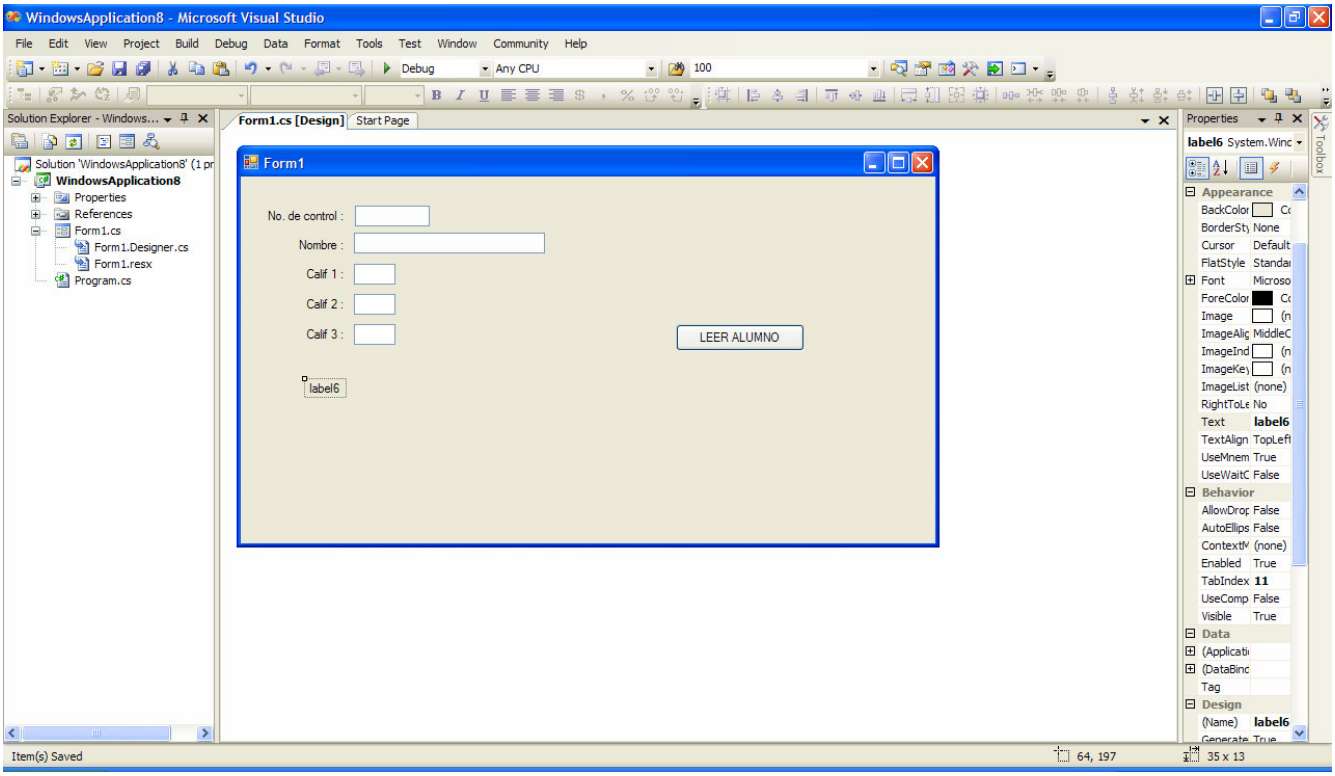


Fig. No. 5.3.3 Interfase gráfica del ejemplo de uso de un parámetro de salida.

La clase `Alumno` debemos agregarla al proyecto pero ahora tiene atributos diferentes :

```
_noControl
_nombre
_calif1
_calif2
_calif3
```

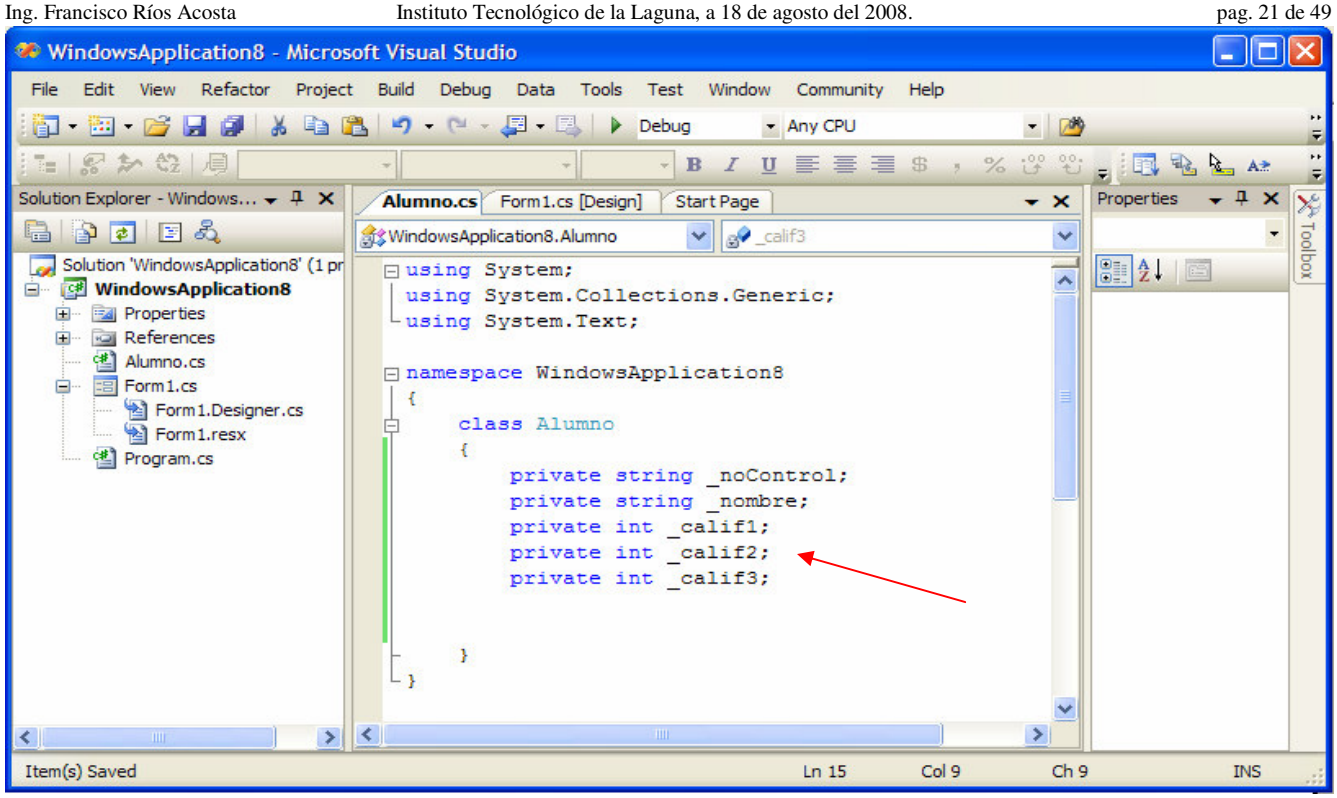


Fig. No. 5.3.4 Clase `Alumno` agregada al proyecto.

Tecleemos la declaración del objeto `oAlu` a la aplicación dentro de la clase `Form1`. Recordemos que la clase `Form1` reside en el archivo `Form1.cs`.

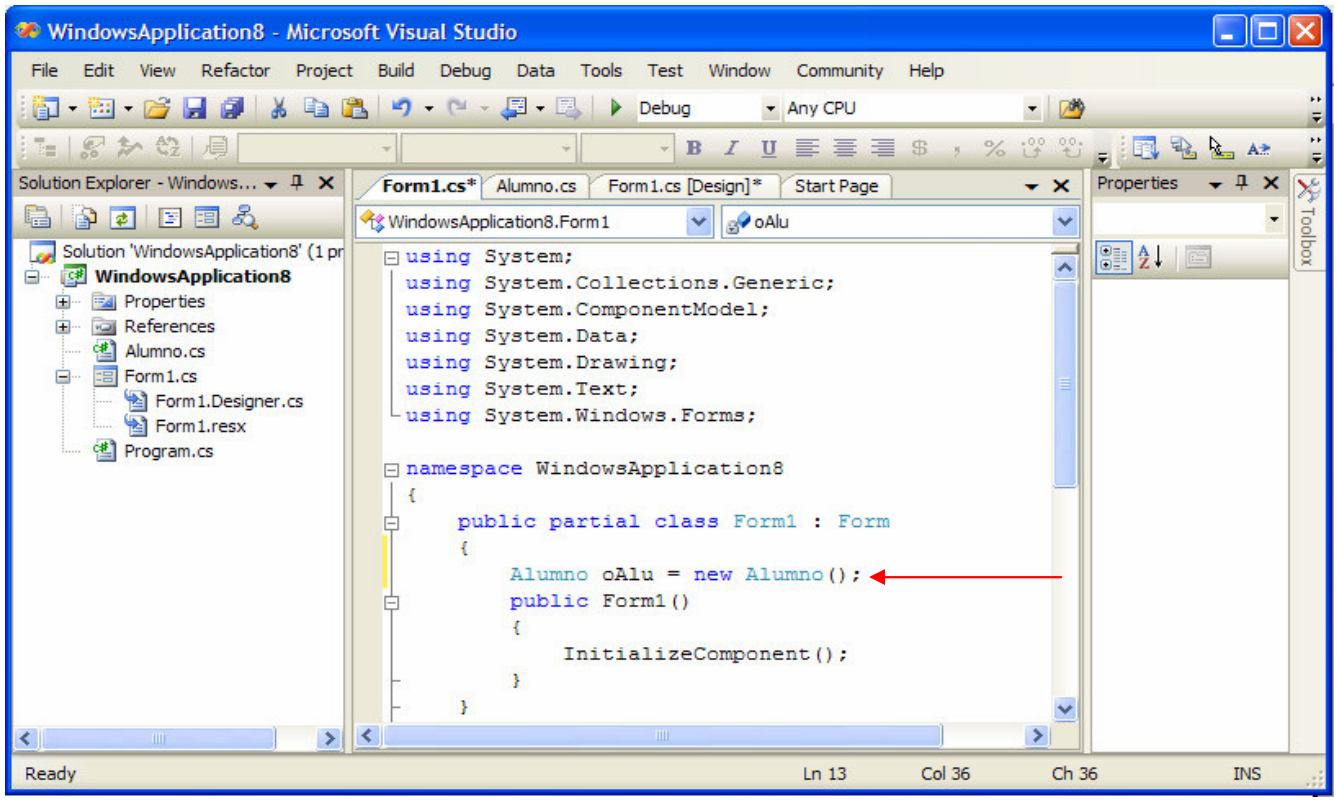
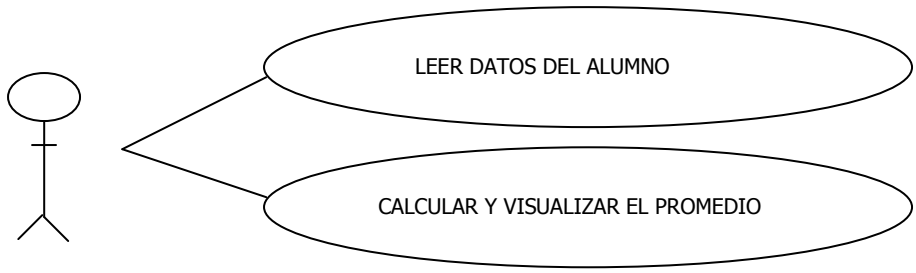
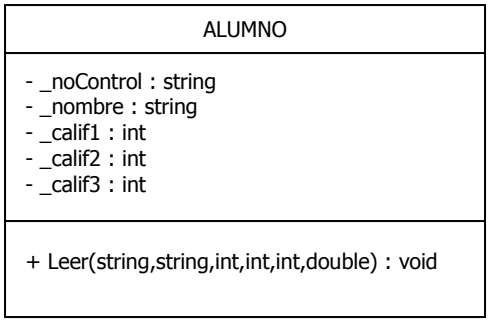


Fig. No. 5.3.5 Definición del objeto `oAlu` en la clase `Form1`.

Entonces el diagrama de casos de uso y el diagrama de clases podrían ser los siguientes :





El último parámetro del método `Leer()` representa el parámetro de salida donde depositaremos el promedio de las 3 calificaciones. Agreguemos el mensaje al alumno `oAlu` que incluye al método `Leer()`, dentro del evento Click del botón `button1`.

```
namespace WindowsApplication8
{
    public partial class Form1 : Form
    {
        Alumno oAlu = new Alumno();
        public Form1()
        {
            InitializeComponent();
        }

        private void button1_Click(object sender, EventArgs e)
        {
            double promedio;
            oAlu.Leer(textBox1.Text, textBox2.Text, Convert.ToInt32(textBox3.Text),
                      Convert.ToInt32(textBox4.Text), Convert.ToInt32(textBox5.Text), ref promedio);
        }
    }
}
```

Notemos que definimos un dato tipo **`double`** al que hemos llamado `promedio`. esta variable la utilizamos para enviarla al método `Leer()` como un parámetros de salida. El método `Leer()` recibe los parámetros con los valores tecleados por el usuario para el número de control, el nombre y las 3 calificacioens del alumno.

Observemos que si un parámetro será modificado en el método que se llama, dicho parámetro deberá ser enviado por referencia. La manera en que se envía un parámetro por referencia es antecediéndolo en la llamada con la palabra reservada **`ref`**.

Ahora definamos el método Leer() en la clase Alumno de acuerdo al código que se muestra enseguida :

```
class Alumno
{
    private string _noControl;
    private string _nombre;
    private int _calif1;
    private int _calif2;
```

```
    private int _calif3;

    public void Leer(string val1, string val2, int val3, int val4, int val5, ref double promedio)
    {
        _noControl = val1;
        _nombre = val2;
        _calif1 = val3;
        _calif2 = val4;
        _calif3 = val5;
        promedio = (_calif1+_calif2+_calif3)/3.0;
    }
}
```

Notemos que el parámetro de salida `promedio` se declara antecediendo a su tipo la palabra reservada **`ref`**. Si ejecutamos la aplicación obtenemos un error según se indica en la figura 5.3.6.

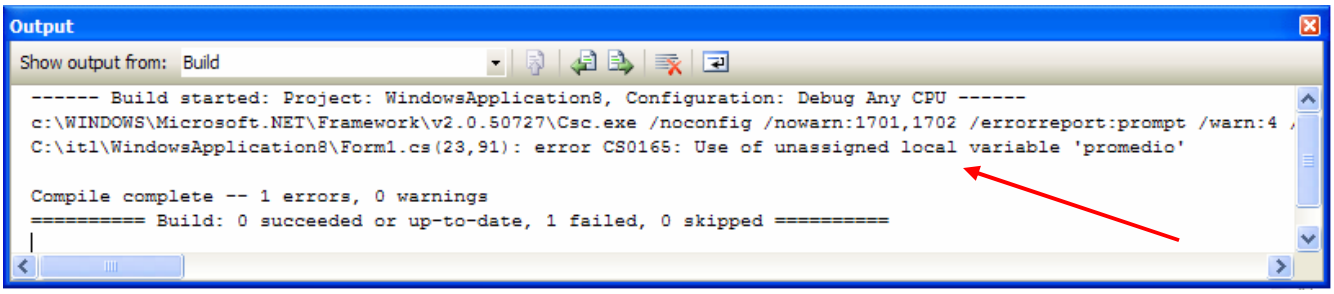


Fig. No. 5.3.6 ERROR, un dato pasado por referencia debe ser previamente asignado.

debemos tener en cuenta que si queremos pasar un parámetro por referencia a un método, el parámetro deberá ser previamente inicializado a cualquier valor. Nuestro error lo corregimos fácilmente, sólo debemos inicializar la variable `promedio` a 0.0 en el momento de su declaración.

```
private void button1_Click(object sender, EventArgs e)
{
    double promedio=0.0;
    oAlu.Leer(textBox1.Text, textBox2.Text, Convert.ToInt32(textBox3.Text),
              Convert.ToInt32(textBox4.Text), Convert.ToInt32(textBox5.Text), ref promedio);
}
```

Ahora ya no tenemos el error pero todavía tenemos que agregar el código que visualiza el valor del promedio de las 3 calificaciones en el componente `label6` en su propiedad Text. Agreguemos el código que visualiza dicho valor según lo indicamos en el Click del `button1`.

```
private void button1_Click(object sender, EventArgs e)
{
    double promedio=0.0;
    oAlu.Leer(textBox1.Text, textBox2.Text, Convert.ToInt32(textBox3.Text),
              Convert.ToInt32(textBox4.Text), Convert.ToInt32(textBox5.Text), ref promedio);
    label6.Text = "EL PROMEDIO ES = " + Convert.ToString(promedio);
}
```

El valor del `promedio` se ha convertido a string usando la clase `Convert` y su método predefinido `ToString()`. Ejecutemos la aplicación Windows para que veamos como el parámetro de salida `promedio` es modificado en su valor de 0.0 al valor correspondiente a la media aritmética de las 3 calificaciones del alumno que han sido leidas. La figura #5.3.7 muestra la ejecución de la aplicación para ciertos valores.

Un buen ejercicio que podemos hacer de manera que veamos lo que sucede si el parámtero de salida no es pasado por referencia, es quitar la palabra resrevada **`ref`** tanto en la llamada como en la definición del método `Leer()`. Los cambios los presentamos en los códigos para el Click del `button1` y para la definición del método en la clase `Alumno`.

```
private void button1_Click(object sender, EventArgs e)
{
    double promedio=0.0;
    oAlu.Leer(textBox1.Text, textBox2.Text, Convert.ToInt32(textBox3.Text),
                      Convert.ToInt32(textBox4.Text), Convert.ToInt32(textBox5.Text), promedio);
```

```
    label6.Text = "EL PROMEDIO ES = " + Convert.ToString(promedio);
}
```

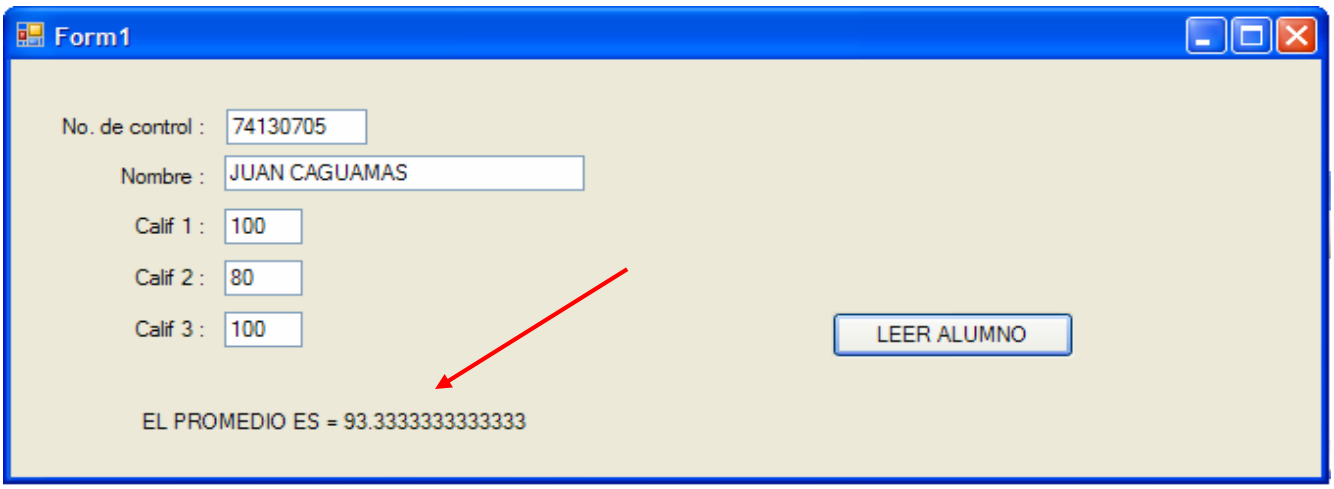


Fig. No. 5.3.7 Cálculo y visualización del parámetro `promedio`.

Ahora eliminamos el **`ref`** en la definición del método `Leer()`.

```
class Alumno
{
    private string _noControl;
    private string _nombre;
    private int _calif1;
    private int _calif2;
    private int _calif3;

    public void Leer(string val1, string val2, int val3, int val4, int val5, double promedio)
    {
        _noControl = val1;
        _nombre = val2;
        _calif1 = val3;
        _calif2 = val4;
        _calif3 = val5;
        promedio = (_calif1+_calif2+_calif3)/3.0;
    }
}
```

Si ejecutamos la aplicación observamos que el promedio visualizado es 0.0 el cual es el valor al que previamente ha sido asignado. Como vemos el método `Leer()` "lo modifica" pero dicho cambio no es retroalimentado al código donde se efectuo la llamada. El cambio no se realizó porque el pasaje del parámetro fue hecho por valor, figura #5.3.8.

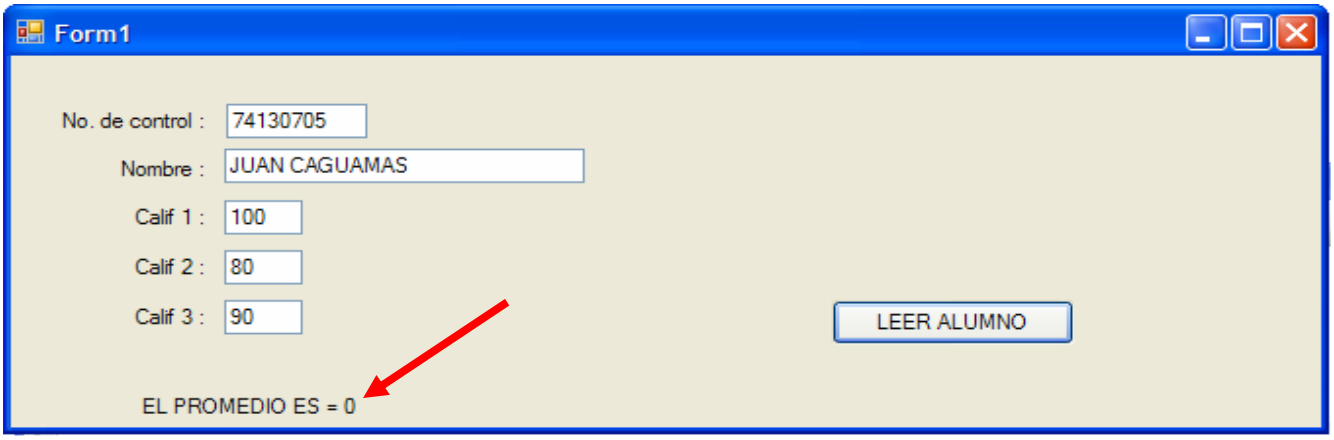
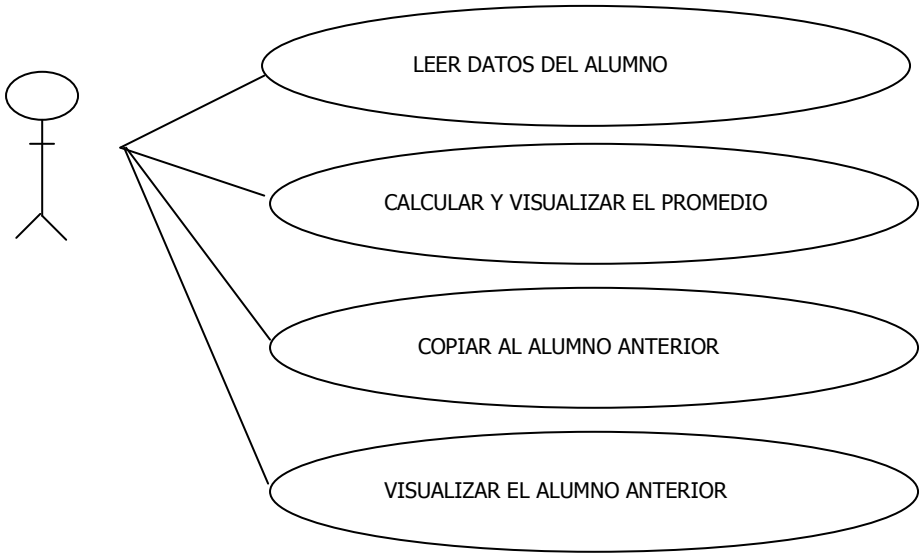
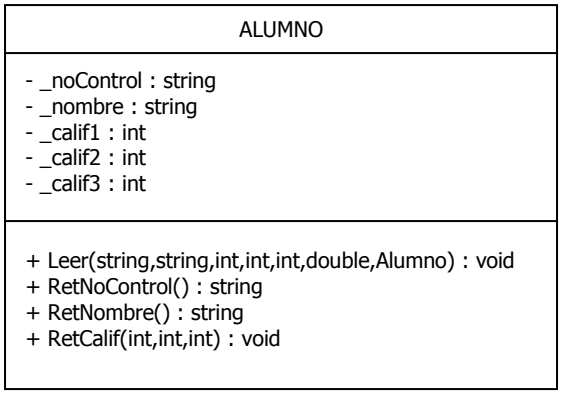


Fig. No. 5.3.8 El parámetro `promedio` no ha sido modificado ya que se pasó por valor al método.

***Ejercicio.*- Modifica el ejercicio anterior de forma que sean visualizados los datos del alumno previamente leido, en el momento en que se leen los datos del nuevo alumno. Utiliza un objeto de la clase `Alumno` como parámetro de salida al método `Leer()`.**

El diagrama de casos de uso y de clases es casi igual al anterior, sólo se añade un nuevo parámetro de salida al método `Leer()` en la clase `Alumno`.





Tenemos 2 de los métodos que habíamos utilizado anteriormente en la clase `Alumno` : `RetNoControl()` y `RetNombre()`, que devuelven el atributo `_noControl` y el atributo `_nombre` respectivamente. El método que es bastante diferente es el `RetCalif()` que ahora recibe 3 parámetros de tipo **int**, que además son de salida. Estos parámetros de salida. retornan a las 3 calificaciones del alumno. Otra forma de conseguir el retorno de las 3 calificaciones es definir 3 métodos que retornen cada uno, a una de las calificaciones del alumno según corresponda, es decir, el primer método retornaría a la primer calificación, el segundo a la segunda calificación y el tercer método a la tercera. Nos inclinaremos por la primer manera de hacerlo y que se muestra en el diagrama de la clase `Alumno`.

Como primer paso agregamos de manera local –dentro del evento Click del `button1`- la declaración y definición de un nuevo objeto `oAluAnt` de la clase `Alumno` y que representa al alumno previamente leido. Notemos que cuando se lee los datos de un primer alumno, este objeto `oAluAnt` tendrá sus atributos sin inicializar. Los valores que puedan tener sus atributos están indeterminados.

```
private void button1_Click(object sender, EventArgs e)
{
    double promedio=0.0;
    Alumno oAluAnt = new Alumno();
    oAlu.Leer(textBox1.Text, textBox2.Text, Convert.ToInt32(textBox3.Text),
              Convert.ToInt32(textBox4.Text), Convert.ToInt32(textBox5.Text),ref promedio, oAluAnt);
    label6.Text = "EL PROMEDIO ES = " + Convert.ToString(promedio)
}
```

Notemos que el parámetro de salida `oAluAnt` no le antecede la palabra reservada **`ref`**, debido a que en C# los objetos son pasados siempre por referencia. Observemos que el mensaje donde se llama al método `Leer()` ya tiene incluido el parámetro de salida `oAluAnt`, donde serán almacenados los atributos del alumno previamente leido.

Ahora vayamos a la definición del método `Leer()` en la clase `Alumno` para agregar este parámetro, además de la asignación de sus atributos. El código que debemos agregar es mostrado a continuación.

```
public void Leer(string val1, string val2, int val3, int val4, int val5, ref double promedio,
                                                                              Alumno oAluAnt)
{
    oAluAnt._noControl = _noControl;
    oAluAnt._nombre = _nombre;
    oAluAnt._calif1 = _calif1;
    oAluAnt._calif2 = _calif2;
    oAluAnt._calif3 = _calif3;
    _noControl = val1;
    _nombre = val2;
    _calif1 = val3;
    _calif2 = val4;
    _calif3 = val5;
    promedio = (_calif1+_calif2+_calif3)/3.0;
}
```

Antes de asignar los nuevos valores a los atributos del alumno que hizo la llamada, almacenamos en los atributos del objeto `oAluAnt` que pasamos como un parámetro de salida, a los atributos del alumno previamente leidos. ¿Por qué se pueden acceder directamente a los atributos del objeto `oAluAnt` sin necesidad de usar un método de la clase?. Discútelo con tus compañeros y con tu profesor. También vemos que no es necesario anteceder la declaración del parámetro `oAluAnt` con la palabra reservada **`ref`** por lo que antes ya hemos explicado.

Sólo queda visualizar los atributos del alumno `oAluAnt` en sus respectivas etiquetas –componentes Label-. Para ello debemos agregar los componentes Label que se muestran en la figura #5.3.9.

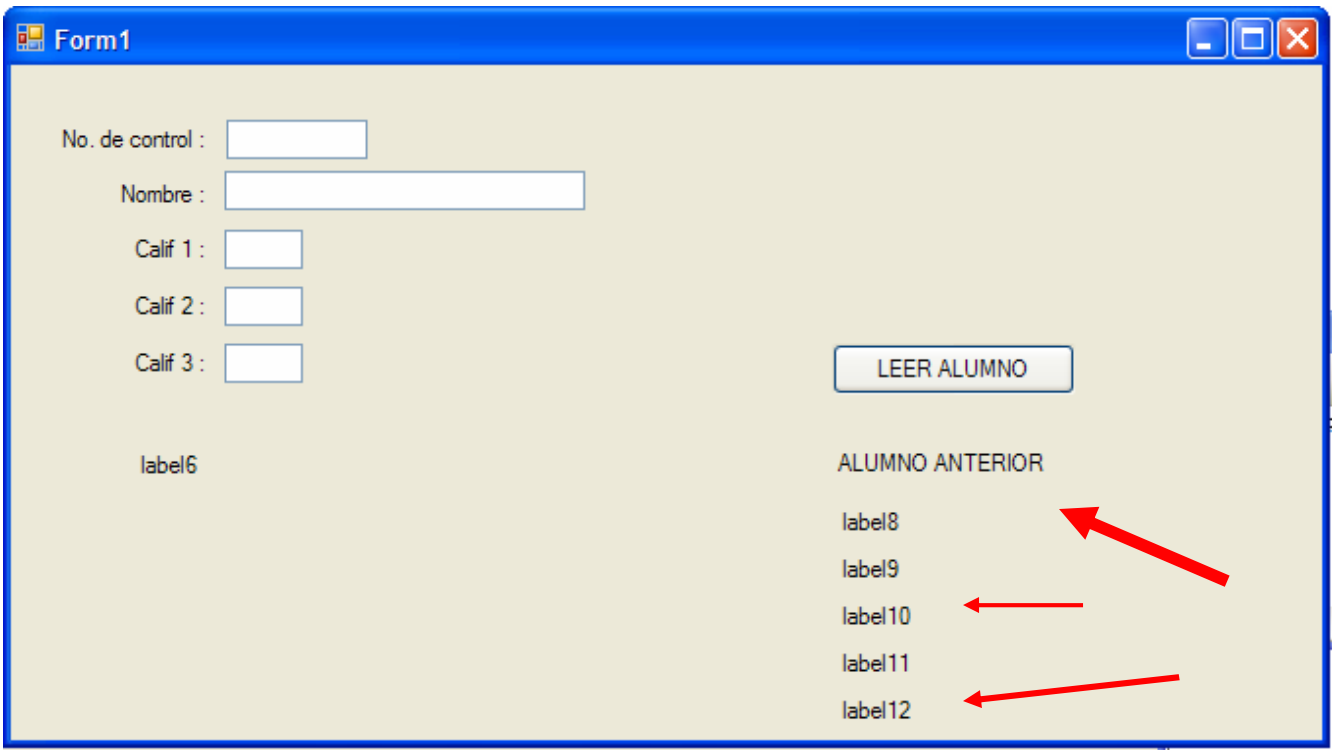


Fig. No. 5.3.9 Interfase gráfica que incluye a los componentes Label utilizados para visualizar los datos del alumno `oAluAnt`.

Antes de ejecutar la aplicación con estas modificaciones, debemos agregar los métodos que antes habiamos mencionado a la clase `Alumno` : `RetNoControl()` y `RetNombre()`. Copialos de tu aplicación anterior o teclealos.

```
class Alumno
{
    private string _noControl;
    private string _nombre;
    private int _calif1;
    private int _calif2;
    private int _calif3;

    public void Leer(string val1, string val2, int val3, int val4, int val5, ref double promedio,
                                                                                    Alumno oAluAnt)
    {
        oAluAnt._noControl = _noControl;
        oAluAnt._nombre = _nombre;
        oAluAnt._calif1 = _calif1;
        oAluAnt._calif2 = _calif2;
        oAluAnt._calif3 = _calif3;
        _noControl = val1;
        _nombre = val2;
        _calif1 = val3;
        _calif2 = val4;
        _calif3 = val5;
        promedio = (_calif1+_calif2+_calif3)/3.0;
    }
    public string RetNoControl()
    {
        return _noControl;
    }
    public string RetNombre()
    {
        return _nombre;
    }
}
```

Necesitamos agregar los mensajes que llaman a estos 2 métodos dentro del evento Click del botón `button1`, una vez que se efectúe la llamada al método `Leer()` en el mensaje al objeto `oAlu`. Hagamoslo según se indica en el código de dicho botón listado enseguida.

```
private void button1_Click(object sender, EventArgs e)
{
    double promedio=0.0;
    Alumno oAluAnt = new Alumno();
    oAlu.Leer(textBox1.Text, textBox2.Text, Convert.ToInt32(textBox3.Text),
              Convert.ToInt32(textBox4.Text), Convert.ToInt32(textBox5.Text),ref promedio, oAluAnt);
    label6.Text = "EL PROMEDIO ES = " + Convert.ToString(promedio);
    label8.Text = oAluAnt.RetNoControl();
    label9.Text = oAluAnt.RetNombre();
}
```

Salvemos los cambios al proyecto y ejecutemos la aplicación. Vemos que efectivamente los atributos del número de control y el nombre del alumno previamente leidos se visualizan en los componentes `label8` y `label9`.

Sólo nos falta agregar el mensaje donde llamemos al método `RetCalif()` con sus 3 parámetros de salida que serán las calificaciones del alumno que hace la llamada, en este caso el objeto `oAluAnt`. Vayamos de nuevo al Click del `button1` para añadir las 3 variables enteras que se enviarán como parámetros de salida al método `RetCalif()`. Notemos que ya hemos agregado en el código que a continuación se muestra, a los 3 parámetros y desde luego que los hemos antecedido de la palabra reservada ref para que puedan ser modificados. Observemos que debemos inicializar a estas 3 variables enteras.

```
private void button1_Click(object sender, EventArgs e)
{
   double promedio=0.0;
   Alumno oAluAnt = new Alumno();
   oAlu.Leer(textBox1.Text, textBox2.Text, Convert.ToInt32(textBox3.Text),
             Convert.ToInt32(textBox4.Text), Convert.ToInt32(textBox5.Text),ref promedio, oAluAnt);
   label6.Text = "EL PROMEDIO ES = " + Convert.ToString(promedio);
   label8.Text = oAluAnt.RetNoControl();
   label9.Text = oAluAnt.RetNombre();
   int cal1 = 0, cal2 = 0, cal3 = 0;
   oAluAnt.RetCalif(ref cal1, ref cal2, ref cal3);
}
```

Siguiendo con nuestro rompecabezas, agreguemos a la clase `Alumno` el método `RetCalif()` según lo mostramos :

```
public void RetCalif(ref int cal1, ref int cal2, ref int cal3)
{
    cal1 = _calif1;
    cal2 = _calif2;
    cal3 = _calif3;
}
```

Sólo falta agregar la visualización de las calificaciones –parámetros de salida- a los componentes Label correspondientes. este código faltante lo agregamos en el Click del `button1`.

```
private void button1_Click(object sender, EventArgs e)
{
    double promedio=0.0;
    Alumno oAluAnt = new Alumno();
    oAlu.Leer(textBox1.Text, textBox2.Text, Convert.ToInt32(textBox3.Text),
              Convert.ToInt32(textBox4.Text), Convert.ToInt32(textBox5.Text),ref promedio, oAluAnt);
    label6.Text = "EL PROMEDIO ES = " + Convert.ToString(promedio);
    label8.Text = oAluAnt.RetNoControl();
    label9.Text = oAluAnt.RetNombre();
    int cal1 = 0, cal2 = 0, cal3 = 0;
    oAluAnt.RetCalif(ref cal1, ref cal2, ref cal3);
    label10.Text = Convert.ToString(cal1);
    label11.Text = Convert.ToString(cal2);
    label12.Text = Convert.ToString(cal3);
}
```

Ejecutemos la aplicación para ver como funciona. La figura #5.3.10 muestra el resultado para ciertos alumnos.

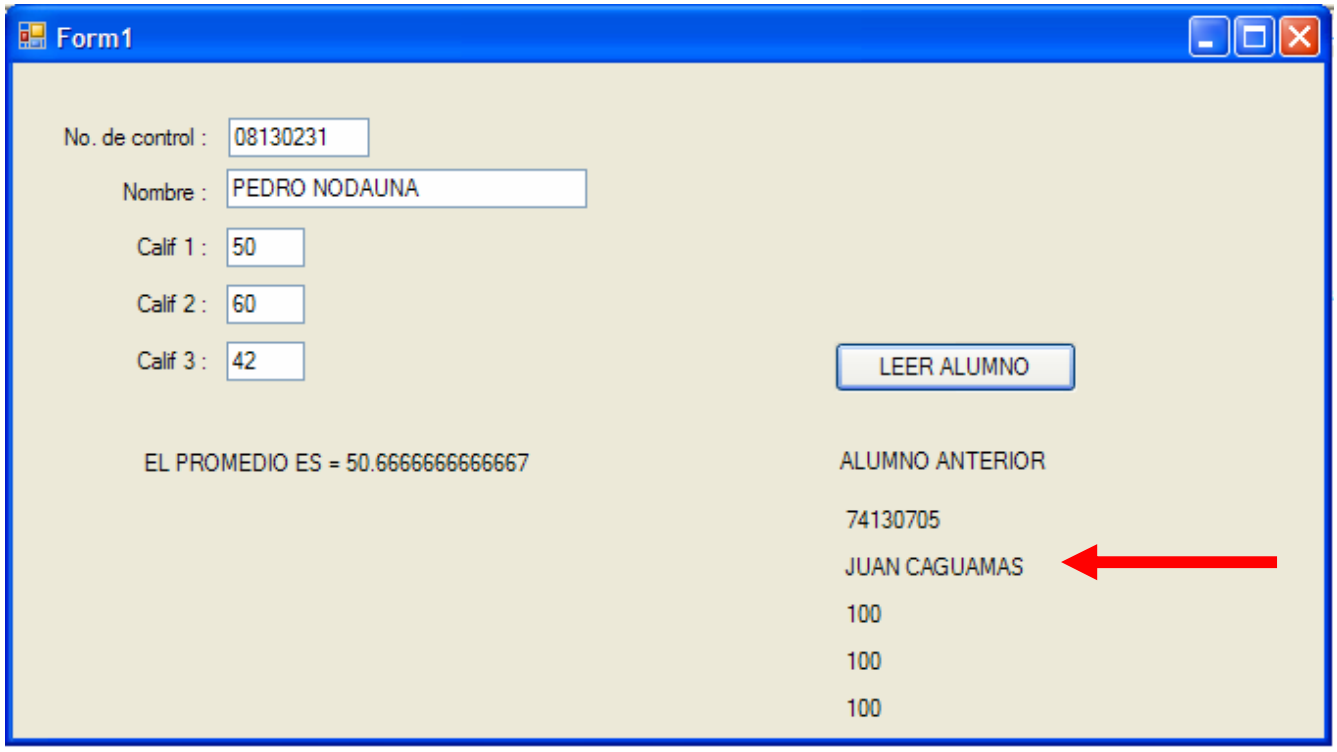


Fig. No. 5.3.10 Aplicación C# que visualiza a los datos del alumno previamente leido `oAluAnt`.

EJERCICIO PROPUESTO.
Modifica la aplicación anterior para que en lugar de usar a un objeto `oAluAnt` pasado como parámetro de salida, se use al mismo objeto `oAlu` que se lee.

out. Existe otra manera de implementar el pasaje por referencia utilizando la palabra reservada `out`. Esta palabra se antecede a los parámetros cuyo valor requerimos modificar o afectar en el cuerpo del método. A diferencia de **ref**, `out` no nos demanda que inicialicemos la variable –dato- que deseamos pasar de parámetro al método. La manera de usar `out` es la misma que cuando utulizamos **ref**. Así que los códigos serán los mismo sólo que no inicializaremos las variables `promedio`, `cal1`, `cal2` y `cal3`, las cuales ahora pasaremos como parámetros de salida a los métodos `Leer()` y `RetCalif()`.

Los eventos Click de los botones en `Form1.cs` tendrán el código mostrado donde se ha sustituido la palabra `ref` por `out`, además que se ha evitado inicializar a cada parámetro ya que el uso de `out` no requiere dicha asignación previa.

```
private void  button1_Click(object sender, EventArgs e)
{
    //double promedio=0.0;
    double promedio;
    Alumno oAluAnt = new Alumno();
   //    oAlu.Leer(textBox1.Text, textBox2.Text, Convert.ToInt32(textBox3.Text),
   //              Convert.ToInt32(textBox4.Text), Convert.ToInt32(textBox5.Text),ref promedio,
                                                                          oAluAnt);
    oAlu.Leer(textBox1.Text, textBox2.Text, Convert.ToInt32(textBox3.Text),
              Convert.ToInt32(textBox4.Text), Convert.ToInt32(textBox5.Text),out promedio, oAluAnt);
    label6.Text = "EL PROMEDIO ES = " + Convert.ToString(promedio);
    label8.Text = oAluAnt.RetNoControl();
    label9.Text = oAluAnt.RetNombre();
  //   int cal1 = 0, cal2 = 0, cal3 = 0;
    int cal1, cal2, cal3;
    oAluAnt.RetCalif(out cal1, out cal2, out cal3);
    label10.Text = Convert.ToString(cal1);
    label11.Text = Convert.ToString(cal2);
    label12.Text = Convert.ToString(cal3);
}

private void button2_Click(object sender, EventArgs e)
{
    label13.Text = oAlu3.RetNoControl();
    label14.Text = oAlu3.RetNombre();
    int cal1, cal2, cal3;
    oAlu3.RetCalif(out cal1, out cal2, out cal3);
    label15.Text = Convert.ToString(cal1);
    label16.Text = Convert.ToString(cal2);
    label17.Text = Convert.ToString(cal3);
}

private void button3_Click(object sender, EventArgs e)
{
    label13.Text = oAlu4.RetNoControl();
    label14.Text = oAlu4.RetNombre();
    int cal1, cal2, cal3;
    oAlu4.RetCalif(out cal1, out cal2, out cal3);
    label15.Text = Convert.ToString(cal1);
    label16.Text = Convert.ToString(cal2);
    label17.Text = Convert.ToString(cal3);
}
```

Se quitó la incialización

Cambio de ref por out.

También debemos modificar la definición de los métodos `RetCalif()` y `Leer()` de la clase `Alumno` en la parte de recepción de parámetros, según se indica en el código siguiente :

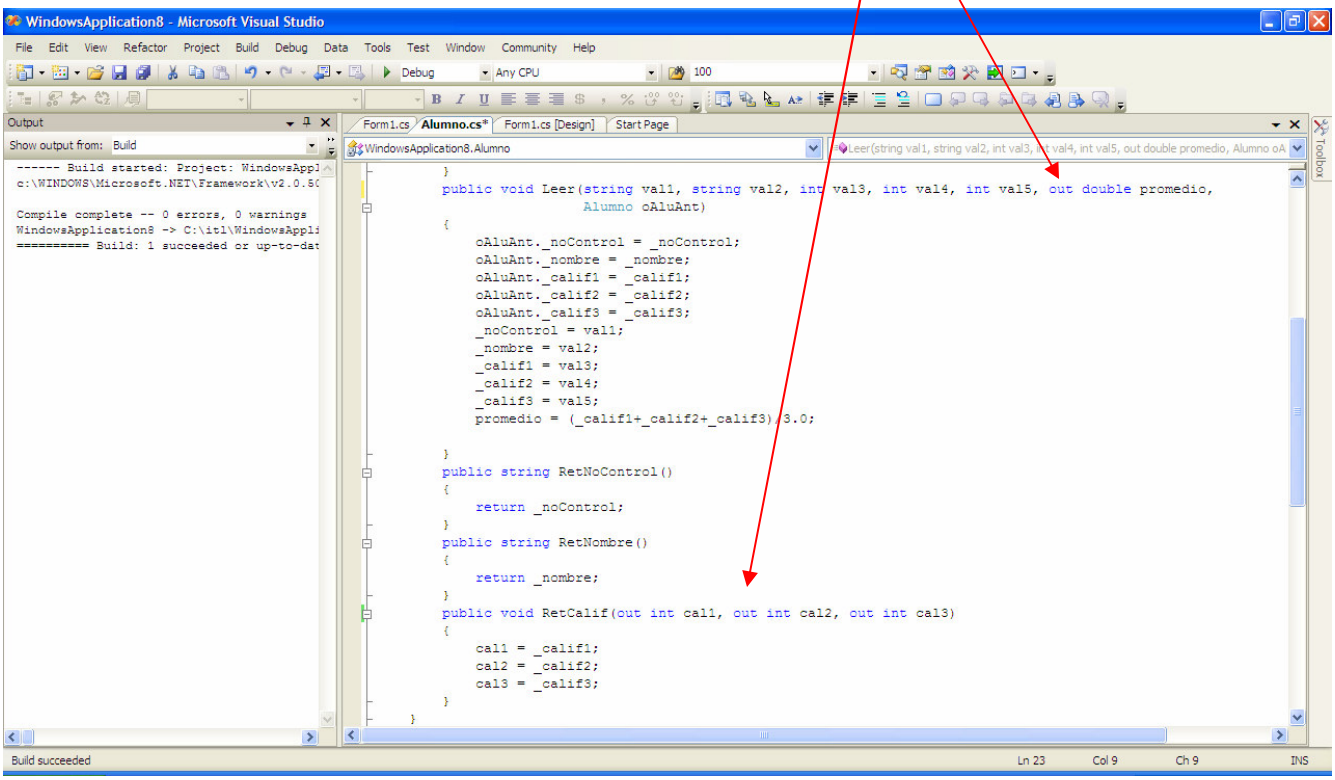
```
public void Leer(string val1, string val2, int val3, int val4, int val5, out double promedio,
                                                                          Alumno oAluAnt)
{
      oAluAnt._noControl = _noControl;
      oAluAnt._nombre = _nombre;
      oAluAnt._calif1 = _calif1;
      oAluAnt._calif2 = _calif2;
      oAluAnt._calif3 = _calif3;
      _noControl = val1;
      _nombre = val2;
      _calif1 = val3;
      _calif2 = val4;
      _calif3 = val5;
      promedio = (_calif1+_calif2+_calif3)/3.0;
}
```

Cambio de ref por out.

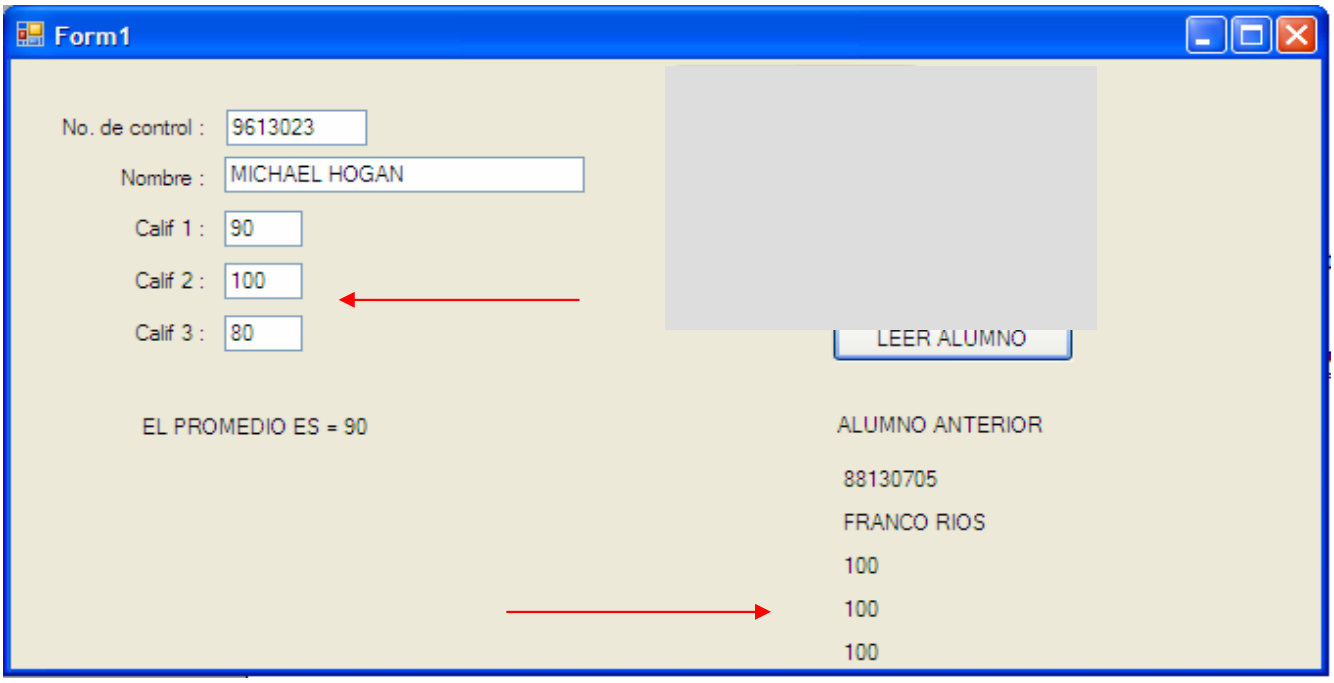
Y para el método `RetCalif()` tenemos :

```
public void RetCalif(out int cal1, out int cal2, out int cal3)
{
      cal1 = _calif1;
      cal2 = _calif2;
      cal3 = _calif3;
}
```

Cambio de ref por out.



La ejecución de la aplicación Windows C# debe realizarse sin problemas. La ventana siguiente muestra la ejecución de la aplicación para 2 alumnos leídos con nombre “FRANCO RIOS” y “MICHAEL HOGAN”.



5.3.4 El constructor.

El constructor es un método que se usa para realizar la inicialización de los atributos de un objeto. Sus características son :

- Es un método de la clase.
- Se ejecuta al momento de definir un objeto de la clase.
- Se llama igual que la clase –identificador-.
- Pueden existir mas de un constructores en la clase –sobrecarga-. Puede recibir parámetros.
- No retorna –nunca- valores. Ni siquiera el tipo void.

Vayamos al ejercicio de la sección anterior. Cuando efectuamos la primer lectura los atributos del alumno leido previamente `oAluAnt` no han sido inicializados por nosotros. El C# le ha puesto por omisión un valor de 0 a cada calificación y al número de control y al nombre los ha asignado a la cadena nula es por eso que la aplicación contesta con lo mostrado en la figura # 5.3.11.

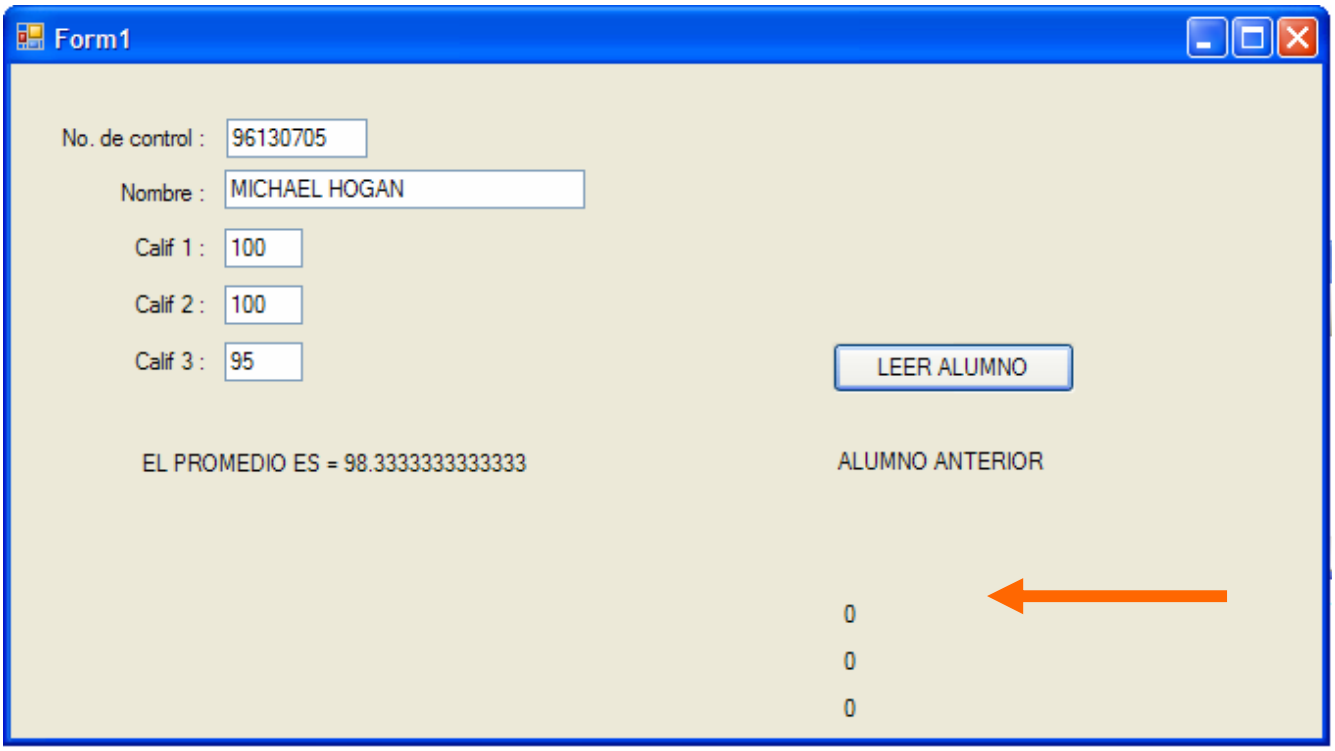


Fig. No. 5.3.11 Aplicación C# que visualiza los valores por omisión para el alumno anterior `oAluAnt`.

Usemos un constructor para inicializar los atributos de un objeto en nuestro caso el de los objetos `oAlu` y `oAluAnt`. Recordemos que el constructor es un método de la clase, por lo tanto el constructor que deseamos lo debemos definir en la clase `Alumno`.

Vamos a inicializar los valores de los atributos según lo indicamos enseguida :

atributo	valor
_noControl	08130000
_nombre	Juan N
_calif1	-1
_calif2	-1
_calif3	-1

Lo que sigue es agregar el constructor a la clase `Alumno` como cualquier método pero conservando las características que hemos enumerado al inicio de esta sección. Seleccionemos en nuestra aplicación la pestaña que contiene la clase `Alumno` y tecleemos el código del constructor.

```
class Alumno
{
    private string _noControl;
    private string _nombre;
    private int _calif1;
    private int _calif2;
    private int _calif3;

    public Alumno()
    {
        _noControl = "08130000";
        _nombre = "JUAN N";
        _calif1 = -1;
        _calif2 = -1;
        _calif3 = -1;
    }
...
...
}
```

Ejecutemos la aplicación para observar que ahora cuando leemos al primer alumno, el objeto `oAluAnt` tiene los valores que el constructor asignó al objeto `oAlu` cuando fue definido, figura #5.3.12. Desde luego que el objeto `oAluAnt` antes de ser asignado en el método `Leer()` tenía sus atributos con el valor dado por el constructor. Dejamos de ejercicio el visualizar los atributos del objeto `oAluAnt` al inicio de la ejecución de la aplicación.

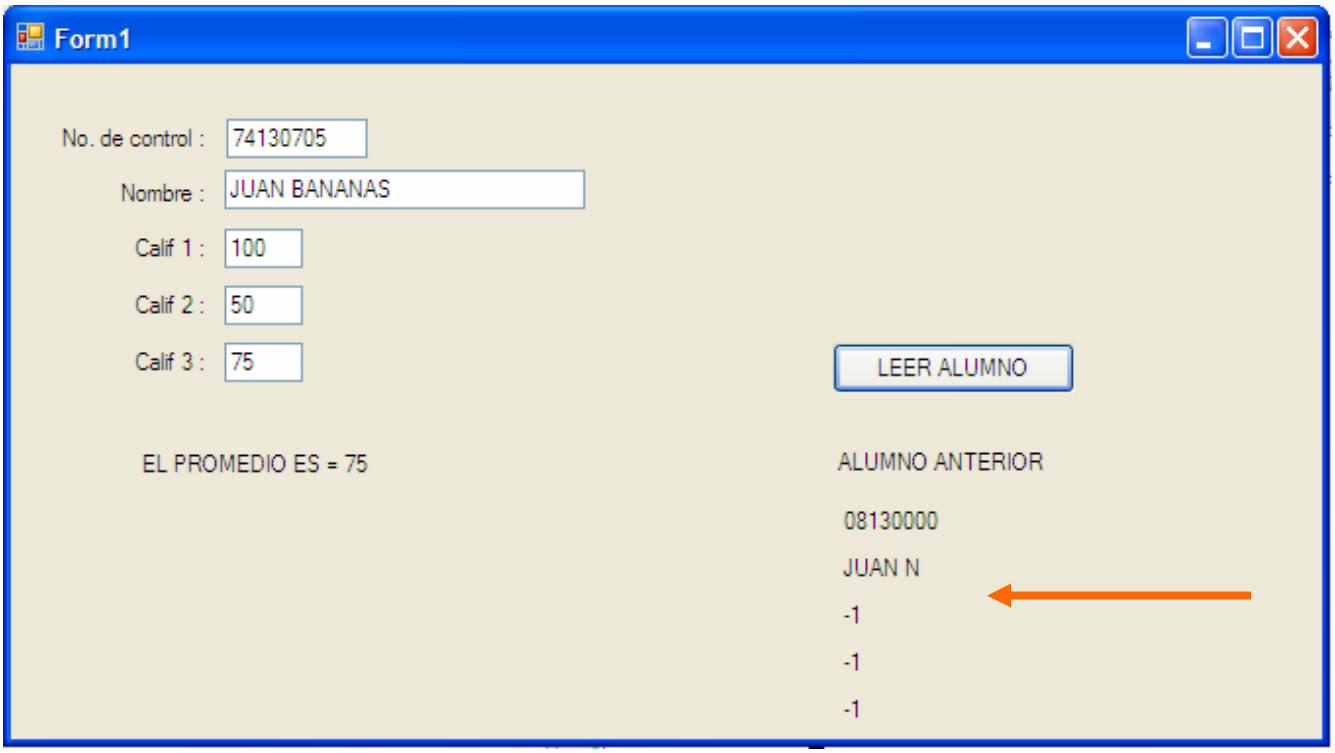


Fig. No. 5.3.12 Aplicación C# que visualiza los datos del `oAluAnt` inicializados por el constructor de la clase.

A este tipo de constructor que no tiene parámetros se le llama constructor por defecto. Sólo puede existir en una determinada clase, un sólo constructor por defecto –ni 2 ni 3 ni4, sólo uno-.

```
public Alumno()
{
    _noControl = "08130000";
    _nombre = "JUAN N";
    _calif1 = -1;
    _calif2 = -1;
    _calif3 = -1;
}
```

El constructor por defecto no tiene parámetros.

En una clase pueden existir mas de un constructores. Otro tipo de constructores son los llamados constructores parametrizados. Un constructor de este tipo recibe parámetros y depende del número de ellos y de su tipo, para que existan mas de un constructores parametrizados.

Un constructor parametrizado difiere de otro constructor parametrizado –o por defecto- según los criterios siguientes :

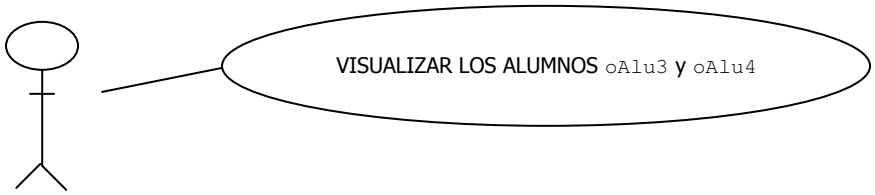
- El número de parámetros.
- El tipo de cada uno de los parámetros.

El lenguaje C# dará por bueno un constructor parametrizado si su número de parámetros es diferente al de otros. Si este criterio falla –existe otro constructor con el mismo número de parámetros-, entonces aplica el criterio del tipo de cada uno de los parámetros. Veamos el ejercicio siguiente para tratar de comprender este concepto de constructores parametrizados.

Ejercicio. Modifica la aplicación Windows C# vista en esta sección de manera que declares 2 objetos mas `oAlu3` y `oAlu4`. Estos 2 objetos se inicializan con constructores parametrizados según se establece a continuación :

- **El constructor parametrizado que utiliza el objeto `oAlu3` recibe las 3 calificaciones del alumno y en su cuerpo hace el número de control igual a "08130007" y al nombre le asigna el valor de "TERENCIO ROSAS".**
- **El constructor parametrizado que utiliza el objeto `oAlu4` recibe el número de control y el nombre del alumno. Las calificaciones las hace igual a 90 cada una.**

El diagrama de casos de uso es muy simple, sólo tiene la tarea de la visualización de los 2 objetos. Tal vez podríamos con el fin de ejemplificar a fondo el problema, añadir las tareas de la inicialización hecha por los constructores, además de la definición de cada uno de los objetos. No lo haremos así, sólo estableceremos el caso de uso : visualización de los alumnos `oAlu3` y `oAlu4`.



Requerimos agergar a la interfase gráfica los componentes necesarios para visualizar los atributos de los alumnos `oAlu3` y `oAlu4`. Con 2 botones `button2` y `button3` haremos la visualización y usaremos los mismos componentes Label para contener a los atributos tanto de `oAlu1` y de `oAlu2`. La figura #5.3.13 muestra los componentes añadidos a la aplicación.

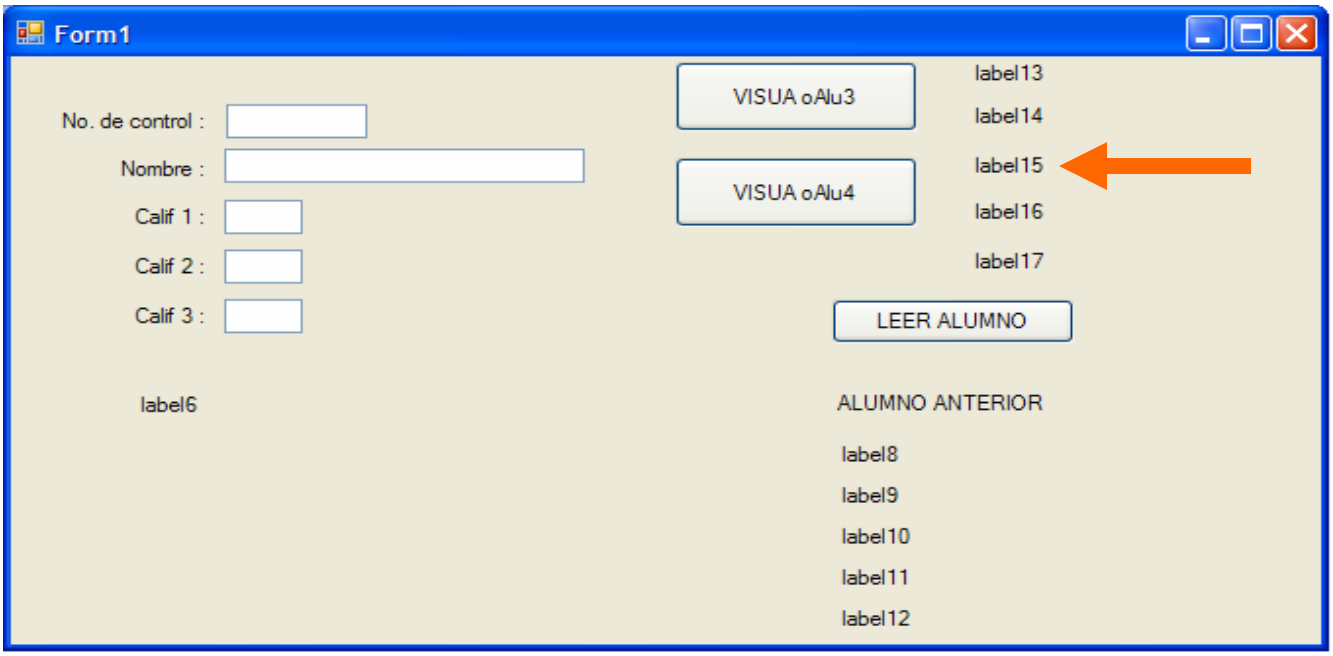


Fig. No. 5.3.13 Nueva interfase gráfica para visualización de los alumnos `oAlu3` y `oAlu4`.

Ahora vamos a definir a los 2 objetos `oAlu3` y `oAlu4` dentro de la clase `Form1`. Agreguemos el código que a continuación se detalla en dicha clase :

```
public partial class Form1 : Form
{
    Alumno oAlu = new Alumno();
    Alumno oAlu3 = new Alumno(85,100,90);
    Alumno oAlu4 = new Alumno("08130234","Venancio Padilla");
    public Form1()
    {
       InitializeComponent();
    }
...
...
```

Observemos que el objeto `oAlu3` hace uso del constructor que recibe 3 parámetros que constituyen a las 3 calificaciones del alumno. El objeto `oAlu4` usa otro constructor que recibe 2 parámetros tipo string que corresponden al número de control y al nombre del alumno. Los 2 nuevos constructores que debemos agregar a la clase no causan error debido a que el primer criterio se cumple : el número de los parámetros es diferente. También notemos que el constructor por defecto tiene 0 parámetros por lo que también cumple el primer criterio al compararlo con los 2 nuevos constructores utilizados para los objetos `oAlu3` y `oAlu4`.
Lo siguiente es añadir los 2 constructores a la clase `Alumno` que sumados al que ya teníamos –por defecto sin parámetros-, la clase `Alumno` tendrá 3 constructores. Hagamos click en la pestaña de la clase `Alumno` y tecleemos el código de los 2 constructores después de la definición de los atributos de la clase. Agrega los 2 nuevos constructores según se ve en la figura #5.3.14.

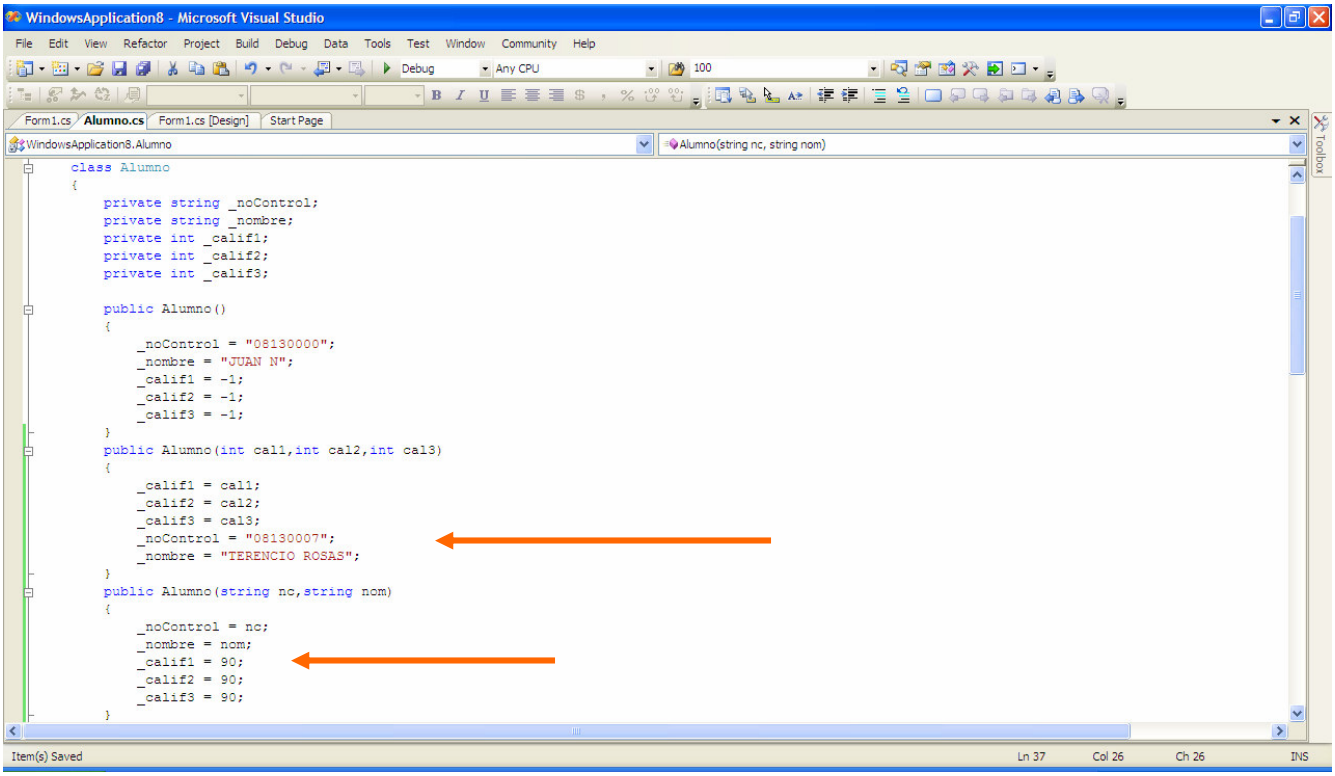


Fig. No. 5.3.14 Inclusión de los 2 nuevos constructores parametrizados en la clase `Alumno`.

El primer constructor recibe las 3 calificaciones como parámetros y en su cuerpo asigna los valores “08130007” y “TERENCIO ROSAS” a los atributos `_noControl` y `_nombre` respectivamente. Mientras que el segundo constructor recibe 2 parámetros correspondientes a los valores que queremos asignar a los atributos `_noControl` y `_nombre`, además que dentro de su cuerpo asigna el valor de 90 a las 3 calificaciones, según lo establecido en el ejercicio.

Lo procedente ahora es agregar el código a los eventos Click de cada botón que sirve para visualizar a cada objeto : `oAlu3` y `oAlu4`.

Vamos a utilizar los métodos que ya hemos definido en la clase Alumno : `RetNoControl()`, `RetNombre() y RetCalif()`. Recordemos que el último método `RetCalif()` recibe 3 parámteros entreos para depositar las calificaciones del alumno, de manera que se usan como parámetros de salida.

```
private void button2_Click(object sender, EventArgs e)
{
    label13.Text = oAlu3.RetNoControl();
    label14.Text = oAlu3.RetNombre();
    int cal1 = 0, cal2 = 0, cal3 = 0;
    oAlu3.RetCalif(ref cal1, ref cal2, ref cal3);
    label15.Text = Convert.ToString(cal1);
    label16.Text = Convert.ToString(cal2);
    label17.Text = Convert.ToString(cal3);
}

private void button3_Click(object sender, EventArgs e)
{
    label13.Text = oAlu4.RetNoControl();
    label14.Text = oAlu4.RetNombre();
    int cal1 = 0, cal2 = 0, cal3 = 0;
    oAlu4.RetCalif(ref cal1, ref cal2, ref cal3);
    label15.Text = Convert.ToString(cal1);
    label16.Text = Convert.ToString(cal2);
    label17.Text = Convert.ToString(cal3);
}
```

Ejecutemos la aplicación y hagamos click en los botones para visualizar a los atributos de los objetos `oAlu3` y `oAlu4` de manera que observemos que todo ha sido tecleado correctamente, figura #5.3.15.

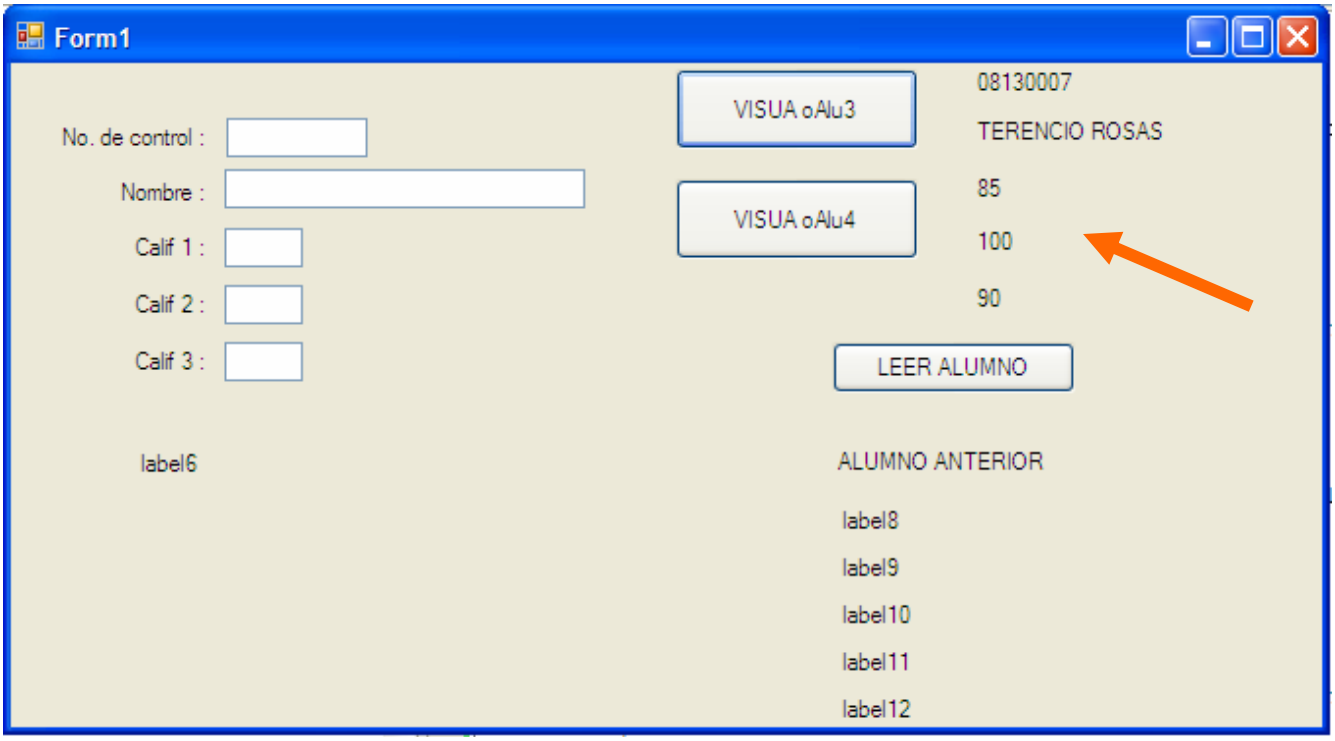
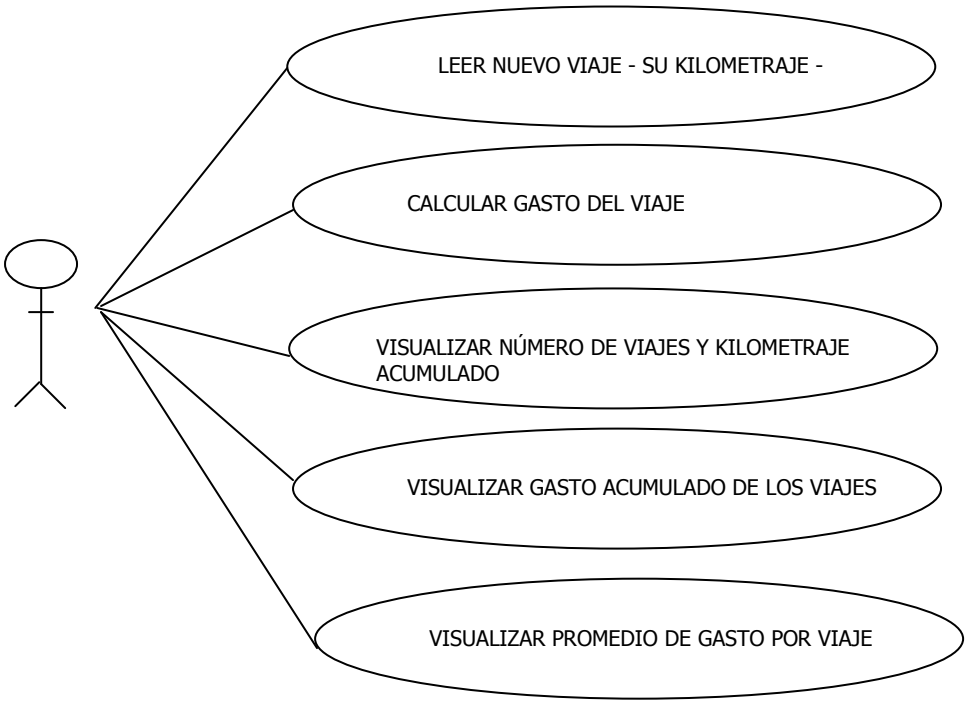


Fig. No. 5.3.15 Visualización de los atributos del objeto `oAlu3`.

Otro ejemplo :

Un señor apodado el viajero, le interesa saber cuantos viajes ha hecho y cuanto se ha gastado en ellos. Quiere que su programa lea el kilometraje de cada viaje, ya que por su experiencia sabe que por cada kilometro gasta $5.00. El programa debe visualizar cuantos viajes ha hecho, lo gastado en todos los viajes (en total) y ademas el promedio que gasta por viaje.

Iniciemos con el diagrama de casos de uso de acuerdo a las tareas que identifiquemos para solucionar el problema. El objeto fundamental es el viajero, así que definamos una clase `Viajero`. Suis atributos serían el número de viajes `_noViajes`, el kilometraje acumulado `_kilometrajeAcum` de los viajes hechos. De las tareas podemos identificar leer un nuevo viaje, su kilometraje, calcular lo gastado en el viaje, visualizar el número de viajes, visualizar el gasto acumulado de los viajes y visualizar el promedio de gasto por viaje.



El diagrama de clase para el problema contiene sólo una clase : `Viajero`.

VIAJERO
- _noViajes : int - _kilometrajeAcum : int
+ Leer(int):void + RetNoViajes():int + RetKilometrajeAcum():int

La tarea CALCULAR GASTO DEL VIAJE seguramente no la vamos a efectuar ya que la tarea VISUALIZAR GASTO ACUMULADO DE LOS VIAJES se puede obtener fácilmente del producto del kilometraje acumulado por 5.0 que es el costo por kilómetro.

Hagamos la interfase gráfica incluyendo uno a uno los casos de uso ilustrados en el diagrama, de manera que expliquemos el porqué de los métodos que hemos añadido a la clase `Viajero`.

Métodos en la clase `Viajero` :

- `Leer()` recibe el parámetro número de kilómetros del nuevo viaje. Incrementará también al número de viajes `_noViajes`, además del kilometraje acumulado `_kilometrajeAcum`.
- `RetNoViajes()` retorna el número de viajes `_noViajes` que el viajero a hecho. Se utiliza para el caso de uso VISUALIZAR NUMERO DE VIAJES y también en el caso de uso VISUALIZAR PROMEDIO DE GASTO POR VIAJE.
- `RetKilometrajeAcum()` retorna el número de kilómetros de todos los viajes realizados por el viajero. Es utilizado para los casos de uso VISUALIZAR GASTO ACUMULADO DE LOS VIAJES y VISUALIZAR PROMEDIO DE GASTO POR VIAJE.

Vamos a crear un nuevo proyecto y en la forma agreguemos los componentes Label, TextBox y Button que nos servirán para implementar al primer caso de uso LEER NUEVO VIAJE - SU KILOMETRAJE -, los cuales se muestran en la figura #5.3.16.

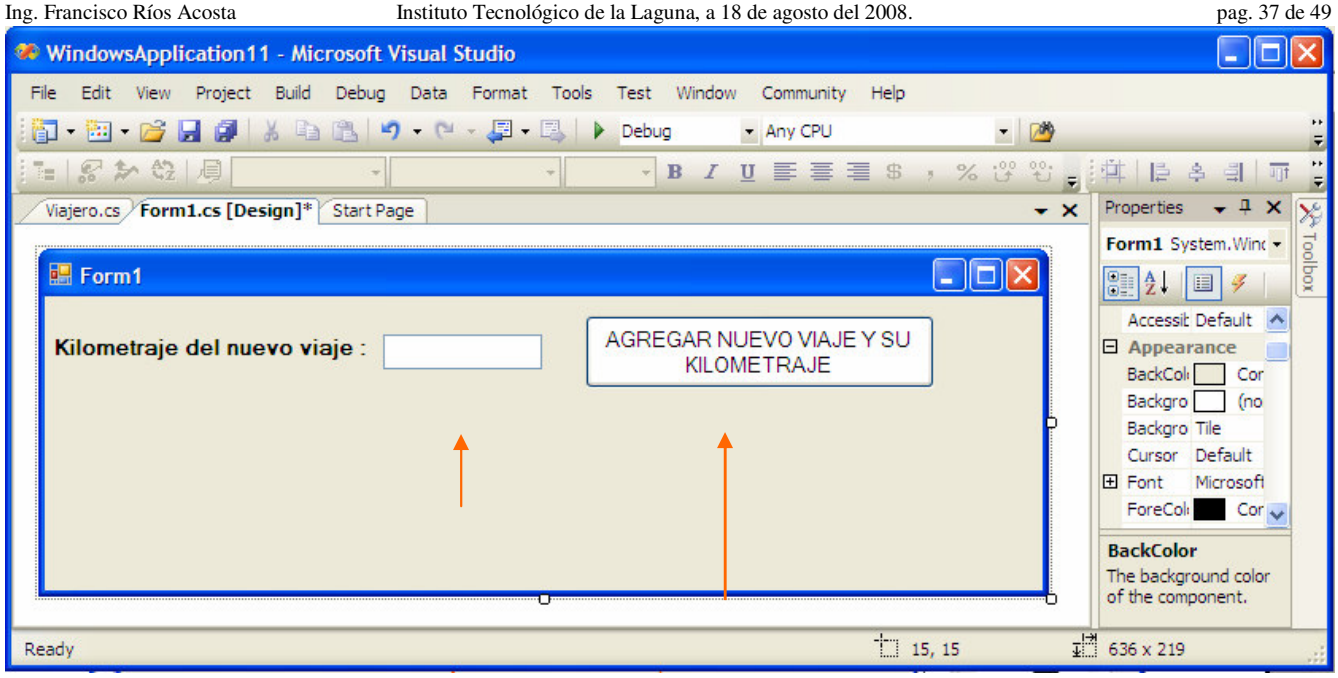


Fig. No. 5.3.16 Interfase gráfica para realizar la lectura del nuevo viaje y su kilometraje.

Ahora agregamos la clase `Viajero` con sus atributos antes descritos en el diagrama de clases y su constructor. El constructor deberá asignar el valor de 0 tanto al número de viajes `_noViajes` como al kilometraje acumulado `_kilometrajeAcum`, por razones obvias : inicialmente no se ha hecho ningún viaje y por consecuencia no se ha recorrido kilometraje alguno. Así que selecciona la opción del menú Project | Add Class y en el cuadro de diálogo que se presenta teclea en nombre de clase `Viajero.cs`. Teclea en el cuerpo de la clase el código visto en la figura 5.3.17.

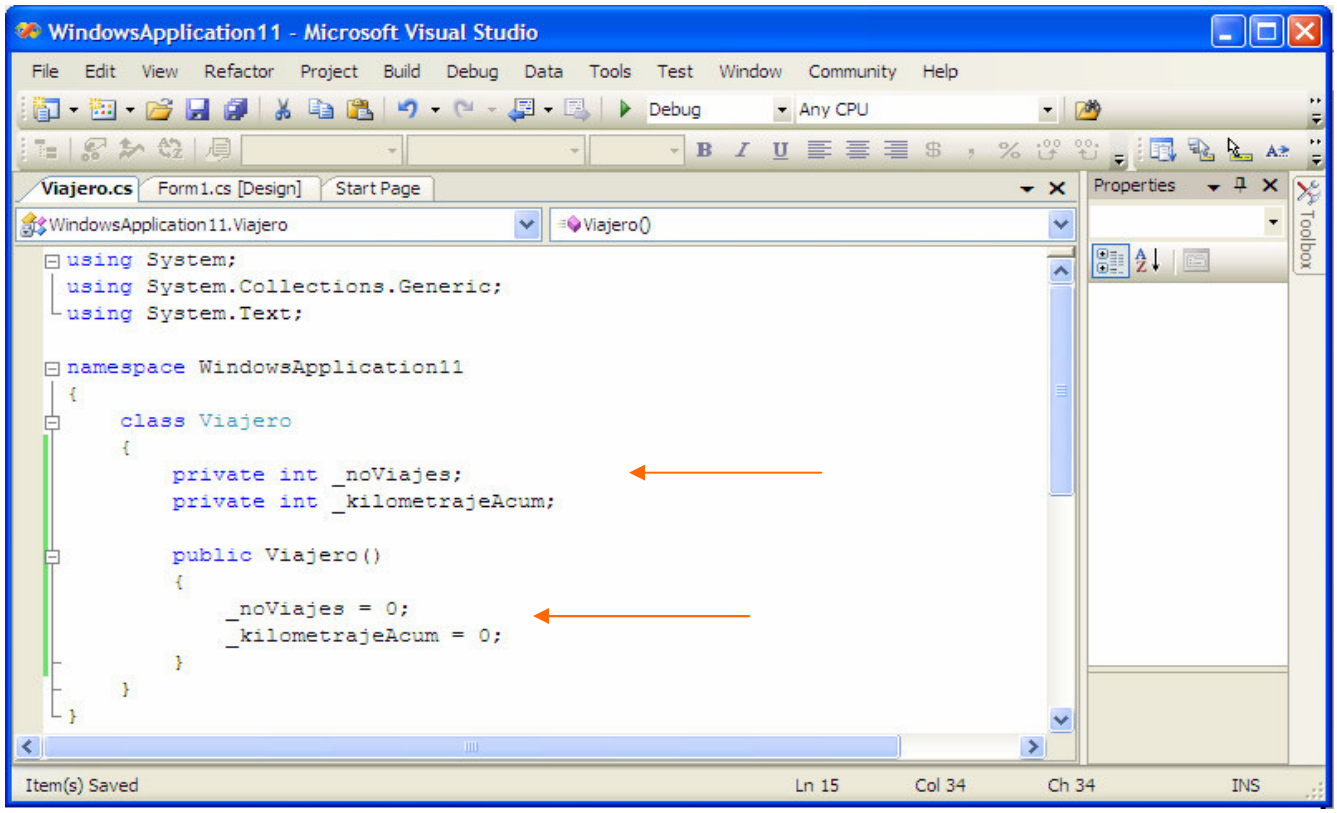


Fig. No. 5.3.17 Atributos y constructor por defecto de la clase `Viajero`.

Con la clase `Viajero` añadida a nuestro proyecto podemos definir al objeto `oTraveler`, el cual se agregará en la forma `Form1` que reside en el archivo Form1.cs. Ver figura #5.3.18.

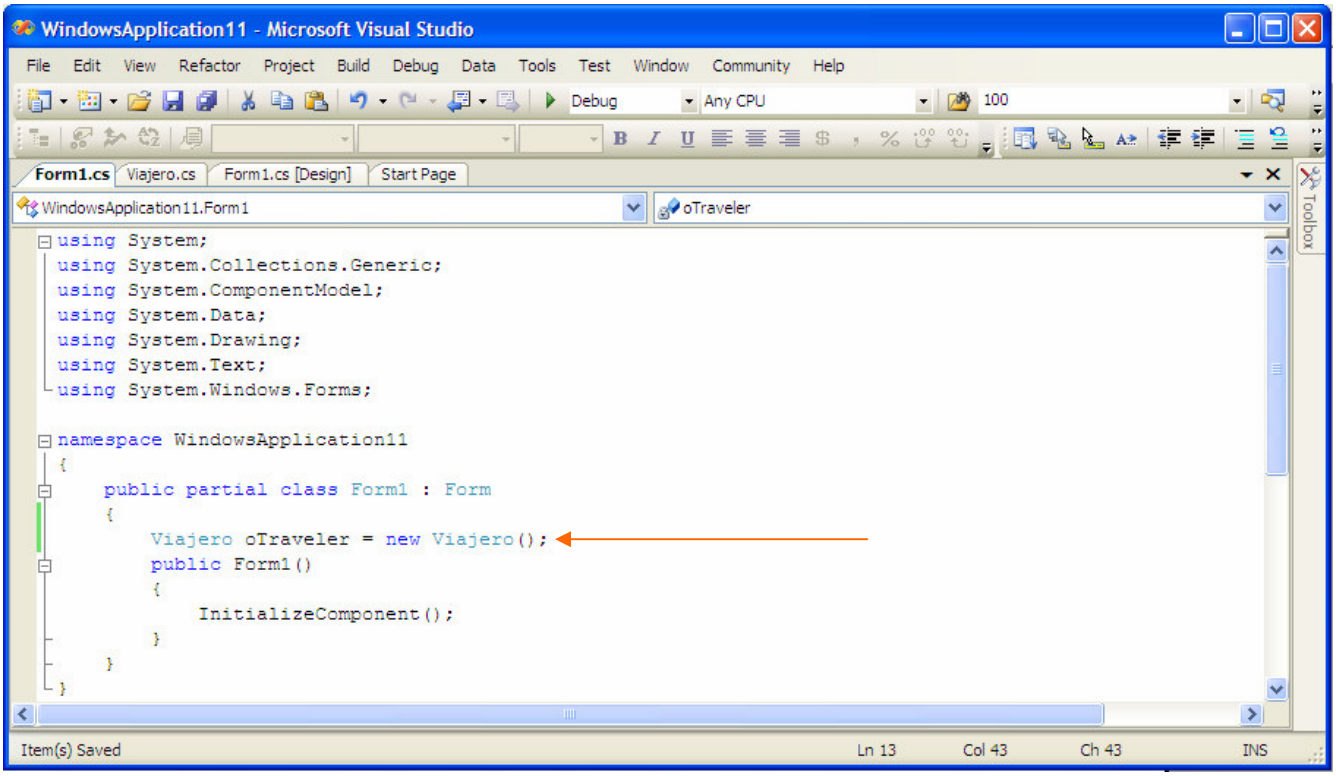


Fig. No. 5.3.18 Definición del objeto `oTraveler`.

Recordemos que cuando es ejecutada la aplicación, el C# hace la llamada al constructor por defecto de la clase `Viajero` cuando encuentra la definición del objeto `oTraveler`. La sentencia `new Traveler()` es la que llama al constructor por defecto de la clase `Viajero`. Esta llamada al constructor nos da certidumbre de que los atributos `_noViajes` y `_kilometrajeAcum` tienen un valor de 0 al inicio de nuestra aplicación.

La interfase gráfica la deberemos completar un poco mas, agregandole los componentes `label2` y `label3` que servirán para visualizar los atributos `_noViajes` y `_kilometrajeAcum`, figura #5.3.19.

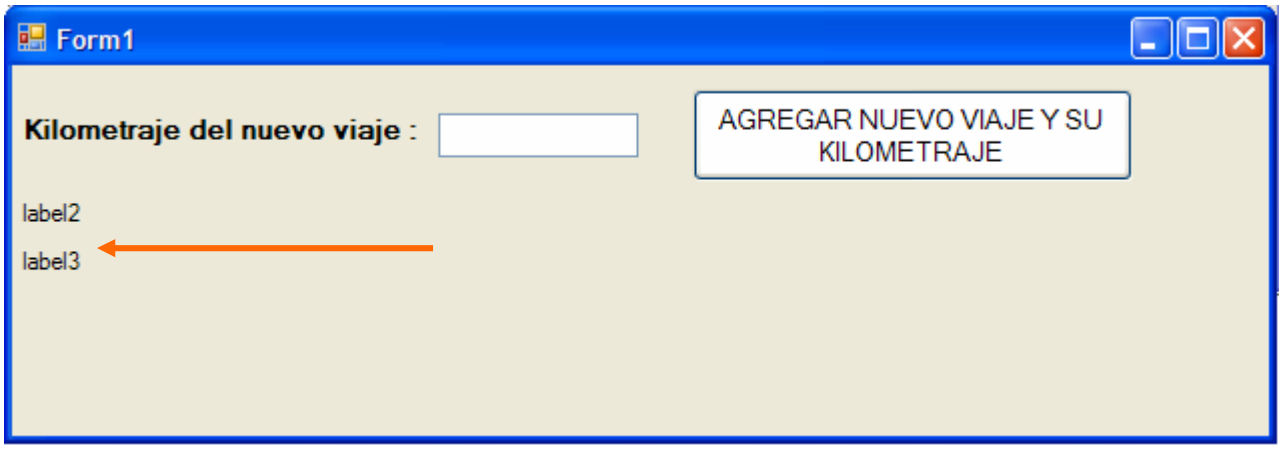


Fig. No. 5.3.19 Componentes `label2` y `label3` para visualizar el número de viajes y el kilometraje acumulado.

Escribamos ahora el código para implementar el caso de uso LEER NUEVO VIAJE - SU KILOMETRAJE - en donde haremos uso del mensaje `oTraveler.Leer()` con un parámetro de entrada que será el valor tecleado en el componente `textBox1`.

Vayamos al evento Click del botón `button1` y tecleemos el código que a continuación se detalla en la figura 5.3.20.

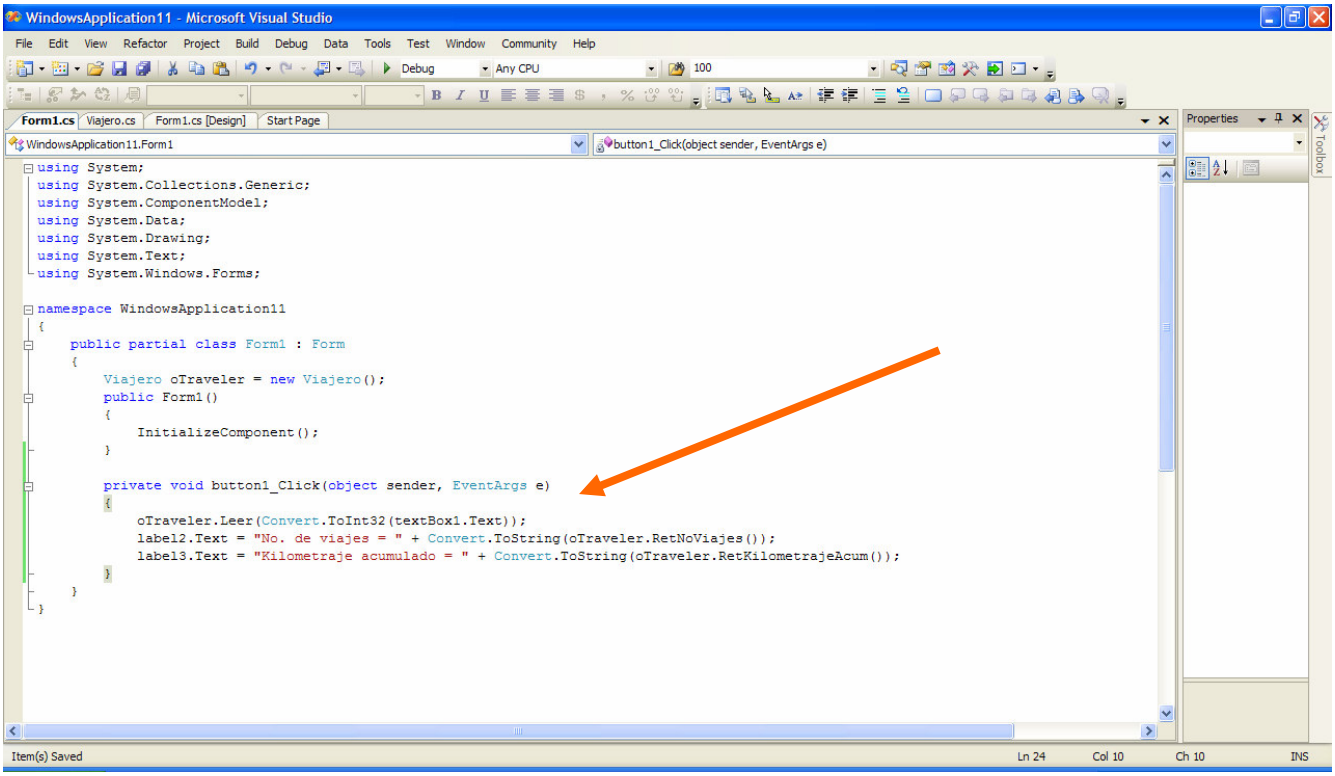


Fig. No. 5.3.20 Código en el evento Click del botón `button1`.

Inicialmente escribimos el mensaje para leer el kilometraje del nuevo viaje, donde hemos convertido la propiedad Text del componente `textBox1` a tipo **int**. Esto lo realizamos debido a que el atributo `_kilometrajeAcum` del objeto `oTraveler` es de tipo **int** por supuesto.

```
oTraveler.Leer(Convert.ToInt32(textBox1.Text));
```

El método `Leer()` incrementa al número de viajes y suma el nuevo kilometraje leido al kilometraje acumulado del objeto `oTraveler`. Desde luego que debemos actualizar esta información en nuestra interfase gráfica, así que por eso hemos escrito los 2 mensajes que se encargan de realizar dicha actualización visual.

```
label2.Text = "No. de viajes = " + Convert.ToString(oTraveler.RetNoViajes());

label3.Text = "Kilometraje acumulado = " + Convert.ToString(oTraveler.RetKilometrajeAcum());
```

La primera sentencia visualiza en el componente `label2` el valor del número de viajes retornado por el mensaje `oTraveler.RetNoViajes()`, donde se llama al método que retorna al atributo `_noViajes`. El mensaje retorna un entero que es necesario convertir a tipo **string** para que pueda ser concatenado a la constante literal "No. de viajes = ".

La segunda sentencia involucra al mensaje `oTraveler.RetKilometrajeAcum()` usado para visualizar al número de kilómetros que el viajero ha efectuado. El método retorna un tipo **int** por lo que debemos convertirlo a tipo **string** igual que lo hicimos para el número de viajes.

Como ya lo deberíamos de suponer, tenemos que ir a la clase `Viajero` y definir los métodos antes utilizados : `Leer()`, `RetNoViajes()` y `RetKilometrajeAcum()`.

```
public void Leer(int noKilom)
{
     _noViajes++;
     _kilometrajeAcum += noKilom;
}
```

Usamos el operador de incremento ++ para incrementar el valor del atributo `_noViajes` en una unidad.

El operador de asignación += acorta la escritura de la sentencia : `_kilometrajeAcum = _kilometrajeAcum + noKilom;`

Los métodos `RetNoViajes()` y `RetKilometrajeAcum()` son simples, sólo retornan al atributo correspondiente utilizando la sentencia **return**. En la figura #5.3.21 tenemos a los 3 métodos añadidos a la clase `Viajero`.

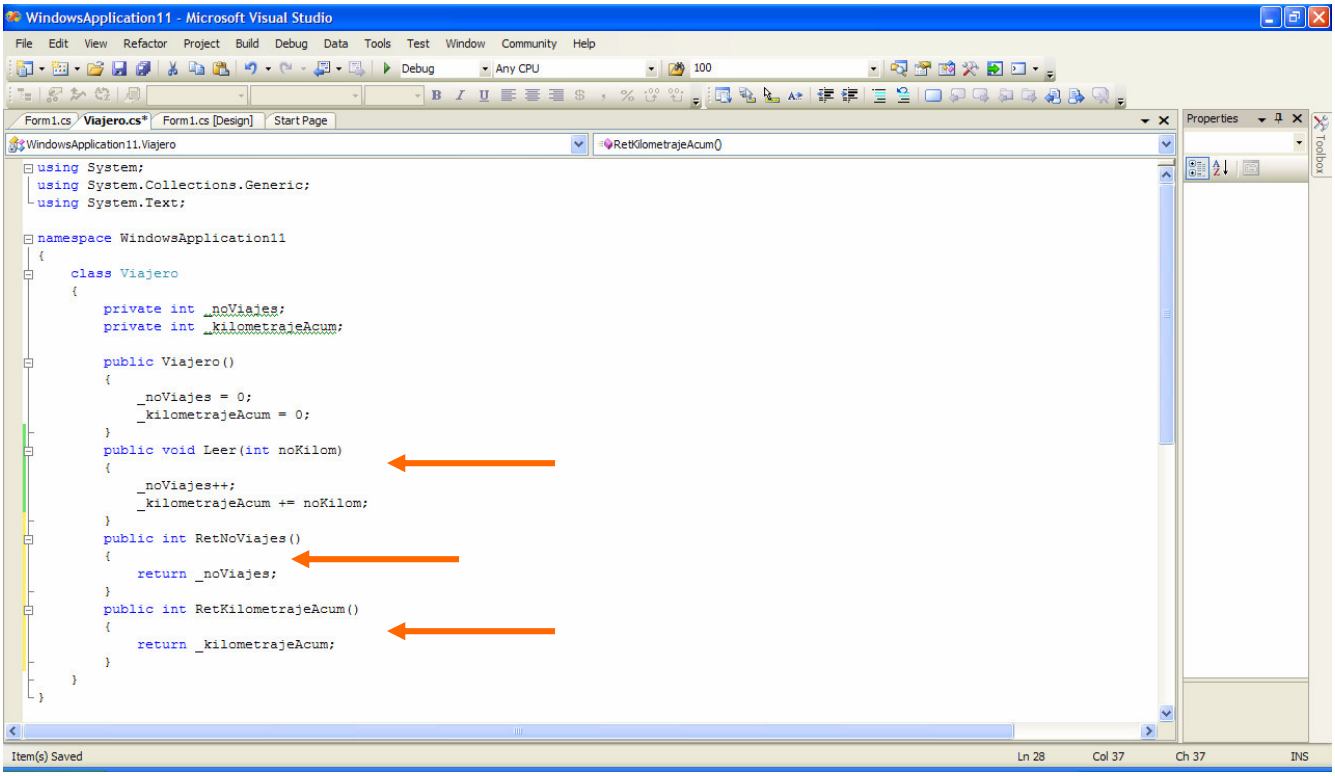


Fig. No. 5.3.21 Métodos `Leer()`, `RetNoViajes()` y `RetKilometrajeAcum()` en la clase `Viajero`.

Si ejecutamos la aplicación veremos que no tenemos errores de sintaxis pero si uno de lógica. Los componentes `label2` y `label3` no exhibirán el valor del número de viajes y del kilometraje acumulado en 0 como deberían hacerlo, ver figura #5.3.22.

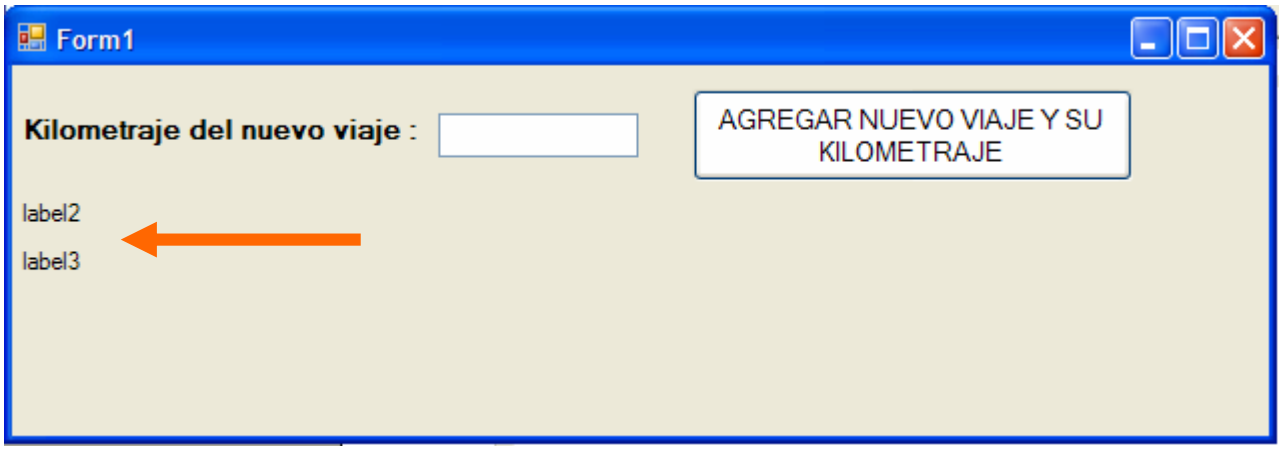


Fig. No. 5.3.22 Componentes `label2` y `label3` sin visualizar los valores de atributos `_noViajes` y `_kilometrajeAcum`.

Recordemos que los atributos `_noViajes` y `_kilometrajeAcum` son asignados inicialmente en el constructor por defecto que antes habíamos añadido a la clase `Viajero`, precisamente con ese fin –la inicialización de los 2 atributos-.

Una buena manera de efectuar la visualización es ingresar en el evento Load de la forma `Form1`, los mensajes al objeto `oTraveler` que incluyen las llamadas a los métodos `RetNoViajes()` y `RetKilometrajeAcum()` y cuyo son asignados a las propiedades Text de los componentes `label2` y `label3`, previamente convertidos a tipo **string**. Seleccionemos a la forma

`Form1` y hagamos un click sobre el botón de eventos en la ventana de propiedades. Luego busquemos el evento Load de la forma `Form1` y hagamos doble click en la caja señalada en la figura #5.3.23.

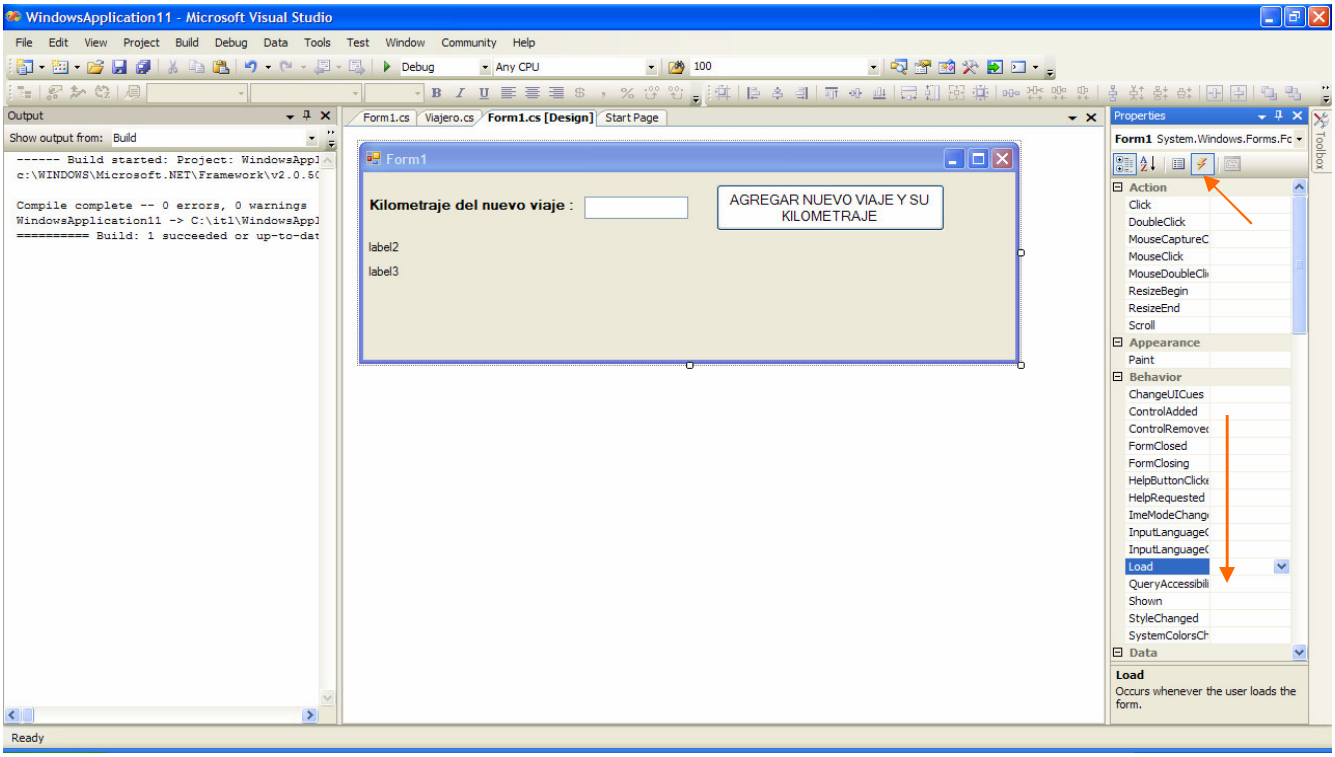


Fig. No. 5.3.23 Accediendo al evento Load de la forma `Form1`.

El evento Load de la forma `Form1` es lanzado solamente una vez durante la ejecución de la aplicación, precisamente al momento de que se inicia la ejecución de ella. Este evento es utilizado para efectuar inicializaciones en nuestra aplicación sobre componentes ingresados en el tiempo de diseño, o bien sobre objetos definidos por nosotros mismos. Introducimos el código entonces en el evento Load según se muestra a continuación.

```
private void Form1_Load(object sender, EventArgs e)
{
    label2.Text = "No. de viajes = " + Convert.ToString(oTraveler.RetNoViajes());
    label3.Text = "Kilometraje acumulado = " + Convert.ToString(oTraveler.RetKilometrajeAcum());
}
```

Notemos que las sentencias incluídas son las mismas que cuando leemos el nuevo viaje claro que sin la inclusión del mensaje donde se involucra la llamada al método `Leer()`. La figura #5.3.24 tiene la mejora en la ejecución de la aplicación.

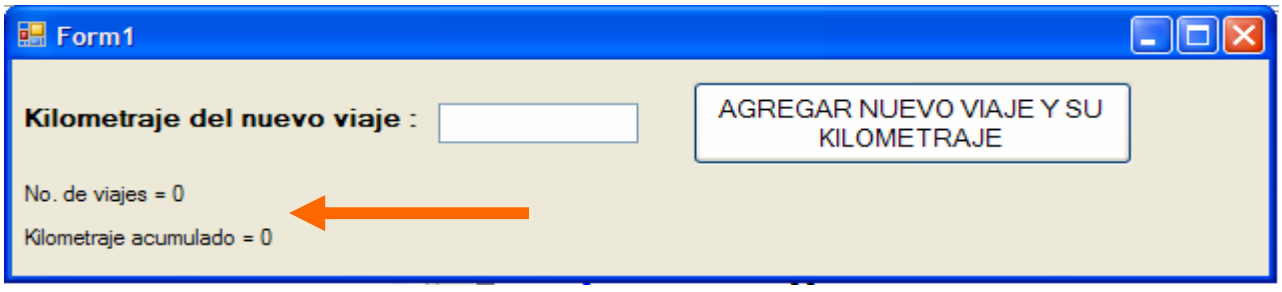


Fig. No. 5.3.24 Los componentes `label2` y `label3` muestran el valor de los atributos `_noViajes` y `_kilometrajeAcum` al inicio de la ejecución de la aplicación.

Bien, ahora probemos la ejecución dando valores para kilometrajes de viajes nuevos y veamos la respuesta de nuestro programa. La figura #5.3.25 muestra la interfase de la aplicación para valores de kilometrajes -100 y 200- ingresados.

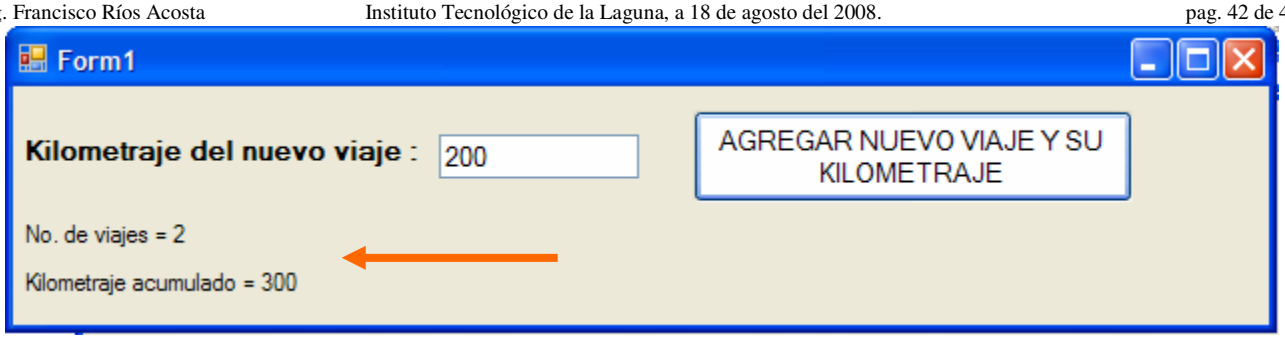


Fig. No. 5.3.25 Prueba de la aplicación para 2 nuevos viajes con 100 y 200 kilómetros registrados.

Seguimos con el caso de uso VISUALIZAR GASTO ACUMULADO DE LOS VIAJES al que implementamos de una manera simple. Añadimos un componente Label con la propiedad Name `label4` y en su propiedad Text le damos el valor del número de kilómetros recorridos multiplicado por $5.00 –costo por cada kilómetro-. Esta sentencia la vamos a incluir tanto en el evento Click donde efectuamos la lectura del nuevo viaje como en el evento Load de la forma `Form1`. La figura #5.3.26 muestra la interfase gráfica con el componente `label4` incluido.

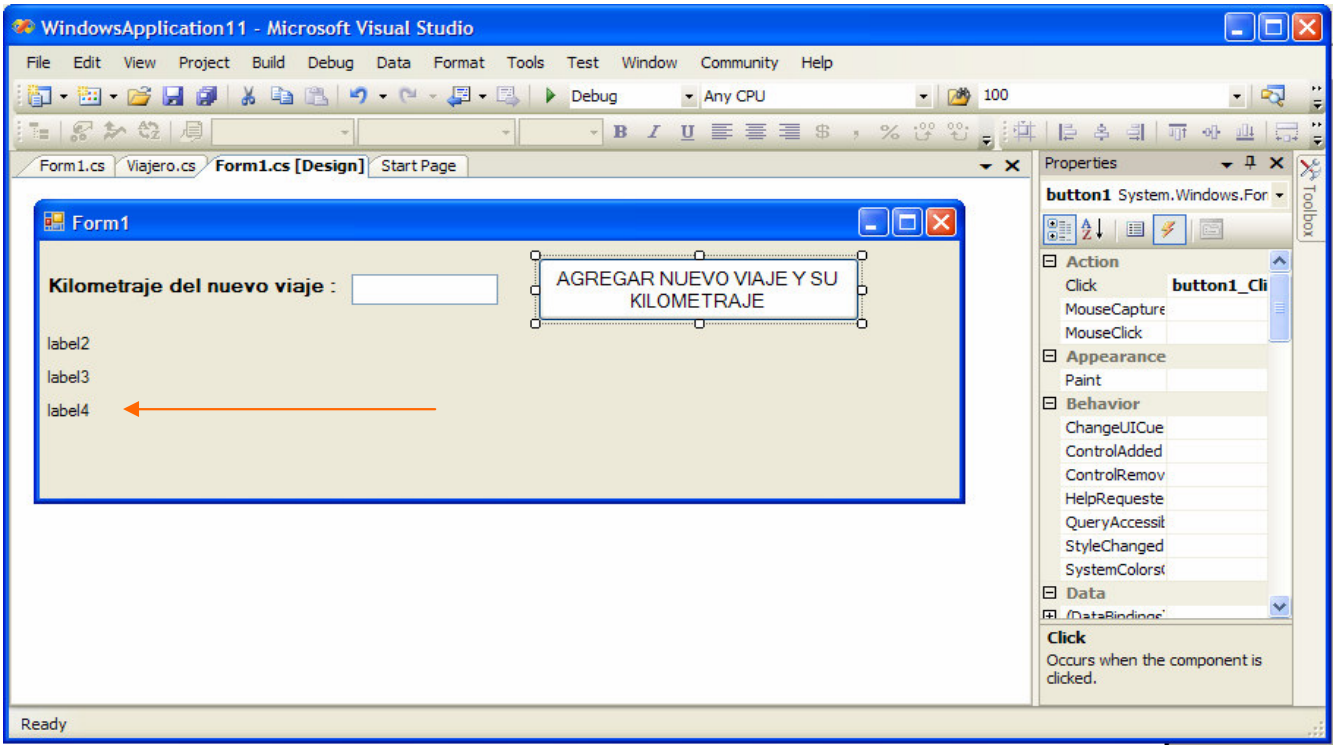


Fig. No. 5.3.26 Interfase de la aplicación con el `label4` incluido para visualizar el gasto total acumulado de los viajes.

El código de los eventos Click y Load con la modificación de la visualización del gasto acumulado en los viajes es :

```
private void button1_Click(object sender, EventArgs e)
{
    oTraveler.Leer(Convert.ToInt32(textBox1.Text));
    label2.Text = "No. de viajes = " + Convert.ToString(oTraveler.RetNoViajes());
    label3.Text = "Kilometraje acumulado = " + Convert.ToString(oTraveler.RetKilometrajeAcum());
    label4.Text="Gasto acumulado = "+Convert.ToString(oTraveler.RetKilometrajeAcum()*5);
}

private void Form1_Load(object sender, EventArgs e)
{
    label2.Text = "No. de viajes = " + Convert.ToString(oTraveler.RetNoViajes());
    label3.Text = "Kilometraje acumulado = " + Convert.ToString(oTraveler.RetKilometrajeAcum());
    label4.Text = "Gasto acumulado = " + Convert.ToString(oTraveler.RetKilometrajeAcum() * 5);
}
```

La ejecución de la aplicación para 2 nuevos viajes y kilometraje de 100 y 200, visualiza el gasto acumulado en $1500, según lo vemos en la figura #5.3.27.

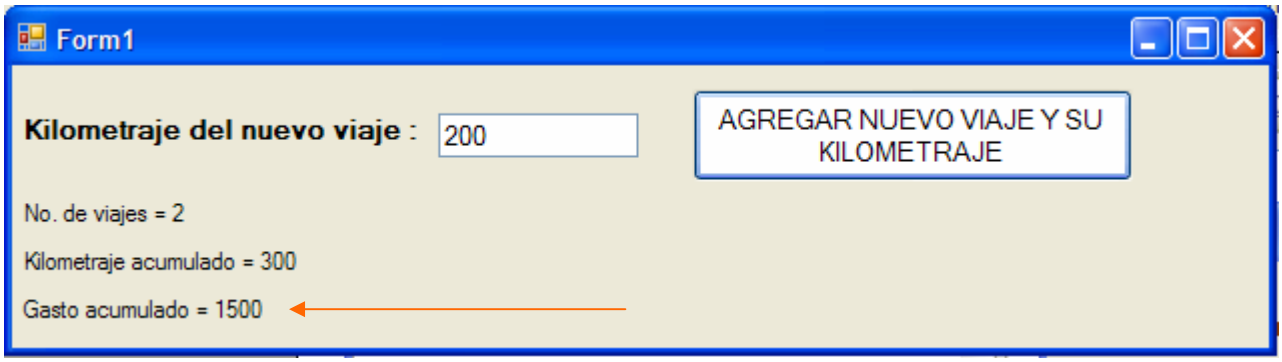


Fig. No. 5.3.27 Gasto acumulado para 2 nuevos viajes de 100 y 200 kilómetros.

Para terminar sólo resta implementar el caso de uso VISUALIZAR PROMEDIO DE GASTO POR VIAJE agregando un nuevo componente `label5` que se encargará de visualizar dicho promedio. Realmente es muy similar a lo que hicimos con el gasto acumulado así que una vez que añadimos el `label5`, modificaremos los eventos Click del `button1` y el Load de la forma `Form1`, de acuerdo a lo que se indica en el código siguiente :

```
private void button1_Click(object sender, EventArgs e)
{
    oTraveler.Leer(Convert.ToInt32(textBox1.Text));
    label2.Text = "No. de viajes = " + Convert.ToString(oTraveler.RetNoViajes());
    label3.Text = "Kilometraje acumulado = " + Convert.ToString(oTraveler.RetKilometrajeAcum());
    label4.Text="Gasto acumulado = "+Convert.ToString(oTraveler.RetKilometrajeAcum()*5);
    label5.Text = "Gasto promedio por viaje = " + Convert.ToString(oTraveler.RetKilometrajeAcum() * 5.0
                                                                   / oTraveler.RetNoViajes());
}

private void Form1_Load(object sender, EventArgs e)
{
    label2.Text = "No. de viajes = " + Convert.ToString(oTraveler.RetNoViajes());
    label3.Text = "Kilometraje acumulado = " + Convert.ToString(oTraveler.RetKilometrajeAcum());
    label4.Text = "Gasto acumulado = " + Convert.ToString(oTraveler.RetKilometrajeAcum() * 5);
    label5.Text = "Gasto promedio por viaje = " + Convert.ToString(oTraveler.RetKilometrajeAcum() * 5.0
                                                                   / oTraveler.RetNoViajes());
}
```

Notemos que hemos usado una constante literal **double** 5.0 para obtener el gasto acumulado en los viajes para luego dividir entre el número de viajes realizados. ¿Por qué lo hicimos así?, discútelo con tus compañeros y con tu profesor. La figura #5.3.28 muestra la ejecución de la aplicación para 2 viajes de 100 y 200 kilómetros.

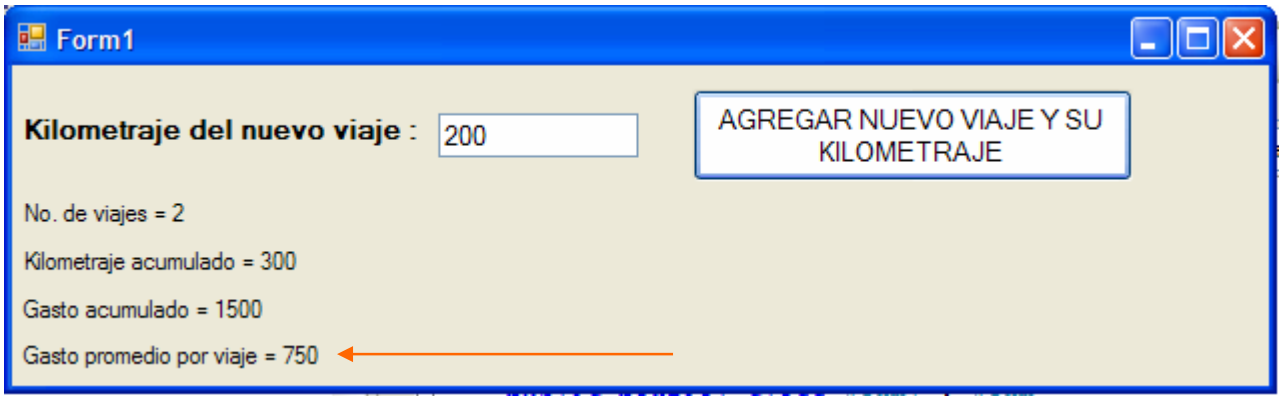


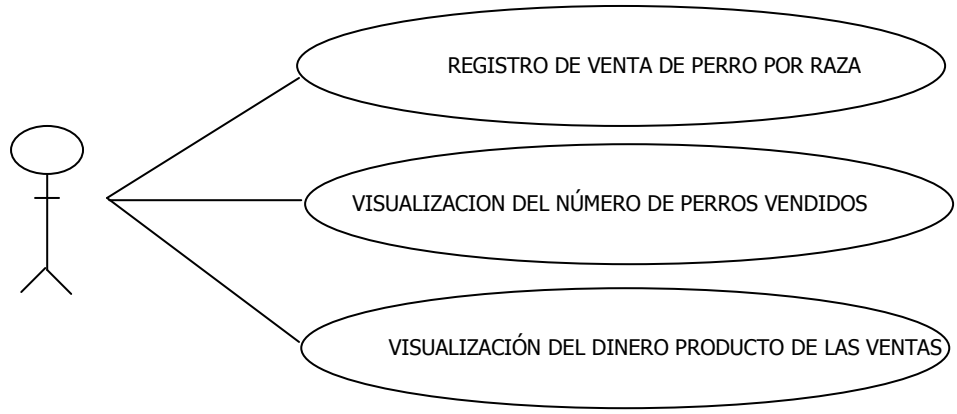
Fig. No. 5.3.28 Gasto promedio por viaje –caso de uso- agregado a la aplicación.

Con el código agregado para el cálculo del gasto promedio por viaje, hemos terminado nuestra aplicación.

EJERCICIO.

Un veterinario quiere registrar cuando vende sus perros de 2 tipos de raza : pastor y boxer. Quiere saber cuantos pastores y cuantos boxer ha vendido, además del dinero que ha recibido por la venta de los perros tanto pastor como boxer. Usa una sola clase Veterinario con los atributos que creas correctos. Escribe la aplicación Windows C# agregando los diagramas de casos de uso y de clase. Se sabe que el veterinario tiene un registro inicial de 1 perro boxer de 1500 su venta y de 2 perros pastor que los vendió en 3600 pesos.

El diagrama de casos de uso requiere de especificar las tareas : Registro de venta del perro, Visualización del número de perros por raza y el dinero resultado de la venta de perros por raza.



La clase `Veterinario` constituye la única clase en nuestra aplicación. Podemos abstraer al perro como un objeto pero realmente sólo nos interesa el número de los perros vendidos por raza y el producto de su venta. Si nos dejarámos llevar por dicha abstracción de la clase `Perro`, deberíamos tener **n** objetos del tipo `Perro` siendo n el número de perros vendidos por el veterinario. De esta manera la abstracción de los objetos `Perro` es inadecuada para solucionar este problema.

VETERINARIO
- _noPerrosBoxer : int - _noPerrosPastor : int - _dineroVentasBoxer : int - _dineroVentasPastor : int
+ RegistroVentaBoxer(int, int) : void + RegistroVentaPastor(int, int) : void + RetNoPerrosBoxer():int + RetNoPerrosPator():int + RetDineroVentasBoxer() : int + RetDineroVentasPastor() : int

Los atributos de la clase `Veterinario` mantienen el número de perros vendidos por cada raza, además de mantener el dinero resultado de las ventas de perros por raza.

Los métodos utilizados para el registro de ventas de perros por raza requieren de 2 parámetros de entrada de tipo **int** : el número de perros vendidos en la transacción que se registra y el precio de venta al que se efectuó la transacción. Los otros 4 métodos son utilizados para retornar al atributo correspondiente.

Los casos de uso los implementaremos con un sólo botón tal y como lo hicimos en el ejercicio anterior. Usaremos componentes Label para visualizar los atributos del objeto de la clase `Veterinario` necesarios para cumplir a todos los casos de uso.

Creamos un nuevo proyecto en VisualStudio C# y en el diseñador de la forma Form1[Designer] empezamos por añadir 6 componentes Label, 2 componentes TextBox y 2 componentes Button y modificamos sus propiedades según se indica en la figura #5.3.29.

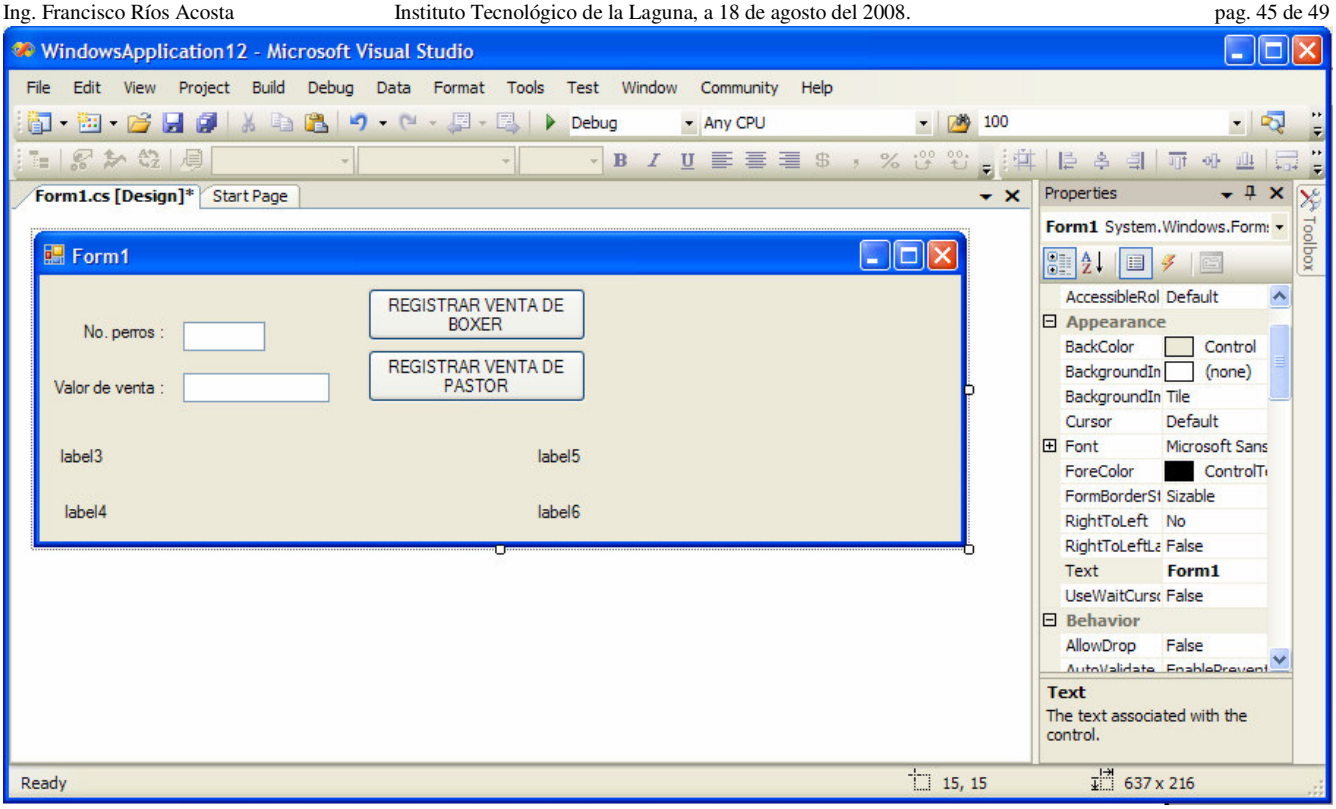


Fig. No. 5.3.29 Interfase gráfica propuesta para el problema de los perros.

Enseguida agregamos la definición del objeto `oMedicoVet` cuyo constructor lo vamos a parametrizar con los valores del número de perros vendidos tanto boxer como pastor y del producto de su venta. La definición del objeto `oMedicoVet` la agregamos en el archivo Form1.cs dentro de la clase `Form1`.

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Text;
using System.Windows.Forms;

namespace WindowsApplication12
{
    public partial class Form1 : Form
    {
        Veterinario oMedicoVet = new Veterinario(1,1500,2,3600);
        public Form1()
        {
            InitializeComponent();
        }
    }
}
```

Constructor con 4 parámetros : número de perros boxer vendidos, valor de venta de los boxer, número de perros pastor vendidos, valor de venta de los perros pastor.

La clase `Veterinario` la añadimos usando la opción del menú Project | Add Class, y en su cuerpo debemos teclear los atributos y el constructor parametrizado según el siguiente código :

```
class Veterinario
{
     private int _noPerrosBoxer;
     private int _noPerrosPastor;
     private int _dineroVentasBoxers;
     private int _dineroVentasPastor;

     public Veterinario(int noBoxer,int ventaBoxer,int noPastor,int ventaPastor)
     {
```

```
            _noPerrosBoxer = noBoxer;
            _dineroVentasBoxers = ventaBoxer;
            _noPerrosPastor = noPastor;
            _dineroVentasPastor = ventaPastor;
        }
    }
```

Ejecutemos la aplicación sólo para comprobar que no hayamos incurrido en errores de dedo. Una vez que veamos por la ausencia de errores seguimos con el evento Click del botón `button1` para el registro de venta de los perros boxer. La figura #5.3.30 contiene el código para el caso de uso del registro de venta de perros –boxer-.

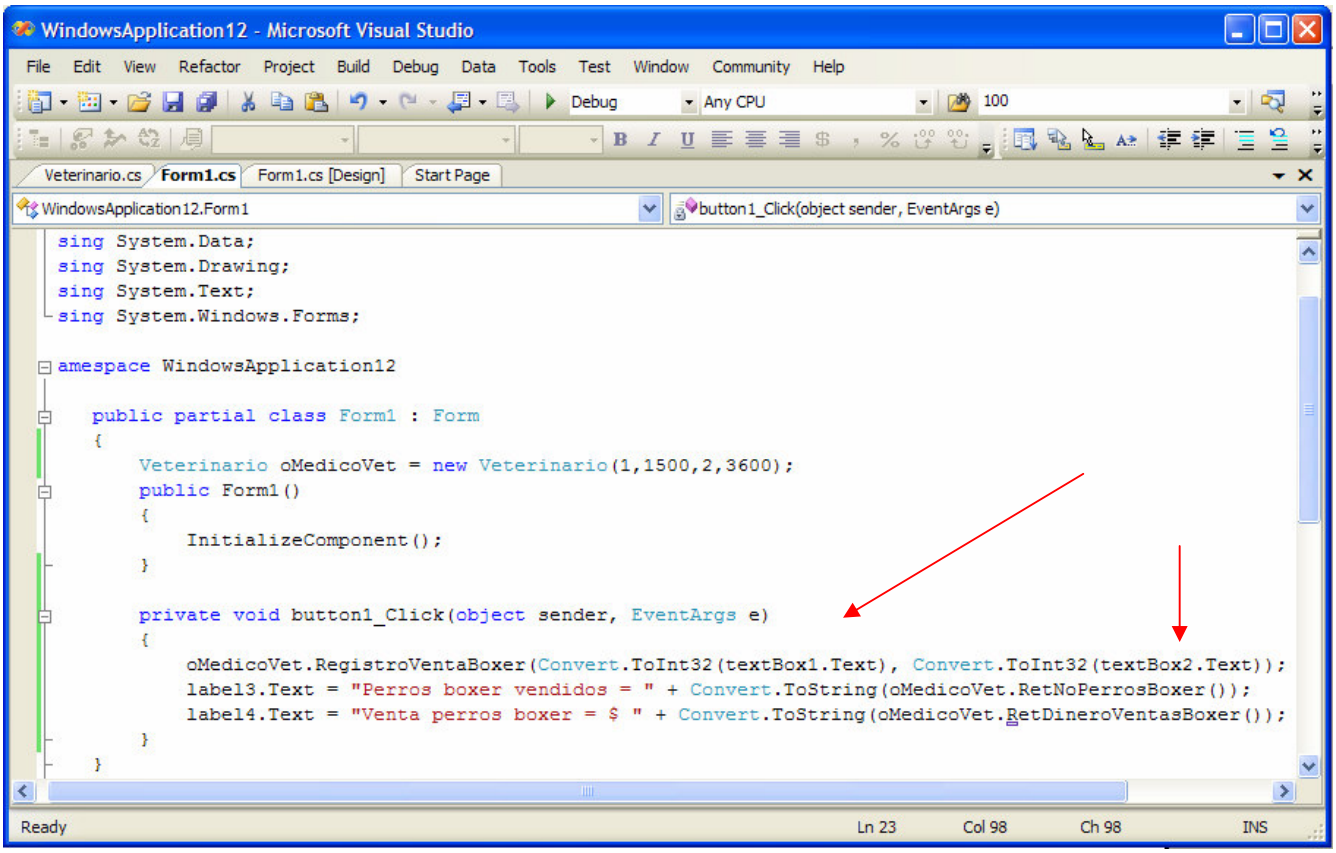


Fig. No. 5.3.30 Código para el evento Click del `button1`.

El método incluido en el mensaje al objeto `oMedicoVet` usado para el registro de la venta de perros boxer, tiene 2 parámetros : el número de perros boxer vendidos en la transacción y el valor de la venta. Los valores tecleados en los componentes `textBox1` y `textBox2` los convertimos a enteros usando la clase `Convert`.

```
oMedicoVet.RegistroVentaBoxer(Convert.ToInt32(textBox1.Text), Convert.ToInt32(textBox2.Text));
```

La definición de este método debemos de ingresarla a la clase `Veterinario` según el código siguiente :

```
public void RegistroVentaBoxer(int noBoxer, int ventaBoxer)
{
    _noPerrosBoxer += noBoxer;
    _dineroVentasBoxer += ventaBoxer;
}
```

Los valores pasados como parámetros a este método son sumados y asignados a los atributos `_noPerrosBoxer` y `_dineroVentasBoxer` usando el operador de asignación +=.

Los nuevos valores de los atributos `_noPerrosBoxer` y `_dineroVentasBoxer` son visualizados en los componentes `label3` y `label4` utilizando mensajes al objeto `oMedicoVet` que llaman a los métodos `RetNoPerrosBoxer()` y `RetDineroVentasBoxer()`.

```
label3.Text = "Perros boxer vendidos = " + Convert.ToString(oMedicoVet.RetNoPerrosBoxer());

label4.Text = "Venta perros boxer = $ " + Convert.ToString(oMedicoVet.RetDineroVentasBoxer());
```

Es necesario usar la clase `Convert` para realizar la conversión de tipo **int** a tipo **string** y poder efectuar la concatenación de manera correcta. Los métodos `RetNoPerrosBoxer()` y `RetDineroVentasBoxer()` se muestran a continuación :

```
public int RetNoPerrosBoxer()
{
    return _noPerrosBoxer;
}
public int RetDineroVentasBoxer()
{
    return _dineroVentasBoxers;
}
```

Hagámos lo mismo para el evento Click del botón `button2` utilizado para registrar la venta de perros pastor. El código del evento Click es el siguiente :

```
private void button2_Click(object sender, EventArgs e)
{
    oMedicoVet.RegistroVentaPastor(Convert.ToInt32(textBox1.Text), Convert.ToInt32(textBox2.Text));
    label5.Text = "Perros pastor vendidos = " + Convert.ToString(oMedicoVet.RetNoPerrosPastor());
    label6.Text = "Venta perros pastor = $ " + Convert.ToString(oMedicoVet.RetDineroVentasPastor());
}
```

Observemos ahora el uso del método `RegistroVentaPastor()` para registrar la venta de los perros pastor, además de los métodos `RetNoPerrosPastor()` y `RetDineroVentasPastor()`. Los 3 métodos debemos definirlos en la clase `Veterinario` ya que son diferentes a los 3 métodos utilizados para los perros boxer.

```
public void RegistroVentaPastor(int noPastor, int ventaPastor)
{
    _noPerrosPastor += noPastor;
    _dineroVentasPastor += ventaPastor;
}
public int RetNoPerrosPastor()
{
    return _noPerrosPastor;
}
public int RetDineroVentasPastor()
{
    return _dineroVentasPastor;
}
```

Si ejecutamos la aplicación observamos que los valores de inicio asignados dentro del constructor no son visualizados según lo mostramos en la figura #5.3.31.

Fig. No. 5.3.31 Valores iniciales en los atributos no han sido visualizados.

Resolvamos este asunto tal y como lo hicimos con el ejercicio anterior. Vayamos al evento Load de la forma `Form1` e insertemos el código que visualiza los valores iniciales de los atributos en los componentes `label3`, `label4`, `label5`, `label6` según lo indicamos a continuación :

```
private void Form1_Load(object sender, EventArgs e)
{
    label3.Text = "Perros boxer vendidos = " + Convert.ToString(oMedicoVet.RetNoPerrosBoxer());
    label4.Text = "Venta perros boxer = $ " + Convert.ToString(oMedicoVet.RetDineroVentasBoxer());
    label5.Text = "Perros pastor vendidos = " + Convert.ToString(oMedicoVet.RetNoPerrosPastor());
    label6.Text = "Venta perros pastor = $ " + Convert.ToString(oMedicoVet.RetDineroVentasPastor());
}
```

Ahora si ejecutamos tenemos resuelto el problema –ver figura #5.3.32. Ingresa valores a tu interfase de manera que observes si el funcionamiento es correcto.

Fig. No. 5.3.32 Valores iniciales visualizados generados por el constructor en la clase `Veterinario`.

5.3.5 El destructor.

A diferencia del constructor que es un método de la clase que es ejecutado cuando inicia "la vida" de un objeto –en su definición-, el destructor es un método que es ejecutado cuando el objeto se destruye -"su vida acaba"-. Su utilidad consiste en liberar memoria cuando un objeto u objetos, ya no sean útiles, cerrar un objeto archivo, hacer algunas operaciones de resúmen, entre otras cosas.

Las características de un destructor son :

- es un método de la clase.
- se ejecuta cuando un objeto perteneciente a la clase es destruido.
- se llama igual que la clase con un caracter tilde (~) como prefijo del nombre.
- no retorna valores.
- no tiene parámetros.
- sólo puede existir un destructor en una clase.
- no podemos asignarle el modificador de acceso `public`.

Para demostrar cuando se ejecuta el destructor agreguemos uno a la aplicación del veterinario, según se indica enseguida :

```
~Veterinario()
{
    MessageBox.Show("estamos en ejecución del destructor");
}
```

Además debemos agregar a la clase `Veterinario` en la parte de inclusión de bibliotecas de clases la sentencia :

```
using System.Windows.Forms;
```

Esta inclusión de biblioteca es necesaria porque en ella se encuentra la clase MessageBox y desde luego su método Show().

Ejecutemos la aplicación Windows del veterinario y de inmediato cerremos la aplicación haciendo click en la tacha o x de la ventana. Notemos que se visualiza la ventana de diálogo indicando el mensaje “estamos en ejecución del destructor” que es cerrada después de unos cuantos segundos.

La figura #5.3.33 muestra a la clase Veterinario con la inclusión del destructor al final de su cuerpo.

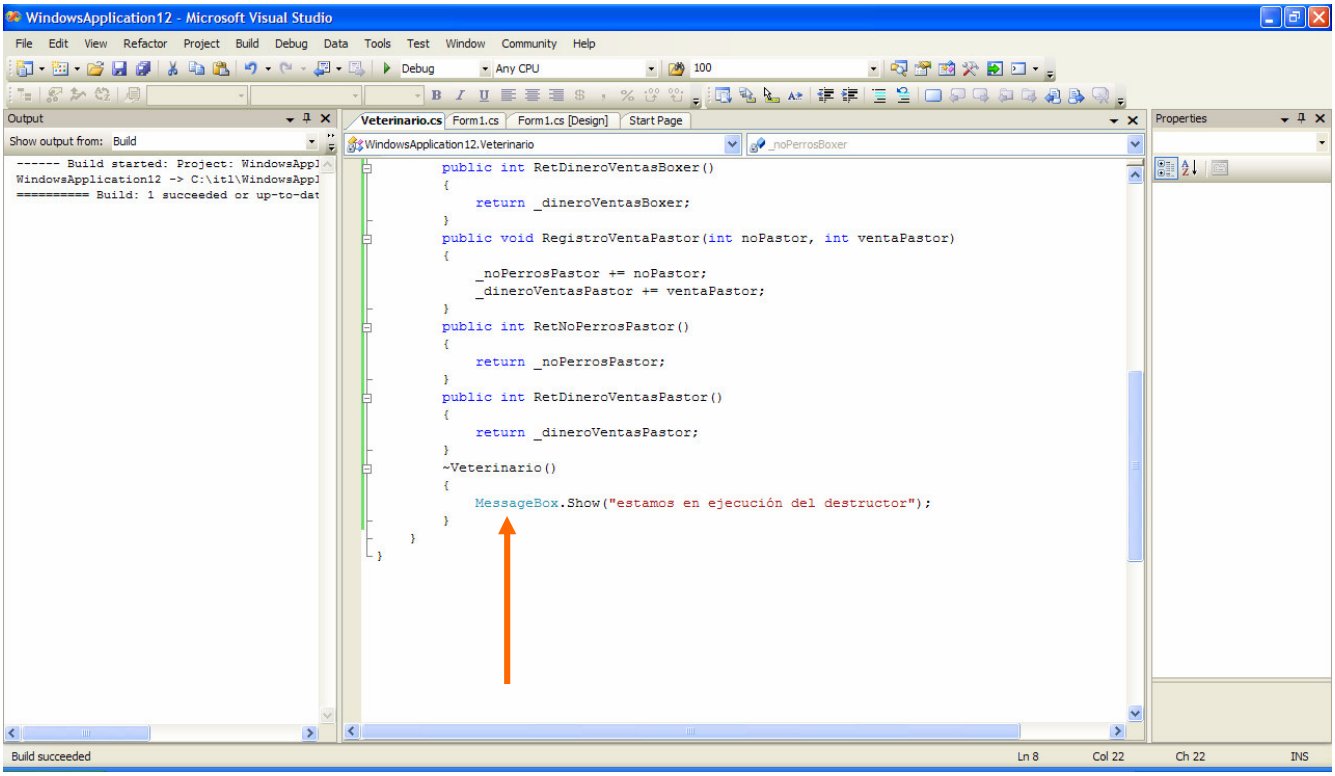


Fig. No. 5.3.33 Destructor en la clase Veterinario.

EJERCICIO PROPUESTO.
Agrega un destructor a la clase Alumno en tu aplicación de la sección 5.3.4.