Apuntes de Fundamentos de Programación UNIDAD 6.

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.

6 Estructuras secuenciales y selectivas.

Desde 1973 Edgser W. Dijkstra establece con sus principios de la programación estructurada, que las instrucciones de un programa deben ejecutarse en secuencia, es decir sin saltos abruptos, por decirlo de alguna manera. El código espaguethi es un ejemplo de saltos de "un lado a otro" dentro o hacia fuera del programa. Una instrucción que solía usarse dentro de un código espaguethi es el goto. Como buenos programadores y siguiendo las enseñanzas de Dijkstra, debemos evitar el uso de senetencias goto.

```
instr1;
instr2;
instr3;
...
...
```

Ejecución en secuencia de las instrucciones de un programa

La secuencia en que son ejecutadas las sentencias –instrucciones- de un programa se le conoce como *flujo de ejecución del programa*. El control de flujo de ejecución de las sentencias de un programa en ocasiones debe variar, según sean realizadas algunas validaciones de datos. Por ejemplo, un algoritmo que efectúe la suma de los primeros **n** números naturales, donde **n** es un dato entero que se lee en nuestra aplicación. Si el valor de **n** leído es 5, el programa deberá calcular la suma de los primeros 5 números naturales. Digamos que las instrucciones del programa en secuencia son :

```
n=oCalc.Leer();
oCalc.EfectuarSuma(n);
visua "la suma de los primeros ", n, "números naturales es = ", oCalc.RetSuma();
```

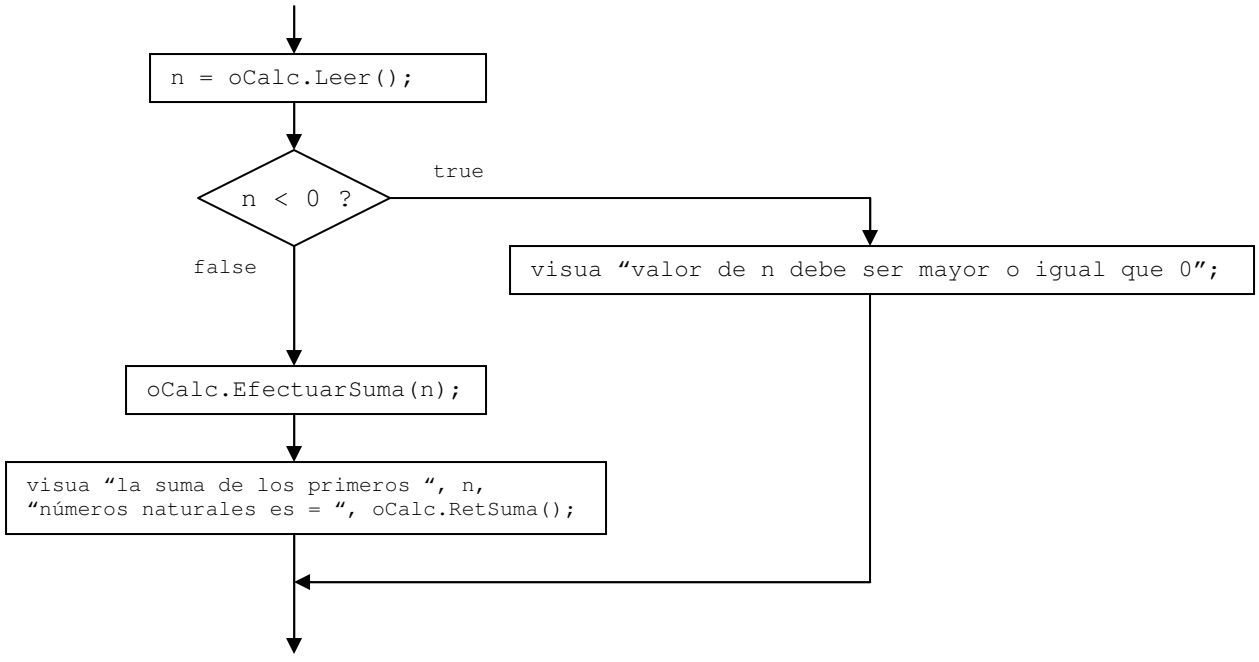
Flujo de ejecución en secuencia de las instrucciones del programa

`oCalc` es un objeto de una determinada clase a la que llamaremos `Calculador`, y es el encargado de leer el valor de **n** y de efectuar la suma de los **n** primeros números naturales, además de visualizar el resultado de la suma. El método `EfectuarSuma(n)` deberá calcular el valor de la suma de alguna forma :

```
suma = 1 + 2 + 3 + 4 + 5;
```

Suma de los primeros 5 números naturales.

Algo muy importante es que los números naturales no son negativos. ¿Qué pasaría si el usuario teclea un valor de **n** negativo?, digamos el -5. El programa debería entender que queremos calcular la suma de los ¡ primeros -5 números naturales !, cuestión que no es lógica. Por lo tanto debemos de modificar el control de flujo de ejecución de las sentencias cuando el valor leído sea negativo. A una instrucción que evalúa una condición (el resultado de una condición es false o true) y de acuerdo a esta evaluación controlar el flujo de ejecución del programa, se le denomina *instrucción selectiva* o *de selección*. A continuación expresamos visualmente cómo el flujo de ejecución del programa varía de acuerdo a la condición **`n < 0`** ?.



Las flechas indican el flujo de ejecución del programa. Notemos que la condición se denota gráficamente mediante un rombo el cuál tiene 2 salidas : **true** cuando la condición probada es verdadera y **false** cuando no lo es. Cuando el valor de **n** es negativo el programa fluye hacia la sentencia de visualización del mensaje de error. Cuando el valor de **n** es positivo el control de ejecución del programa fluye hacia las sentencias que calculan la suma y visualizan los resultados.

En el transcurso de la exposición de esta unidad 6 explicaremos este tipo de sentencias selectivas, además de repasar las sentencias de entrada y salida de datos. También daremos el repaso a los modificadores de acceso `public` y `private`.

6.1 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`

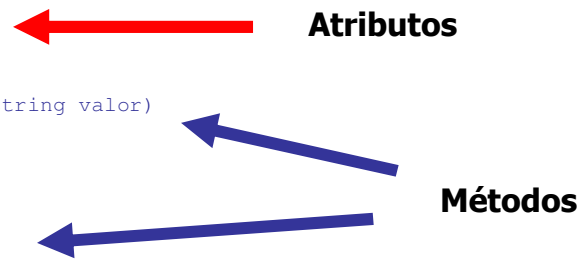
Un beneficio fundamental en la orientación a objetos es el ocultamiento de datos –atributos-, que consiste en que sólo los métodos definidos en la clase, son los únicos que pueden acceder a los atributos de los objetos pertenecientes a una clase. La implementación del concepto de ocultamiento de datos 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;
  }

}
```

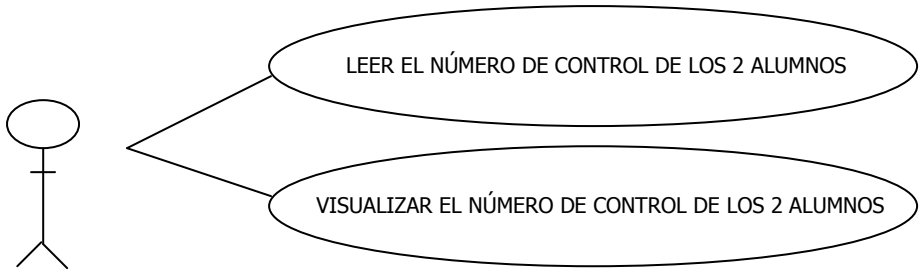


No hemos escrito los métodos para asignar y retornar los atributos `_nombre` y `_calif` sólo por no hacer mas abultada la clase `Alumno`, después los agregaremos.

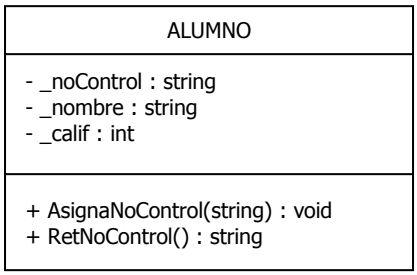
Notemos que los 3 atributos han sido declarados antecediéndolos de la palabra reservada `private`, que indica que el atributo no puede ser accedido desde cualquier parte de la aplicación. El atributo así declarado, sólo podrá ser accedido por los métodos de la clase `Alumno`, en este caso por lo tanto, 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 #6.1.1. Notemos que las etiquetas `label5` y `label6` tendrán la utilidad de visualizar los números de control leídos.

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 #6.1.2 muestra a la clase Alumno.cs.

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`.

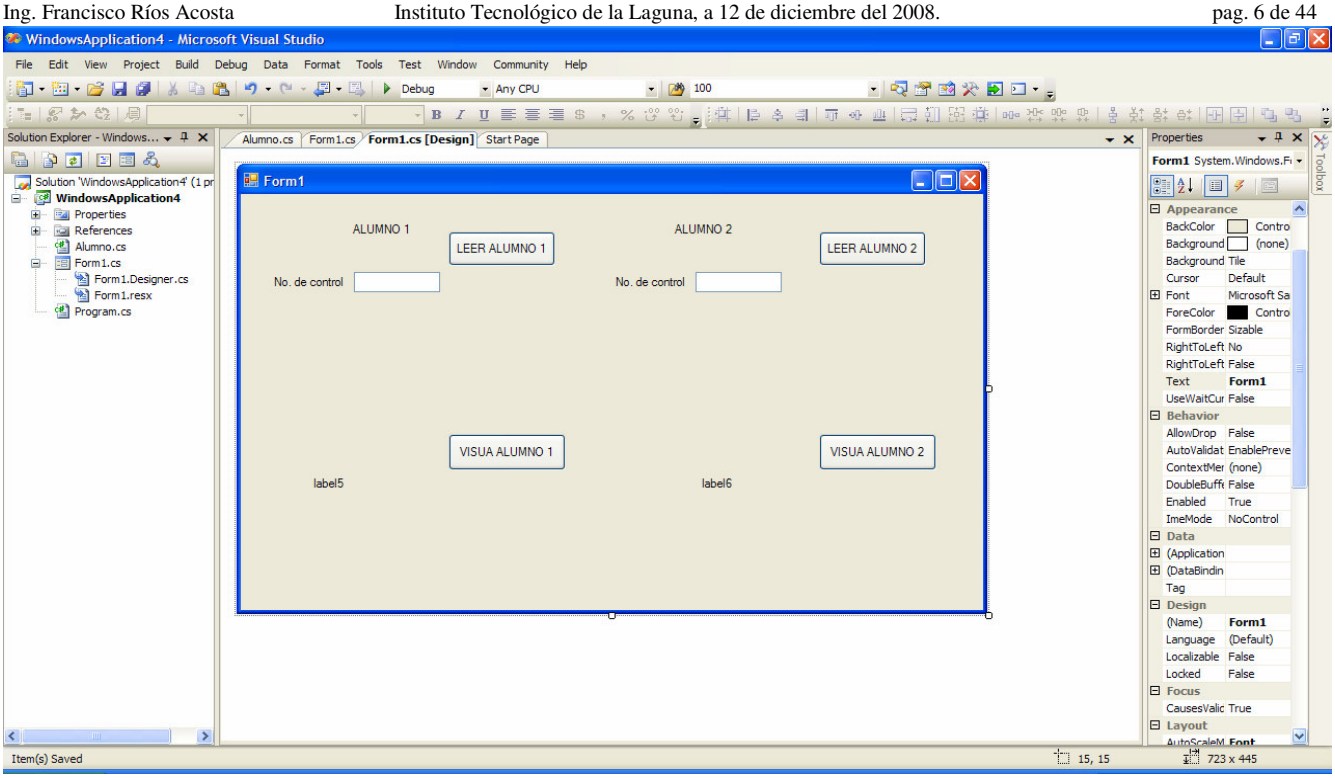


Fig. No. 6.1.1 Interfase gráfica de usuario Form1.cs[Design].

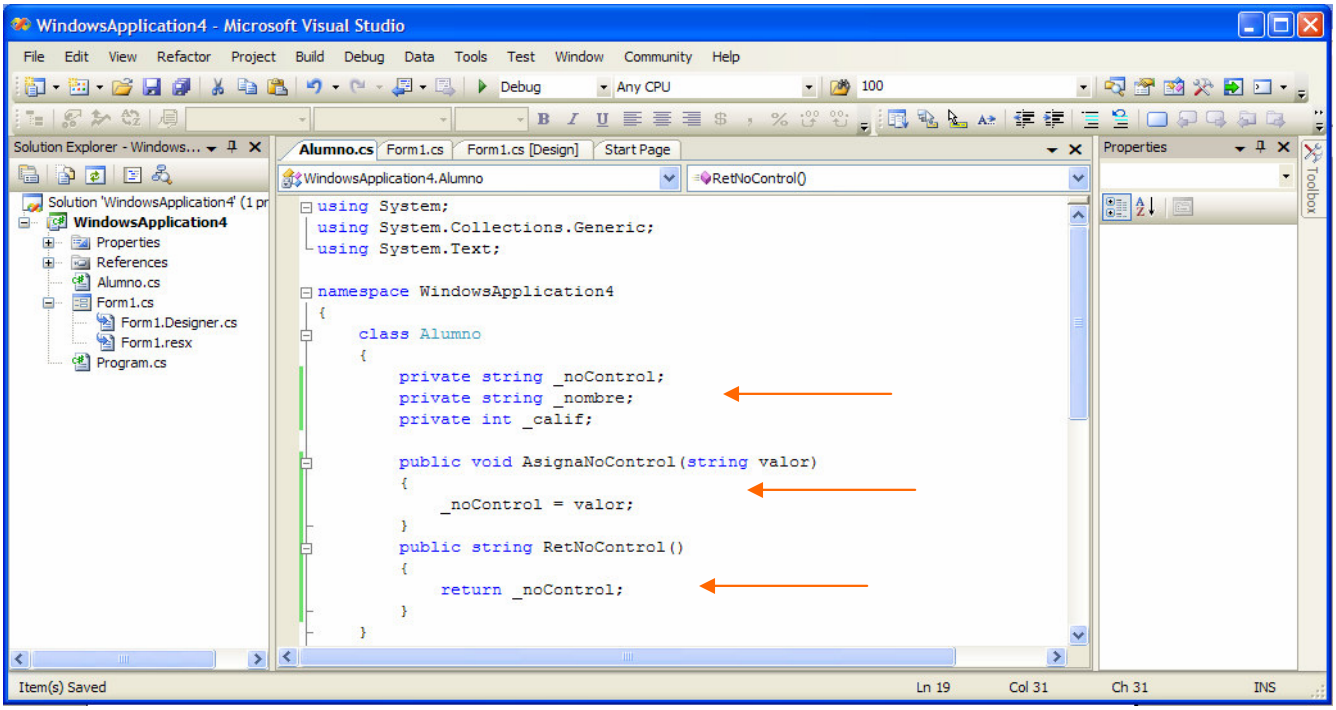


Fig. No. 6.1.2 Clase `Alumno`.

¿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 6.1.3.

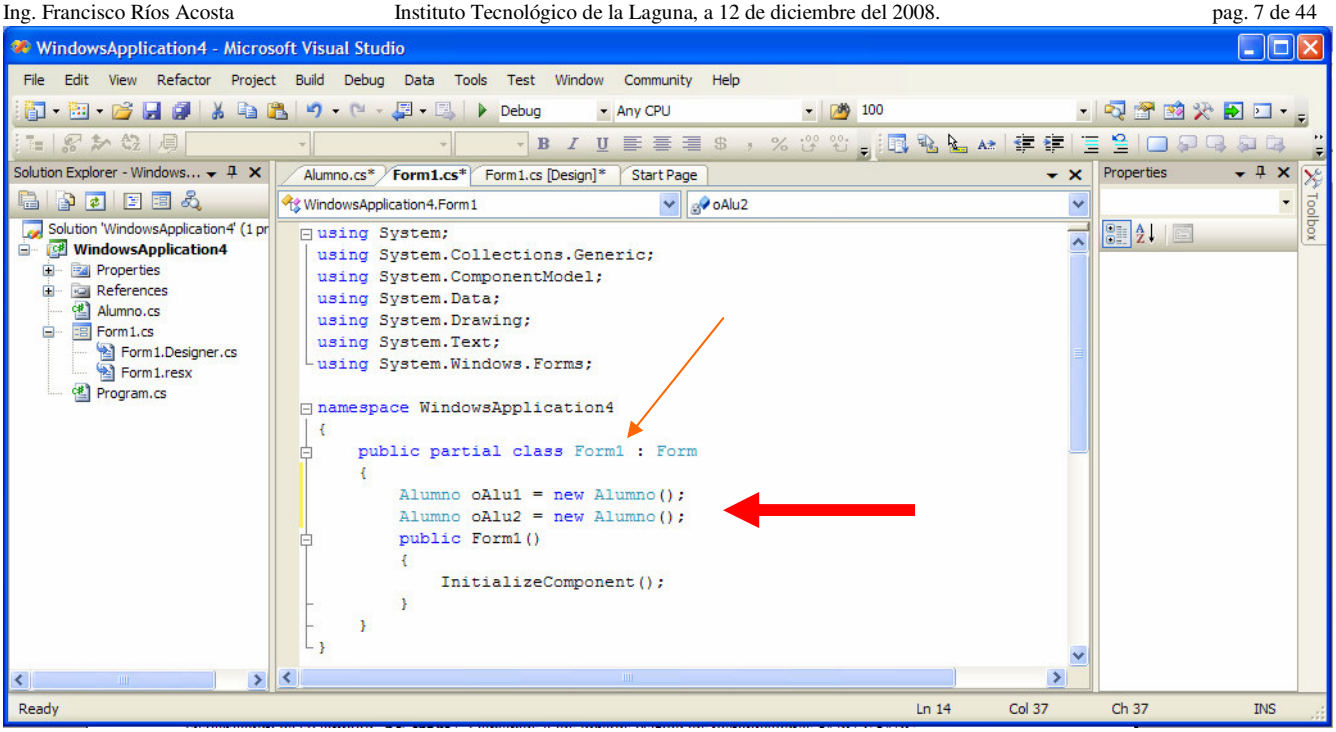


Fig. No. 6.1.3 Definición de los objetos `oAlu1` y `oAlu2` en la clase `Form1`.

Ejecutemos la aplicación sólo con el fin de conocer si no hemos cometido algún error de dedo, figura #6.1.4.

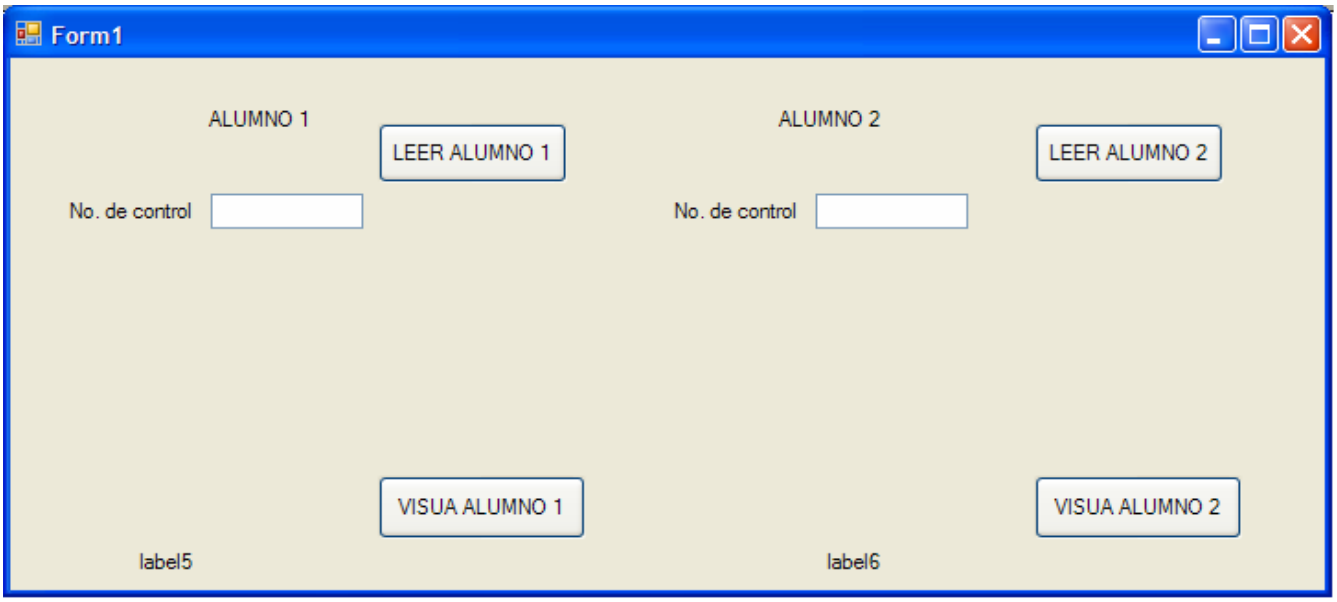


Fig. No. 6.1.4 Nuestra aplicación en ejecución.

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 #6.1.5. Observemos que los métodos Click de cada botón residen dentro de la clase `Form1`.

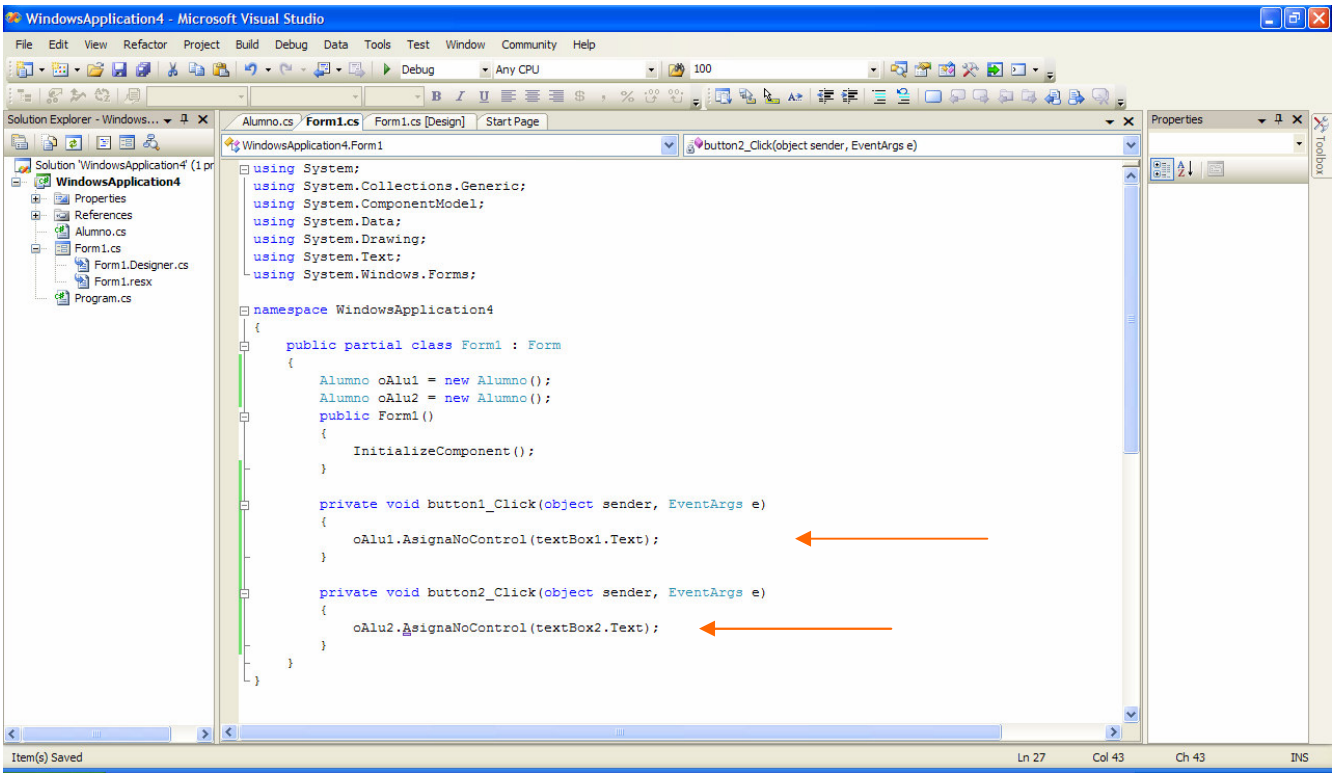


Fig. No. 6.1.5 Inclusión de los mensajes para leer el número de control de un alumno.

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 la figura 6.1.6.

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;
```

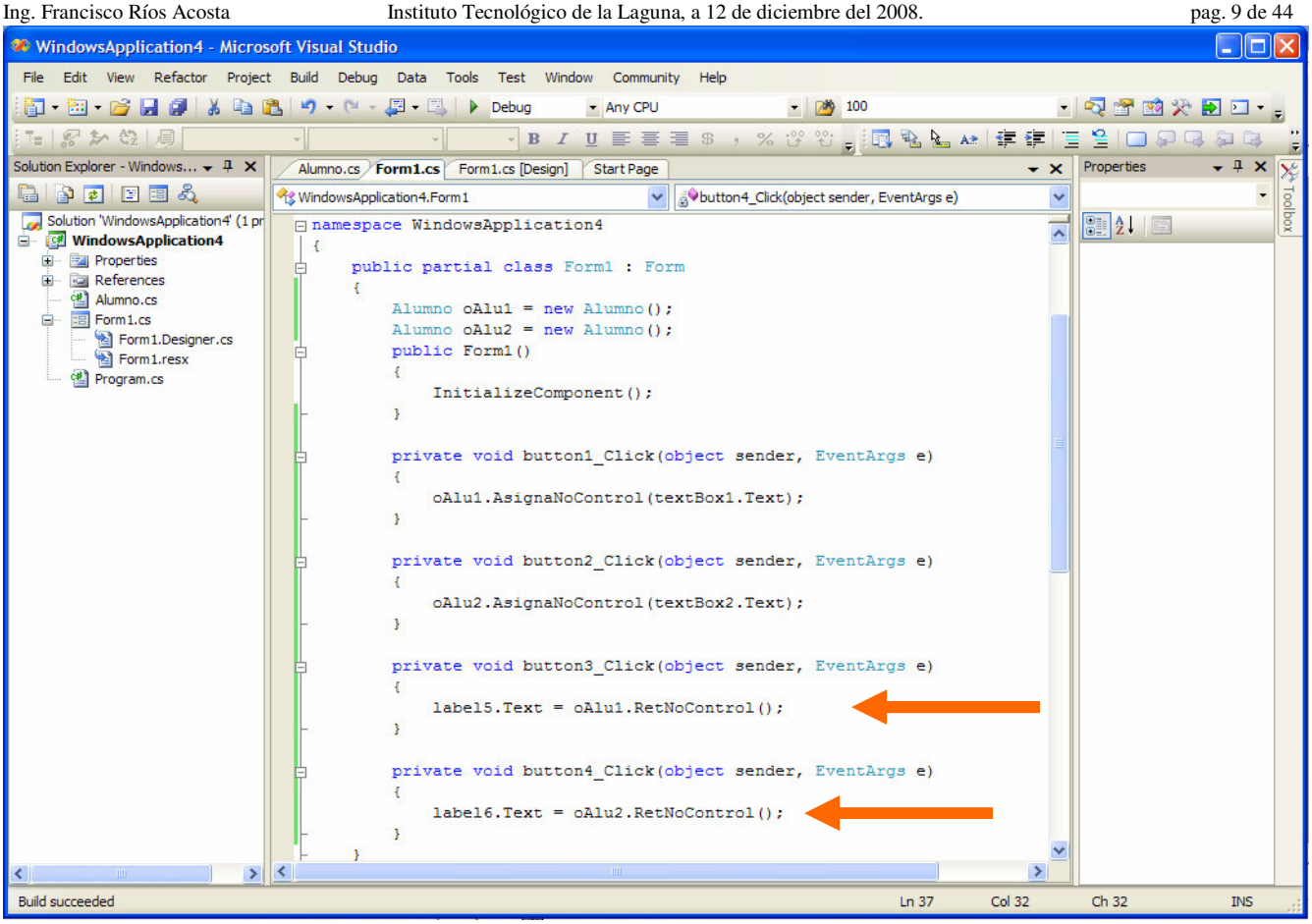


Fig. No. 6.1.6 Inclusión de los mensajes para visualizar el número de control de un alumno.

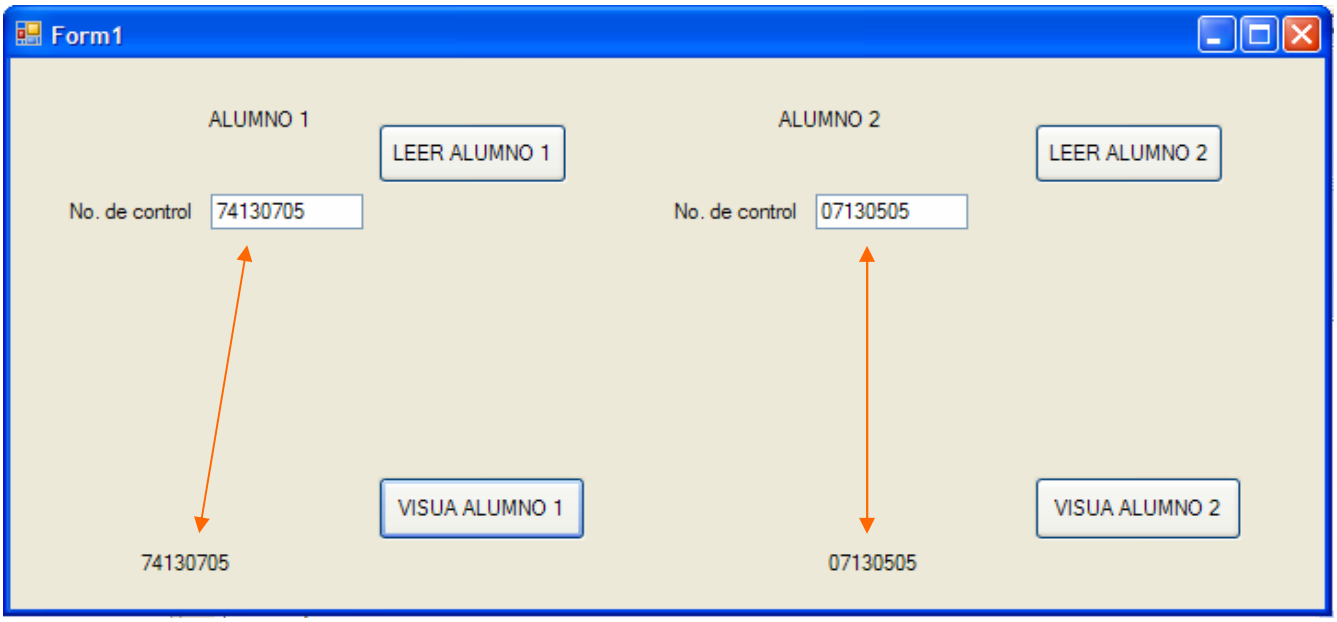


Fig. No. 6.1.7 Ejecución de la aplicación vista después de hacer click en los botones de lectura y de visualización.

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 #6.1.8 muestra al mensaje previamente escrito como un comentario –no es tomado en cuenta por la compilación-, y el nuevo mensaje que lo sustituye.

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 #6.1.8 muestra al mensaje previamente escrito como un comentario –no es tomado en cuenta por la compilación-, y el nuevo mensaje que lo sustituye.

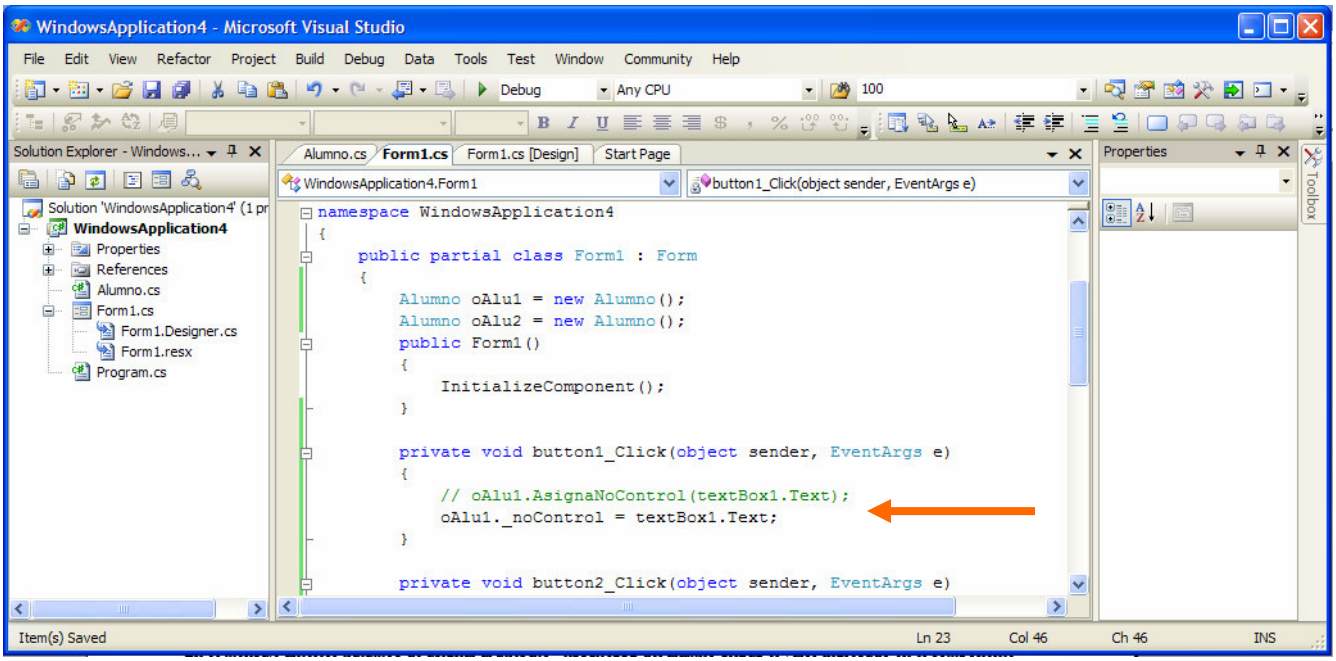


Fig. No. 6.1.8 Acceso directo al atributo `_noControl` del objeto `oAlu1`.

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

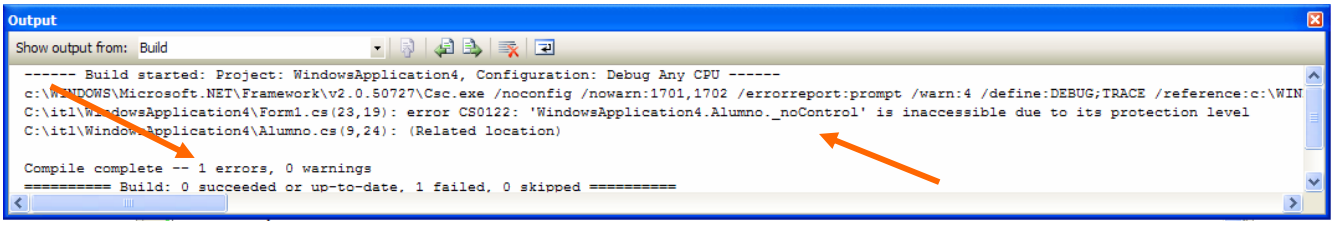


Fig. No. 6.1.9 ERROR al tratar de accesar al atributo `_noControl` del objeto `oAlu1`.

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.

6.2 Entrada y salida de datos. 6.3 Interacción de la aplicación y la clase.

Tal y como hemos venido explicando en las aplicaciones construidas en la unidad 5 y en la sección 6.1 de esta unidad, la entrada de datos en una aplicación Windows C# la hacemos utilizando el componente TextBox. La salida de datos la implementamos mediante el uso del componente Label. Desde luego que estas maneras de implementar la entrada y salida de datos, no son únicas. Aquí se han propuesto sólo estos 2 componentes, de manera que el alumno se sitúe en el aprendizaje de los fundamentos de programación y no en los detalles de uso de los diferentes componentes que nos proporciona un ambiente tan completo como es el Visual C#.
En cuanto a la interacción entre la clase y la aplicación C#, diremos que se efectúa de varias maneras :

- Definiendo objetos en el archivo Form1.cs dentro de la clase `class Form1`, cuyo ámbito sea precisamente todos los métodos de dicha clase. Al definir estos objetos dentro de la clase, realmente estamos estableciendo que dichos objetos son atributos de la clase `Form1`. Recordemos que la clase `Form1` representa a la aplicación Windows. En la figura #6.1.3 se muestra la definición de 2 objetos de la clase `Alumno oAlu1` y `oAlu2` dentro del cuerpo de la forma `Form1`.
- Otra forma de interacción es el pasaje de parámetros a los métodos de nuestras clases definidas. Estos parámetros son valores leídos en los componentes TextBox insertados en la aplicación Windows. Desde luego que los métodos son parte de mensajes a los objetos definidos previamente. Generalmente es usado un componente Button para efectuar una acción de lectura o de asignación. En las páginas 7 y 8 se muestran 2 mensajes donde se asignan los números de control de 2 alumnos `oAlu1` y `oAlu2` que son ingresados por el usuario de la aplicación, en los componentes `textBox1` y `textBox2`. La propiedad Text de los componentes TextBox es la que contiene a los números de control ingresados.

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

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

- Los objetos pueden también retornar valores que pueden ser asignados a las propiedades –o por métodos- de los componentes que forman parte de la aplicación Windows. Los parámetros también pueden ser de entrada-salida tal y como lo vimos en la sección 5.3.3 de la unidad 5. En la página 8 de esta unidad 6 tenemos 2 asignaciones a la propiedad `Text` de 2 componentes Label, de los números de control de 2 alumnos `oAlu1` y `oAlu2` :

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

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

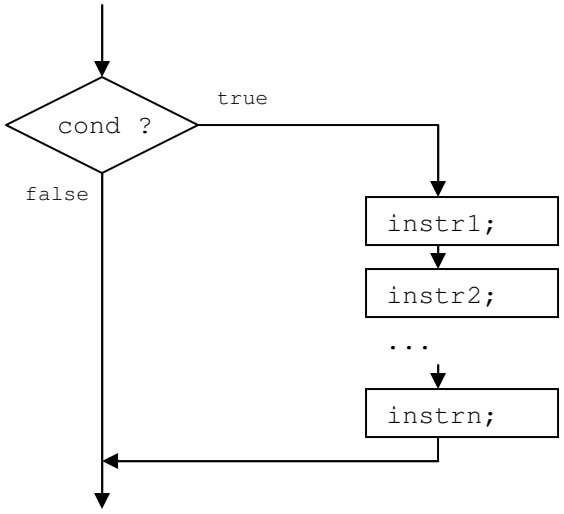
Para más sobre entrada y salida de datos, además de la interacción entre clase y aplicación, efectúa un repaso de la unidad 5.

6.4 Estructuras selectivas.

En esta sección desarrollaremos las sentencias de selección :
- selectiva simple –si de una rama- **`if`**.
- selectiva doble –si de dos ramas- **`if-else`**.
- selectiva anidada –**`if-else if`**-.
- selectiva múltiple –**`switch-case-default`**-.
- selectiva intenta –**`try-catch`**-.

6.4.1 Selectiva simple –si de una rama-.

Esta sentencia se caracteriza por tener sólo una rama donde se efectúan instrucciones dado que la condición probada sea verdadera. Su diagrama de flujo es :



if de una rama

La implementación en C# se realiza usando la palabra reservada **`if`**. Si se desea ejecutar mas de una instrucción en la rama **true** del **`if`**, es necesario encerrar a dichas sentencias dentro de un bloque de código, es decir entre los caracteres **{ }**. La condición siempre debe ser acotada por los paréntesis circulares (cond).

```
if (cond)
   instrucción;
```
← if de una rama con una sola sentencia.

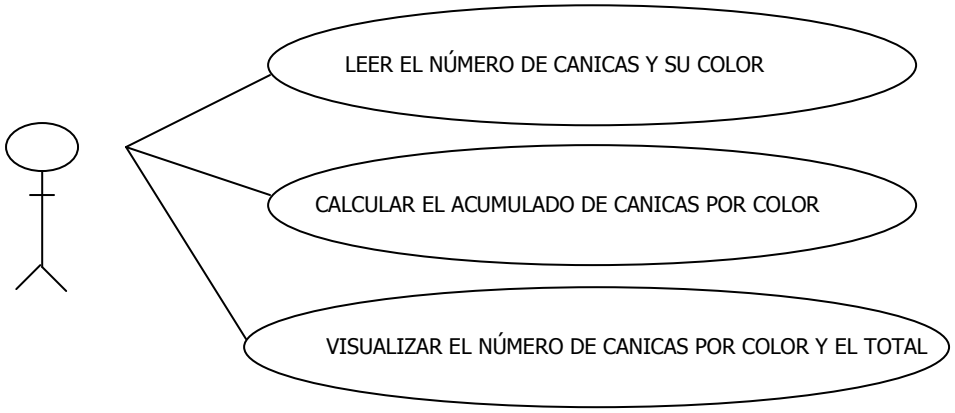
```
if (cond)
{
  instr1;
  instr2;
  ...
  instrn;
}
```
← if de una rama con mas de una sentencia.

Ejercicio 6.4.1

Un niño requiere registrar la cantidad de canicas que recibe de parte de sus seres queridos. Las canicas pueden ser azules, rojas y verdes. Le interesa saber cuántas canicas tiene azules, cuantas rojas y cuantas verdes, además el total de las canicas que le han regalado.

Solución :
Iniciamos por el diagrama de casos de uso el cual tiene 3 tareas :

- Leer el número de canicas recibidas y su color.
- Calcular el acumulado de canicas para el color leído.
- Visualizar el número de canicas por color y el total de ellas.



El diagrama de clase tiene 3 atributos : el número de canicas azules `_noCanAzules`, el número de canicas rojas `_noCanRojas` y el número de canicas verdes `_noCanVerdes`.

NINO
- _noCanAzules : int - _noCanRojas : int - _noCanVerdes : int
+ Leer(noCan:int,colorCan:string) : void + RetNoCanAzules() : int + RetNoCanRojas() : int + RetNoCanVerdes() : int

Los métodos los abstraemos de acuerdo a las tareas identificadas en el diagrama de casos de uso :

```
Leer(noDeCanicas, Color);
RetNoCanAzules()
RetNoCanRojas()
RetNoCanVerdes().
```

El cálculo del acumulado de canicas lo vamos a realizar dentro del método `Leer()`.
La visualización del total de canicas la efectuamos sumando el resultado de las llamadas a los 3 métodos :

```
RetCanAzules()+RetCanRojas()+RetCanVerdes().
```

Sigamos con la construcción de la interfase gráfica que permite lograr las tareas en el diagrama de casos de uso. La figura #6.4.1 contiene los componentes que integran la realización de todas las tareas abstraídas.

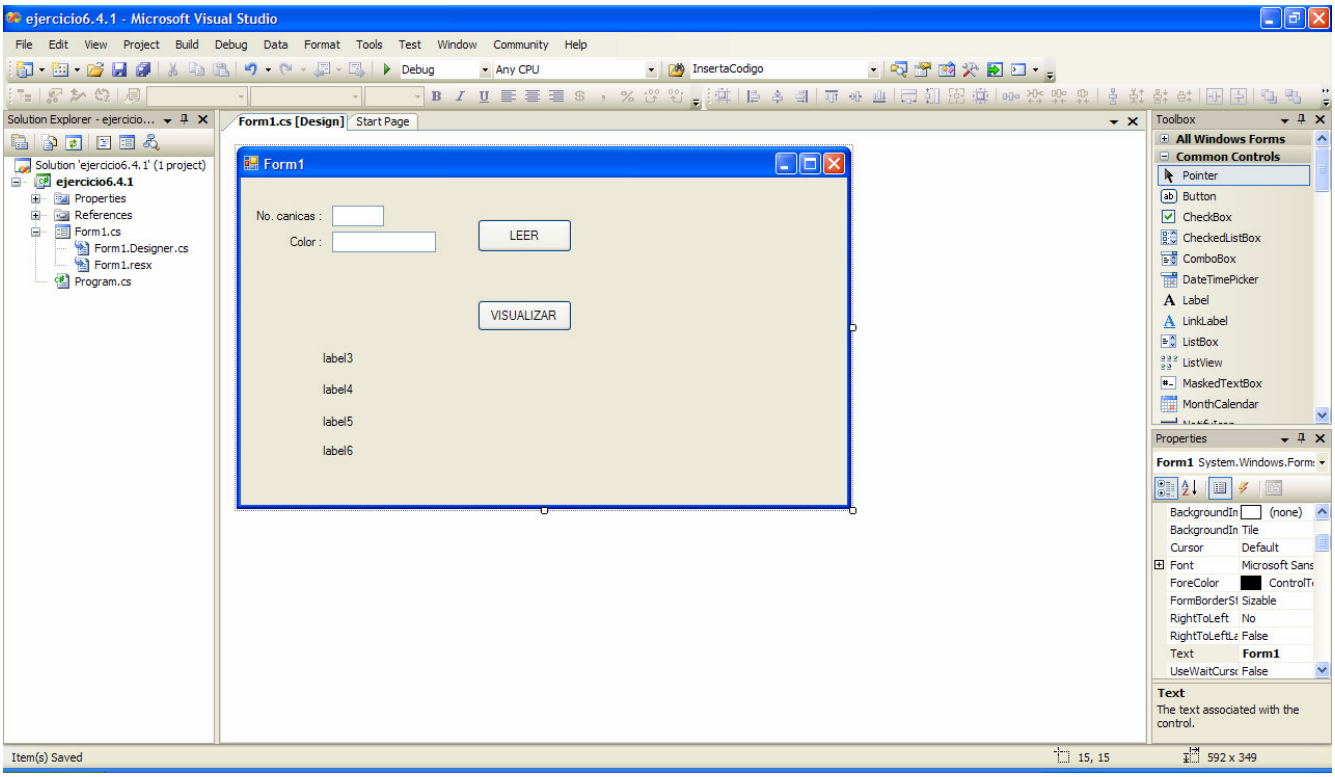


Fig. No. 6.4.1 Interfase gráfica.

Las etiquetas `label3` a `label6` servirán para la visualización del total de canicas y del número de canicas por color que ha registrado el niño. Agreguemos a la clase `Nino` al proyecto. Sólo por ahora añadimos los atributos y el constructor que inicializa el número de canicas de cada color a 0, figura #6.4.2.

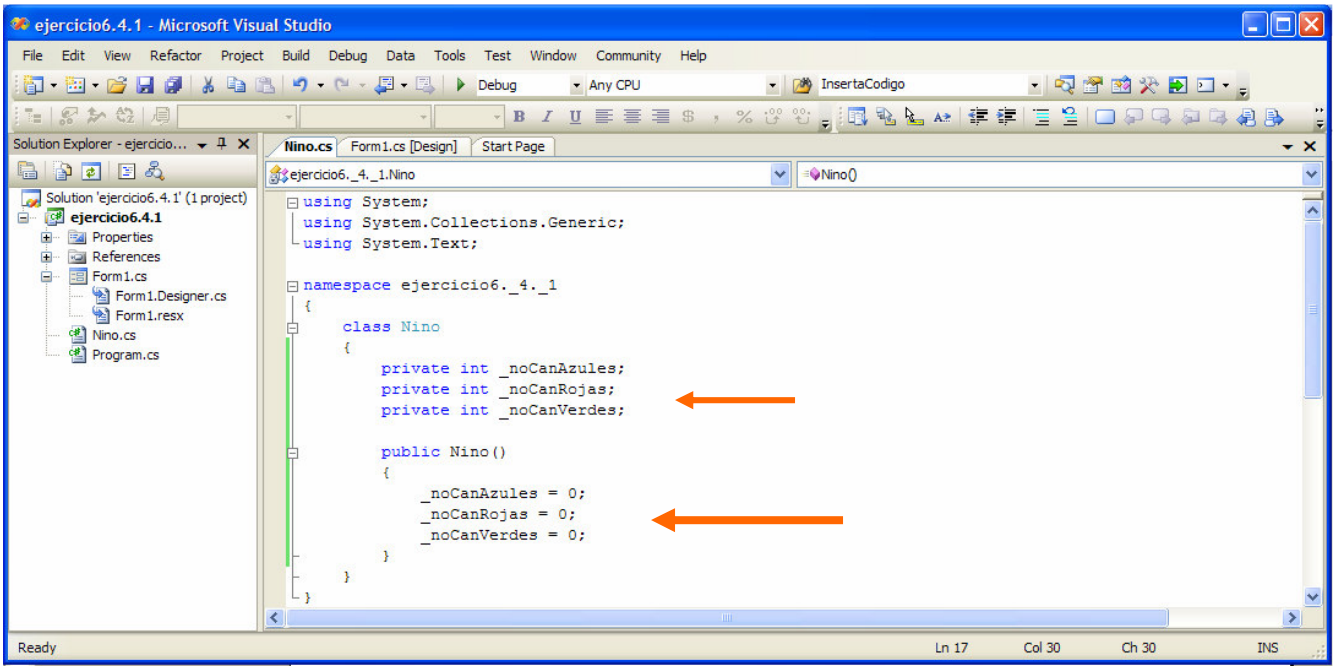


Fig. No. 6.4.2 Clase `Nino` incluyendo su constructor.

Lo que sigue es agregar la definición del objeto `oNino` perteneciente a la clase `Nino`, en la clase Form1, según se muestra en la figura #6.4.3.

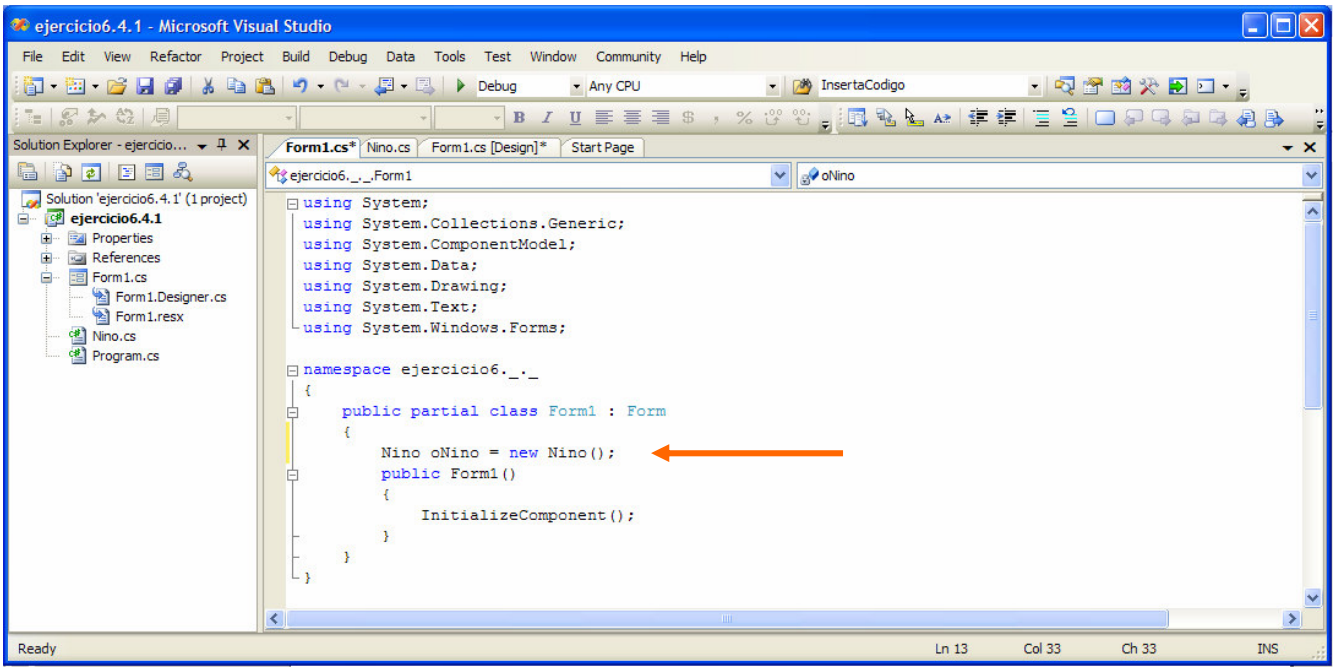


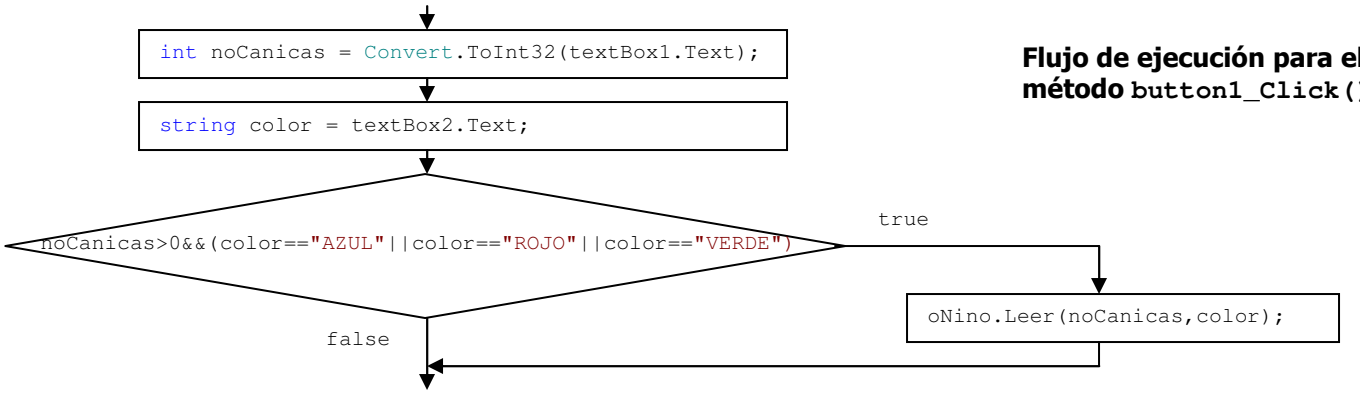
Fig. No. 6.4.3 Definición del objeto `oNino`.

Veamos las siguientes consideraciones antes de iniciar con la codificación del evento Click del botón `button1` con leyenda LEER :

- El usuario puede teclear valores negativos en el componente `textBox1` que sirve para ingresar el número de canicas a registrar. También puede teclear un 0 que significan 0 canicas a registrar. En ambos casos, no debemos mandar el mensaje `Leer()` al objeto `oNino`, ya que sólo registramos cantidades de canicas mayores que 0. Aquí debemos aplicar una validación del dato leído `noDeCanicas`.
- El usuario puede teclear de manera diferente el color de las canicas, es decir “Azul”, “azules”, “rojas”, “rojo”, “verdes”, “berde”, “asul”, etcétera. En este caso seguiremos el criterio de aceptar sólo las cadenas “AZUL”, “ROJO” y “VERDE” el color en mayúscula y en singular. También aquí debemos efectuar una validación del dato leído `color`, de manera que si no es igual a cualquiera de las 3 constantes cadenas anteriores, no debemos llamar al método `Leer()`.

Teniendo en cuenta las anteriores consideraciones, procedemos a codificar el Click del `button1`, según se indica a continuación.

```
private void button1_Click(object sender, EventArgs e)
{
    int noCanicas = Convert.ToInt32(textBox1.Text);
    string color = textBox2.Text;
    if (noCanicas>0 && (color=="AZUL" || color=="ROJO" || color=="VERDE"))
       oNino.Leer(noCanicas,color);
}
```



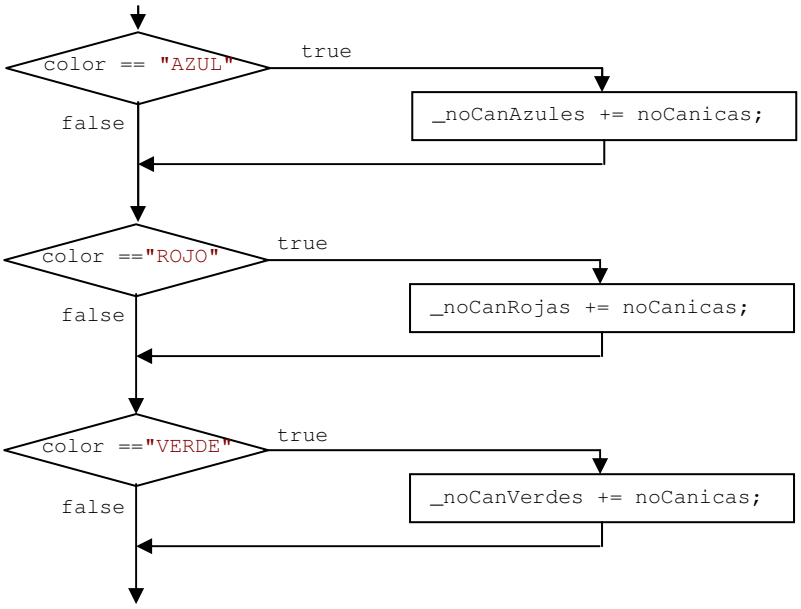
Flujo de ejecución para el método `button1_Click()`

En el código anterior estamos empleando un **`if`** de una rama con el fin de realizar el mensaje `oNino.Leer()` sólo si se cumplen valores válidos para las variables locales `noCanicas` y `color`.

El método `Leer()` debemos añadirlo en la clase `Nino` y tiene la siguiente definición :

```
public void Leer(int noCanicas, string color)
{
     if (color == "AZUL")
         _noCanAzules += noCanicas;
     if (color == "ROJO")
         _noCanRojas += noCanicas;
     if (color == "VERDE")
         _noCanVerdes += noCanicas;
}
```

Flujo de ejecución para el método `Leer()` de la clase `Nino`.



Notemos que en el método `Leer()` de la clase `Nino`, la evaluación de las 3 condiciones siempre es efectuada no importando que por ejemplo la variable `color` sea "AZUL", la prueba de las condiciones a "ROJO" y a "VERDE" de la variable `color` es realizada. Ésto sucede ya que el control de flujo regresa a la secuencia de ejecución de cada condición. Esta cuestión puede corregirse usando **if's** anidados que serán vistos en la sección 6.4.3. Lo lógico es que si ya se sabe que el color es "AZUL" entonces la ejecución del método debería terminar sin llegar a probar las condiciones "ROJO" y "VERDE" de la variable local `color`. Lo mismo podríamos decir para cuando `color` sea "ROJO" la prueba del "VERDE" debería no ejecutarse.
Ejecutemos la aplicación Windows y observemos que ingresando valores válidos y no válidos el programa no retroalimenta al usuario ningún mensaje que le ayude a saber el estado de su ejecución. Modifiquemos el **if** en el método `button1_Click` de Form1 en la rama **`true`**, de manera que mostremos un mensaje al usuario que le indique que la operación de registro de las canicas de un determinado color, se ha realizado.

```
private void button1_Click(object sender, EventArgs e)
{
    int noCanicas = Convert.ToInt32(textBox1.Text);
    string color = textBox2.Text;
    if (noCanicas > 0 && (color == "AZUL" || color == "ROJO" || color == "VERDE"))
    {
        oNino.Leer(noCanicas, color);
        MessageBox.Show("EL REGISTRO DE "+noCanicas.ToString()+" CANICAS COLOR "+color+" FUE
                                                                                REALIZADO");
    }
}
```

Observemos que se han agregado los caracteres **{** y **}** para acotar las 2 sentencias que se ejecutan cuando la condición en el **`if`** es **true**. También hemos usado la clase **`MessageBox`** y su método **`Show()`** para visualizar el mensaje al usuario.

Ejecutemos de nuevo la aplicación y demos valores válidos para observar la manera en que responde ahora nuestra aplicación. La figura #6.4.4 muestra la respuesta para el registro de 10 canicas color ROJO.

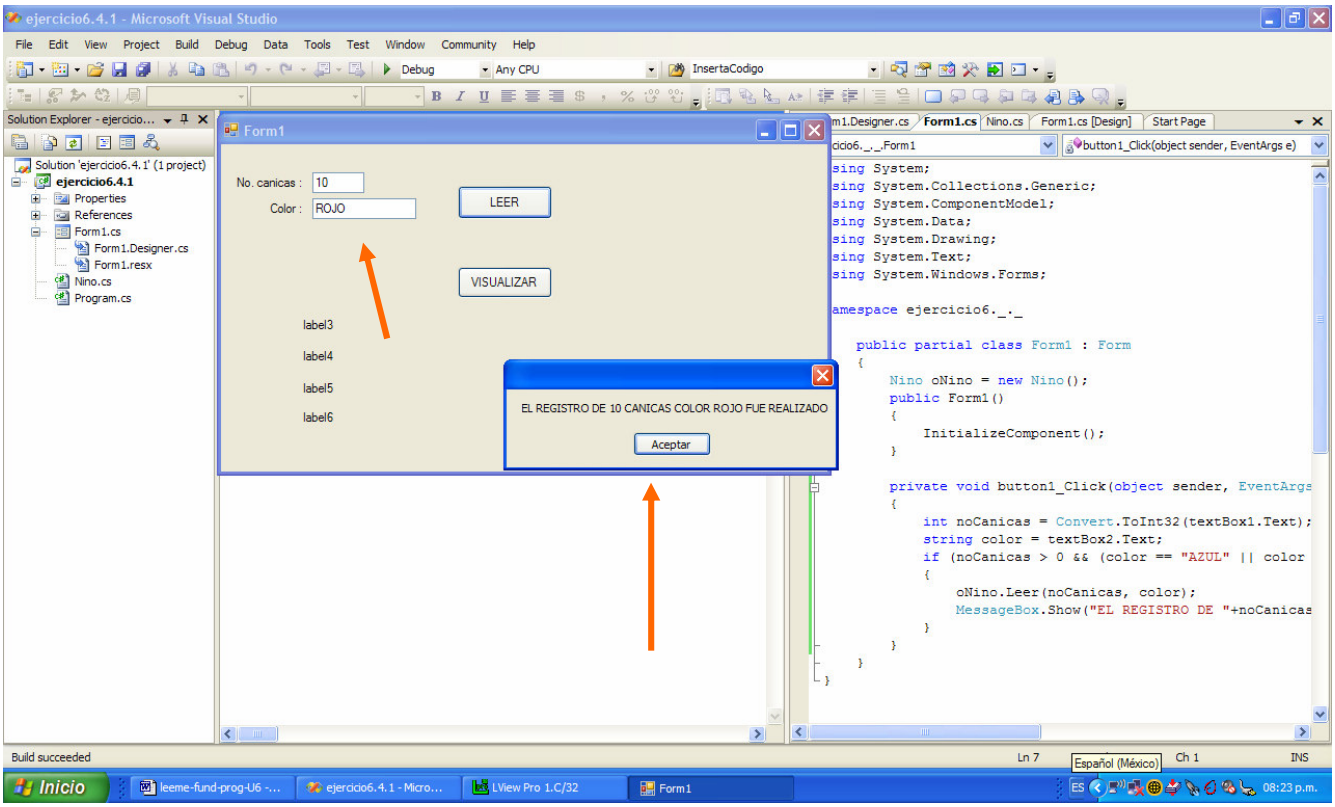
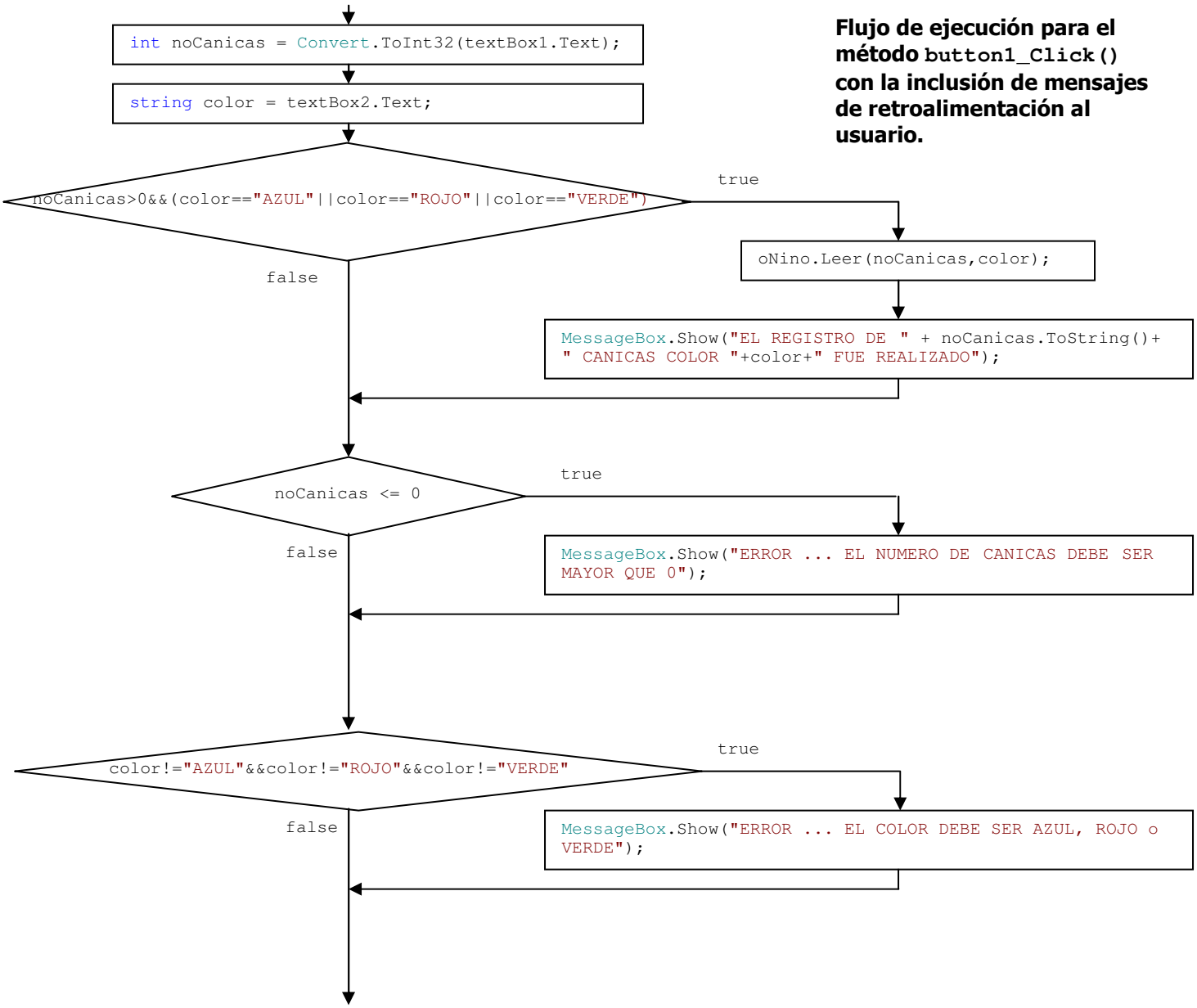


Fig. No. 6.4.4 Comunicación de la aplicación con el usuario para datos leídos válidos.

Observemos que aún cuando los datos no sean válidos la aplicación NO responde con ningún mensaje al usuario, es decir el usuario no sabe que pasó con su operación de registro ¿se hizo? o ¿no se hizo?, la aplicación es la que tiene la respuesta. La aplicación podrá responder a las preguntas anteriores si el usuario hace click sobre el botón button2 con leyenda VISUALIZAR, de manera que el usuario que tenga una cierta memoria podrá hacer sus cálculos y tal vez establezca que su operación de registro se realizó.
Realmente lo que debemos hacer es de inmediato retroalimentar al usuario si su registro se realizó o no se realizó. Ya hicimos lo primero, ahora debemos codificar la notificación cuando no se efectuó el registro. Aquí sería muy útil saber el uso del **if** de 2 ramas, pero como sólo hasta ahora sabemos el uso del **if** de una rama, éste es el que utilizaremos. Modifica el método `button1_Click()` de acuerdo al código siguiente :

```
private void button1_Click(object sender, EventArgs e)
{
    int noCanicas = Convert.ToInt32(textBox1.Text);
    string color = textBox2.Text;
    if (noCanicas > 0 && (color == "AZUL" || color == "ROJO" || color == "VERDE"))
    {
        oNino.Leer(noCanicas, color);
        MessageBox.Show("EL REGISTRO DE "+noCanicas.ToString()+" CANICAS COLOR "+color+" FUE
                                                                                    REALIZADO");
    }
    if (noCanicas <= 0)
        MessageBox.Show("ERROR ... EL NUMERO DE CANICAS DEBE SER MAYOR QUE 0");
    if (color != "AZUL" && color != "ROJO" && color != "VERDE")
        MessageBox.Show("ERROR ... EL COLOR DEBE SER AZUL, ROJO o VERDE");
}
```

Ejecutemos la aplicación y demos valores 0 o negativos al número de canicas para obtener el mensaje de error para la lectura del número de canicas. También hagamos lo similar pero para los valores del color de las canicas registradas. El flujo de ejecución del método button1_Click() se muestra a continuación.



Flujo de ejecución para el método `button1_Click()` con la inclusión de mensajes de retroalimentación al usuario.

Sólo resta añadir el código para visualizar las canicas que tiene el niño. El botón `button2` con leyenda VISUALIZAR es el encargado de hacer esta tarea. Este código no necesita validar ningún dato así que sólo son necesarios los mensajes al objeto `oNino` que retornan las cantidades de canicas de color AZUL, ROJO, y VERDE. Agrega el código del método `button2_Click()` en tu aplicación según se indica enseguida :

```
private void button2_Click(object sender, EventArgs e)
{
    label3.Text = "EL NIÑO TIENE REGISTRADAS " + (oNino.RetNoCanAzules() + oNino.RetNoCanRojas() +
                                                  oNino.RetNoCanVerdes()).ToString() + " CANICAS";
    label4.Text = oNino.RetNoCanAzules().ToString() + " CANICAS AZULES";
    label5.Text = oNino.RetNoCanRojas().ToString() + " CANICAS ROJAS";
    label6.Text = oNino.RetNoCanVerdes().ToString() + " CANICAS VERDES";
}
```

La figura 6.4.5 muestra la ejecución de la aplicación para un registro de 10 canicas color AZUL, 10 canicas color ROJO y 10 canicas color VERDE.

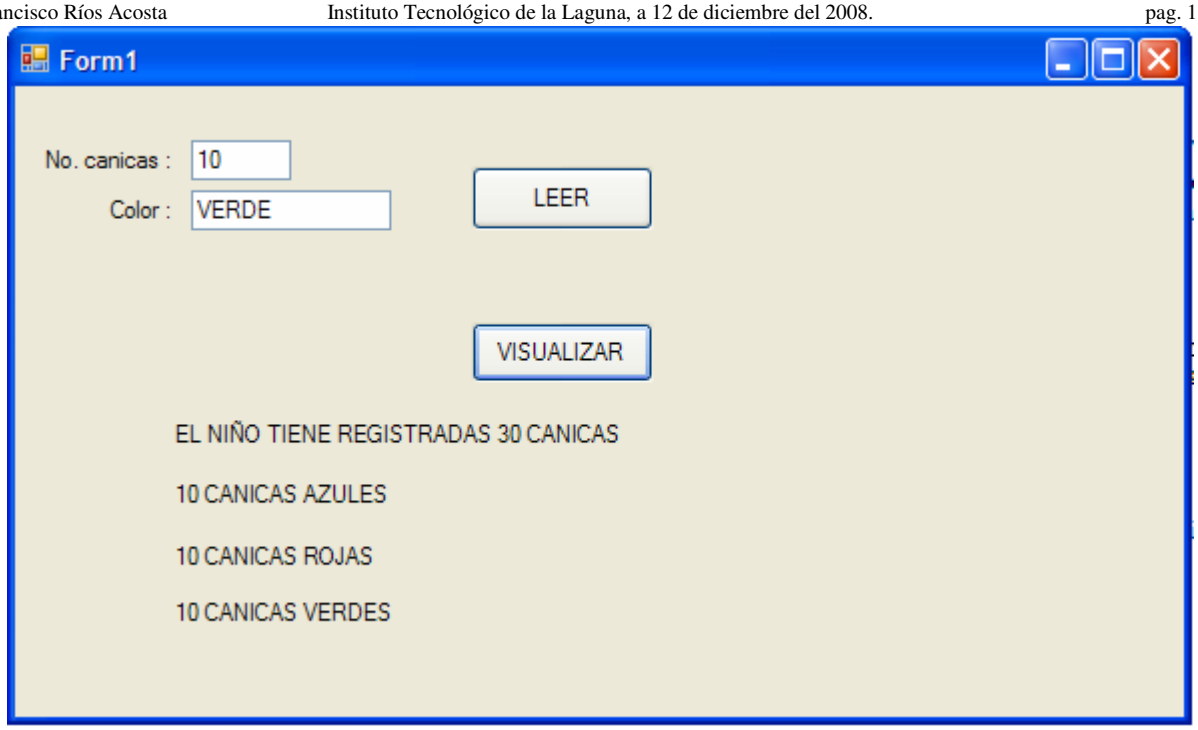
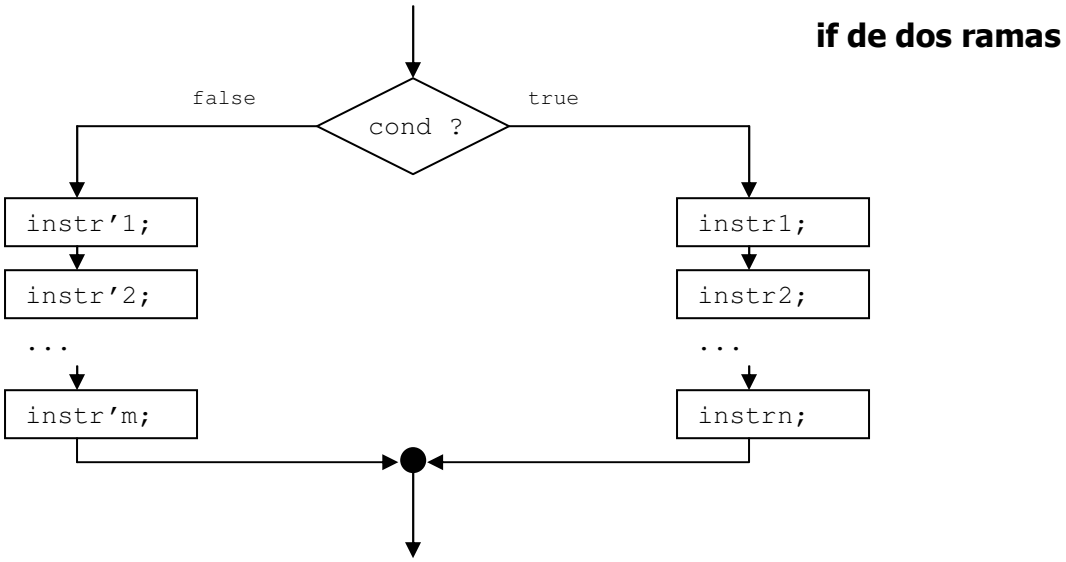


Fig. No. 6.4.5 Visualización del número de canicas registradas.

6.4.2 Selectiva doble –si de dos ramas-.

La sentencia **if-else** permite ejecutar código en ambas ramas : la true y la false. Su flujo de ejecución tiene 2 ramas según se muestra enseguida. Ambas ramas son mutuamente excluyentes : el flujo de ejecución se va por la rama del true o bien, se va por la rama del false, pero no por ambas.



La implementación en C# se realiza usando las palabras reservadas **`if – else`**. Si se desea ejecutar mas de una instrucción en cualquiera de las 2 ramas del **`if`**, es necesario encerrar a dichas sentencias dentro de un bloque de código, es decir entre los caracteres **{ }**. La condición siempre debe ser acotada por los paréntesis circulares (cond).

```
if (cond)
   instr;
else
   instr';
```

← if de dos ramas con una sola sentencia en ambas.

```
if (cond)
{
  instr1;
  instr2;
  ...
  instrn;
}
else
  instr';
```

← if de 2 ramas con mas de una sentencia en la rama `true` y una sóla en la rama `false`.

```
if (cond)
  instr';
else
{
  instr1;
  instr2;
  ...
  instrn;
}
```

← if de 2 ramas con una sentencia en la rama `true` y mas de una sentencia en la rama `false`.

```
if (cond)
{
  instr1;
  instr2;
  ...
  instrn;
}
else
{
  instr'1;
  instr'2;
  ...
  instr'm;
}
```

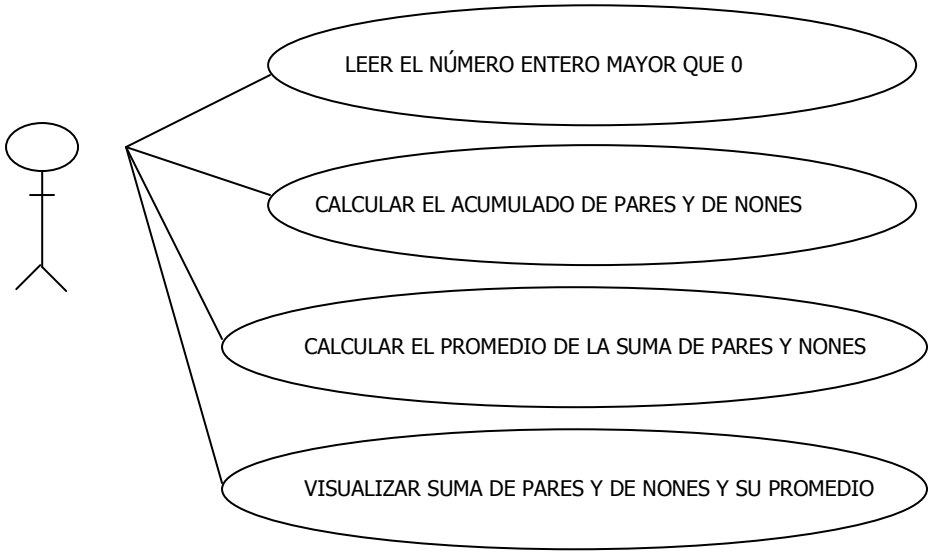
← if de 2 ramas con mas de una sentencia en ambas ramas.

Ejercicio 6.4.2
Un jugador requiere de un programa que le permita registrar números enteros positivos pares y nones, e irlos sumando por separado (pares con pares y nones con nones). Además quiere visualizar el promedio de la suma de los números pares registrados y el promedio de la suma de los números nones registrados. Desde luego que requiere visualizar también la suma acumulada de los pares y de los nones, así como del número de pares y de nones leídos. Se sabe que el jugador sólo ingresa hasta 20 números en cada ejecución del programa.

Solución :
El enunciado es la especificación de requerimientos si queremos compararlo con las etapas en el desarrollo de un programa, visto en la unidad 1. Así que de éste, debemos abstraer las tareas que nos permiten resolver –desarrollar- el programa. Es relativamente simple analizar que sólo tendremos una clase : `Jugador`. Este proceso de abstracción de tareas nos permitirán seguir con la abstracción de los métodos sobre la clase `Jugador`, ya que estos métodos deberán implementar las tareas previamente identificadas y mostradas en el diagrama de casos de uso. Este proceso que relaciona a las tareas con los métodos, constituye una parte de la etapa del análisis.

Bien, entonces las tareas que ilustraremos enseguida en el diagrama de casos de uso son :
- Lectura del número entero mayor que 0.
- Calcular la suma acumulada de pares y nones.
- Calcular el promedio de la suma de pares y el promedio de la suma de nones acumulados.
- Visualizar los pares y nones leídos, la suma de pares y de nones, además de su promedio.

El diagrama de casos de uso no es mas que una consecuencia de las tareas identificadas o abstraídas :



El diagrama de clases corresponde a una sola clase : `Jugador`.

JUGADOR
- _noPares : int - _noNones : int - _sumaPares : int - _sumaNones : int
+ Leer(num:int) : void + RetNoPares() : int + RetNoNones() : int + RetSumaPares() : int + RetSumaNones() : int

En el método `Leer()` vamos efectuar el cálculo de los atributos `_noPares`, `_noNones`, `_sumaPares` y `_sumaNones`. El promedio de los números pares y de los nones leídos lo obtendremos usando los métodos que retornan el número de pares y de nones leídos, en conjunto con los métodos que retornan la suma acumulada de pares y de nones leídos.

Bueno, pues ya estamos listos para empezar la construcción de la aplicación Windows C#, así que manos a la obra. Crea un nuevo proyecto y agrega en el archivo Form1.cs en la clase `Form1`, al objeto oJug perteneciente a la clase `Jugador`. Antes de definir a este objeto será bueno que agregues primero la clase `Jugador`, incluyendo a sus atributos y al constructor, según se te indica en el código siguiente.

```
class Jugador
{
     private int _noPares;
     private int _noNones;
     private int _sumaPares;
     private int _sumaNones;

     public Jugador()
     {
         _noPares = _noNones = _sumaPares = _sumaNones = 0;
     }
}
```

Asignación transitiva, pregúntale a tu maestro si no lo entiendes. Al final los 4 atributos son asignados al valor de 0.

La definición del objeto `oJug` de la clase `Jugador`, házla según el código siguiente :

```
public partial class Form1 : Form
{
    Jugador oJug = new Jugador();
    public Form1()
    {
        InitializeComponent();
    }
}
```

Para crear la interfase gráfica debemos pensar que vamos a leer al número entero solamente, así que requerimos de un componente TextBox. Además requeriremos de algunos componentes Label para visualizar los resultados : el número de pares y de nones leídos, la suma de pares y de nones, además de sus promedios. La figura #6.4.6 muestra a la interfase gráfica a la que paso a paso le daremos vida. El componente Label con propiedad name=`label7` sirve para indicarle al usuario la cantidad de números leídos sean pares o nones, de manera que pueda saber cuántos números puede aún teclear o ingresar al programa.

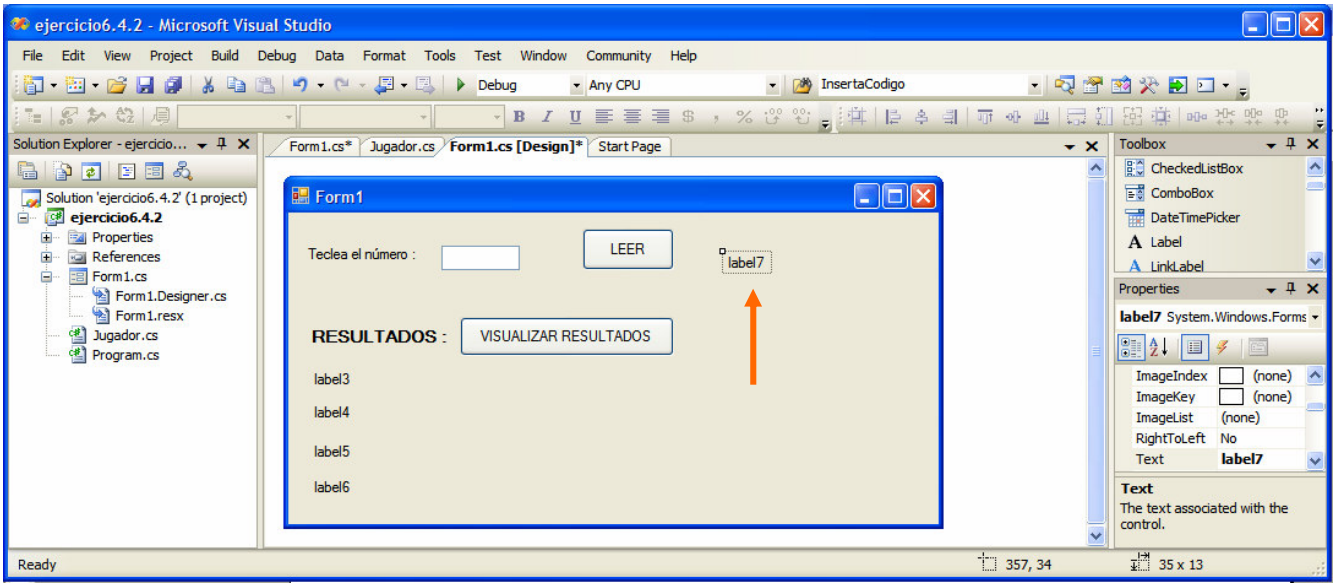


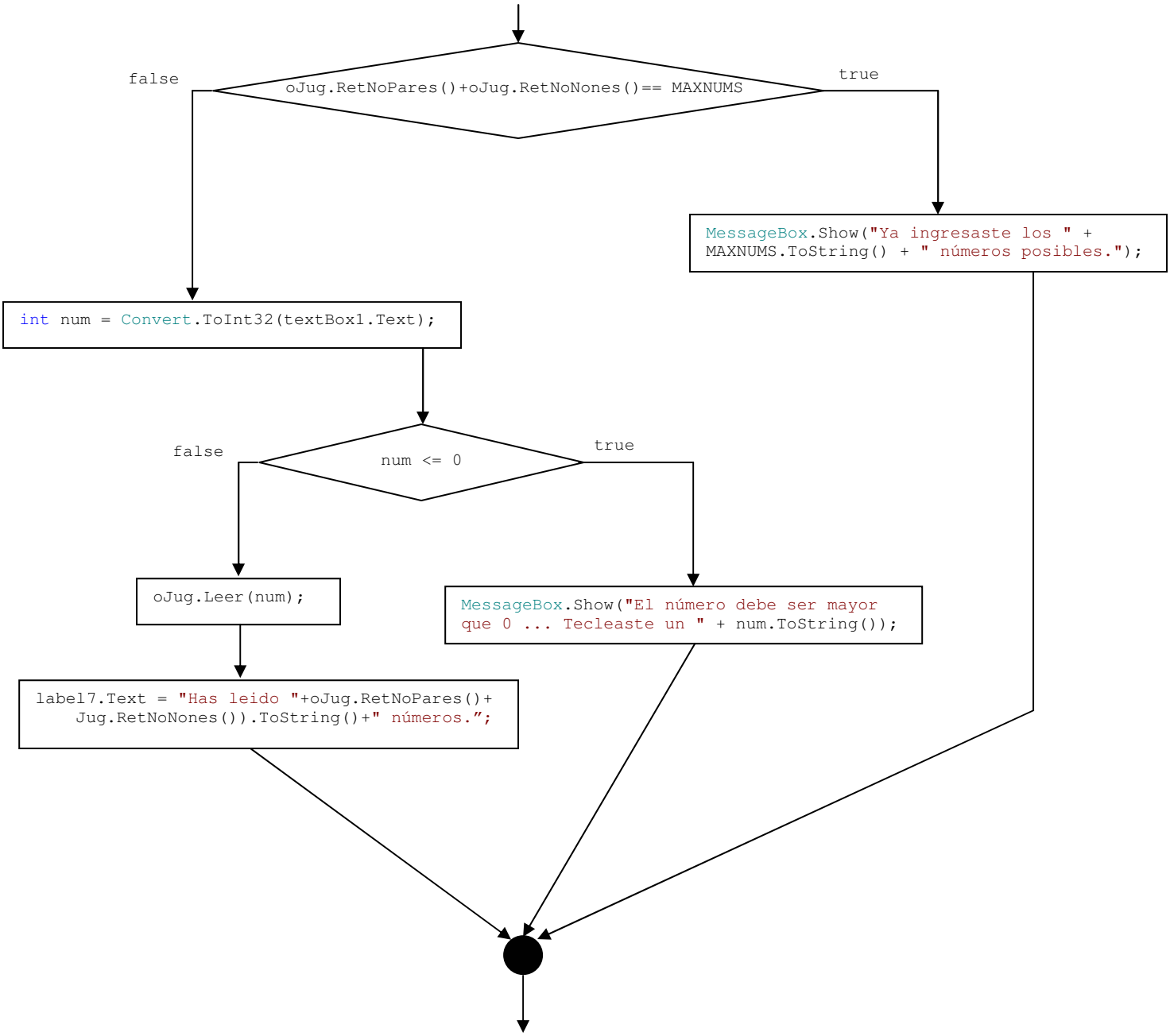
Fig. No. 6.4.6 Interfase gráfica para el ejercicio del Jugador.

Lo que sigue es codificar el método `button1_Click()` que es llamado cuando hacemos click sobre el botón `button1` con leyenda LEER. El diseño del código para este botón deberá incluir :

- Validación de la cantidad de números leídos sin importar sea par o non, de manera que no se lean mas de 20 números (repasa el enunciado).
- Validación sobre el número leído para aceptar sólo números enteros mayores que 0.
- Uso del mensaje `oJug.Leer()` para efectuar la lectura del número y el cálculo de pares-nones leídos, así como de la suma correspondiente sea par o non.
- Actualización del componente `label7` para visualización de la cantidad de números leídos sean pares, sean nones.

```
private void button1_Click(object sender, EventArgs e)
{
    if (oJug.RetNoPares() + oJug.RetNoNones() == MAXNUMS)
        MessageBox.Show("Ya ingresaste los " + MAXNUMS.ToString() + " números posibles.");
    else
    {
        int num = Convert.ToInt32(textBox1.Text);
        if (num <= 0)
           MessageBox.Show("El número debe ser mayor que 0 ... Tecleaste un " + num.ToString());
        else
        {
           oJug.Leer(num);
           label7.Text = "Has leido " + (oJug.RetNoPares() + oJug.RetNoNones()).ToString() + "
                                                                                    números.";
         }
    }
}
```

El flujo de ejecución del método `button1_Click()` inicia con un **if** de 2 ramas, y dentro de la rama `false` de este primer **if**, se encuentra la lectura del número entero que luego es validado para saber si es mayor que 0. Esta validación la efectúa un **if** de 2 ramas que se encarga de mostrar un mensaje al usuario si el número no es positivo, de lo contrario realiza el mensaje `oJug.Leer(num)` y la visualización de la cantidad de números leídos en el componente `label7`.



Antes de probar el código del `button1_Click()` deberás definir la constante MAXNUMS en la clase Form1, precisamente junto a la definición del objeto `oJug`.

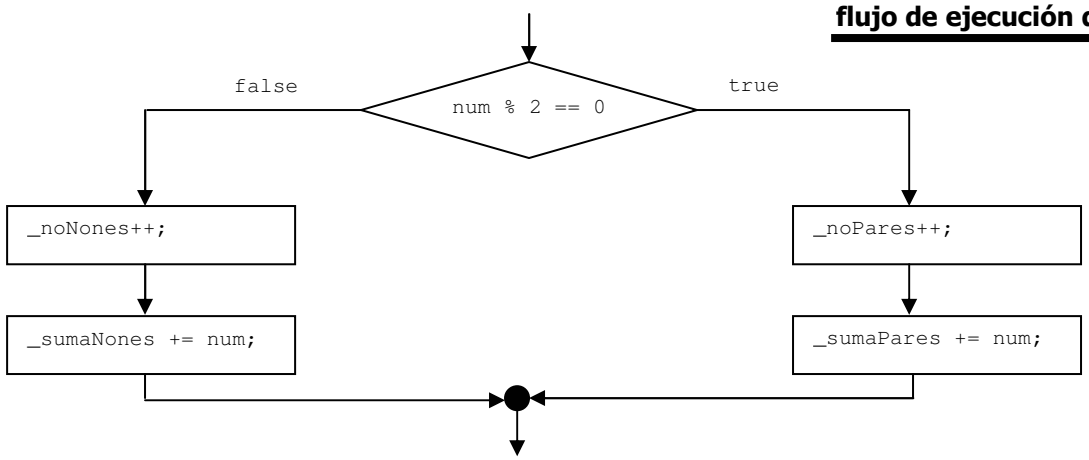
```
public partial class Form1 : Form
{
    Jugador oJug = new Jugador();
    const int MAXNUMS = 20;
    ...
    ...
```

Se nos ha olvidado que debemos definir los métodos `Leer()`, `RetNoPares()` y `RetNoNones()` en la clase `Jugador`, ya que se usan en el método `button1_Click()`.

```
public void Leer(int num)
{
    if (num % 2 == 0)
    {
        _noPares++;
        _sumaPares += num;
    }
    else
    {
        _noNones++;
        _sumaNones += num;
    }
}
public int RetNoPares()
{
    return _noPares;
}
public int RetNoNones()
{
    return _noNones;
}
```

Observemos que el método `Leer()` usa un **if** de 2 ramas para efectuar su tarea : incrementar y sumar los pares si el número leído es par, de lo contrario incrementa y suma los nones. Para saber si el número es par, es aplicado el operador residuo **%** a los operandos `num` y `2`. Recordemos que el operador residuo **%** retorna el residuo de dividir sus operandos. El flujo de ejecución para el método Leer() es el que se muestra a continuación :

flujo de ejecución del método `Leer()`



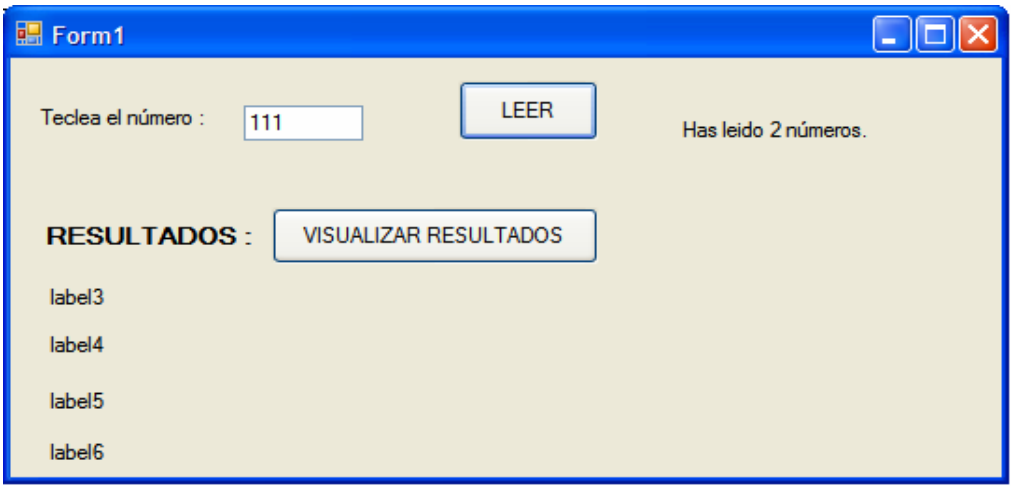


Fig. No. 6.4.7
Ejecución de la aplicación y lectura de 2 números.

Resta incluir el código del método `button2_Click()` del botón `button2` que es el encargado de visualizar los resultados. La visualización no requiere de mayor explicación, ya que esta operación la hemos efectuado de manera repetitiva desde los ejercicios de la unidad 5 y de la presente unidad. A continuación se muestra el código del botón el cual deberás de teclear según se indica.

```
private void button2_Click(object sender, EventArgs e)
{
     label3.Text = "Se leyeron " + oJug.RetNoPares().ToString() + " números pares, su suma fue " +
                                                                    oJug.RetSumaPares().ToString();
     if (oJug.RetNoPares() == 0)
         label4.Text = "NO HAY PROMEDIO YA QUE NO SE LEYERON NUMEROS PARES.";
     else
         label4.Text = "EL PROMEDIO DE NUMEROS PARES ES = " + (oJug.RetSumaPares() / (float)
                                                                    oJug.RetNoPares()).ToString();
     label5.Text = "Se leyeron " + oJug.RetNoNones().ToString() + " números nones, su suma fue " +
                                                                    oJug.RetSumaNones().ToString();
     if (oJug.RetNoNones() == 0)
         label6.Text = "NO HAY PROMEDIO YA QUE NO SE LEYERON NUMEROS NONES.";
     else
         label6.Text = "EL PROMEDIO DE NUMEROS NONES ES = " + (oJug.RetSumaNones() / (float)
                                                                    oJug.RetNoNones()).ToString();
}
```

En este código también fue necesario utilizar 2 **if**'s de 2 ramas, para validar que el promedio exista, de lo contrario nuestra aplicación Windows tendría un error de ejecución cuando no se hayan leído números pares o números nones o ambos. Recordemos que la computadora no puede dividir por 0.
Falta agregar en la clase `Jugador` los métodos `RetSumaPares()` y `RetSumaNones()`. Házlo de acuerdo al siguiente código.

```
public int RetSumaPares()
{
    return _sumaPares;
}
public int RetSumaNones()
{
    return _sumaNones;
}
```

Como ejercicio dibuja el flujo de ejecución para el código de este método `button2_Click()` de manera que observes gráficamente dicho flujo. La figura #6.4.8 muestra la ejecución de la aplicación para 2 números pares leídos el 10 y el 20, y ningún número non.

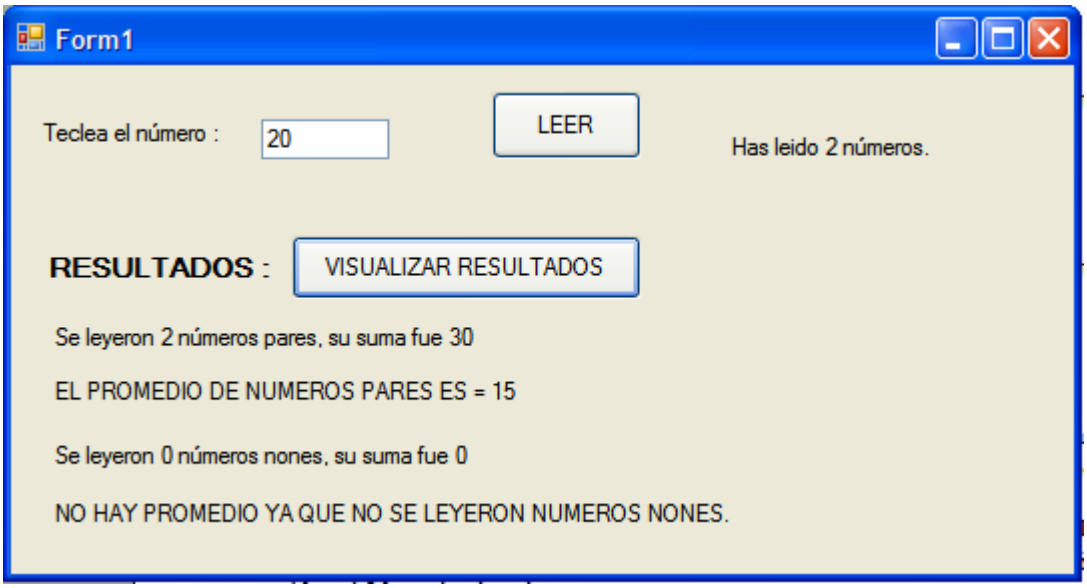
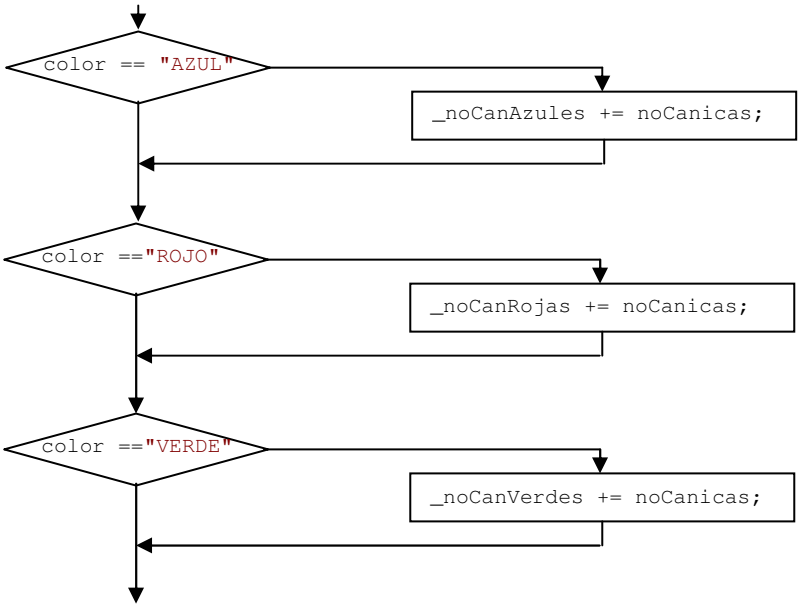


Fig. No. 6.4.8 Ejecución para los pares 10 y 20 leídos y ningún número non leído.

6.4.3 Selectiva anidada if-else-if.

Realmente esta sentencia es un conjunto de **if's** de 2 ramas, donde uno de ellos se encuentra "anidado" en la rama `false` de otro. Generalmente el flujo de ejecución "sale" de la estructura de los **if's** anidados por la rama `true` de uno de ellos. Veamos un ejemplo : en el ejercicio 6.4.1 el método `Leer()` de la clase `Nino` tiene 3 **if's** de una rama según lo recordamos enseguida :



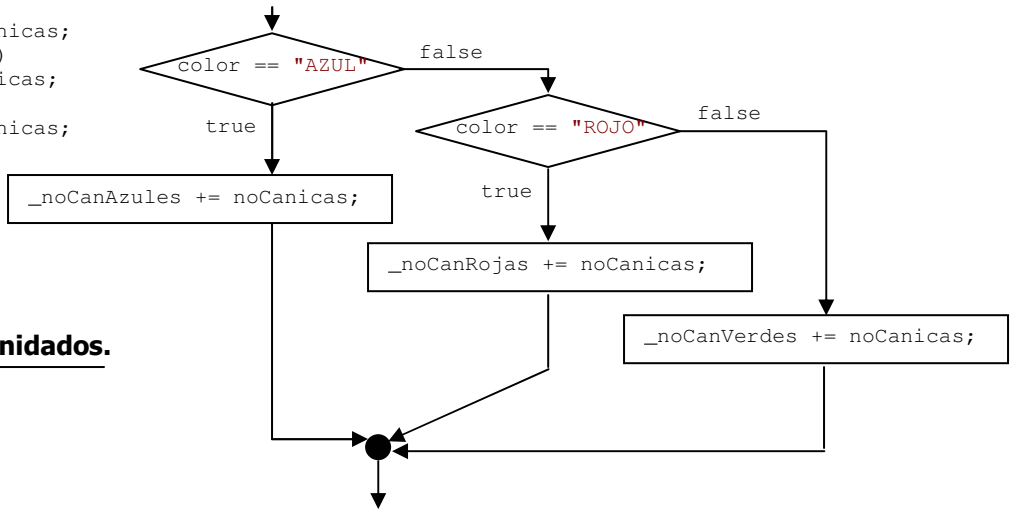
También discutimos que si el color es "AZUL", no tendríamos por que probar los siguientes 2 **if's** que prueban si el color es "ROJO" o si es "VERDE". Este es un caso típico del uso de **if's** anidados, donde sólo la rama `true` de uno de los **if's** es ejecutado. Cambiemos los **if's** de una rama en el método `Leer()` que nos está ocupando por un **if** anidado.

```
public void Leer(int noCanicas, string color)
{
     if (color == "AZUL")
         _noCanAzules += noCanicas;
     if (color == "ROJO")
         _noCanRojas += noCanicas;
     if (color == "VERDE")
         _noCanVerdes += noCanicas;
}
```

Lo cambiamos por los **if** anidados :

```
public void Leer(int noCanicas, string color)
{
     if (color == "AZUL")
         _noCanAzules += noCanicas;
     else if (color == "ROJO")
         _noCanRojas += noCanicas;
     else
         _noCanVerdes += noCanicas;
}
```

Flujo de ejecución de los if anidados.



Sobre este nuevo código con **if** anidados podemos decir que :

- El flujo de ejecución inicia con la prueba de la condición si el color es "AZUL", si es verdad se realiza la asignación del número de canicas leídas al atributo `_noCanAzules`, y el flujo se sigue con las sentencias siguientes al conjunto de **if's** anidados. Es decir, ya no se prueba la condición si el color de las canicas es el "ROJO".
- Si las canicas no tienen el color "AZUL" entonces el flujo sigue con la prueba de la condición si el color es "ROJO", y en el caso que sea verdad, efectuar la asignación del atributo `_noCanRojas`.
- En la rama false del **if** anidado correspondiente al que prueba si el color es "ROJO", no es necesario efectuar otro **if** – prueba si el color es "VERDE"- ya que el color ha sido previamente validado antes de entrar al método `Leer()` de la clase `Nino`.

Realiza las adecuaciones anteriores al ejercicio 6.4.1 y observa que el programa funciona de manera correcta. Notemos que con los **if** anidados el flujo de ejecución es mas claro y eficiente.

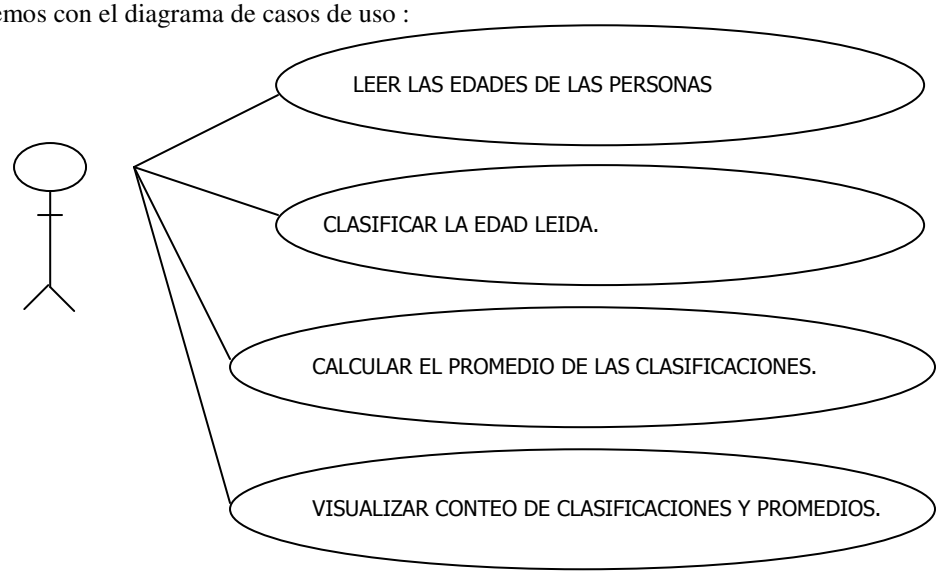
Ejercicio 6.4.3

Un encuestador necesita leer hasta 30 edades de personas con el fin de clasificarlas de acuerdo a la tabla :

edad	clasificación
1-11	niño
12-17	adolescente
18-52	joven
53-...	adulto

La aplicación deberá visualizar cuantas de las edades pertenecen a personas que son niños, cuantas a adolescentes, cuantas a jóvenes y cuantas a adultos. También deberá visualizar el promedio de las edades de cada clasificación. Utiliza objetos contadores.

Iniciemos con el diagrama de casos de uso :



Usaremos 4 objetos contadores : uno para conteo de los níños, otro para la cuenta de los adolescentes, otro para los jóvenes y el cuarto para los adultos. La clase `Contador` tiene sólo un atributo : `_cuenta`, sus métodos son los mostrados en el diagrama de clase.

JUGADOR
- _cuenta : int
+ Incr(num:int) : void + RetCuenta() : int

El método `Incr()` para objetos de la clase `Contador`, incrementa en 1 al atributo `_cuenta`. También puede recibir un parámetro para que en vez de incrementar en 1, incremente el valor de `_cuenta` tantas veces como sea el valor del parámetro recibido. De esta manera, un objeto `Contador` puede comportarse como un objeto `Sumador`. Con esta característica en cuenta, la suma de las edades para cada clasificación que nos permitirán obtener los promediosde las edades, las llevaremos a cabo usando objetos de la clase `Contador`.
Por lo tanto, definiremos otros 4 Contadores : uno para llevar la suma de las edades de los niños, otro para la suma de las edades de los adolescentes, otro para la suma de las edades de los jóvenes y por último, otro para la suma de las edades de los adultos.
También usaremos un contador que nos permita llevar el conteo de cuantas edades hemos leído, de tal forma que con este conteo validaremos no leer mas de 30 edades de personas. El método `RetCuenta()` sirve para devolver el valor del conteo para el objeto `Contador`.
Empecemos por crear un nuevo proyecto con la interfase gráfica que se muestra en la figura 6.4.9.

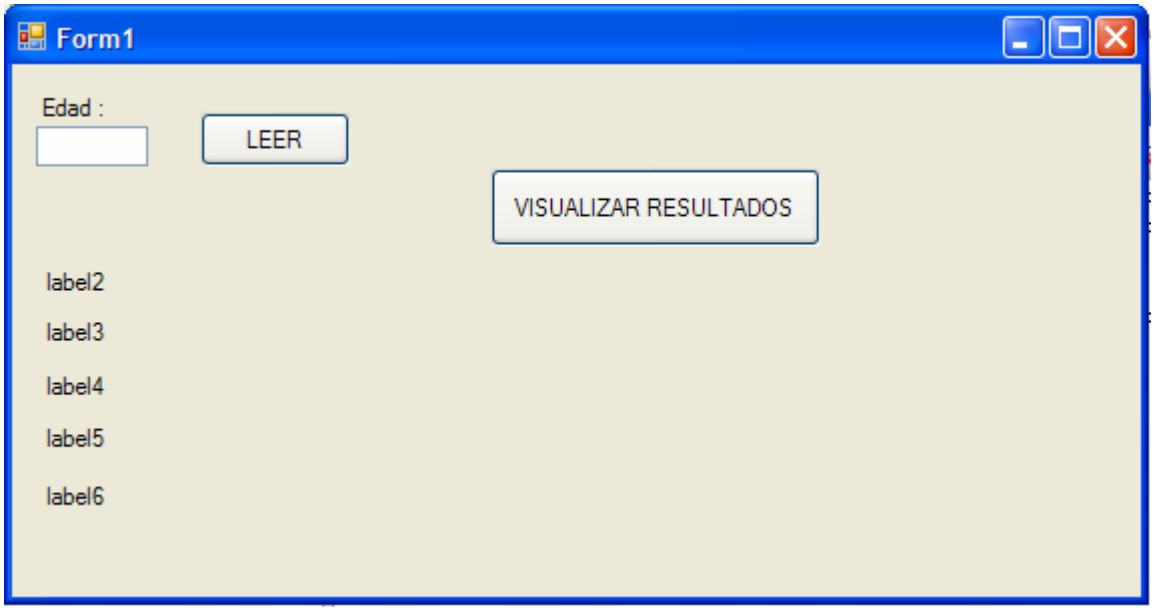


Fig. No. 6.4.9 Interfase gráfica del ejercicio de clasificación de edades.

Los componentes Label sirven para visualizar el conteo de todas las edades leídas, de niños, de adolescentes, de jóvenes y de adultos. En estos mismos Label's serán visualizados los promedios de las edades por cada clasificación.
Ahora añadimos la clase `Contador` al proyecto, con su atributo `_cuenta`, y los métodos constructor y `RetCuenta()`.

```
class Contador
{
    private int _cuenta;

    public Contador()
    {
        _cuenta = 0;
    }
    public int RetCuenta()
    {
        return _cuenta;
    }
}
```

El método `Incr()` lo codificaremos hasta que veamos el código del método `button1_Click()` donde se efectúa la lectura de la edad y se clasifica dicha lectura.
Enseguida debemos agregar los objetos que tendrán la tarea de llevar el conteo de cuantos niños, cuantos adolescentes, cuantos jóvenes y cuantos adultos han sido leídos. Además del contador del total de edades leídas. Recordemos que la definición de estos objetos la hacemos dentro de la clase `Form1` que reside dentro del archivo Form1.cs. Agrega los objetos mencionados segun lo indica el código a continuación. También definiremos la constante MAXNUMEDADES que contiene el número de edades máximo que pueden ser ingresadas a la aplicación.

```
public partial class Form1 : Form
{
      const int MAXNUMEDADES = 30;
      Contador oTot = new Contador();
      Contador oNinos = new Contador();
      Contador oAdoles = new Contador();
      Contador oJovenes = new Contador();
      Contador oAdultos = new Contador();

      public Form1()
      {
          InitializeComponent();
      }
}
```

El constructor de la clase `Contador` se ejecutará 5 veces, una vez para cada objeto definido, de manera que el atributo `_cuenta` para cada objeto iniciará en 0. ¿Dónde es que el constructor es ejecutado? ¿Cuándo?. Contesta las preguntas y coméntalas con tu profesor o con tus compañeros.
En tiempos de diseño teclea en la propiedad Text del componente `label2` la cadena "Se han leido 0 edades". La interfase gráfica ahora se verá como la mostrada en la figura 6.4.10.

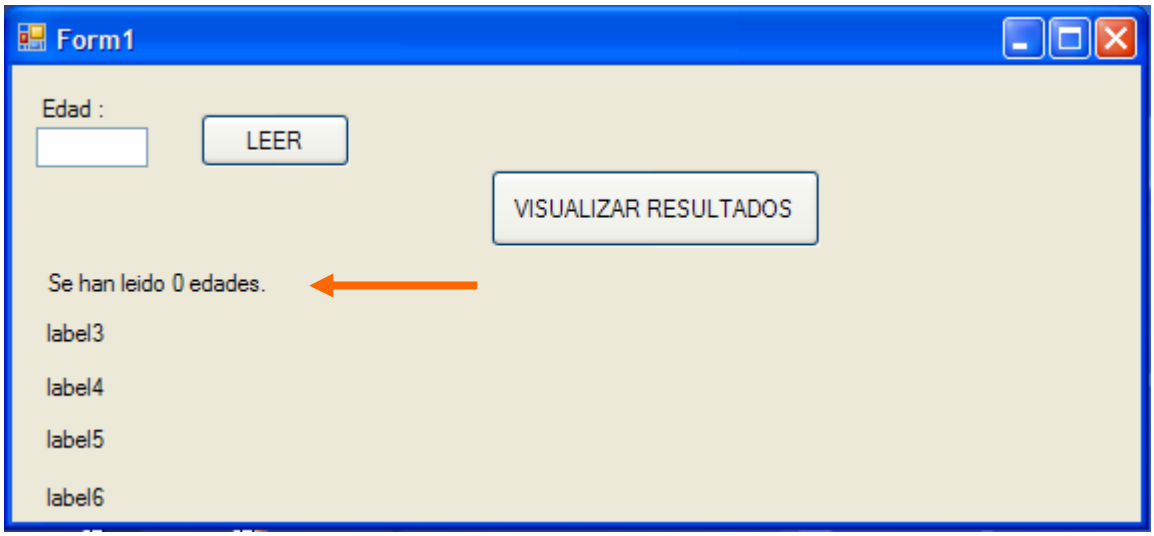
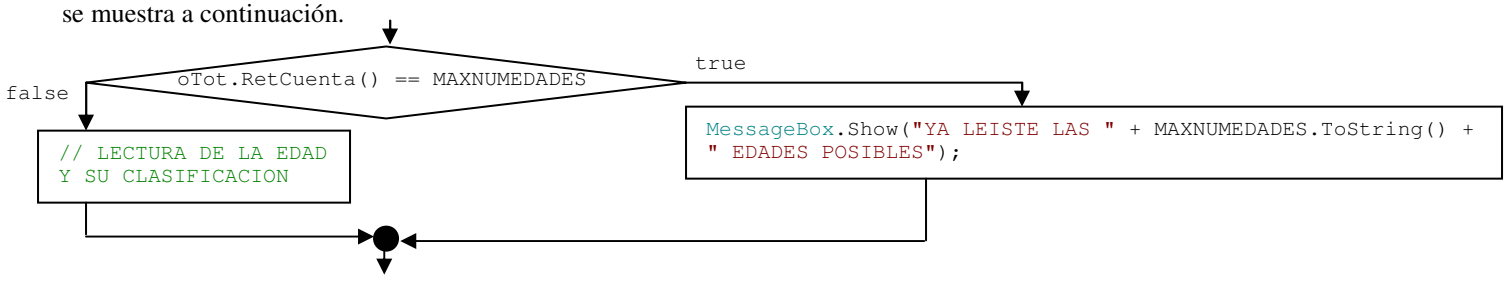


Fig. No. 6.4.10 Interfase gráfica modificadaen el label2.

Sigamos con el código del método `button1_Click()` que es el encargado de efectuar la lectura de la edad, para luego clasificarla. Inicialmente tecleemos el código que valida que todavía no se hayan leído las 30 edades de personas.

```
private void button1_Click(object sender, EventArgs e)
{
      if (oTot.RetCuenta() == MAXNUMEDADES)
            MessageBox.Show("YA LEISTE LAS " + MAXNUMEDADES.ToString() + " EDADES POSIBLES");
      else
      {
            // LECTURA DE LA EDAD Y SU CLASIFICACION
      }
}
```

El mensaje `oTot.RetCuenta()` retorna el conteo del objeto `oTot`, el cual es un contador que almacena el número de edades leídas que se han clasificado. Este conteo es comparado con la constante entera MAXNUMEDADES de manera que si es igual, significa que ya se han leído y clasificado las 30 edades posibles. El flujo de ejecución del método button1_Click() se muestra a continuación.

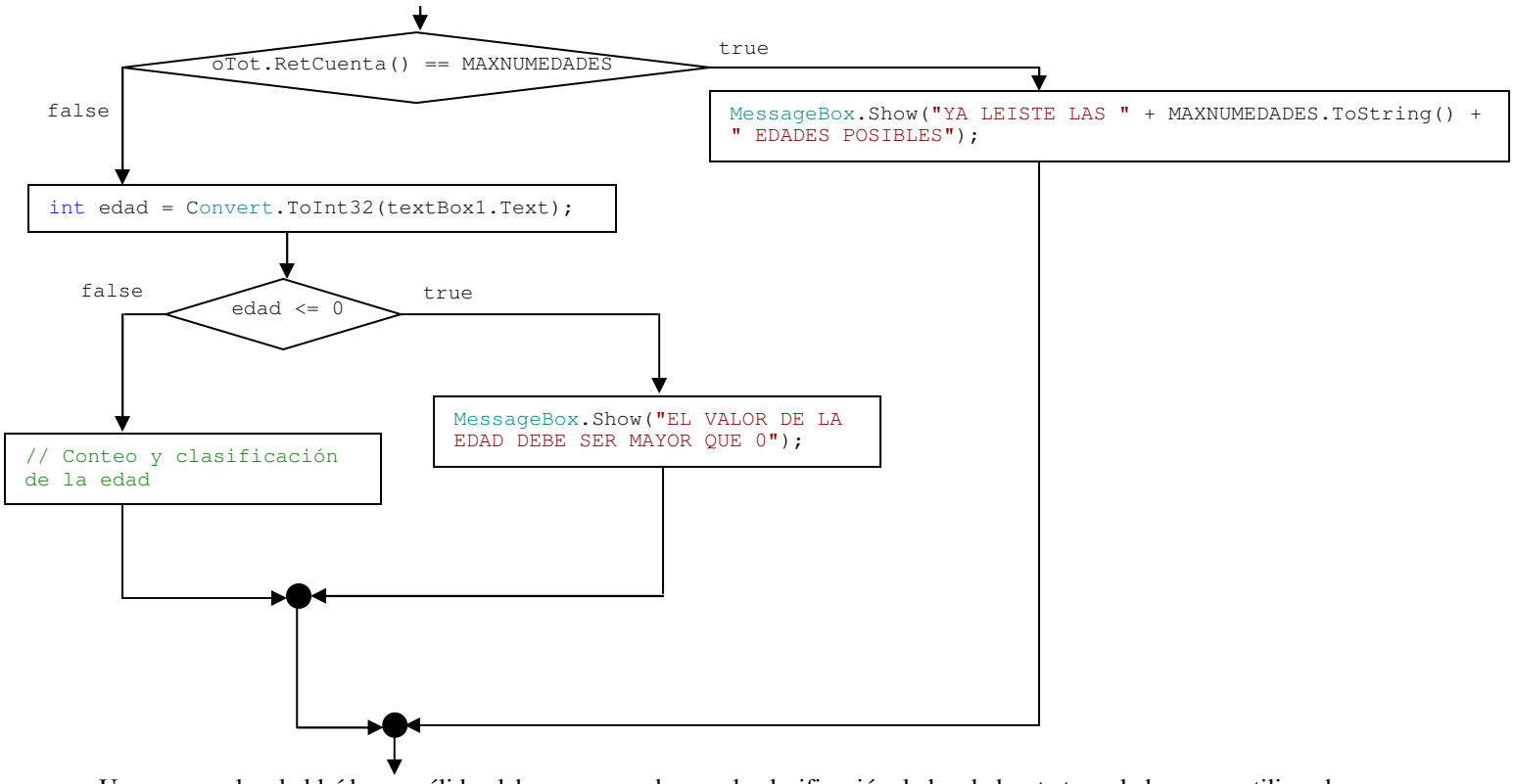


Una vez que validamos el número de edades leídas y clasificadas, seguimos con la rama `false` del **if** de 2 ramas donde es efectuada la lectura de la edad y también es realizada la clasificación. Entonces primero leemos la edad ingresada en el componente TextBox `textBox1`, asignándola a una variable entera a la que llamamos `edad`. Luego corresponde efectuar la validación de la edad leída –sea mayor que 0-. Agreguemos el código que se indica dentro de la rama `false` del **if** previamente tecleado.

```
private void button1_Click(object sender, EventArgs e)
{
    if (oTot.RetCuenta() == MAXNUMEDADES)
        MessageBox.Show("YA LEISTE LAS " + MAXNUMEDADES.ToString() + " EDADES POSIBLES");
    else
    {
        int edad = Convert.ToInt32(textBox1.Text);
        if (edad <= 0)
            MessageBox.Show("EL VALOR DE LA EDAD DEBE SER MAYOR QUE 0");
        else
        {
           // Conteo y clasificación de la edad
        }

    }
}
```

Notemos que se requiere de un **if** de 2 ramas para efectuar la validación de la edad leída. El flujo de ejecución ahora es :

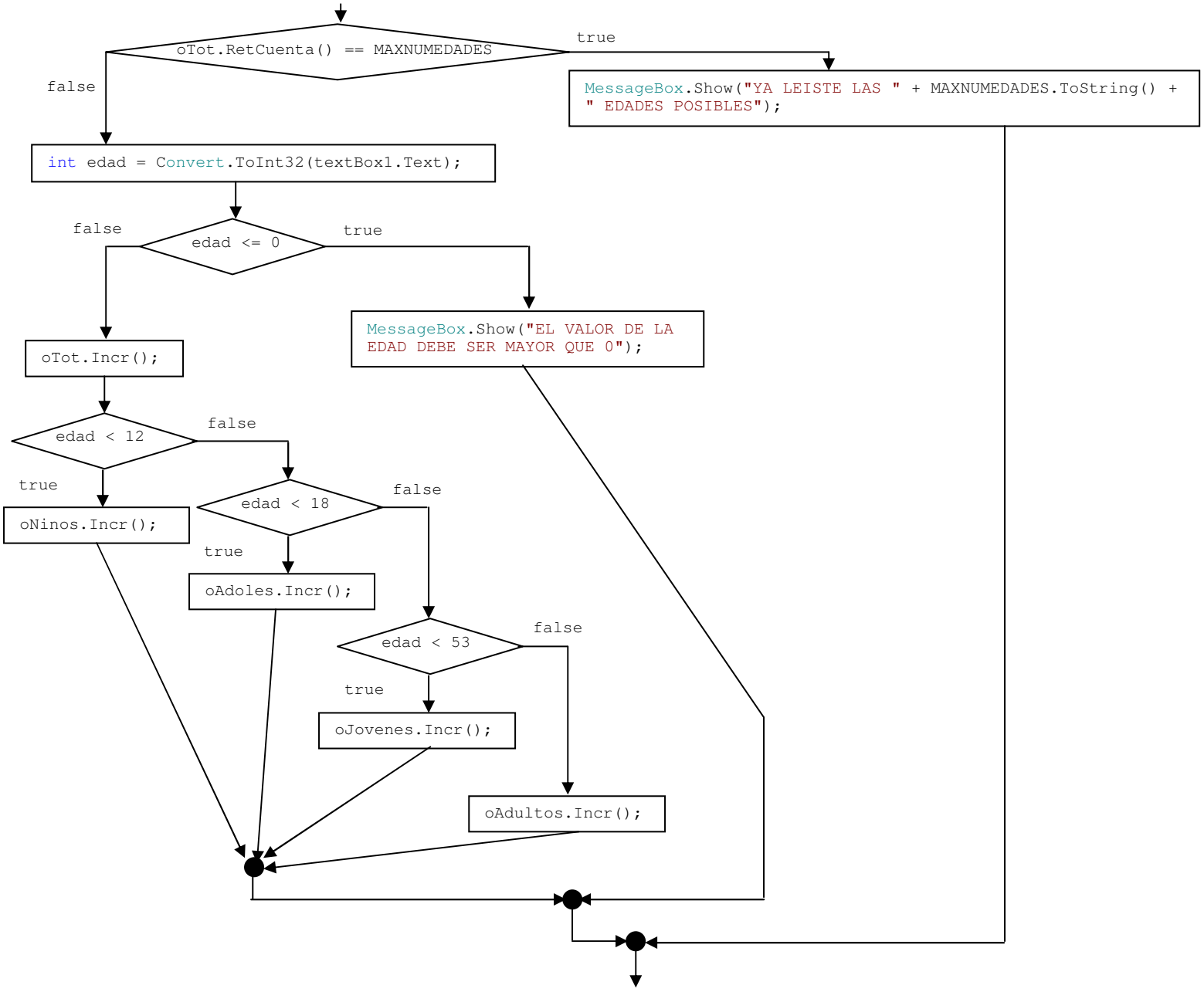


Una vez que la edad leída sea válida, debemos proceder con la clasificación de la edad. esta tarea la haremos utilizando un **if** anidado, donde las condiciones en los **if's** que lo componen, aprovechen el hecho de que la edad ha sido validada, es decir que es un entero mayor que 0. Antes de clasificar la edad, incrementaremos el conteo del contador `oTot`. El código que se lista a continuación efectúa el conteo de la edad, además de su clasificación. La clasificación de la edad no es otra cosa mas que el incremento del contador adecuado, sea `oNinos` para la edad entre 1 y 11 inclusive, `oAdoles` para la edad de 12 a 17, `oJovenes` para edades entre 18 y 52, y `oAdultos` para el intervalo de 53 en adelante.

```
private void button1_Click(object sender, EventArgs e)
{
     if (oTot.RetCuenta() == MAXNUMEDADES)
          MessageBox.Show("YA LEISTE LAS " + MAXNUMEDADES.ToString() + " EDADES POSIBLES");
     else
```

```
        {
            int edad = Convert.ToInt32(textBox1.Text);
            if (edad <= 0)
                MessageBox.Show("EL VALOR DE LA EDAD DEBE SER MAYOR QUE 0");
            else
            {
                oTot.Incr();
                if (edad < 12)
                    oNinos.Incr();
                else if (edad < 18)
                    oAdoles.Incr();
                else if (edad < 53)
                    oJovenes.Incr();
                else
                    oAdultos.Incr();
            }
        }
}
```

¿Por qué las condiciones sólo prueban el valor superior y no el inferior para cada rango?. Contesta la pregunta y discútela con tu profesor. El flujo de ejecución para el método `button1_Click()` ahora es :



Antes de ejecutar la aplicación debemos añadir el método `Incr()` a la clase `Contador`. Házlo según se te indica en el código a continuación.

```
class Contador
{
    private int _cuenta;

    public Contador()
    {
        _cuenta = 0;
    }
    public int RetCuenta()
    {
        return _cuenta;
    }
    public void Incr()
    {
        _cuenta++;
    }
}
```

Observa que existe una falta de retroalimentación al usuario cuando es leída una edad válida. Esta consiste, en que a pesar de que se leen las edades, el componente `label2` sigue con la leyenda "Se han leido 0 edades", ver la figura #6.4.11.

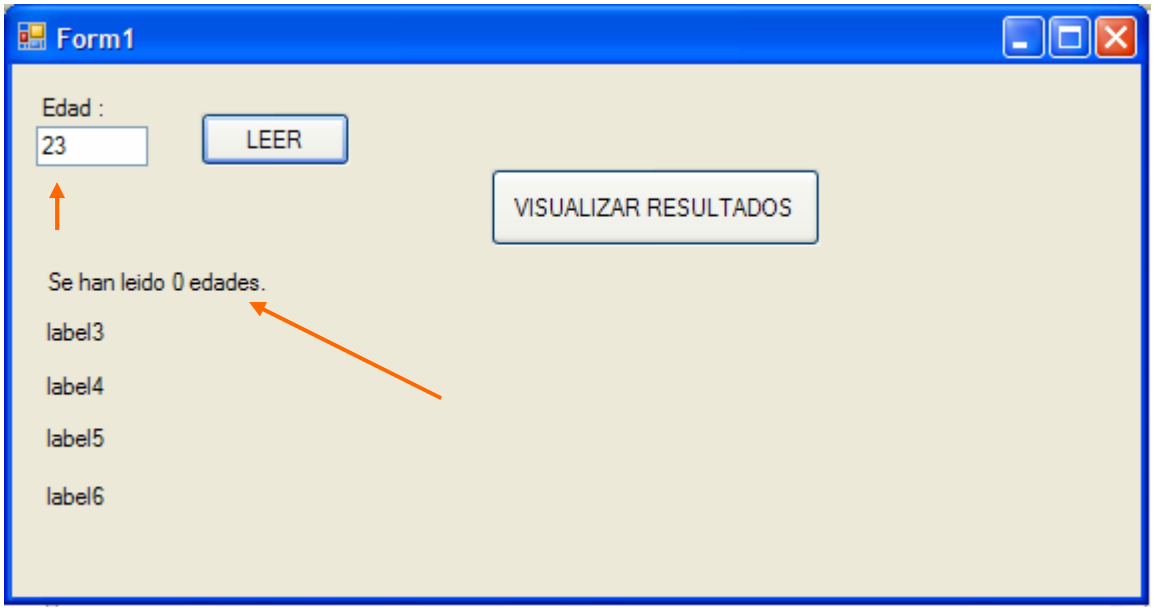


Fig. No. 6.4.11 `label2` sigue sin modificarse a pesar de haber leído la edad 23.

Agrega la instrucción que modifica la proiedad Text del componente `label2`, después de haber validado el valor de la edad ingresada, y una vez que hayamos incrementado el conteo del objeto `oTot`.

```
private void button1_Click(object sender, EventArgs e)
{
    if (oTot.RetCuenta() == MAXNUMEDADES)
        MessageBox.Show("YA LEISTE LAS " + MAXNUMEDADES.ToString() + " EDADES POSIBLES");
    else
    {
        int edad = Convert.ToInt32(textBox1.Text);
        if (edad <= 0)
            MessageBox.Show("EL VALOR DE LA EDAD DEBE SER MAYOR QUE 0");
        else
        {
            oTot.Incr();
            label2.Text = "Se han leído " + oTot.RetCuenta().ToString() + " edades.";
            if (edad < 12)
                oNinos.Incr();
```

```
            else if (edad < 18)
                    oAdoles.Incr();
            else if (edad < 53)
                    oJovenes.Incr();
            else
                    oAdultos.Incr();
        }
    }
}
```

Como ejercicio modifica el flujo de ejecución del método `button1_Click()` de manera que contenga la asignación a la propiedad Text del componente `label2`, que hemos añadido.
Ejecutemos de nuevo la aplicación y observemos que ya existe la retroalimentación al usuario en cuanto al número de edades leídas y clasificadas, ver figura #6.4.12.

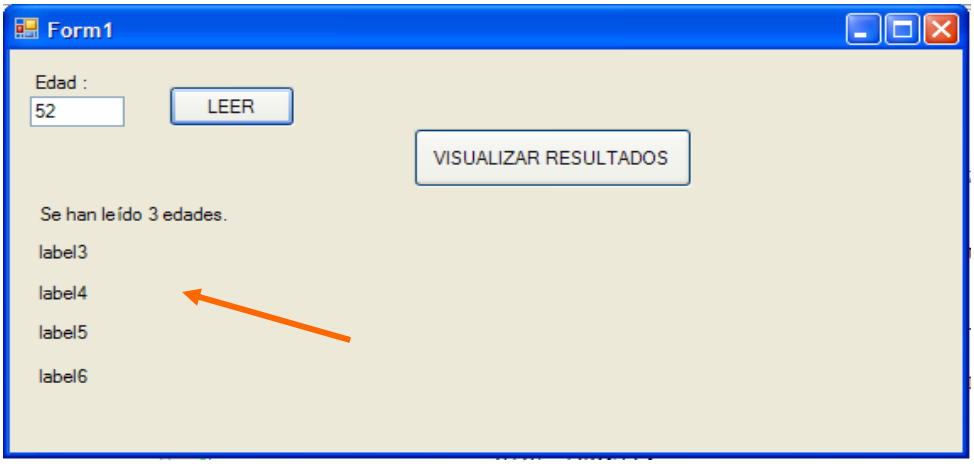


Fig. No. 6.4.12 `label2` muestra las edades leídas y clasificadas.

Proseguimos con la visualización de los objetos `oNinos`, `oAdoles`, `oJovenes` y `oAdultos` que tienen el conteo de la clasificación de las edades leídas. El código lo tecleamos en el método `button2_Click()` :

```
private void button2_Click(object sender, EventArgs e)
{
    label3.Text = oNinos.RetCuenta().ToString() + " son niños.";
    label4.Text = oAdoles.RetCuenta().ToString() + " son adolescentes.";
    label5.Text = oJovenes.RetCuenta().ToString() + " son jóvenes.";
    label6.Text = oAdultos.RetCuenta().ToString() + " son adultos.";
}
```

La figura #6.4.13 muestra la ejecución de la aplicación para una lectura de 6 edades válidas.

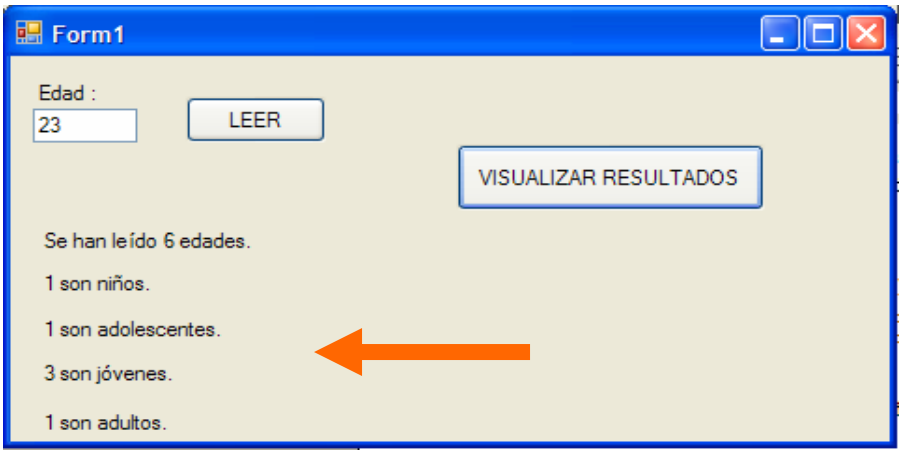


Fig. No. 6.4.13 Visualización de los contadores de edades.

El ejercicio ya casi está terminado, sólo nos falta implementar el cálculo de los promedios de las edades por clasificación, es decir, el promedio de las edades de los niños leídos, de los adolescentes, de los jóvenes y de los adultos. para lograr estos promedios necesitamos llevar la suma de las edades por grupo de clasificación y dividir entre la cuenta de cada objeto `oNinos`, `oAdoles`, `oJovenes` y `oAdultos`.

Agreguemos otros contadores –sumadores- que en lugar de incrementar en 1 el atributo `_cuenta`, reciban un parámetro en nuestro caso la edad leída, y que incrementen su atributo `_cuenta` por el valor del parámetro recibido. Así llevaremos la suma de las edades leídas por grupo de clasificación. Iniciemos por añadir los contadores que servirán para llevar la suma de las edades según lo hemos comentado, dentro de la clase `Form1`.

```
public partial class Form1 : Form
{
      const int MAXNUMEDADES = 30;
      Contador oTot = new Contador();
      Contador oNinos = new Contador();
      Contador oAdoles = new Contador();
      Contador oJovenes = new Contador();
      Contador oAdultos = new Contador();
      Contador oSumaNinos = new Contador();
      Contador oSumaAdoles = new Contador();
      Contador oSumaJovenes = new Contador();
      Contador oSumaAdultos = new Contador();
```

Ahora lo que tenemos que hacer es incluir en el **if** anidado del método `button1_Click()`, el mensaje a cada uno de los objetos que hemos incluido `oSumaNinos`, `oSumaAdoles`, `oSumaJovenes` y `oSumaAdultos`. Hagámoslo de acuerdo al código siguiente :

```
private void button1_Click(object sender, EventArgs e)
{
    if (oTot.RetCuenta() == MAXNUMEDADES)
        MessageBox.Show("YA LEISTE LAS " + MAXNUMEDADES.ToString() + " EDADES POSIBLES");
    else
    {
        int edad = Convert.ToInt32(textBox1.Text);
        if (edad <= 0)
            MessageBox.Show("EL VALOR DE LA EDAD DEBE SER MAYOR QUE 0");
        else
        {
           oTot.Incr();
           label2.Text = "Se han leído " + oTot.RetCuenta().ToString() + " edades.";
           if (edad < 12)
           {
               oNinos.Incr();
               oSumaNinos.Incr(edad);
           }
           else if (edad < 18)
           {
               oAdoles.Incr();
               oSumaAdoles.Incr(edad);
           }
           else if (edad < 53)
           {
               oJovenes.Incr();
               oSumaJovenes.Incr(edad);
           }
           else
           {
               oAdultos.Incr();
               oSumaAdultos.Incr(edad);
           }
       }
}
```

Primeramente debemos observar que requerimos de añadir las llaves en las ramas del **if** anidado, ya que ahora debemos ejecutar 2 sentencias.

Otra cuestión que nos produce asombro es el hecho de que estamos utilizando el método `Incr()` pero ahora con un parámetro –la edad leída-. C# permite el concepto de sobrecarga de métodos, el cual permite usar el mismo nombre para 2 o mas métodos que deben diferir en al menos uno de los 2 criterios siguientes :

- número de parámetros no igual, ó bien
- si el número de parámetros es el mismo, entonces deben diferir en el tipo de al menos uno de ellos.

En nuestro caso se cumple el primer criterio, ya que un método `Incr()` el que ya tenemos definido, no tiene parámetros y el método `Incr(int)` que estamos usando para la suma de las edades por clasificación, tiene un parámetro. Bien, entonces debemos de agregar al nuevo método `Incr(int)` en la clase `Contador`.

```
class Contador
{
    private int _cuenta;

    public Contador()
    {
        _cuenta = 0;
    }
    public int RetCuenta()
    {
        return _cuenta;
    }
    public void Incr()
    {
        _cuenta++;
    }
    public void Incr(int incremento)
    {
        _cuenta+=incremento;
    }
}
```

Sobrecarga del método `Incr()` en la clase `Contador`.

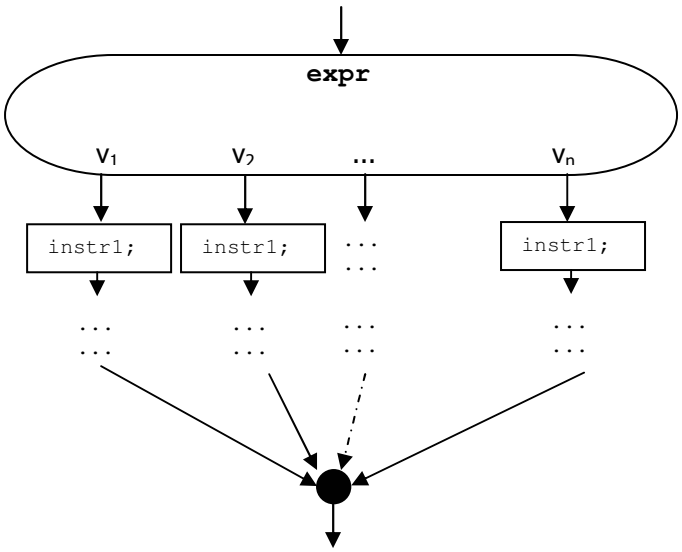
Ya que hemos hecho la suma de las edades, ahora debemos de añadir el código que servirá para visualizar los promedios. Este código lo insertamos en el método `button2_Click()`, únicamente teniendo la precaución de no efectuar una división por 0. La división por 0 nos produce un error en tiempos de ejecución, y en nuestro caso se podría presentar cuando el número de edades leídas para una determinada clasificación es 0. Usaremos **if's** de 2 ramas para efectuar la validación de manera que no dividamos por 0 al calcular los promedios.

```
private void button2_Click(object sender, EventArgs e)
{
     label3.Text = oNinos.RetCuenta().ToString() + " son niños. Promedio de edades = ";
     if (oNinos.RetCuenta() == 0)
         label3.Text += "NO HAY PROMEDIO.";
     else
         label3.Text += (oSumaNinos.RetCuenta() / (float)oNinos.RetCuenta()).ToString();
     label4.Text = oAdoles.RetCuenta().ToString() + " son adolescentes. Promedio de edades = ";
     if (oAdoles.RetCuenta() == 0)
         label4.Text += "NO HAY PROMEDIO.";
     else
         label4.Text += (oSumaAdoles.RetCuenta() / (float)oAdoles.RetCuenta()).ToString();
     label5.Text = oJovenes.RetCuenta().ToString() + " son jóvenes. Promedio de edades = ";
     if (oJovenes.RetCuenta() == 0)
         label5.Text += "NO HAY PROMEDIO.";
     else
         label5.Text += (oSumaJovenes.RetCuenta() / (float)oJovenes.RetCuenta()).ToString();
     label6.Text = oAdultos.RetCuenta().ToString() + " son adultos. Promedio de edades = ";
     if (oAdultos.RetCuenta() == 0)
         label6.Text += "NO HAY PROMEDIO.";
     else
         label6.Text += (oSumaAdultos.RetCuenta() / (float)oAdultos.RetCuenta()).ToString();
}
```

Notemos que hemos utilizado moldes `(float)` para efectuar la conversión explícita del entero que retorna el mensaje correspondiente : `oNinos.RetCuenta()`, `oAdoles.RetCuenta()`, `oJovenes.RetCuenta()`, `oAdultos.RetCuenta()`.
Ejecutemos la aplicación Windows C# para observar los resultados. Dibuja el flujo de ejecución del método `button2_Click()` con las modificaciones que incluyen los 4 **if's** de 2 ramas.

6.4.4 Selectiva múltiple, switch-case-default.

Un **if** anidado es efectivamente una sentencia compuesta que hace las funciones de una selección múltiple. Existe una sentencia en C# que permite realizar una selección múltiple, la sentencia **switch**. Una sentencia de selección múltiple permite controlar el flujo de ejecución del programa, de forma que siga por n posibles caminos diferentes.
La sentencia **switch** prueba el valor de una expresión entera, booleana, enumerada, cadena, caracter, y establece por medio de sentencias **case** caminos de flujo de ejecución para los diferentes valores que pueda tomar la expresión. La sentencia **switch** tiene el flujo de ejecución según se muestra :



Dependiendo del valor de la expresión `expr`, el flujo de ejecución puede irse por el lado del valor v_1, del valor v_2, hasta el camino del valor v_n. La sentencia **switch** se implementa :

```
switch(expr)
{
  case v1 :
    instr11;
    ...
    break;
  case v2 :
    instr21;
    ...
    break;
  ...
  ...
  case vn :
    instrn1;
    ...
    break;
 [ default :
    instr;
    ...
    break; ]
}
```

El uso de **default** es opcional.

Las consideraciones que debemos tener en cuenta en el **switch** son :

- La expresión se encierra entre paréntesis.
- Después de escribir el valor v_i debe ser agregado el caracter (**:**).
- El conjunto de sentencias en un **case** no requiere de ser acotado por llaves { }, pero si se obliga al uso de la sentencia **break** que realmente es la encargada de efectuar la salida del **switch** y seguir con el flujo de ejecución fuera del **switch**. Si no se emplea el **break**, el C# seguirá ejecutando las instrucciones del **case** siguiente hasta que se encuentre con una sentencia **break**.
- El uso de la sentencia **default** es opcional. **default** permite ejecutar sus instrucciones cuando ninguno de los valores en los **case** se iguale al valor de la expresión probada por el **switch**.

- Si no usamos el **default** y no se iguale ningún valor establecido en un **case**, el flujo de ejecución seguirá con la siguiente sentencia después del cuerpo del **switch**.

Ejercicio 6.4.4
Utiliza un switch en el ejercicio 6.4.1 que sustituya en el método `Leer()` de la clase `Nino`, a los **if** de una rama que efectúan el conteo del número de canicas por color.

Recordemos que el método `Leer()` de la clase `Nino` tiene 2 parámetros : el número de canicas y su color, que previamente han sido ingresados por el usuario.

```
public void Leer(int noCanicas, string color)
{
     if (color == "AZUL")
         _noCanAzules += noCanicas;
     if (color == "ROJO")
         _noCanRojas += noCanicas;
     if (color == "VERDE")
         _noCanVerdes += noCanicas;
}
```

Estos **if's** son los que pretendemos sustituir por un **switch**. Notemos que la expresión sería el parámetro `color`.

Pongamos entre comentarios a los 3 **if** de una rama, para luego sustituirlos por el **switch**. Usemos los caracteres /* y */ para comentar todo el código.

```
public void Leer(int noCanicas, string color)
{
    /* if (color == "AZUL")
           _noCanAzules += noCanicas;
       if (color == "ROJO")
           _noCanRojas += noCanicas;
       if (color == "VERDE")
           _noCanVerdes += noCanicas; */
}
```

Hemos convertido al código de los **if's** a un comentario. Un comentario no lo interpreta el C# de manera que no será ejecutado el código que se encuentre en él.

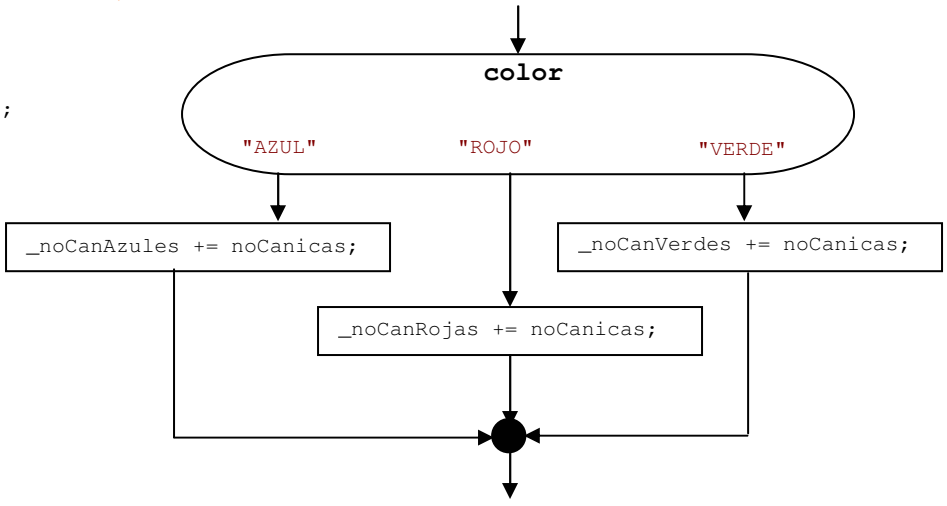
Ahora tecleemos el código para el **switch** que sustituye a los 3 **if's** de una rama.

```
public void Leer(int noCanicas, string color)
{
    /* if (color == "AZUL")
           _noCanAzules += noCanicas;
       if (color == "ROJO")
           _noCanRojas += noCanicas;
       if (color == "VERDE")
           _noCanVerdes += noCanicas; */

    switch (color)
    {
        case "AZUL" :
              _noCanAzules += noCanicas;
              break;
        case "ROJO" :
              _noCanRojas += noCanicas;
              break;
        case "VERDE" :
              _noCanVerdes += noCanicas;
              break;
    }
}
```

La expresión que prueba el **switch** es el parámetro `color`. Si su valor es "AZUL" se incrementa el atributo `_noCanAzules`, si es "ROJO" se incrementa el atributo `_noCanRojas` y si es "VERDE" se incrementa el atributo `_noCanVerdes`.

Flujo de ejecución del método `Leer()`.

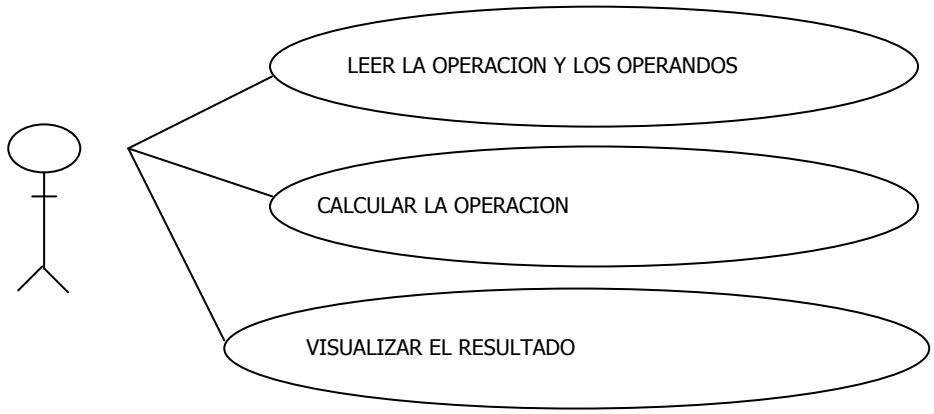


Ejecutemos la aplicación con este cambio. Veremos que todo sigue funcionando igual. El código ahora es mas eficiente debido a que sólo es probado una sola vez el valor del parámetro `color` para seleccionar el flujo de ejecución correspondiente.

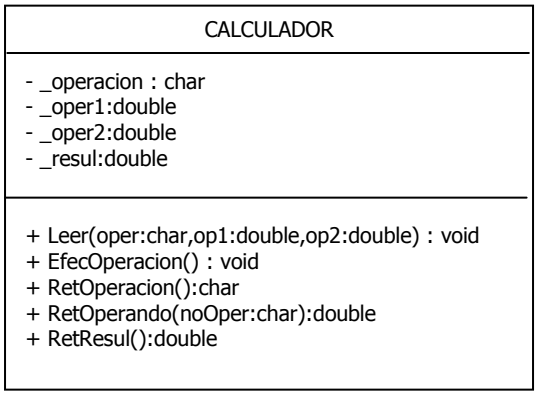
Ejercicio 6.4.5

Una persona necesita un calculador para obtener sumas, restas, multiplicaciones y divisiones. Requiere de ingresar el símbolo de la operación y los 2 operandos para luego hacer el cálculo y visualizarlo. Utiliza la sentencia **switch**.

El diagrama de casos de uso es simple :



Podemos pensar en un objeto `oCalc` perteneciente a la clase `Calculador`. *R.A. Francisco* propone una metodología para el uso de objetos `Calculador` que establece "Todo lo que se lea y se visualice deberá ser un atributo de la clase calculador". En este caso los atributos serían los que se especifican en el diagrama de clases siguiente :



La operación a realizar y sus operandos se leen, el resultado de efectuar la operación se visualiza, por lo tanto según la metodología ellos deberán ser atributos, cuestión que se ve en el diagrama de la clase. Los métodos son muy obvios, salvo tal vez el método `RetOperando()` el cual recibe un parámetro que indica al operando que deseamos que nos retorne el método. El método `Leer()` sólo establecerá labores de asignación a los atributos `_operacion`, `_oper1` y `_oper2`. El método encargado de realizar la operación indicada por el atributo `_operacion`, es el método `EfecOperacion()`. Este método deberá validar que en caso de que la operación requerida sea la división, el operando `_oper2` que hace las veces de divisor, no sea 0. La computadora no puede realizar divisiones por 0.
Iniciemos por agregar la clase `Calculador`, sus 4 atributos además de los métodos que retornan a dichos atributos, según el código que sigue :

```
class Calculador
{
     private char _operacion;
     private double _oper1;
     private double _oper2;
     private double _resul;
```

```
    public char RetOperacion()
    {
        return _operacion;
    }
    public double RetOperando(char noOper)
    {
        if (noOper == '1')
            return _oper1;
        else
            return _oper2;
    }
    public double RetResul()
    {
        return _resul;
    }
}
```

Luego agregamos la definición del objeto `oCalc` en la clase `Form1` residente en el archivo Form1.cs.

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

Construyamos la interfase gráfica de usuario según la figura #6.4.14.

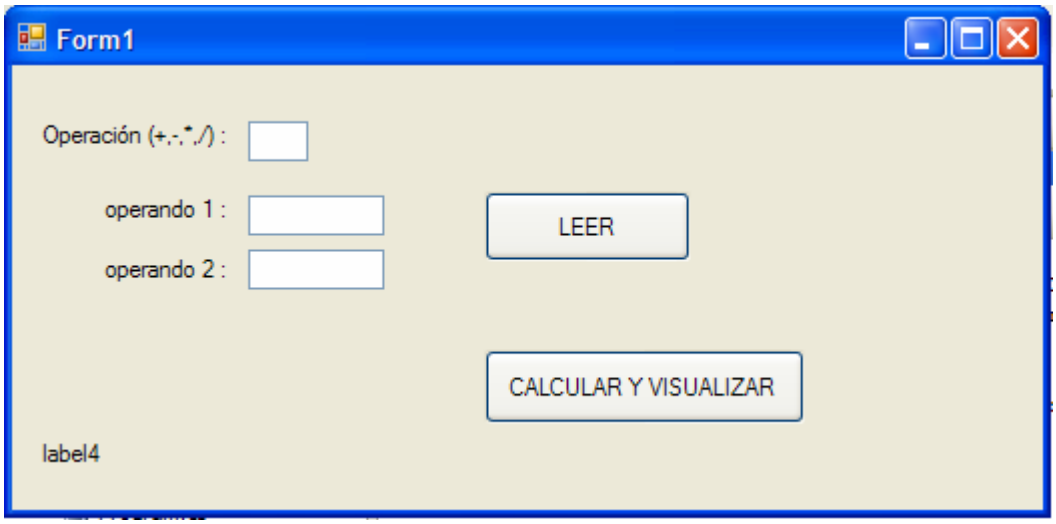


Fig. No. 6.4.14 Interfase gráfica del ejercicio 6.4.5.

Sigamos con el código del método `button1_Click()` de la clase `Form1`. Este método sólo asigna los valores tecleados por el usuario en los componentes TextBox, a los atributos correspondientes.

```
private void button1_Click(object sender, EventArgs e)
{
    char c=Convert.ToChar(textBox1.Text);
    if (c == '+' || c == '-' || c == '*' || c == '/')
        oCalc.Leer(c, Convert.ToDouble(textBox2.Text), Convert.ToDouble(textBox3.Text));
    else
        MessageBox.Show("ERROR ... LA OPERACION DEBE SER CUALQUIERA DE LOS CARACTERES : +, -, *, /");
}
```

Notemos que estamos comparando caracteres, por lo que debemos utilizar apóstrofes en la escritura de las constantes tipo `char`.

Antes de ejecutar el programa debemos definir el método `Leer()` en la clase `Calculador`, según se indica en el código siguiente :

```
public void Leer(char oper, double op1, double op2)
{
     _operacion = oper;
     _oper1 = op1;
     _oper2 = op2;
}
```

Si ejecutamos la aplicación e ingresamos valores en los componentes 3 TextBox, vemos que todo funciona de manera correcta. Ingresemos un valor diferente a '+', a '-', a '*' y a '/' para obtener el mensaje mostrado en la figura 6.4.15.

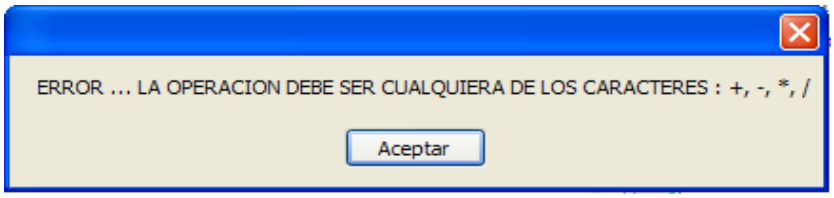
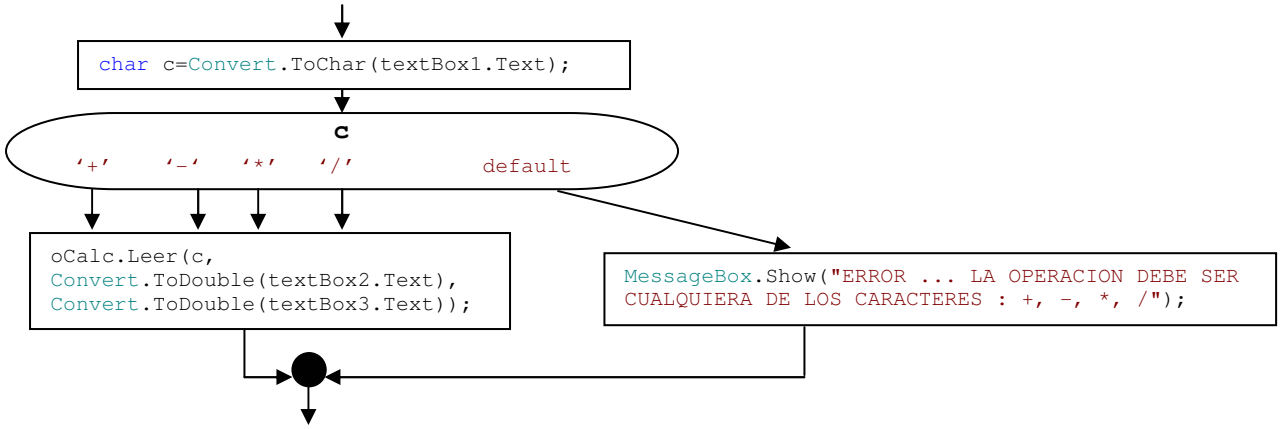


Fig. No. 6.4.15 validación de la operación a efectuar.

Como estamos explicando el uso del **switch**, vamos a modificar el código en el método `button1_Click()` de manera que en lugar de utilizar el **if** de 2 ramas para realizar la validación de la operación, usaremos una sentencia **switch**. Encierra entre comentarios al **if** de 2 ramas e inserta el código del **switch**.

```
private void button1_Click(object sender, EventArgs e)
{
   char c=Convert.ToChar(textBox1.Text);
   /*  if (c == '+' || c == '-' || c == '*' || c == '/')
           oCalc.Leer(c, Convert.ToDouble(textBox2.Text), Convert.ToDouble(textBox3.Text));
       else
           MessageBox.Show("ERROR ... LA OPERACION DEBE SER CUALQUIERA DE LOS CARACTERES : +, -, *,
                                                                                        /"); */
   switch (c)
   {
       case '+' :
       case '-' :
       case '*' :
       case '/' :
            oCalc.Leer(c, Convert.ToDouble(textBox2.Text), Convert.ToDouble(textBox3.Text));
            break;
       default :
            MessageBox.Show("ERROR ... LA OPERACION DEBE SER CUALQUIERA DE LOS CARACTERES : +, -, *,
                                                                                        /");
            break;
   }
}
```

Observemos que listamos los **case** que prueban los valores '+', '-', '*' y '/' de la operación `c` sin incluir la sentencia **break**, sino hasta el último **case** '/'. Recordemos que si no incluimos la sentencia **break**, al entrar al **switch** digamos por el valor '+', el flujo de ejecución seguirá por todos los **case** hasta encontrar la sentencia **break**.
Si el valor de la operación ingresada no corresponde a ninguno de los valores especificados en los **case**, entonces el flujo de ejecución seguirá por el **default**, en el cual incluimos la caja de mensaje del error.



Compilemos y ejecutemos la aplicación, provoquemos el error en la lectura de la operación, y la aplicación responderá según lo muestra la figura #6.4.15.

Terminaremos la aplicación construyendo el método `button2_Click()` que es el encargado de efectuar la operación deseada, además de visualizar el resultado. Este método inicia con la validación del atributo `_operacion` del objeto `oCalc`, mediante el uso de un **switch**.

```
private void button2_Click(object sender, EventArgs e)
{
    switch (oCalc.RetOperacion())
    {
        case '+':
        case '-':
        case '*':
        case '/':
            // efectuamos la operacion
            // visualizamos resultados
            break;
        default:
            MessageBox.Show("ERROR ... LA OPERACION NO ES VALIDA.");
            break;
    }
}
```

El mensaje `oCalc.RetOperacion()` retorna la operación registrada –leída-.

Si ejecutamos la aplicación y hacemos click en el botón con leyenda CALCULAR Y VISUALIZAR, provocamos el error de operación no valida, figura #6.4.16.

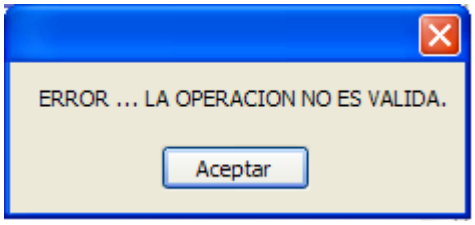


Fig. No. 6.4.16 Operación no válida –no leída-.

Bien, ahora añadimos el mensaje al objeto `oCalc` de manera que la operación la calcule. El mensaje deberá llamar al método `EfecOperacion()`. Agreguemos el mensaje dentro del **switch**, además de eliminar el comentario.

```
private void button2_Click(object sender, EventArgs e)
{
     switch (oCalc.RetOperacion())
     {
         case '+':
         case '-':
         case '*':
         case '/':
              oCalc.EfecOperacion();
              // visualizamos resultados
              break;
         default:
              MessageBox.Show("ERROR ... LA OPERACION NO ES VALIDA.");
              break;
     }
}
```

El método `EfecOperacion()` deberá validar que el operando 2 no sea 0, cuando el caso sea de una operación de división.

Ahora debemos agregar la definición del método `EfecOperacion()` dentro de la clase `Calculador`.

```
public void EfecOperacion()
{
    switch (_operacion)
    {
        case '+' :
            _resul = _oper1 + _oper2;
            break;
        case '-':
            _resul = _oper1 - _oper2;
```

```
            break;
        case '*':
            _resul = _oper1 * _oper2;
            break;
        case '/':
            if (_oper2 != 0.0)
                _resul = _oper1 / _oper2;
            break;
    }
}
```

Notemos la validación con el **if** de una rama, del valor del atributo `_oper2` con el fin de no dividir por 0.

Sólo falta añadir la visualización del resultado en el componente `label4`. El código que realiza la visualización es el mostrado a continuación.

```
private void button2_Click(object sender, EventArgs e)
{
    switch (oCalc.RetOperacion())
    {
        case '+':
        case '-':
        case '*':
        case '/':
            oCalc.EfecOperacion();
            label4.Text = oCalc.RetOperando('1').ToString() + " " + oCalc.RetOperacion().ToString() +
                                                    " " + oCalc.RetOperando('2').ToString() + " es = ";
            if (oCalc.RetOperando('2') == 0.0 && oCalc.RetOperacion() == '/')
                label4.Text += " ERROR ... NO PUEDO DIVIDIR POR 0";
            else
                label4.Text += oCalc.RetResul().ToString();
            break;
        default:
            MessageBox.Show("ERROR ... LA OPERACION NO ES VALIDA.");
            break;
    }
}
```

Ejecutemos la aplicación de manera que la probemos en todas sus formas de flujo de ejecución del programa.

6.4.5 Selectiva intenta, try-catch.

En ocasiones nuestro código puede producir errores en tiempos de ejecución, no obstante que la compilación haya sido exitosa. Los errores producidos durante la ejecución de un programa se les denomina excepciones.
Las excepciones pueden ser manejadas o tratadas con sentencias selectivas que controlan el flujo de ejecución de un programa, cuando se presente alguna excepción –error-.
Este tipo de sentencia selectiva trata de ejecutar una o mas instrucciones en un bloque de código etiquetado con una sentencia **try**, de manera que si se presenta un error –excepción- al intentar ejecutar las instrucciones en el bloque de código, envía el flujo de ejecución a otro bloque de código que recibe la notificación de que se ha atrapado una excepción.
Este bloque de código que es ejecutado cuando una excepción ha sido detectada por el código **try**, se le etiqueta con el encabezado **catch**.
En C# la selectiva intenta se escribe de la siguiente forma :

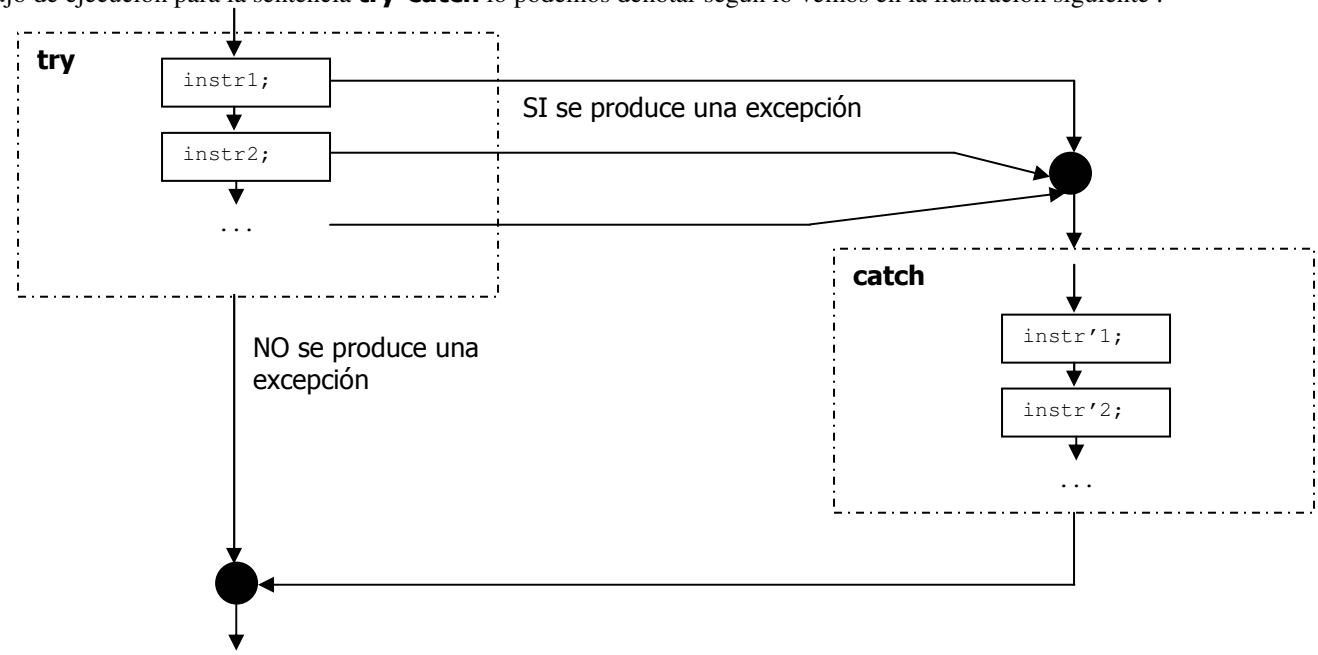
```
try
{
  instr1;
  instr2;
  ...
}
catch
{
  instr'1;
  instr'2;
  ...
}
```

Bloque de código que se intenta ejecutar para detectar alguna excepción –error- en tiempos de ejecución.

Bloque de código que recibe el control de ejecución del programa cuando el bloque **try** detecta una excepción.

Realmente el encabezado **catch** puede tener parámetros. El parámetro es el tipo y el identificador de la excepción que se quiere atrapar en el bloque **catch**. Entonces pueden definirse mas de un bloque **catch** para un mismo bloque **try**, donde cada bloque **catch** deberá aceptar la excepción para la cual está definido, como parámetro. La revisión del uso del **catch**

con parámetros se ha dejado para un siguiente curso donde se profundiza en el uso del **try-catch**. Asi que para este curso de fundamentos de programación, queda fuera del alcance lo que se ha mencionado en este párrafo.
El flujo de ejecución para la sentencia **try-catch** lo podemos denotar según lo vemos en la ilustración siguiente :



Notemos que cualquier sentencia en el bloque **try** puede cambiar el flujo de ejecución del programa, ya que la prueba por una excepción es implícita en la ejecución de cada sentencia en dicho bloque **try**.

Ejercicio 6.4.6
Analiza el ejercicio 6.4.1 de manera que detectes cuando puede ser lanzada una excepción por C# en tiempos de ejecución del programa.

Veamos la interfase gráfica del ejercicio 6.4.1 ilustrada en la figura #6.4.17, y tratemos de contestarnos la pregunta ¿Qué sucede si en lugar de ingresar un entero válido en el componente TextBox `textBox1` que recibe el número de canicas para su registro, ingresamos por ejemplo el `5i`?.

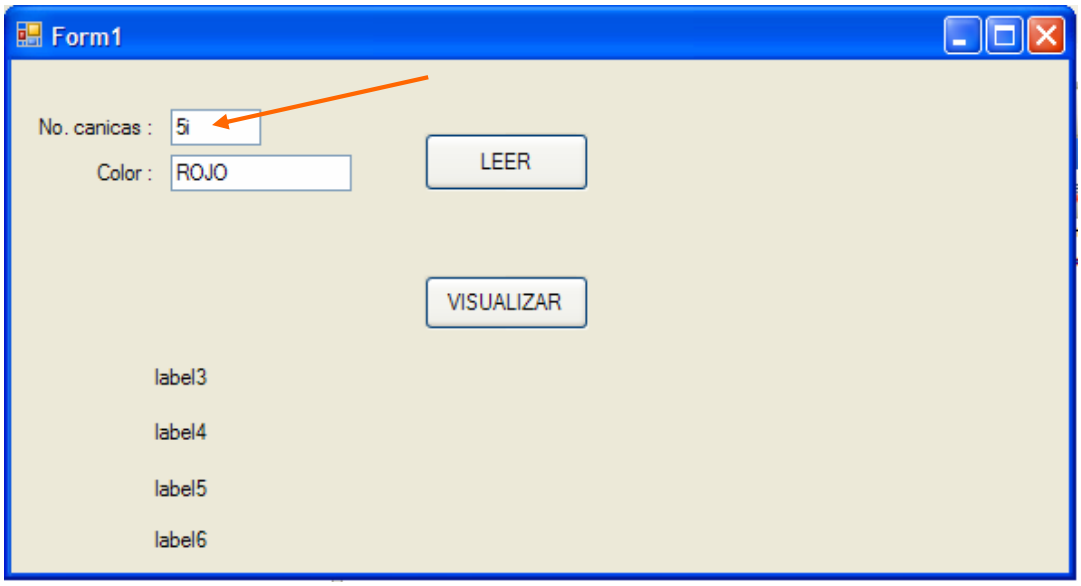


Fig. No. 6.4.17 Ingreso del número de canicas NO válido `5i`.

La respuesta es que el C# nos envía una caja de mensajes de error en el cual nos indica que ha ocurrido una excepción cuando se ha tratado de efectuar la conversión de **`string`** a **`Int32`**, figura #6.4.18.

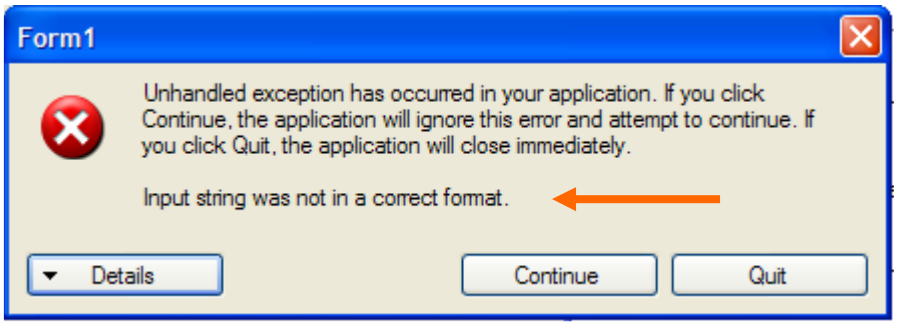


Fig. No. 6.4.18 Excepción en la conversión de tipo cadena a entero.

Observemos que en el mensaje se indica que la excepción que ha ocurrido, no ha sido manejada por nuestro código –Unhandled exception-.
Entonces debemos usar la selectiva intenta **try** en la ejecución de la sentencia que efectúa la lectura del número de canicas en formato cadena para luego convertirla a formato entero, utilizando la clase **Convert**.
El código del botón `button1` que se encarga de efectuar la lectura del número de canicas y de su color, es el que se muestra a continuación.

```
private void button1_Click(object sender, EventArgs e)
{
    int noCanicas = Convert.ToInt32(textBox1.Text);
    string color = textBox2.Text;
    if (noCanicas > 0 && (color == "AZUL" || color == "ROJO" || color == "VERDE"))
    {
        oNino.Leer(noCanicas, color);
        MessageBox.Show("EL REGISTRO DE "+noCanicas.ToString()+" CANICAS COLOR "+color+" FUE
                                                                                    REALIZADO");
    }
    if (noCanicas <= 0)
        MessageBox.Show("ERROR ... EL NUMERO DE CANICAS DEBE SER MAYOR QUE 0");
    if (color != "AZUL" && color != "ROJO" && color != "VERDE")
        MessageBox.Show("ERROR ... EL COLOR DEBE SER AZUL, ROJO o VERDE");
}
```

Conversión con posibilidades de excepción.

Notemos que la lectura y a la vez la conversión se efectúa en la sentencia señalada por la flecha color naranja. Desde luego que en la asignación –lectura- del color de las canicas, no hay posiblidades de que se genere una excepción ya que no se realiza ninguna conversión ni otra operación que pudiera generarla.
Reescribamos el código del método `button1_Click()` de manera que ahora manejemos la posible excepción que se genera cuando la cadena a convertir no representa a un número entero.

```
private void button1_Click(object sender, EventArgs e)
{
    int noCanicas=0;
    bool huboExcepcion = false;
    try
    {
        noCanicas = Convert.ToInt32(textBox1.Text);
    }
    catch
    {
        MessageBox.Show("ERROR ... número de canicas ingresado no es un entero.");
        huboExcepcion = true;
    }
    if (! huboExcepcion)
    {
        string color = textBox2.Text;
        if (noCanicas > 0 && (color == "AZUL" || color == "ROJO" || color == "VERDE"))
        {
            oNino.Leer(noCanicas, color);
            MessageBox.Show("EL REGISTRO DE " + noCanicas.ToString() + " CANICAS COLOR " + color + "
                                                                                FUE REALIZADO");
```

Declaramos la variable booleana `huboExcepcion` para conocer si hubo o no hubo un error en la conversión.

Asignación de `huboExcepcion` a **true** en el caso de que exista el error de conversión.

```
        }
        if (noCanicas <= 0)
            MessageBox.Show("ERROR ... EL NUMERO DE CANICAS DEBE SER MAYOR QUE 0");
        if (color != "AZUL" && color != "ROJO" && color != "VERDE")
            MessageBox.Show("ERROR ... EL COLOR DEBE SER AZUL, ROJO o VERDE");
    }
}
```

La lectura del color y los **if's** que corresponden a las diferentes validaciones de rangos y valores, sólo se efectúan si no existió excepcion, cuestión que es mantenida por la variable `huboExcepcion`. Esta variable es puesta a `true` si el flujo de ejecución entra al bloque de código del **catch**.
Compilemos y ejecutemos la aplicación para ver si todo está bien, es decir, la aplicación debe funcionar igual y ahora mejor –ya que se está controlando la excepción-, que el código anterior –sin manejo de la excepción-. La figura #6.4.19 muestra el mensaje generado por nosotros cuando existe la excepción durante la conversión de cadena a entero.

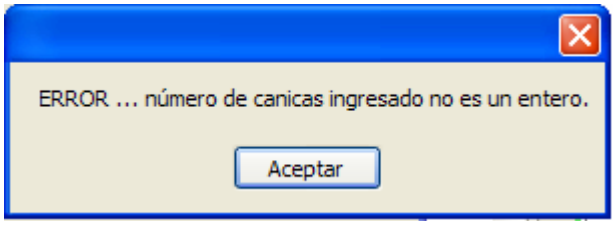


Fig. No. 6.4.19 Excepción manejada existente en la conversión de cadena a entero.