VARIABLES Y CÁLCULOS - TETXTBOX

Guía de laboratorio - Capítulo III

Contenidos:

- ✓ Tipos de variables numéricas
- ✓ Declarar variables
- ✓ Cálculos
- ✓ Operadores aritméticos
- ✓ Uso de números con etiquetas y cuadros de texto
- ✓ Métododos Substring() e IndexOf()
- ✓ Clase Array
- ✓ Evento keyPress

Fuente: B. Douglas & P. Mark (2010). "C# para estudiantes". Pearson Educación. DF México.

En casi todos los programas se utilizan números de un tipo u otro; por ejemplo, para dibujar imágenes mediante el uso de coordenadas en la pantalla, para controlar trayectorias de vuelos espaciales, o para calcular sueldos y deducciones fiscales.

En este capítulo veremos los dos tipos básicos de números:

- Números sin decimales, conocidos como enteros en matemáticas, y como el tipo **int** en C#;
- Números con "punto decimal", conocidos como "reales" en matemáticas, y como el tipo **double** en C#. El término general para los números con punto decimal en computación es números de *punto flotante*.

Una variable es un espacio de almacenamiento que se utiliza para recordar valores, de forma que éstos puedan utilizarse o modificarse más adelante en el programa.

Mayormente se utilizan números **int** son para representar o calcular *cantidades*; en cambio, las situaciones que exigen el uso de números **double** podrían ser generalmente *medidas, promedios*, etc.

Sin embargo, algunas veces el tipo de número a utilizar no es obvio; por ejemplo, si queremos tener una variable para almacenar la calificación de un examen, ¿debemos emplear un número **double** o **int**? No podemos determinar la respuesta con base en lo que sabemos; debemos contar con más detalles. Por ejemplo, podemos preguntar a la persona que se encarga de calificar si redondea al número entero más cercano, o si utiliza números decimales. En consecuencia, la elección entre **int** y **double** se determina a partir de cada problema en particular.

Tipo de variables numéricas

La naturaleza de int

Cuando utilizamos un número **int** en C#, puede tratarse de un número entero en el rango de

22,147,483,648 a 12,147,483,647, o aproximadamente de 22,000,000,000 a 12,000,000,000.

Todos los cálculos con números **int** son precisos, en cuanto a que toda la información en el número se preserva sin errores.

La naturaleza de double

Cuando utilizamos un número **double** en C# su valor puede estar entre 21.79 310308 y 11.79 310308.

En términos no tan matemáticos, el mayor valor es 179 seguido de 306 ceros; ¡sin duda un valor extremadamente grande! Los números se guardan con una precisión aproximada de 15 dígitos.

El principal detalle respecto de las cantidades **double** está en que, en casi todos los casos, éstas se guardan en forma aproximada. Para comprender mejor esta característica, realice la siguiente operación en una calculadora:

Si utilizamos siete dígitos (por ejemplo) la respuesta es 2.333333, pero sabemos que una respuesta más exacta sería: 2.33333333333333333

Y aun así, ¡ésa no es la respuesta exacta!

En resumen, como las cantidades **double** se almacenan utilizando un número limitado de dígitos, pueden acumularse pequeños errores en el extremo menos significativo. Para muchos cálculos (por ejemplo, calificaciones de exámenes) esto no es importante, pero para aquellos relacionados con digamos, el diseño de una nave espacial, cualquier diferencia podría ser relevante. Sin embargo, el rango de precisión de los números **double** es tan amplio que es posible emplearlos sin problemas en los cálculos de todos los días.

Para una mayor precisión de los números utilicen el tipo **decimal** (29-29 dígitos de precisión), aunque se perdería el rendimiento. Útil para operaciones financieras importantes.

Para un procesamiento más intensivo utilicen el tipo **float**, aunque se perderá precisión y habrá posibles errores de redondeo (7 dígitos de precisión). Útil para gráficos y juegos.

Para escribir valores **double** muy grandes (o muy pequeños) se requieren grandes secuencias de ceros. Para simplificar esto podemos usar la notación "científica" o "exponencial", con **e** o **E**, como en el siguiente ejemplo: **double valorGrande = 12.3E+23;** lo cual representa 12.3 multiplicado por 10123. Esta característica se utiliza principalmente en programas matemáticos o científicos.

Declaración de variables

Una vez elegido el tipo de nuestras variables, necesitamos darles un nombre. Podemos imaginarlas como cajas de almacenamiento con un nombre en su exterior y un número (valor) en su interior.

El valor puede cambiar a medida que el programa realiza su secuencia de operaciones, pero el nombre es fijo. El programador tiene la libertad de elegir los nombres, y recomendamos escoger aquellos que sean significativos. A pesar de esa libertad, al igual que en casi todos los lenguajes de programación, en C# hay ciertas reglas que debemos seguir. Por ejemplo, los nombres:

- deben empezar con una letra (de la **A** a la **Z** o de la **a** a la **z**);
- pueden contener cualquier cantidad de letras o dígitos (un dígito es cualquier número del 0 al 9);
- pueden contener el guión bajo '_';
- pueden tener hasta 255 caracteres de longitud.

Tengan en cuenta que C# es sensible al uso de mayúsculas y minúsculas. Por ejemplo, si usted declara una variable llamada **ancho**, no podrá referirse nunca a ella como **Ancho** o **ANCHO**, ya que el uso de las mayúsculas y minúsculas es distinto en cada caso.

También hay un estilo de C#, una forma de usar las reglas que se implementa cuando el nombre de una variable consta de varias palabras: las reglas no permiten separar los nombres con espacios, así que en lugar de utilizar nombres cortos o guiones bajos, el estilo aceptado es poner en mayúscula la primera letra de cada palabra (o exceptuando la primera). Se denominan estilo **PascalCase** ó **camelCase**.

En este capítulo trabajaremos con variables que sólo se utilizan dentro de un método (en vez de que varios métodos las compartan). Las variables de este tipo se conocen como ***locales*** y sólo se pueden emplear entre los caracteres **{** y **}** en donde se declaren.

Es por ello, y siguiendo con los estilos, la metodología de C# dicta *no* poner en mayúscula la primera letra de las variables locales.

Más adelante veremos que otros tipos de nombres, como los de métodos y clases, empiezan por convención con letra mayúscula.

Por lo tanto, en vez de:

Alturadecaja / h / hob / altura_de_caja

usaremos:

alturaDeCaja

He aquí algunos nombres permitidos:

Cantidad, x, pago2003

y éstos son algunos nombres no permitidos (ilegales):

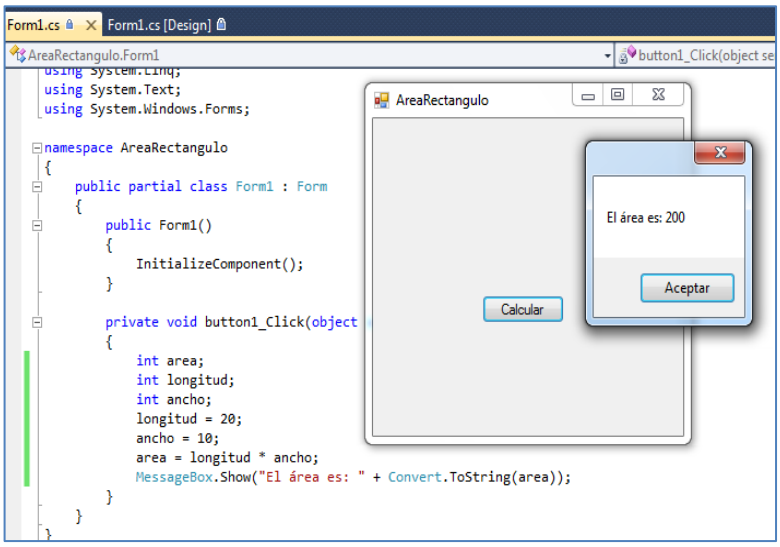
2001pago (empieza con número)**, _area** (empieza con signo)**, mi edad** (tiene espacio)

También existen algunos nombres reservados para utilización exclusiva de C#, de manera que el programador no los puede reutilizar. Se denominan *palabras clave* o *palabras reservadas*, como ser:

Private, int, new (entre otras)

El código siguiente corresponde a un programa de ejemplo llamado **AreaRectangulo**, mismo que analizaremos con detalle a continuación. Supongamos que las medidas de los lados del rectángulo que nos interesa están representadas en números enteros (**int**). Sólo hay un control en el formulario: un botón con el texto “Calcular” en su propiedad **Text**. Todo el código que agregaremos estará dentro del método **button1_Click**.

```
private void button1_Click(object sender, EventArgs e)
{
    int area;
    int longitud;
    int ancho;
    longitud = 20;
    ancho = 10;
    area = longitud * ancho;
    MessageBox.Show("El área es: " + Convert.ToString(area));
}
```



En el programa utilizamos tres variables int, que en un momento dado guardarán los datos de nuestro rectángulo. Recuerde que podemos elegir cualquier nombre para nuestras variables; sin embargo, optamos por utilizar nombres claros en vez de nombres cómicos o de una sola letra, que no resultan lo suficientemente claros.

Una vez elegidos los nombres debemos declararlos en el sistema de C#. Aunque esto parece tedioso al principio, el propósito de introducirlos radica en permitir que el compilador detecte errores al escribirlos en el código del programa. He aquí las **declaraciones**:

```
int area;
int longitud;
int ancho;
```

Al declarar variables anteponemos al nombre que elegimos el **tipo** que necesitamos (en las tres variables anteriores utilizamos el tipo **int**, de manera que cada variable contendrán un número entero).

También podríamos utilizar como alternativa una sola línea de código:

```
int area, longitud, ancho;
```

Ustedes pueden emplear el estilo de su preferencia; no obstante, recomendamos usar el primero, ya que nos permite insertar comentarios en cada nombre, en caso de ser necesario. Si usted opta por el segundo estilo, úselo para agrupar nombres relacionados.

Por ejemplo, emplee:

```
int alturaImagen, anchoImagen;
int miEdad;
```

en vez de:

```
int alturaImagen, anchoImagen, miEdad;
```

Cálculos y operadores

Veamos de nuevo nuestro programa del rectángulo, en el que se incluye la siguiente instrucción:

```
area = longitud * ancho;
```

La forma general de la instrucción de **asignación** es: **variable = expresión;** (se asigna un valor a la variable)

Una expresión puede tomar varias formas, como un solo número o como un cálculo. En nuestro ejemplo específico la secuencia de eventos es:

1. El operador ***** hace que se multipliquen los valores almacenados en **longitud** y **ancho**, obteniéndose como resultado el valor 200.

2. El símbolo igual = hace que el número 200 se asigne a (se almacene en) **área**.

Es importante comprender el flujo de los datos, ya que esto nos permite entender el significado de código como el siguiente:

```
int n = 10;
n = n + 1;
```

Lo que ocurre aquí es que la expresión que está al lado derecho del signo = se calcula utilizando el valor actual de **n**, con lo cual se obtiene **11**. Después este valor se almacena en **n**, sustituyendo el valor anterior, que era **10**.

C# cuenta con una versión abreviada de esta instrucción, llamada instrucción de *incremento: l*os operadores **++** y **- -** realizan el incremento y el decremento (o resta de una unidad). Su uso más frecuente es en los ciclos o bucles. He aquí una forma de utilizar el operador **++**: `n = 3; n++;` **// ahora n vale 4**

Los operadores aritméticos

En esta sección le presentaremos un conjunto básico de operadores: los aritméticos, similares a los botones de cualquier calculadora.

Operador Significado

***** multiplicación	**/** división	**%** módulo
+ suma	**-** resta	

La multiplicación, división y módulo (*****, **/** y **%**) se llevan a cabo antes que la suma y la resta (**+** y **–**). También podemos usar paréntesis para agrupar los cálculos y forzarlos a llevarse a cabo en un orden específico. Si un cálculo incluye operadores de la misma precedencia, las operaciones se realizarán de izquierda a derecha. Ejemplo:

```
int i;
int n = 3;
double d;

i = n + 3; // se convierte en 6
i = n * 4; // se convierte en 12
i = 7 + 2 * 4; // se convierte en 15
n = n * (n + 2) * 4; // se convierte en 60
d = 3.5 / 2; // se convierte en 1.75
n = 7 / 4; // se convierte en 1
```

Veamos a continuación algunas fórmulas matemáticas y su conversión a C#. Supongamos que todas las variables están declaradas como tipos **double**, y que su valor inicial ha sido establecido.

	Versión matemática	Versión de C#
1	$y = mx + c$	y = m * x + c;
2	$x = (a - b)(a + b)$	x = (a - b) * (a + b);
3	$y = 3[(a - b)(a + b)] - x$	y = 3 * ((a - b)*(a + b)) - x;
4	$y = 1 - \frac{2a}{3b}$	y = 1 - (2 * a) / (3 * b);

En el ejemplo 1 insertamos el símbolo de multiplicación. En C# **mx** se consideraría un nombre de variable.
En el ejemplo 2 necesitamos un signo de multiplicación explícito entre los paréntesis.
En el ejemplo 3 sustituimos los corchetes matemáticos por paréntesis.
En el ejemplo 4 podríamos haber cometido el error de usar esta versión incorrecta: **y = 1 - 2 * a / 3 * b;**

Recuerde la regla según la cual los cálculos se realizan de izquierda a derecha cuando los operadores tienen igual precedencia. El problema tiene que ver con los operadores ***** y **/**. El orden de evaluación es como si hubiéramos utilizado: **y = 1 - (2 * a / 3) * b;** es decir, la **b** ahora está multiplicando en vez de dividir. La forma más simple de manejar los cálculos potencialmente confusos consiste en utilizar paréntesis adicionales; hacerlo no implica penalización alguna en términos de tamaño o velocidad del programa.

El uso de los operadores **+**, **–** y ***** es razonablemente intuitivo, pero la división es un poco más engañosa, ya que exige diferenciar entre los tipos **int** y **double**. En este sentido, lo importante es tomar en cuenta que:

- Cuando el operador **/** trabaja con dos números **double** o con una mezcla de **double** e **int** se produce un resultado **double**. Para fines de cálculo, cualquier valor **int** se considera como **double**. Así es como funciona la división en una calculadora de bolsillo.
- Cuando **/** trabaja con dos números de tipo **int** se produce un resultado entero. El resultado se trunca, lo cual significa que se borran los dígitos que pudiera haber después del "punto decimal". Ésta *no* es la forma en que funcionan las calculadoras.

El operador %

Por último veremos el operador **%** (módulo). A menudo se utiliza junto con la división de enteros, ya que provee la parte del residuo. Su nombre proviene del término "módulo" que se utiliza en una rama de las matemáticas conocida como aritmética modular.

Anteriormente dijimos que los valores **double** se almacenan de manera aproximada, a diferencia de los enteros, que lo hacen de forma exacta. Entonces ¿cómo puede ser que **33/44** genere un resultado entero de **0**? ¿Acaso perder **0.75** significa que el cálculo no es preciso? La respuesta es que los enteros *sí* operan con exactitud, pero el resultado exacto está compuesto de dos partes: el cociente (es decir, el resultado principal) y el residuo. Por lo tanto, si dividimos **4** entre **3** obtenemos como resultado **1**, con un residuo de **1**. Esto es más exacto que **1.3333333**, etc.

En consecuencia, el operador **%** nos da el residuo como si se hubiera llevado a cabo una división. Ejemplo:

```
int i;
double d;
i = 12 % 4; // se convierte en 0
i = 13 % 4; // se convierte en 1
i = 15 % 4; // se convierte en 3
d = 14.9 % 3.9; // se convierte en 3.2 (se divide 3.2 veces)
```

Hasta ahora el uso más frecuente de **%** es con números **int**, pero cabe mencionar que también funciona con números **double**. Imaginen que queremos convertir un número entero de centavos en dos cantidades: la cantidad de pesos y el número de centavos restantes. La solución es:

```
int centavos = 234;
int pesos, centavosRestantes;
pesos = centavos / 100;           // se convierte en 2
centavosRestantes = centavos % 100; // se convierte en 34
```

Conversiones entre cadenas y números

Uno de los usos más importantes del tipo de datos **string** son las operaciones de entrada y salida, en donde procesamos los datos que introduce el usuario y desplegamos los resultados en pantalla.

Muchos de los controles de la GUI (interfaz gráfica de usuario) de C# trabajan con cadenas de caracteres en vez de hacerlo con números, por lo cual es preciso que aprendamos a realizar conversiones entre números y cadenas.

La clase **Convert** proporciona varios métodos convenientes para ese propósito.

Para convertir una variable o cálculo (una expresión en general) podemos utilizar el método **ToString**. He aquí algunos ejemplos:

```
string s1, s2;
int num = 44;
double d = 1.234;

s1 = Convert.ToString(num); // s1 es "44"
s2 = Convert.ToString(d);   // s2 es "1.234"
```

Por lo general el nombre del método va precedido por el de un objeto con el que debe trabajar, pero aquí suministramos el objeto como un *argumento* (entre paréntesis). Los métodos que funcionan de esta forma se denominan estáticos (**static**); cada vez que los utilicemos deberemos identificar la clase a la que pertenecen.

Éste es el motivo por el que colocamos **Convert** antes de **ToString**. Analizaremos más adelante a fondo los métodos **static**.

En el ejemplo anterior el método **ToString** nos regresa una cadena que podemos almacenar en una variable, o utilizarla de alguna otra forma.

En el programa para calcular el área de un rectángulo utilizamos el operador **+** y el método **ToString** con un cuadro de mensaje desplegable. En vez de mostrar sólo el número, lo concatenamos (unimos) a un mensaje:

```
MessageBox.Show("El área es: " + Convert.ToString(area));
```

El siguiente código no compila, ya que el método **Show** espera un valor **string** como parámetro:

```
MessageBox.Show(area); //NO - ¡no compilará!
```

Debemos utilizar:

```
MessageBox.Show(Convert.ToString(area));
```

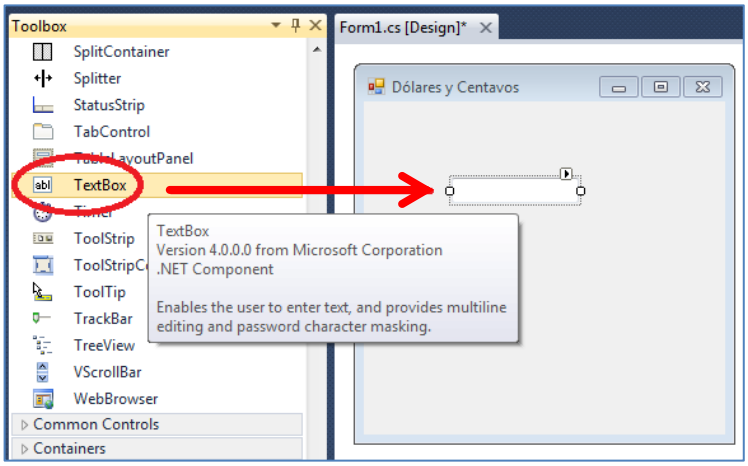
Para complementar el método **ToString** tenemos los métodos **ToInt32** y **ToDouble**, los cuales convierten las cadenas de caracteres en números. Observe que no hay un método **ToInt**. La clase **Convert** está disponible para cualquier lenguaje que utilice el marco de trabajo (framework) .NET, y el nombre de clase a nivel de marco de trabajo para los elementos **int** en C# es **Int32** (enteros de 32 bits). He aquí algunos ejemplos:

```
double d;
int i;
string s1 = "12.3";
string s2 = "567";
d = Convert.ToDouble(s1);
i = Convert.ToInt32(s2);
```

Cuadros de texto y etiquetas

En los programas en que hemos venido trabajando utilizamos instrucciones de asignación con el propósito de establecer valores iniciales para los cálculos; pero, en la práctica no conoceremos esos valores al escribir el programa ya que el usuario los introducirá a medida que éste se vaya ejecutando.

En esta sección veremos el control **TextBox**, el cual permite que un usuario introduzca datos, y utilizaremos el control **Label** (que ya hemos conocido) utilizado para mostrar información (por ejemplo, los resultados de un cálculo, o instrucciones para el usuario) en un formulario.



Como sabemos, para usar un control todo lo que tenemos que hacer es seleccionarlo en el cuadro de herramientas y colocarlo en un formulario. Los cuadros de texto, o **TextBox** tienen muchas propiedades, pero la principal es **Text**, que nos proporciona la cadena escrita por el usuario. Para acceder a esta propiedad utilizamos la ya conocida notación de "punto", como en el siguiente ejemplo:

```
string s;
s = textBox1.Text;
```

Es bastante común que el programador elimine el contenido de la propiedad **Text** del control en tiempo de diseño (mediante la ventana de propiedades), para que el usuario pueda escribir en un área en blanco.

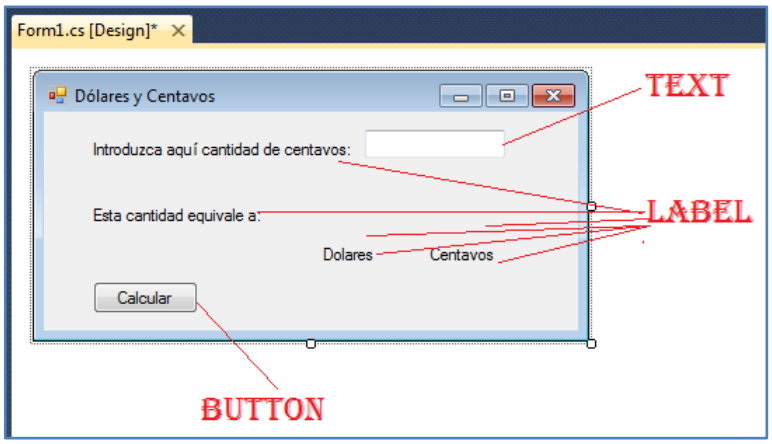
Al igual que en el caso de los cuadros de texto, la principal propiedad del control **Label** (disponible también en el cuadro de herramientas) es **Text**, pues nos permite establecer la cadena que la etiqueta mostrará en pantalla. Podemos acceder a esta propiedad de la siguiente manera:

```
string s = "Alto";
label1.Text = s;
```

Algunas etiquetas se utilizan para mostrar mensajes de ayuda al usuario; por lo general establecemos su propiedad **Text** en tiempo de diseño mediante la ventana de propiedades. No es necesario que el texto que contienen cambie durante la ejecución del programa. Por otro lado, en el caso de las etiquetas que despliegan resultados hay que establecer su propiedad **Text** en tiempo de ejecución, como se muestra en el ejemplo anterior. El usuario puede sobrescribir los cuadros de texto, pero las etiquetas están protegidas.

En general, las clases tienen métodos y propiedades. Los métodos hacen que los objetos realicen acciones, mientras que las propiedades nos permiten acceder al estado actual de un objeto. Ya veremos estos conceptos con mayor profundidad.

He aquí un programa de ejemplo (Dólares y centavos), en el que una cantidad en centavos se convierte a dólares y centavos. Anteriormente en este capítulo vimos cómo usar los operadores **/** y **%**. En la Figura se muestra este programa en ejecución; en él se utiliza un cuadro de texto y varias etiquetas.



Los principales controles que utilizamos en este programa son:

- 1 botón para iniciar la conversión;
- 1 cuadro de texto en donde el usuario introduce una cantidad en centavos;
- 2 etiquetas (Label): para mostrar el número de dólares y el número de centavos.

A continuación en la tabla, se verán del lado izquierdo las propiedades de cada control a utilizar y del lado derecho el valor que deberán asignar a cada una de ellas:

button1	Valor
Text	Calcular
textBox	
Text	(vacía)
dolaresEtiqueta	
Text	(vacía)
centavosEtiqueta	
Text	(vacía)

Además hay tres etiquetas debajo del cuadro de texto y las dos etiquetas que muestran el resultado para ayudar al usuario a entender el formulario. Los valores de texto de las etiquetas son:

“Introduzca aquí la cantidad de centavos”
“Dólares”
“Centavos”

En este programa:

- hay sólo un botón y sólo un cuadro de texto, por lo que podemos dejar a estos controles el nombre que C# les asignó.
- hay dos etiquetas que muestran resultados. Como tenemos dos controles “label” (nombre por defecto) podría causar confusión, les damos un nombre específico a cada uno de ellos.
- al resto de las etiquetas se les asigna su propiedad de texto en tiempo de diseño, y el programa nunca las manipulará. Podemos dejar a estas etiquetas los nombres que C# les asignó.

El código para ingresar en el Button (haciendo doble click sobre el botón agregado al formulario) es el siguiente:

```
private void button1_Click(object sender, EventArgs e)
{
    int centavos;
    centavos = Convert.ToInt32(textBox1.Text);
    dolaresEtiqueta.Text = Convert.ToString(centavos / 100);
    centavosEtiqueta.Text = Convert.ToString(centavos % 100);
}
```

Recuerde que es conveniente cambiar el nombre de los controles tan pronto como los coloque en el formulario, antes de hacer doble clic para crear el código de cualquier evento.

Cuando el programa se ejecuta el usuario introduce un número en el cuadro de texto. Luego, cuando este hace clic en el botón se lleva a cabo el cálculo detallado en el código anterior y los resultados se colocan en las dos etiquetas. Sin embargo, hay que realizar conversiones de cadenas a números y viceversa:

```
centavos = Convert.ToInt32(textBox1.Text);
dólaresEtiqueta.Text = Convert.ToString(centavos / 100);
```

El programa ilustra el uso de un cuadro de texto y de etiquetas para mostrar resultados que pueden cambiar, junto con mensajes que no se modifican.

Métododos Substring() e IndexOf()

Una variable de tipo string permite almacenar una cadena de caracteres y manipularla utilizando los métodos proporcionadas por C#. Uno de ellos es “Substring”, veamos un ejemplo:

```
string nombreCompleto = "Mariano, Vega";
string nombre = nombreCompleto.Substring(0, 7);
```

Este código nos devuelve como resultado en la variable “nombre” la cadena “Mariano”, recortando un fragmento de la cadena original. Esto sucede ya que en la segunda línea de código, hacemos uso del método Substring, donde en sus parámetros debemos indicar en qué posición comenzamos a *recortar* y cuántos caracteres deseamos obtener en este recorte.

En el caso de querer obtener el apellido, indicamos desde donde comenzar, pero como es hasta el final, no haría falta indicar la cantidad de caracteres de la cadena a obteber. La línea de código sería la siguiente:

```
string apellido = nombreCompleto.Substring(9);
```

o bien:

```
string apellido = nombreCompleto.Substring(nombreCompleto.Length - 4);
```

Otra forma de obtener fragmentos de cadenas o buscar caracteres en objetos de tipo string, es con el método **IndexOf**. Siguiendo con el mismo ejemplo, podríamos utilizar este método de la siguiente manera:

```
string nombreCompleto = "Mariano, Vega";
string nombre = nombreCompleto.Substring(0, nombreCompleto.IndexOf(','));

int espacio = nombreCompleto.IndexOf(' ');
string apellido = nombreCompleto.Substring(espacio).Trim();
```

Lo que permite este método es encontrar en la cedana el primer caracter especificado y devolver su índice en un valor de tipo int. Es decir, que si se desconoce el índice desde donde debería comenzar el recorte de la cadena pero existe un caracter que delimite el sector deseado, esto podría ser de gran utilidad.

La segunda línea de código, busca la primera posición en la que se encuentra el caracter `(',')`, obteniendo como resultado el valor **8**, valor que es usado para indicar en el método substring como longitud de caracteres para recortar.

La tercera línea, busca el primer caracter igual a `(' ')` para identificar la separación entre el nombre y el apellido, obteniendo como resultado el número del índice desde donde se comienza con el recorte en la siguiente línea de código dentro del método Substring.

Otro método que utilizamos es **Trim()**, el cual recorta todo espacio que pueda existir al inicio o al final de un valor de tipo string.

Clase Array

La clase array del espacio de nombres System sirve como clase base para todas las matrices. Proporciona métodos para la creación, manipulación, búsqueda y ordenación de matrices.

Un array puede almacenar varios valores simultáneos, cada uno de estos valores se identifica mediante un numero al cual se lo llama índice.

Un array se declara e instancia de la siguiente manera:

tipo [ ] variable = new tipo [6];

O bien se puede declarar en una línea e instanciar en otra, como ya lo hemos hecho con otros objetos.

Una segunda forma de declarar e instanciar una arrays se puede generar de forma preliminar:

tipo [ ] variable = { x, x, x, x, x };

Algunos de sus métodos son:

- **Equals** *Devuelve un resultado booleano de la comparación de dos valores*

```
bool ok = Array.Equals(arrayAnimales[0], arrayAnimales[1]);
```

- **Clone** *Clona el array sobre una variable de tipo object*

```
object arrayNuevo = arrayAnimales.Clone();
```

- **CopyTo** *Copia los elementos de un array a otro que concuerde*

```
string[] array2 = new string[5];
arrayAnimales.CopyTo(array2, 0);
```

- **getLength** *Devuelve la longitud del array en la dimensión especificada*

```
int cant = arrayAnimales.GetLength(0);
```

- **Clear** *Limpia todos los elementos de un array o desde determinado índice y cantidad*

```
Array.Clear(array2, 0, 2); //limpia desde el índice 0, dos elementos
```

- **Sort** *Ordena descendentemente*

```
Array.Sort(arrayAnimales);
```

- **Reverse** *Invierte todos los elementos*

```
Array.Reverse(array2);
```

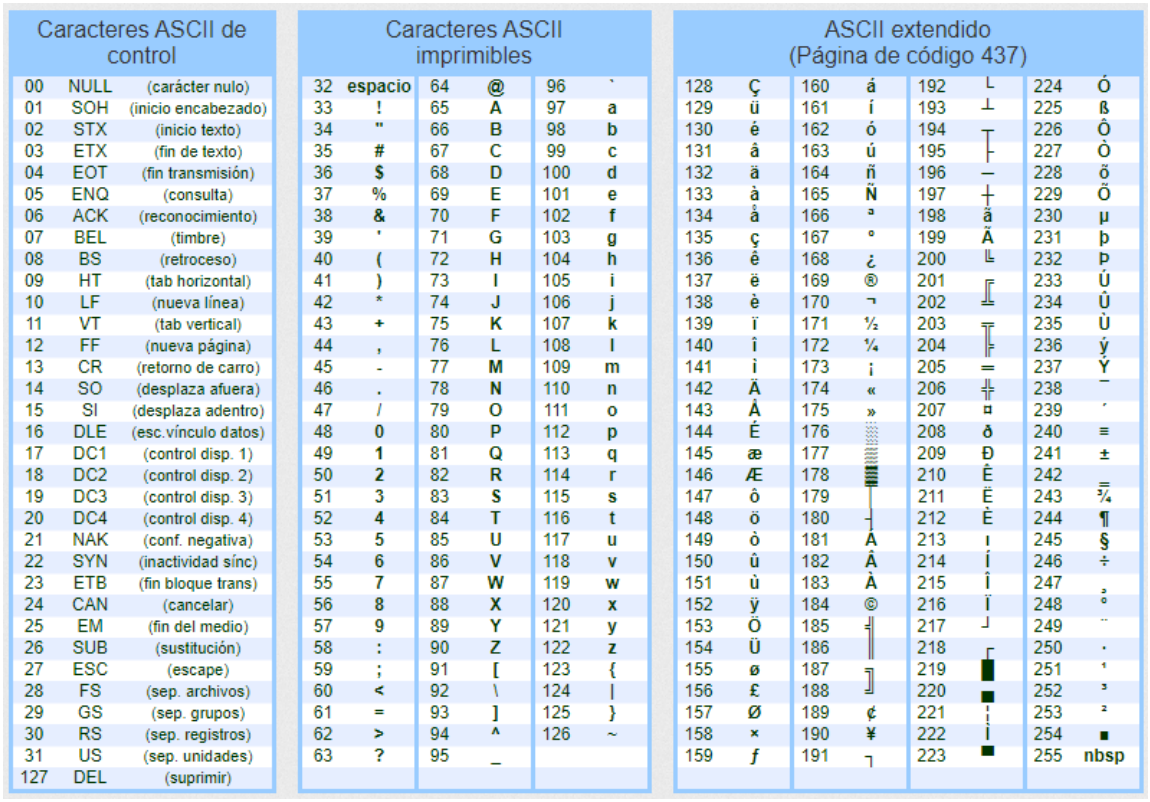
Evento KeyPress

Se ejecuta cuando se presiona una tecla con el foco en el objeto TextBox al que se asocia el evento.

Si el foco está en dicho cuadro de texto, cada tecla presionada va a desencadenar el código que se encuentre dentro de este evento:

```
private void txt_costo_KeyPress(object sender, KeyPressEventArgs e)
{
    //Este es el evento KeyPress del txt_costo
}
```

Este evento se puede aprovechar para decidir si el valor de la tecla presionada, puede o no mostrarse en el TextBox. Para eso debemos tener en cuenta los valores de cada carácter, según la tabla ASCII:



Caracteres ASCII de control

00	NULL	(carácter nulo)
01	SOH	(inicio encabezado)
02	STX	(inicio texto)
03	ETX	(fin de texto)
04	EOT	(fin transmisión)
05	ENQ	(consulta)
06	ACK	(reconocimiento)
07	BEL	(timbre)
08	BS	(retroceso)
09	HT	(tab horizontal)
10	LF	(nueva línea)
11	VT	(tab vertical)
12	FF	(nueva página)
13	CR	(retorno de carro)
14	SO	(desplaza afuera)
15	SI	(desplaza adentro)
16	DLE	(esc.vínculo datos)
17	DC1	(control disp. 1)
18	DC2	(control disp. 2)
19	DC3	(control disp. 3)
20	DC4	(control disp. 4)
21	NAK	(conf. negativa)
22	SYN	(inactividad sínc)
23	ETB	(fin bloque trans)
24	CAN	(cancelar)
25	EM	(fin del medio)
26	SUB	(sustitución)
27	ESC	(escape)
28	FS	(sep. archivos)
29	GS	(sep. grupos)
30	RS	(sep. registros)
31	US	(sep. unidades)
127	DEL	(suprimir)

Caracteres ASCII imprimibles

32	espacio	64	@	96	`
33	!	65	A	97	a
34	"	66	B	98	b
35	#	67	C	99	c
36	$	68	D	100	d
37	%	69	E	101	e
38	&	70	F	102	f
39	'	71	G	103	g
40	(	72	H	104	h
41	)	73	I	105	i
42	*	74	J	106	j
43	+	75	K	107	k
44	,	76	L	108	l
45	-	77	M	109	m
46	.	78	N	110	n
47	/	79	O	111	o
48	0	80	P	112	p
49	1	81	Q	113	q
50	2	82	R	114	r
51	3	83	S	115	s
52	4	84	T	116	t
53	5	85	U	117	u
54	6	86	V	118	v
55	7	87	W	119	w
56	8	88	X	120	x
57	9	89	Y	121	y
58	:	90	Z	122	z
59	;	91	[	123	{
60	<	92	\	124	|
61	=	93	]	125	}
62	>	94	^	126	~
63	?	95	_		

ASCII extendido (Página de código 437)

128	Ç	160	á	192	└	224	Ó
129	ü	161	í	193	┴	225	ß
130	é	162	ó	194	┬	226	Ô
131	â	163	ú	195	├	227	Ò
132	ä	164	ñ	196	─	228	õ
133	à	165	Ñ	197	┼	229	Õ
134	å	166	ª	198	ã	230	µ
135	ç	167	º	199	Ã	231	þ
136	ê	168	¿	200	╚	232	Þ
137	ë	169	®	201	╔	233	Ú
138	è	170	¬	202	╩	234	Û
139	ï	171	½	203	╦	235	Ù
140	î	172	¼	204	╠	236	ý
141	ì	173	¡	205	═	237	Ý
142	Ä	174	«	206	╬	238	¯
143	Å	175	»	207	¤	239	´
144	É	176	▒	208	ð	240	≡
145	æ	177	▓	209	Ð	241	±
146	Æ	178	▓	210	Ê	242	‗
147	ô	179	│	211	Ë	243	¾
148	ö	180	┤	212	È	244	¶
149	ò	181	Á	213	ı	245	§
150	û	182	Â	214	Í	246	÷
151	ù	183	À	215	Î	247	¸
152	ÿ	184	©	216	Ï	248	°
153	Ö	185	╣	217	┘	249	¨
154	Ü	186	║	218	┌	250	·
155	ø	187	╗	219	█	251	¹
156	£	188	╝	220	▄	252	³
157	Ø	189	¢	221	¦	253	²
158	×	190	¥	222	Ì	254	■
159	ƒ	191	┐	223	▀	255	nbsp

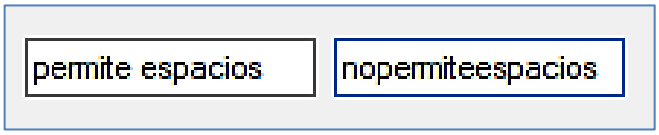
https://elcodigoascii.com.ar/

Conociendo estos valores, podremos tratar cada valor ingresado como un carácter, que forma parte de la clase **Char**.

Por ejemplo, podemos validar que en el cuadro de textos no se permita ingresar espacios. Si buscamos en la tabla ASCII el valor del espacio, vemos que es el número 32, por lo que, podríamos hacer lo siguiente:

```
private void txt_costo_KeyPress(object sender, KeyPressEventArgs e)
{
    if ((e.KeyChar) == 32)
    {
        e.Handled = true;
    }
}
```

El resultado, comparando un TextBox que no tiene ninguna validación con el txt_costo, sería el siguiente:



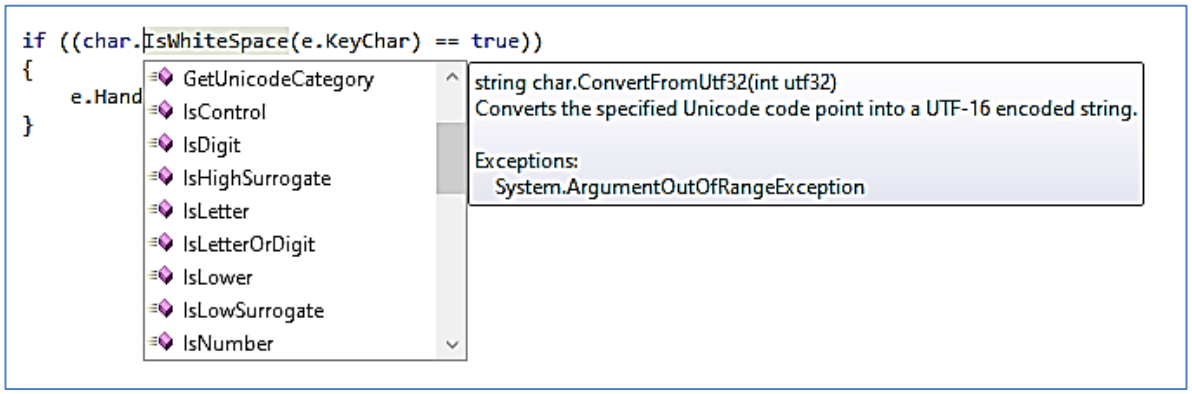
Esto se logra ya que en la variable “e” se guardan varios datos que el evento puede almacenar, uno de ellos es justamente el código ASCII de la tecla presionada.

Esta variable, de tipo `KeyPressEventArgs`, a su vez tiene propiedades, una de ellas es “Handled”, quien determina si el valor de la tecla finalmente será admitido en el control o no.

Otra forma de evitar el ingreso de espacios es la siguiente:

```
private void txt_costo_KeyPress(object sender, KeyPressEventArgs e)
{
    if ((char.IsWhiteSpace(e.KeyChar) == true))
    {
        e.Handled = true;
    }
}
```

La clase char contiene varios métodos que nos permitirán validar de forma mucho más general y no comparando el valor ingresado con cada carácter habilitado o no en nuestro control.



Lo que resta es simplemente usar condicionales que nos ayuden a validar el formato que estamos deseando.

Algunos ejemplos útiles:

```
// CONTROLAR QUE NO HAYA ESPACIOS PRIMERO NI DOS VECES SEGUIDAS

if ((e.KeyChar == 32 && txt_costo.Text.Length == 0) || (e.KeyChar == 32 &&
    e.KeyChar == txt_costo.Text[txt_costo.Text.Length - 1]))
{
    e.Handled = true;
}
```

```
//// FORMATO HORA (HH:MM)

// Si no es un número se descarta
if (!(char.IsDigit(e.KeyChar) == true))
{
    e.Handled = true;
}
// Si el caracter es número pero la pocisión de este caracter en el
// string del Text es 2, se descarta, porque ahí van los dos puntos únicamente
if (char.IsDigit(e.KeyChar) == true && txt_costo.Text.Length == 2)
{
    e.Handled = true;
}
// Si son los dos puntos y la posición en el string es 2, se permite
if (e.KeyChar == 58 && txt_costo.Text.Length == 2)
{
    e.Handled = false;
}
// Si es la tecla de "borrar", se permite
if ((e.KeyChar) == 8)
{
    e.Handled = false;
}
```

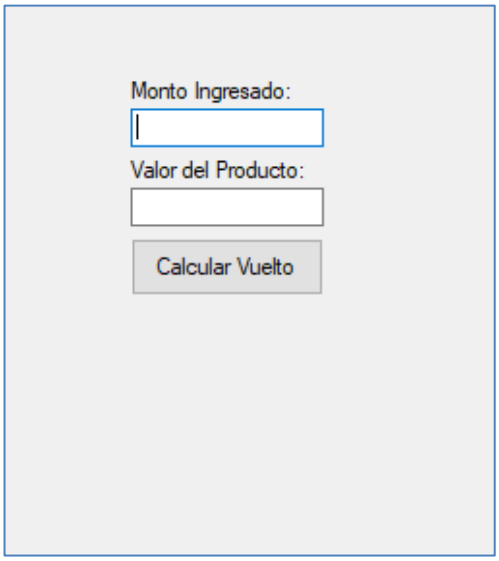
Por supuesto que existen otras herramientas más que válidas y útiles para validación de campos, será cuestión de investigar y analizar qué método o herramienta será de mayor utilidad para el requerimiento actual.

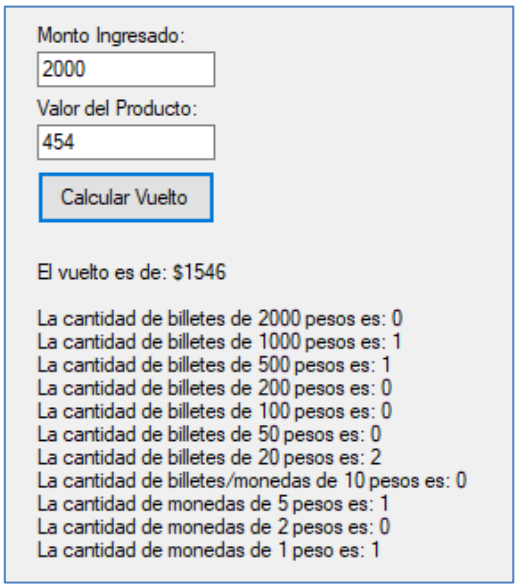
3.1. **EJERCICIO ENTREGABLE:** Utilizar el Formulario creado en la Guía 2 y desarrollar un software que calcule el vuelto de un pago en efectivo. El mismo deberá permitirle al usuario ingresar en un textBox el monto abonado, en otro textBox el costo del producto y mediante un botón deberá poder visualizar en un label qué cantidad de billetes/monedas de cada valor se requieren para el vuelto. Por ejemplo:

*El cliente abona con **$2000** un producto que cuesta **$454***

*El cajero ingresa **2000** en el campo "Monto ingresado" (txtMontoIngresado) y **454** en el campo "Valor del producto" (txtValorProducto) y presiona el botón "Calcular Vuelto" (btnCalcularVuelto).*

El resultado deseado sería similar al siguiente:





Usar el operador %. Recuerden que el resultado de este operador es lo que sobra de una división.

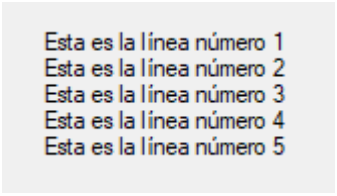
Ej: 23 **%** 10 = **3**, en cambio,

si dividimos: 23 **/** 10 = **2**. (Esta parte es clave para resolver el ejercicio)

Para mostrar el resultado usarán una sola etiqueta (label). Deberán concatenar cada resultado. Para lograr un salto de línea o "Enter", concatenen los caracteres `"\n"` al final de cada línea. Ej:

```
int contador = 0;
label1.Text = "Esta es la línea número " + (contador += 1) + "\n" +
    "Esta es la línea número " + (contador += 1) + "\n" +
    "Esta es la línea número " + (contador += 1) + "\n" +
    "Esta es la línea número " + (contador += 1) + "\n" +
    "Esta es la línea número " + (contador += 1) + "\n";
```

Resultado:



Usar KeyPress para aceptar solo el ingreso de valores enteros

Recuerden que el valor de la propiedad text de los textBox son de tipo string, por lo que deberán convertirlos a enteros antes de realizar los cálculos necesarios.