

## MÉTODOS Y ARGUMENTOS

### Guía de laboratorio – Capítulo V

#### Contenidos:

- ✓ Clase, objeto y método
- ✓ Métodos y argumentos
- ✓ Método return

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

#### CLASES, OBJETOS Y MÉTODOS

La programación orientada a objetos se basa en la programación de clases; a diferencia de la programación estructurada, que está centrada en las funciones.

Una clase es un molde del que luego se pueden crear múltiples objetos, con similares características.

Una clase es una plantilla (molde), que define atributos (variables) y métodos (funciones). En esta guía nos centraremos en estos últimos.

En C# se utiliza la palabra **new** para proveer al programador objetos recién creados con los que pueda trabajar. Al utilizar new también debemos especificar qué tipo de nuevo objeto requerimos. En otras palabras, seleccionamos la clase del objeto. C# tiene una extensa colección de clases listas para usar (como botones, etiquetas, formularios, etc.).

Por ejemplo, cuando creamos una "instancia" de una clase:

```
frmSecundario F1 = new frmSecundario();
```

Estamos creando un nuevo objeto de la clase, en este caso, un objeto al que llamaremos **F1** y que pertenece a la clase **frmSecundario**.

Para comenzar, analizaremos el concepto de un equipo real para dibujo de gráficos, y después desde la perspectiva de la programación orientada a objetos.

En el mundo real nuestro equipo de dibujo podría consistir en un montón de hojas de papel en blanco, algunos lápices y un conjunto de herramientas para dibujar formas (por ejemplo, una regla y una plantilla de figuras recortadas). Por supuesto, los lápices tendrían que ser adecuados para el papel que usemos: por ejemplo, si se tratara de hojas de acetato podríamos emplear lápices con tinta a base de aceite.

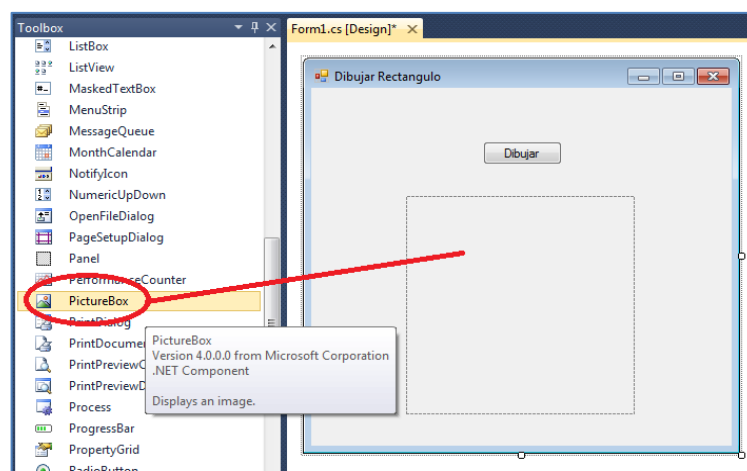
Tengamos en cuenta que no es suficiente contar con papel y plantillas; lo que nos da la oportunidad de crear dibujos y gráficos es la combinación de estos elementos.

En el desarrollo orientado a objetos, debemos solicitar a C# un área de dibujo en blanco (como hacemos al seleccionar un "nuevo" documento antes de empezar a escribir en un procesador de textos). Esta área de dibujo incluye un conjunto de métodos (funciones, operaciones) para dibujar formas. Para decirlo de otro modo: obtenemos una hoja de papel 'inteligente' que incluye un conjunto de herramientas.

Crearemos un nuevo proyecto Aplicación de Windows Forms, llamado (por ejemplo) DibujoRectangulo.

Una vez creado, debemos colocar los siguientes controles en el formulario:

button1	
Name	btnDibujar
Text	"Dibujar"
pictureBox1	
Name	picDibujo
Size	150;200 ( <i>cambiamos su tamaño</i> )
BackColor	White



Por ahora ignoraremos los detalles sobre el color y la posición del rectángulo. Lo importante es que **papel** es un objeto. Para dibujar sobre él utilizamos uno de los métodos de la clase **Graphics** que nos proporciona C#. En este caso elegiremos **DrawRectangle** (dibujar rectángulo).

Al proceso de utilizar un método se le conoce como "llamar" o "invocar" el método. Para invocar el método de un objeto utilizamos la notación "punto", es decir, colocamos un "." entre el objeto y el método que estamos invocando. Por ejemplo: `frmSecundario.Show()`; estamos llamando al método `Show` del objeto `frmSecundario`.

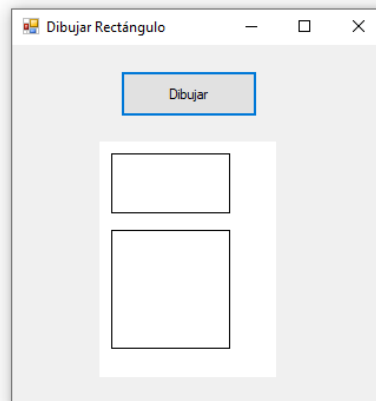
En nuestro programa, nuestra área de dibujo (a la que llamaremos **papel**) será **PictureBox**:

Crearemos un código que muestra dos rectángulos en un cuadro de imagen cuando se hace clic en el botón:

```
private void button1_Click(object sender, EventArgs e)
{
    //preparamos las herramientas de dibujo:
    Graphics papel; //nombramos "papel" a nuestra área de dibujo de tipo Graphics
    papel = pictureBox1.CreateGraphics(); //indicamos que el papel será nuestro pictureBox
    Pen lapiz = new Pen(Color.Black); //creamos el objeto llamado "lapiz" de color negro

    //dibujamos:
    //dos rectángulos con el objeto lápiz y en las ubicaciones indicadas
    //      (objeto Pen, X, Y, ancho, alto)
    papel.DrawRectangle(lapiz, 10, 10, 100, 50);
    papel.DrawRectangle(lapiz, 10, 75, 100, 100);
}
```

El resultado sería el siguiente:



Los gráficos de C# se basan en píxeles. Cada píxel se identifica mediante sus coordenadas, empezando desde cero:

- El primer número corresponde a su posición horizontal; a menudo se le denomina **x** en matemáticas, y también en la documentación de C#. Este valor se incrementa de izquierda a derecha;
- El segundo señala su posición vertical, o **y**; este valor se incrementa de arriba hacia abajo.

Cuando colocamos un objeto visual en la pantalla establecemos su posición **x** y **y**. Al dibujar en el cuadro de imagen (PictureBox) consideramos su esquina superior izquierda como el punto cero de las coordenadas horizontal y vertical. En otras palabras, dibujamos en relación con la esquina superior izquierda del PictureBox, y no de acuerdo con la esquina superior izquierda del formulario. Esto significa que si cambiamos la posición del cuadro de imagen no se verán afectados los dibujos que éste contenga. Utilizamos este sistema cuando pedimos a C# que dibuje formas simples.

El tamaño de los dibujos depende de la calidad de la pantalla del equipo en donde se ejecute el programa, y de la configuración gráfica del sistema. Los gráficos de alta calidad tienen más píxeles (aunque de menor tamaño), por lo que sus dibujos serán más pequeños.

Volviendo al programa, la siguiente instrucción es una invocación (o llamada) al método:

```
papel.DrawRectangle(lapiz, 10, 10, 100, 50);
```

En la cual le pedimos que lleve a cabo la tarea de dibujar un rectángulo. También se le dice método debido a que es un procedimiento para (o medio de) realizar cierta acción.

Cuando utilizamos el método DrawRectangle es necesario que le proporcionemos un instrumento para realizar el dibujo (el lápiz) y los valores para fijar su posición y tamaño. Tenemos que suministrarle estos valores en el orden correcto:

- un objeto Pen;
- el valor horizontal de la esquina superior izquierda del rectángulo (x);
- el valor vertical de la esquina superior izquierda del rectángulo (y);
- el ancho del rectángulo;
- la altura del rectángulo.

En C# estos elementos reciben la denominación de **argumentos**. Otros lenguajes utilizan también el término “**parámetro**”. En este caso los argumentos constituyen entradas de información (inputs) para el método DrawRectangle. Los argumentos deben ir encerrados entre paréntesis y separados por comas.

Varios métodos, al igual que este método específico, tienen varias versiones distintas, con diferentes números y tipos de argumentos.

El que utilizamos en este programa tiene cinco argumentos: un objeto Pen seguido de cuatro números enteros. Si tratamos de utilizar el número incorrecto de argumentos, o el tipo equivocado de éstos, obtendremos un mensaje de error del compilador. Para evitarlo necesitamos asegurarnos de:

- proveer el número correcto de argumentos;
- proporcionar el tipo correcto de argumentos;
- ordenarlos de la forma correcta.

Algunos métodos no requieren argumentos. En este caso también debemos utilizar paréntesis, como en este ejemplo:

```
pictureBox1.CreateGraphics();
```

En nuestro programa invocamos métodos preexistentes, como el evento Click del btnDibujar. No necesitamos invocarlo de manera explícita, ya que C# se encarga de ello cuando el usuario hace clic en el botón Button1.

## Métodos y argumentos

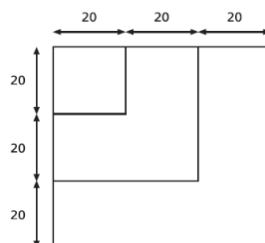
Los programas grandes pueden ser complejos, difíciles de comprender y de depurar. La técnica más importante para reducir esta complejidad consiste en dividir el programa en secciones (relativamente) independientes. Esto nos permite enfocarnos en una sección específica sin distraernos con el programa completo. Además, si la sección tiene nombre podemos “llamarla” o “invocarla” (es decir, hacer que sea utilizada por otra instancia) con sólo hacer referencia a ella. Trabajar de esta manera nos permite, en cierto sentido, pensar a un nivel más alto. En C# a dichas secciones se les conoce como métodos.

Haremos un ejemplo donde realizaremos un logo de una empresa en el cual utilizaremos una buena cantidad de métodos gráficos predefinidos para dibujar figuras en pantalla, como el método DrawRectangle, al cual podemos invocar con cinco argumentos de la siguiente forma: *(pueden reutilizar el mismo formulario y comentar el código de los dibujos anteriores)*

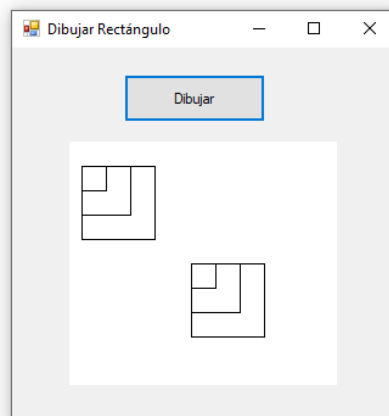
```
papel.DrawRectangle(lapiz, 10, 20, 60, 60);
```

Al utilizar argumentos (los elementos entre paréntesis) podemos controlar el tamaño y la posición del rectángulo, y garantizar que DrawRectangle será lo suficientemente flexible como para funcionar en diversas circunstancias. Los argumentos modifican sus acciones. Además, cabe mencionar que podríamos producir un rectángulo mediante el uso de cuatro llamadas a DrawLine. Sin embargo, es mucho más sensato agrupar las cuatro instrucciones DrawLine en un método conocido como DrawRectangle, ya que hacerlo de esta manera permite que el programador aproveche al máximo sus habilidades.

El logo de la empresa es el siguiente:



La idea de la aplicación siguiente es dibujar el logo dos veces mediante un pictureBox y se realiza el pedido a través de un button. La pantalla quedaría así:



Codificaremos el btnDibujar con las siguientes instrucciones:

```
// Dibuja el logotipo en la esquina superior izquierda
papel.DrawRectangle(lapiz, 10, 20, 60, 60);
papel.DrawRectangle(lapiz, 10, 20, 40, 40);
papel.DrawRectangle(lapiz, 10, 20, 20, 20);

// Dibuja el logotipo en la esquina superior derecha
papel.DrawRectangle(lapiz, 100, 100, 60, 60);
papel.DrawRectangle(lapiz, 100, 100, 40, 40);
papel.DrawRectangle(lapiz, 100, 100, 20, 20);
```

Observe que los cuadrados son de 20, 40 y 60 píxeles, y que su esquina superior izquierda está en el mismo punto. Si analiza el código observará que, en esencia, se repiten las tres instrucciones para dibujar el logotipo, independientemente de la posición de su esquina superior izquierda. A continuación agruparemos esas tres instrucciones para crear un método, de manera que pueda dibujarse un logotipo con una sola instrucción.

## Nuestro primer método

Pueden reutilizar el formulario anterior, crear uno nuevo o crear un nuevo proyecto.

Para que al ejecutar el programa se muestre otro formulario, deben acceder a la clase `Program.cs` ubicada en el explorador de la solución y cambiar el nombre del formulario en el método Run, al que desean que inicie el programa:

```
static void Main()
{
    Application.EnableVisualStyles();
    Application.SetCompatibleTextRenderingDefault(false);
    Application.Run(new Form2());
}
```

El concepto de los métodos y argumentos es una importante habilidad que los programadores necesitan dominar. Lo que haremos, entonces, será crear y utilizar un método desde cero.

La forma general de la declaración de un método cuyo propósito *no es calcular un resultado* y al que los argumentos le son transferidos por valor es:

```
private void UnNombre(lista de parámetros)
{
    cuerpo
}
```

El programador elige el nombre del método.

La lista de parámetros es una lista de tipos y nombres. Si un método no necesita argumentos utilizamos paréntesis vacíos inmediatamente después de declararlo, y también para la lista de argumentos al momento de invocarlo.

```
private void MiMétodo()
{
    cuerpo
}
```

y la llamada al método sin argumentos tendría esta forma:

```
MiMétodo();
```

Una clase puede contener cualquier número de métodos, en el orden que se desee. Los programas que empleamos como ejemplo en este capítulo sólo incluyen una clase. En esencia su distribución es:

```
public class Forma1
{
    private void UnNombre(lista de parámetros...)
    {
        cuerpo
    }

    private void OtroNombre(lista de parámetros...)
    {
        cuerpo
    }
}
```


Teniendo esta guía, la idea será realizar el mismo logo que dibujamos anteriormente, pero reducir la repetición de código y al mismo tiempo encapsularlo en un método.

El código completo sería el siguiente:

```
private void button1_Click(object sender, EventArgs e)
{
    Graphics papel;
    papel = pictureBox1.CreateGraphics();
    Pen lapiz = new Pen(Color.Black);

    DibujarLogo(papel, lapiz, 10, 20); //invocamos al evento creado
    DibujarLogo(papel, lapiz, 100, 100); //invocamos al evento creado
}

//evento creado
private void DibujarLogo(Graphics areaDibujo, Pen lapizAUsar, int posX, int posY)
{
    areaDibujo.DrawRectangle(lapizAUsar, posX, posY, 60, 60);
    areaDibujo.DrawRectangle(lapizAUsar, posX, posY, 40, 40);
    areaDibujo.DrawRectangle(lapizAUsar, posX, posY, 20, 20);
}
```



Analizaremos a detalle el programa:

```
private void DibujarLogo(Graphics areaDibujo, Pen lapizAUsar, int posX, int posY)
```

Aquí se declara (introduce, crea) el método; a esto se le conoce como encabezado del método. El encabezado declara el nombre del método (el que deseemos) y los elementos que deben suministrarse para controlar su operación (parámetros).

Al resto del método se le conoce como cuerpo, y va encerrado entre llaves `{ }`; aquí es donde se realiza el trabajo. A menudo el encabezado es una línea extensa, y nosotros podemos optar por dividirlo en puntos adecuados (aunque no en medio de una palabra).

Ej:

```
private void DibujarLogo(Graphics areaDibujo,
    Pen lapizAUsar,
    int posX,
    int posY)
```

Una importante decisión que debe tomar el programador es el lugar desde donde se puede invocar el método; en este sentido, tenemos dos opciones:

- El método sólo puede ser invocado desde el interior de la clase actual (dentro del código del mismo formulario, por ejemplo); en este caso utilizamos la palabra clave **private**.
- El método puede ser invocado desde otras clases (por ejemplo, otros formularios); en este caso utilizamos la palabra clave **public**.

Los métodos como **DrawRectangle** son ejemplos de métodos que se han declarado como **public**; son de uso general.

Otra decisión que debe tomar el programador es:

- ¿El método producirá un resultado? En el caso de que no devuelva nada, utilizamos la palabra clave **void** después de **private**.
- ¿El método calculará un resultado y lo devolverá a la sección de código que lo invocó? En este caso tenemos que declarar el tipo del resultado, en vez de usar **void**. Más adelante veremos cómo hacerlo.

En el caso del método **DibujarLogo**, su tarea consiste en dibujar líneas en la pantalla, y no en proveer la respuesta de un cálculo. Por ende, utilizamos **void**.

## Cómo invocar métodos

Para invocar un método privado en C# es preciso indicar su nombre junto con una lista de argumentos entre paréntesis. En nuestro programa la primera llamada es:

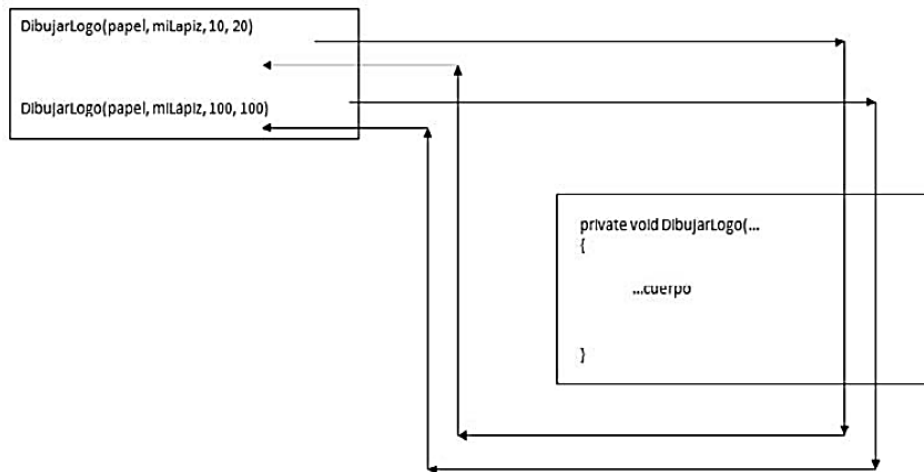
```
DibujarLogo(papel, lapiz, 10, 20);
```

Esta instrucción tiene dos efectos:

- Los valores de los argumentos se transfieren al método de manera automática.
- El programa salta al cuerpo del método (la instrucción después del encabezado) y ejecuta las instrucciones. Cuando termina con todas las instrucciones y llega al carácter `}`, la ejecución continúa en el punto desde el que se hizo la llamada al método.

Luego lleva a cabo la segunda llamada:

```
DibujarLogo(papel, lapiz, 100, 100);
```



### Cómo pasar argumentos

Es imprescindible comprender lo mejor posible cómo se transfieren (pasan) los argumentos a los métodos. En nuestro ejemplo el concepto se muestra en las siguientes líneas:

```
namespace DibujarLogo
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
        }

        private void button1_Click(object sender, EventArgs e)
        {
            Graphics papel;
            papel = pictureBox1.CreateGraphics();
            Pen miLapiz = new Pen(Color.Black);
            DibujarLogo(papel, miLapiz, 10, 20);
            DibujarLogo(papel, miLapiz, 100, 100);
        }

        private void DibujarLogo(Graphics áreaDibujo, Pen lapizAUsar, int posX, int posY)
        {
            áreaDibujo.DrawRectangle(lapizAUsar, posX, posY, 60, 60);
            áreaDibujo.DrawRectangle(lapizAUsar, posX, posY, 40, 40);
            áreaDibujo.DrawRectangle(lapizAUsar, posX, posY, 20, 20);
        }
    }
}
```

El área en la que debemos enfocarnos está constituida por las dos listas de elementos que se hallan entre paréntesis:

(papel, lapiz, 10, 20)  
(papel, lapiz, 100, 100)

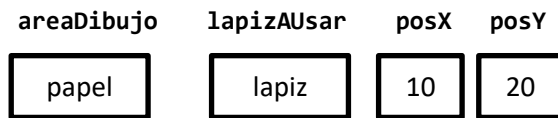
En una llamada a un método, estos elementos se denominan **argumentos**. En el encabezado del método los elementos se denominan **parámetros**. Para aclarar esta situación extraigamos los parámetros y los argumentos:

Argumentos: papel, lapiz, 10, 20

Parámetros: áreaDibujo, lapizAUsar, posX, posY



Supongamos que dentro del método hay un conjunto de cajas vacías (los parámetros) que esperan la transferencia de los valores de los argumentos. Después de la transferencia tenemos la situación que se muestra en la Figura siguiente.



La transferencia se realiza de izquierda a derecha. La llamada debe proporcionar el número y tipo correctos de cada argumento. Si el que hace la llamada (el usuario) recibe accidentalmente los argumentos en el orden incorrecto, el proceso de la transferencia no los regresará a su orden correcto.

## Parámetros y argumentos

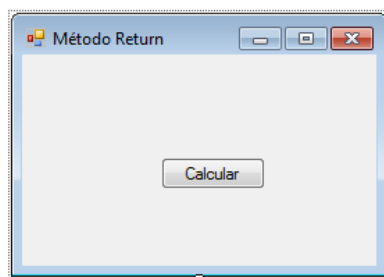
En el análisis que estamos llevando a cabo están involucradas dos listas entre paréntesis, y es importante aclarar el propósito de cada una de ellas:

- El programador que escribe el método debe elegir cuáles son los elementos que éste solicitará por medio de parámetros. En el método **DibujarLogo** las medidas de los cuadrados anidados siempre se establecen en 20, 40 y 60, de manera que la instancia que invoca el método no necesita suministrar esos datos. Sin embargo, tal vez quien haga la llamada quiera variar la posición del logotipo, utilizar un lápiz distinto o incluso dibujarlo en un componente distinto (como un botón). Estos elementos se han convertido en parámetros.
- El escritor del método debe elegir el nombre de cada parámetro. Si se utilizan nombres similares en otros métodos no hay problema alguno, pues cada uno de ellos tiene una copia propia de sus parámetros. En otras palabras, el escritor tiene la libertad de elegir cualquier nombre.
- Se debe proporcionar el tipo de cada parámetro; esta información debe ir antes del nombre del mismo. Los tipos dependen del método en particular. Se utiliza una coma para separar los parámetros entre sí.
- Quien hace la llamada debe proveer una lista de argumentos entre paréntesis, y éstos tendrán que ser del tipo correcto y estar en el orden adecuado para el método. Los dos beneficios que conlleva la utilización de un método para dibujar el logotipo son: evitamos duplicar las tres instrucciones **DrawRectangle** cuando se requieren varios logos, y al dar un nombre a esta tarea podemos ser decididamente más creativos.

## INSTRUCCIÓN RETURN

En nuestros ejemplos anteriores de argumentos y parámetros pasamos valores hacia los métodos, para que éstos los utilizaran. Sin embargo, con frecuencia es necesario codificar métodos que realicen un cálculo y envíen un resultado al resto del programa, de manera que pueda emplearlo en cálculos posteriores.

Para poner en práctica esto, en un formulario vamos a agregar un button para realizar el cálculo de un área:



En estos casos podemos utilizar la instrucción **return**. Veamos un método que calcula el área de un rectángulo, dados sus dos lados como argumentos de entrada. El siguiente es el código completo:

```
private void button1_Click(object sender, EventArgs e)
{
    int a;
    a = areaRectangulo(10, 20);
}

private int areaRectangulo(int longitud, int ancho)
{
    int area;
    area = longitud * ancho;
    return area;
}
```

Este ejemplo incluye varias nuevas características relacionadas entre sí.

Considere el encabezado del método:

```
private int areaRectangulo(int longitud, int ancho)
```

En vez de **void** especificamos el tipo de elemento que el método debe regresar a la instancia que lo invocó. Como en este caso estamos multiplicando dos valores **int**, la respuesta también es de tipo **int**.

La elección del tipo de este elemento depende del problema. Por ejemplo, el resultado que estamos buscando podría ser un entero o una cadena de caracteres, pero también podría ser un objeto más complicado, como un cuadro de imagen o un botón. El programador que escribe el método elige el tipo de valor que se devolverá.

Para devolver un valor como resultado del método utilizamos la instrucción **return** de la siguiente manera:

```
return expresión;
```

La **expresión** (como siempre) puede ser un número, una variable o un cálculo (o incluso la llamada a un método), pero es necesario que sea del tipo correcto, según lo especificado en la declaración del método (su encabezado). Además, la instrucción **return** hace que el método actual deje de ejecutarse, y regresa de inmediato al lugar en el que se encontraba dentro del método que hizo la llamada.

Ahora veamos cómo se puede invocar un método que devuelve un resultado.

La siguiente es una manera de no invocar dicho método. No debe utilizarse como una instrucción completa; por ejemplo:

```
areaRectangulo(10, 20); //NO
```

En lugar de ello, el programador debe asegurarse de utilizar el valor devuelto. Para comprender cómo devolver un valor imagine que la llamada al método (el nombre y la lista de argumentos) se elimina, y se sustituye por el resultado devuelto. Si el código resultante tiene sentido, C# le permitirá realizar dicha llamada. Vea este ejemplo:

```
a = areaRectangulo(10, 20);
```

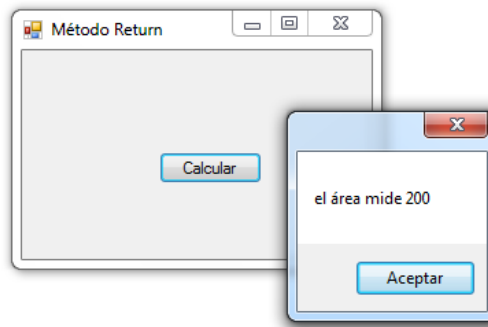
El resultado es 1200, y si sustituimos la llamada por este resultado, obtenemos lo siguiente:

```
a = 1200;
```

Por último, mostramos el resultado de la operación mostrándolo mediante un **msgbox**:

```
private void button1_Click(object sender, EventArgs e)
{
    int a;
    a = areaRectangulo(10, 20);
    MessageBox.Show("El área mide " + Convert.ToString(a));
}
```

El resultado de la ejecución sería la siguiente:



### 5.1. EJERCICIO ENTREGABLE: Crear un formulario donde deberán permitir al usuario ingresar diferentes frases a un ListBox:

El usuario podrá escribir una frase en un textBox y solo presionando la tecla **Enter**, esta frase se agregará automáticamente al listBox. (Es decir, deberán controlar esta acción en el evento KeyPress del textBox – el valor ASCII para la tecla “Enter” es 13).

Al momento de presionar la tecla “Enter” y antes de agregar la nueva frase a la lista, se deberá consultar al usuario si desea realmente agregarla (Aceptar ó Cancelar): si presiona Cancelar, volverán a hacer foco en el textBox sin agregar la frase a la lista. Si presiona Aceptar, se deberá agregar tal como se solicita, limpiar el textBox y hacer foco en él nuevamente.

Al seleccionar una frase de la lista, se deberá llamar a varios métodos que calcularán lo siguiente:

- Cantidad de caracteres en la frase.
- Cantidad de espacios vacíos en la frase.
- Primera palabra de la frase.
- Última palabra de la frase.

Verificar que la frase no contenga espacios al final.

La frase **será enviada por parámetro**, es decir que el método ya debe recibir la variable de tipo *string* con la frase (listBox1.SelectedItem.ToString()). En los métodos no podrán obtener la frase desde el objeto listBox.

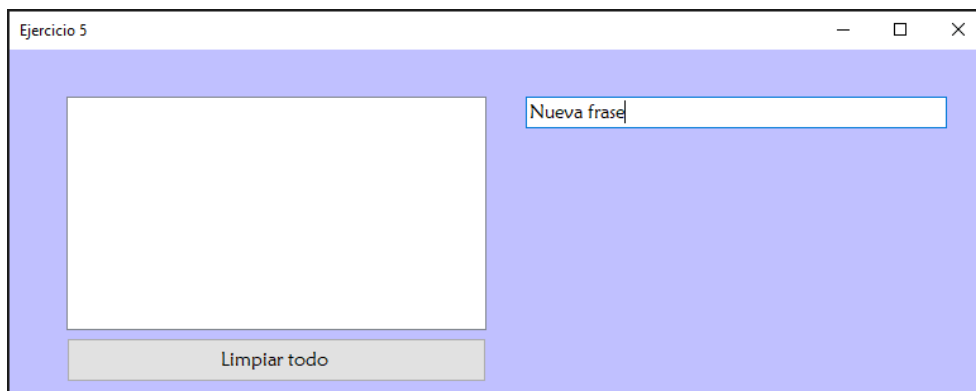
Cada método a su vez **deberá retornar** el resultado de los cálculos obtenidos, para concatenarlo a la propiedad Text de un Label. En los métodos no podrán asignar el resultado obtenido directamente al Label ni a ninguna variable global, sí o sí deberán retornar el resultado para que, desde el contexto en donde se llamaron, se muestre el valor en el label correspondiente.

También deberán agregar un botón que permita limpiar/inicializar todo el contenido del formulario. Al presionarlo deberán permitirle al usuario confirmar o no dicha acción.

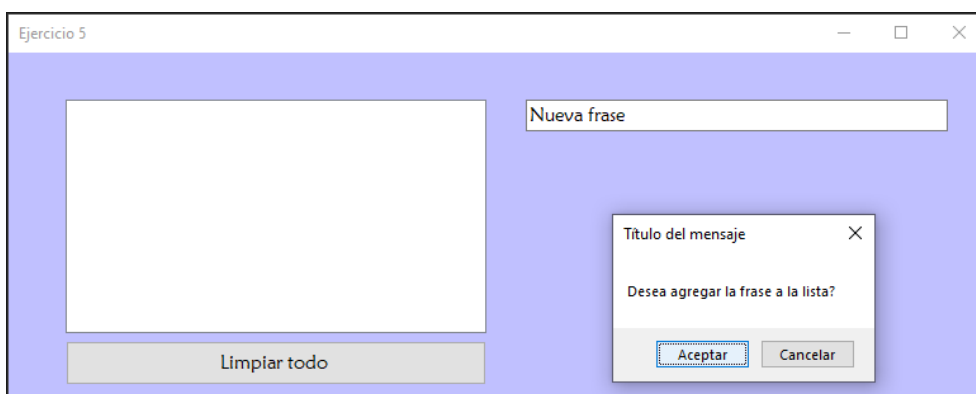
- En el caso de que el usuario confirme, se deberá vaciar el textBox que contenga la frase, los ítems de la lista y el label donde estarán los resultados de los métodos.
- En caso de que el usuario presione “No” o “Cancelar”, retorna al formulario sin realizar nada, ninguna acción ni cambios.

Ejemplo parcial a modo de orientación:

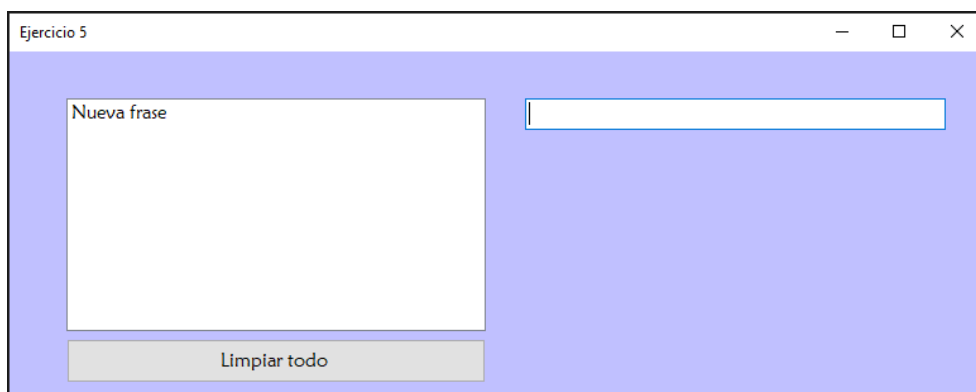
Escribo una frase:



Presiono Enter, muestra el mensaje, presiono Aceptar:



Se agrega a la lista y se limpia el textBox:



Selecciono la frase y obtengo el resultado de los cálculos:

