#2

UNDERCODE

TALLER DE PROGRAMACION C#

TEMAS

TIPOS DE DATO I/O
OPERADORES
METODOS
PARAMETROS
MODIFICADORES

TUTOR

CRAZYKADE

ERES LIBRE DE COPIAR, DISTRIBUIR Y COMPARTIR ESTE MATERIAL.

Taller de C#.NET

Entrega 2

Índice:

1. Tipos de datos de entrada/salida………………………………………………………….. 1
2. Operadores……………………………………………………………………………………….. 2
3. Métodos…………………………………………………………………………………………... 4
4. Parámetros……………………………………………………………………………………… 8
5. Modificadores…………………………………………………………………………………….. 10

1. **INTRODUCCIÓN**:

Hola a todos y bienvenidos a esta segunda entrega del taller de programación en **C#.NET**. Antes que nada quiero empezar agradeciendo por la enorme repercusión que tuvo la entrega anterior. Me llegaron un montón de mails, comentarios, sugerencias y felicitaciones. ¡Muchas gracias! Ahora sí, ¡vamos a lo nuestro!

2. **Tipos de Datos de Entrada/Salida:**

En la entrega anterior hemos hablado de los distintos tipos de datos con los que podemos trabajar en **C#.NET**. También habíamos dejado un ejercicio que ustedes tenían que tratar de resolver basado en el cálculo de un promedio de cinco calificaciones obtenidas por un alumno. La verdadera dificultad del problema era descubrir que para ingresar valores numéricos obtenidos por consola a una variable, es necesario "**parsearlos**" al tipo de dato de la variable en cuestión. ¿Se marearon?, no se preocupen, sigan leyendo.

Tal como las variables, los métodos y propiedades también poseen un tipo de dato con el que se identifican. El tipo de dato de un método, es el tipo de dato del valor que tal método devuelve. En caso de que el método no devuelva valores, será del tipo **`void`**. Veremos esto en detalle cuando estudiemos los métodos.

A estas alturas, ya debemos estar familiarizados con los métodos **`Read()`** y **`ReadLine()`** de la clase **`Console`**. Tales métodos permiten leer desde la entrada estándar de la consola, un valor determinado.

Si nos fijamos en la descripción del texto de ayuda, veremos que antes de **`Console.Readline()`**, tenemos la palabra **`string`**. Esto significa que el método, devuelve un valor de tipo cadena (**string**). Este valor podemos asignarlo de forma directa, (mediante el operador de asignación "**=**"), solamente a una variable de tipo cadena. Por esto mismo es que de seguro, han obtenido el siguiente error mientras trataban de hacer el ejercicio del promedio:

```
Console.Write("\nNota de lengua: ");
lengua = Console.ReadLine();
    class System.Console
    Represents the standard input, output, and error streams for console applications. This class cannot be inherited.

    Error:
      Cannot implicitly convert type 'string' to 'float'
```

Este error, por demás de típico, se debe a que estamos queriendo asignar a una variable de tipo **`float`**, el valor de la salida de consola que es siempre de tipo **`string`**. Entonces, el framework no puede convertir de manera implícita, (automática, sin intervención del programador), el tipo de datos **`string`** a **`float`**. Por tanto, tenemos que ser nosotros, quienes hagamos la conversión de datos explícitamente.

```
Console.ReadLine(
    string Console.ReadLine()
    Reads the next line of characters from the standard input stream.
```

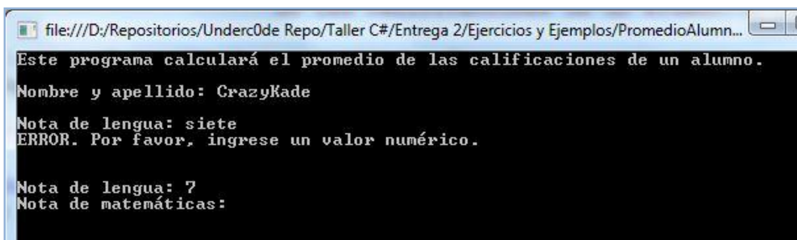
```
Console.Write("\nNota de lengua: ");
lengua = float.Parse(Console.ReadLine());
```

Para **parsear** un tipo de dato a otro diferente, tenemos que utilizar el método **`Parse()`** de la clase correspondiente al tipo de dato de destino. En este caso necesitamos parsear un valor de tipo **`string`** a un valor de tipo **`float`**. Para esto simplemente escribimos lo siguiente: **`[variable de tipo float] = float.Parse([valor de tipo string]);`** Aplicado a nuestro ejemplo nos quedaría tal y como se ve en la imagen. En este punto, los estudiantes más avanzados se estarán preguntando qué pasa si por error el usuario ingresa un valor que no sea un número. Pues el **framework** de todas formas intentará parsearlo a tipo **`float`** y cuando vea que no puede lanzará un error en tiempo de ejecución (aquí llamado **Excepción**).

NOTA: La explicación que viene a continuación es <u>solamente</u> para estudiantes avanzados que tienen ya un conocimiento previo en las tecnologías .NET o bien provienen de lenguajes como Java, C o C++. El resto de los estudiantes pueden saltear esta parte ya que estudiaremos el control de excepciones más adelante a su debido tiempo.

Este es un problema muy común con el que nos toparemos más de una vez. La solución, si bien es sencilla, requiere un cierto análisis. La idea consiste en tener inicializada la variable numérica en cero y realizar el parseo dentro de una estructura repetitiva hasta que la variable numérica sea distinta de cero. También necesitaremos una estructura **TRY/CATCH** que maneje la excepción y nos informe por pantalla que se está cometiendo un error.

Ésta es la manera más prolija y ordenada de resolver una situación tan típica como esta. De esta forma queda totalmente comprobado el gran poder que tiene el programador frente al usuario. En el ejemplo, se fuerza a este último a realizar una acción concreta decidida de antemano por el programador. Como resultado tenemos un comportamiento como el que se puede apreciar en la imagen.



Vemos claramente cómo el usuario se equivoca e ingresa la nota en forma de palabra. El programa reacciona y le muestra el error. Luego, el usuario recompone y sigue adelante.

```
float lengua = 0;

do
{

    try
    {

        Console.Write("\nNota de lengua: ");
        lengua = float.Parse(Console.ReadLine());

    }
    catch (Exception ex)
    {
        Console.WriteLine("ERROR. Por favor, ingrese un valor numérico.\n");
    }

} while (lengua == 0);
```

3. <u>Operadores Matemáticos:</u>

Symbol	Description
+	Add (and string concatenation)
–	Subtract
*	Multiply
/	Divide
%	Modulo (whole numbers remaining after division)
++	Increment by 1, may be pre- or post-increment
--	Decrement by 1, may be pre- or post-decrement
Compound Operators	
+=	Takes current value and adds to assigned value
–=	Takes current value and subtracts from assigned value
*=	Takes current value and multiplies by assigned value
/=	Takes current value and divides by assigned value
%=	Takes current value and performs modulo of assigned value

Otro de los aspectos esenciales de un lenguaje de programación son los famosos **operadores matemáticos**. Gracias a ellos y en combinación con los diversos tipos de datos numéricos, podemos realizar cualquier operación matemática que necesitemos. Para ilustrar un poco el tema, he tomado "prestada" la siguiente tabla:

Esta tabla resulta por demás de ilustrativa ya que hace diferencia entre los operadores simples o simbólicos y los

operadores compuestos. Este capítulo está dedicado a explicar el correcto y completo funcionamiento de los mismos.

Podemos observar, que dentro de los operadores simples encontramos el operador de suma, resta, producto (multiplicación) y cociente (división). También tenemos el operador de módulo, que nos devolvería el resto o residuo de una división, y los operadores incrementales y decrementales (++ y --).

En **C#.NET** es muy fácil trabajar con operadores matemáticos. Para demostrarlo veremos el siguiente ejemplo en donde realizaremos un programa simple cuyo objetivo es sumar 2 números enteros.

```
 7 namespace Operadores
 8 {
 9     class Program
10     {
11         static void Main(string[] args)
12         {
13             float n1, n2, resul;
14 
15             Console.WriteLine("Este programa sumará 2 números.\n");
16 
17             Console.Write("Ingrese el primer número: ");
18             n1 = float.Parse(Console.ReadLine());
19 
20             Console.Write("Ingrese el segundo número: ");
21             n2 = float.Parse(Console.ReadLine());
22 
23             resul = n1 + n2;
24 
25             Console.WriteLine("\nEl resultado de la suma es: {0}\n",resul);
26 
27             Console.Write("Presione una tecla para continuar...");
28             Console.Read();
29         }
30     }
31 }
32 
```

En el ejemplo vemos que en la línea 13 hay una declaración de tres variables de tipo `float`: `n1`, `n2` y `resul`. Dentro de ellas almacenaremos tanto los valores que el usuario desee sumar como también el resultado de la suma correspondiente. En la línea 18 y 21 vemos que nos volvemos a enfrentar con el problema típico de inconsistencia entre tipos de dato. Para solucionarlo hacemos uso del método `parse()` como ya hemos estudiado anteriormente. La línea 23 muestra la ejecución de la operación algebraica que en este caso es una suma. Al mismo tiempo asigna el resultado de la operación en la variable `resul`.

También podríamos realizar la operación algebraica en la misma línea en donde tenemos la llamada al método WriteLine de la clase console como lo muestra la imagen a continuación.

```
Console.WriteLine("\nEl resultado de la suma es: {0}\n",n1 + n2);
```

Esta acción realizará la operación correspondiente y luego la imprimirá en la consola a modo de salida.

- **Ejercicio De Aplicación:**

Para aplicar todo lo que hemos aprendido hasta ahora, vamos a desarrollar un programa que nos permita convertir una dirección IP expresada en base 256 a su correspondiente en base 10 (decimal). Una dirección IP (v4) no es más que un conjunto de cuatro cifras en base 256 (cuyo intervalo numérico va de 0 a 255) separados por un punto. Para realizar el algoritmo de forma algebraica, vamos a designar con una letra a cada una de estas cifras. Entonces vamos a decir que nuestra IP sería: a.b.c.d

El algoritmo para convertir una IP de base 256 a base 10 es el siguiente:

$$a * 256^3 + b * 256^2 + c * 256^1 + d * 256^0$$

Según la imagen, tenemos que tomar la primera cifra y multiplicarla por 256 elevado a la 3 y así sucesivamente con las demás cifras solamente decrementando el exponente en cada término. Traduzcamos ahora este algoritmo a código fuente C#.

```
static void Main(string[] args)
{
    int s1, s2, s3, s4;
    double resul;

    Console.Write("Ingrese la primera cifra: ");
    s1 = int.Parse(Console.ReadLine());

    Console.Write("Ingrese la segunda cifra: ");
    s2 = int.Parse(Console.ReadLine());

    Console.Write("Ingrese la tercera cifra: ");
    s3 = int.Parse(Console.ReadLine());

    Console.Write("Ingrese la cuarta cifra: ");
    s4 = int.Parse(Console.ReadLine());

    resul = s1 * (Math.Pow(256, 3)) + s2 * (Math.Pow(256, 2)) + s3 *
        (Math.Pow(256, 1)) + s4 * (Math.Pow(256, 0));

    Console.WriteLine("\nLa ip en base 10 es: {0}\n",resul);

    Console.Write("Presione una tecla para continuar...");
    Console.Read();
}
```

En primer lugar declaramos 4 variables que nos van a servir para almacenar cada una de las cifras de la dirección IP. También declaramos una variable **`resul`** del tipo **`double`** para almacenar el resultado de la operación algebraica. Esta variable tiene que ser de tipo **`double`** ya que haremos uso del método **`Pow()`** de la clase **`Math`**, el cual devuelve un tipo de dato **`Double`**.

Las ocho líneas que siguen le piden al usuario que ingrese cada una de las cifras de la IP y las almacena en las variables correspondientes. Luego realizamos la operación algebraica que vimos en el algoritmo. En **C#.NET**, para obtener una potencia de un número, el **framework** nos regala la clase **`Math`**, la cual contiene muchos métodos que realizan operaciones matemáticas. Uno de ellos es el método **`Pow()`**. Este método lo usaremos cada vez que queramos obtener la potencia de un número. Por ejemplo, para obtener el resultado de 100 al cuadrado deberemos usar el método **`Pow()`** de la siguiente forma: **`Math.Pow(100,2)`**. El primer parámetro es la base y el segundo es el exponente. Presten atención al valor que devuelve el método **`Pow()`**, ya que es de tipo **`double`** y si no lo asignan a una variable de ese mismo tipo tendrán que **parsearlo**. El uso de paréntesis en la resolución, indican el orden en el que se deben resolver cada una de las operaciones dentro de cada término.

4. Métodos:

Como habíamos visto anteriormente, **C#** implementa al 100% el paradigma de ***programación orientada a objetos*** o **POO** (de sus siglas en inglés *Object Oriented Programming*). Este paradigma se apoya sobre 4 pilares fundamentales: **Abstracción, Encapsulamiento, Herencia y Polimorfismo**. Parte del concepto de **abstracción** y de **encapsulamiento** tiene su base racional en los **métodos**, ya que estos nos permiten definir comportamiento de forma aislada y encapsulada. Pero, ¿qué es en sí un método? Un método, también llamado **función** o **procedimiento** en lenguajes **no orientados a objetos**, es un bloque de sentencias que ejecuta una tarea específica y al que nos referimos mediante un nombre. El bloque es el cuerpo del método y el nombre del bloque es el nombre del método. Cuando se escribe un método, además del nombre y el cuerpo, en general hay que especificar también los **parámetros** en los que se apoyan las operaciones que tiene que realizar y el tipo de dato del resultado que retornará. Por ejemplo:

```
int Sumar(int num1, int num2)
{
    return num1 + num2;
}
```

```
void Escribir(string texto)
{
    Console.WriteLine(texto);
}
```

Aquí podemos observar un método llamado sumar, el cual recibe como parámetro 2 valores de tipo **`entero`** y retorna un valor de tipo también **`entero`**. Existen métodos que realizan tareas que no necesitan devolver ningún valor. Para estos métodos, el tipo de datos será siempre **`void`**.

Este tipo de métodos, en los lenguajes no orientados a objetos, generalmente reciben el nombre de procedimientos.

Los métodos, como bien decíamos al principio, fueron concebidos con la idea de definir funcionalidad adicional o aislada del programa principal. Esto cumple con la premisa de reutilización de código. Lo que significa que siempre que tengamos código que se repita en dos o más partes de nuestro programa, podemos aislar este código e implementarlo dentro de un método. Este método será llamado o invocado desde las partes del programa que lo necesiten. Un programa bien estructurado en métodos es mucho más fácil de mantener en el largo plazo que uno que no lo esté; ya que a la hora de hacer una modificación en la funcionalidad, solo debemos hacerla en el método. De otra manera tendríamos que modificar tal funcionalidad en todas las partes del programa, de forma manual, en donde apareciese.

Veamos el siguiente ejemplo:

```
Class program
{
        Static void Main(string[] args)
        {
                int horas;

                Console.WriteLine("Vamos a calcular el sueldo de un empleado normal.\n");

                Console.Write("Cantidad de horas trabajadas: ");
                horas = int.Parse(Console.ReadLine());

                Console.WriteLine("\nConsiderando que la hora del empleado vale 40 pesos.\n");
                Console.Write("SUELDO: ${0}", CalcularSueldo(horas, 40));

                Console.WriteLine("\n------------------\n");

                Console.WriteLine("Ahora vamos a calcular el sueldo de un jefe.\n" +
                "Consideramos para esto el valor de la hora a 60 pesos.\n");

                Console.Write("Cantidad de horas trabajadas: ");
                horas = int.Parse(Console.ReadLine());

                Console.Write("\nSUELDO: ${0}", CalcularSueldo(horas, 60));
                Console.WriteLine("\n------------------\n");

                Console.Read();
        }

        Static float CalcularSueldo(int Horas, float precioHora)
        {
                return Horas * precioHora;
        }
}
```

Tenemos aquí un programa que calcula el sueldo de un empleado regular y un jefe basándose en la cantidad de horas trabajadas y el valor de la hora. Por supuesto que el valor de la hora de un empleado regular no será el mismo que el de un jefe. Vamos a utilizar entonces un método llamado **`CalcularSueldo`** de tipo **`float`** que permita pasar por parámetro la cantidad de horas y el valor de la misma. Como se puede apreciar, el método está declarado como **`estático`** (palabra reservada **`static`** antes del tipo de dato). No voy a explicar qué es esto todavía porque aún no vimos en profundidad la teoría de objetos. Solo diré que un método estático es un método que pertenece a la clase y no al objeto, por tanto puede ser llamado sin problemas solamente invocando el nombre de la clase, o si estamos dentro de la misma, como en este caso, simplemente usando el nombre del método. Como aún no sabemos cómo crear objetos ni como instanciarlos, vamos a usar métodos estáticos para los ejemplos. En la vida real, tendríamos todo organizado en objetos por tanto para acceder a sus métodos deberíamos invocar al objeto en cuestión. Si se marearon no den mucha importancia, estudiaremos esto en detalle en el futuro.

Vemos en el ejemplo que el método **CalcularSueldo** es declarado y definido fuera del ámbito del método **main()** pero dentro del ámbito de la clase, (en este caso la clase se llama **program**). De esta forma, indicamos que el método **CalcularSueldo**, pertenece a la clase **program** y es de tipo **float**, esto significa que retornará un valor de coma flotante (coloquialmente llamado decimal). En la firma vemos que requiere dos parámetros: **horas**, de tipo **entero** y **precioHora** de tipo coma flotante. Ya en el cuerpo del método encontramos su funcionalidad: Simplemente multiplica la cantidad de horas por el precio de la hora. El resultado es devuelto al punto del programa en donde fue llamado el método.

Pasemos ahora a desarrollar un ejercicio en donde veremos un problema planteado en forma secuencial y el mismo problema planteado desde un punto de vista modular (dividiendo el programa principal en sub programas llamados métodos). El enunciado será el siguiente: Un motor consume 1.7 litros de combustible en el lapso de una hora de funcionamiento. El litro de tal combustible tiene un precio de 9 pesos. Escribir un programa que ayude al usuario a saber cuánto tiene que gastar en combustible según las horas que necesite tener encendido el motor. Para esto el usuario ingresará las horas de trabajo, y el programa le dirá el costo operativo del motor. ¡Vamos al código!

```
using System;

namespace Motor
{
    class Program
    {
        static void Main(string[] args)
        {
            //La constante del consumo del motor. 1.7 litros cada una hora
            const float CONSUMO = 1.7f;
            const float PRECIO_L = 9;

            float horas;
            float litros;
            float precioTotal;

            Console.WriteLine("Este programa hace un calculo de relación de costo/beneficio.");
            Console.WriteLine("-------------------------------------------------------------\n");

            Console.Write("Ingrese las horas: ");
            horas = float.Parse(Console.ReadLine());

            //Calculamos los litros a usarse en total
            litros = horas * CONSUMO;

            //Calculamos el valor total del combustible
            precioTotal = litros * PRECIO_L;

            Console.WriteLine();

            Console.WriteLine("El precio total del combustible es: ${0}\n", precioTotal);

            //Pausamos la ejecución
            Console.Write("Presione una tecla para salir...");
            Console.ReadLine();
        }
    }
}
```

Este código resuelve nuestra situación problemática de una forma lineal, y secuencial. Cada acción comienza cuando finaliza la anterior y el sentido de ejecución es de arriba hacia abajo. Según la teoría de la programación estructurada, tenemos que dividir una situación problemática, en cápsulas, o situaciones problemáticas más pequeñas y puntuales de tal forma que el código resultante sea más prolijo y más fácil de mantener. Si bien esto es lo que reza la teoría, veremos que en la práctica no siempre es así.

Pasemos ahora a estudiar el mismo código pero planteado desde el paradigma de la programación modular o procedural:

```
using System;

namespace Motor
{
    class Program
    {
        static void Main(string[] args)
        {
            //La constante del consumo del motor. 1.7 litros cada una hora
            const float CONSUMO = 1.7f;
            const float PRECIO_L = 9;

            float horas;
            float litros;
            float precioTotal;

            Escribir_Bajar("Este programa hace un calculo de relación "
                + "de costo/beneficio.");
            Console.WriteLine("----------------------------------------"
                + "---------------------\n");

            Escribir("Ingrese las horas: ");
            horas = float.Parse(Console.ReadLine());

            //Calculamos los litros a usarse en total
            litros = Calcular_Litros(horas, CONSUMO);

            //Calculamos el valor total del combustible
            precioTotal = Calcular_Valor_Comb(litros, PRECIO_L);

            Escribir_Bajar("");

            Escribir_Bajar("El precio total del combustible es: $"
                + precioTotal.ToString() + "\n");

            //Pausamos la ejecución
            Pausar_Ejecucion();
        }

        private static void Escribir(string texto)
        {
            Console.Write(texto);
        }

        private static void Escribir_Bajar(string texto)
        {
            Console.WriteLine(texto);
        }

        private static float Calcular_Litros(float horas, float consumo)
        {
            return horas * consumo;
        }

        private static float Calcular_Valor_Comb(float Litros, float Precio)
        {
            return Litros * Precio;
        }

        private static void Pausar_Ejecucion()
        {
            Escribir("Presione una tecla para salir...");
            Console.ReadLine();
        }
    }
}
```

En este código, podemos observar de qué manera, cada operación importante es aislada y comprendida dentro de un método creado a tal fin. A primera vista vemos que el código ha crecido en

tamaño, pero se ha vuelto más prolijo y estructurado. A medida que el código crezca, posiblemente sea necesario calcular el valor del combustible o los litros consumidos desde varias partes diferentes del programa. Teniendo los cálculos expresados en métodos, no necesitamos más que llamarlos y obtener su resultado. De otra forma, tendríamos que escribir el mismo código tantas veces como lo necesitáramos.

Saliéndonos ya un poco de la teoría y adentrándonos en un terreno más práctico, es menester mencionar que si bien esta forma de trabajar parece, a los ojos del neófito, la octava maravilla del mundo, en realidad, si lo aplicamos en exceso puede complicarnos mucho la vida. Hay que recordar siempre que ningún extremo es bueno, y esto no es la excepción. Para decirlo de otra forma, quienes hemos estudiado normalización de bases de datos, sabemos a la perfección que muchas veces normalizar más de la cuenta es contraproducente para el correcto funcionamiento de la lógica de un sistema. Bueno, aquí pasa lo mismo, muchas veces si se nos va la mano aislando procesos, podemos terminar metidos en un embrollo de código inentendible hasta para nosotros mismos.

5. Parámetros:

Los métodos (procedimientos, funciones, rutinas, sub rutinas) como ya vimos anteriormente, pueden o no recibir parámetros de la misma forma en la que pueden o no retornar algún valor.

```
private static int Sumar (int N1, int N2)
{
    return N1 + N2;
}
```

En el ejemplo vemos un método llamado **Sumar**, de tipo entero, que recibe 2 parámetros: **N1** y **N2**. Ya dentro del cuerpo vemos que se realiza la suma entre ambos y se devuelve el resultado. Esta forma de pasarle parámetros a un método se llama "**pasaje por valor**"; es decir, lo que en realidad se le pasa al método es una copia de una variable o un valor en forma literal. Continuemos con el ejemplo:

```
static void Main(string[] args)
{
    int Num1;
    int Num2;
    int Resultado;
    //-----------------

    Num1 = 10;
    Num2 = 15;

    //-----------------

    Resultado = Sumar(Num1, Num2);

    Console.WriteLine(Resultado);
    Console.Read();
}
```

Cuando llamamos a la función **Sumar**, le pasamos como parámetro las variables **Num1** y **Num2**, aunque en realidad, lo que hace el **framework**, es crear en memoria una copia de estas variables y pasarle al método un **puntero** (dirección de memoria) a las mismas. Esto quiere decir que, como estaremos trabajando con una copia de las variables originales, si los datos se modifican, solo estaremos modificando las copias y no los valores base. Veamos un ejemplo:

```
private static int Sumar(int N1, int N2)
{
    N1 = N1 + 3;
    N2 += 3;

    return N1 + N2;
}
```

Vamos a modificar el método **Sumar** para que realice un incremento en 3 en las variables que han sido pasadas por valor, o sea son copias. Al método **Main** de la clase también vamos a hacerle una pequeña modificación:

```
static void Main(string[] args)
{
    int N1;
    int N2;
    int Resultado;
    //-----------------

    N1 = 10;
    N2 = 15;

    //-----------------

    Resultado = Sumar(N1, N2);

    Console.WriteLine("Variables Originales:");
    Console.WriteLine("---------------------\n");
    Console.WriteLine("N1 = {0}", N1);
    Console.WriteLine("N2 = {0}\n", N2);
    Console.WriteLine("---------------------\n");
    Console.WriteLine("Resultado de la suma: {0}\n",Resultado);

    Console.Read();
}
```

Cuando ejecutemos, veremos que la salida de la consola es la siguiente:

```
Variables Originales:
---------------------

N1 = 10
N2 = 15

---------------------

Resultado de la suma: 31
```

Pues… ¿Qué ha pasado? Es obvio que la suma de 10 más 15 no es 31. La respuesta es sencilla, solo se han modificado los valores de las variables copia, pero nunca el valor original de las variables base. Imagino que te debes estar preguntando: ¿Y qué pasaría si yo necesito modificar los valores originales de las variables base dentro de un método? En este caso, solo basta con cambiar la forma en la que el método recibe los parámetros. Hay que indicarle al **framework** que en vez de pasarle al método una copia de una variable, le pase directamente el **puntero** a la variable base. ¿Cómo se logra eso? Veámoslo con un ejemplo:

```
private static int Sumar(ref int N1, ref int N2)
{
    N1 = N1 + 3;
    N2 += 3;

    return N1 + N2;
}
```

Vemos en el código que ahora aparece la palabra "`ref`" en la firma del método. Esto le indica al **frammework** que le pase una **referencia** (**puntero**) a la variable base, por tanto cuando se produzcan modificaciones de tal variable en el método, se modificará también el contenido original. También es necesario modificar la forma en la que se llama al método y añadir la palabra reservada "`ref`" antes de cada parámetro.

```
Resultado = Sumar(ref N1, ref N2);
```

De esta forma, cuando compilemos y ejecutemos el programa obtendremos la siguiente salida de consola:

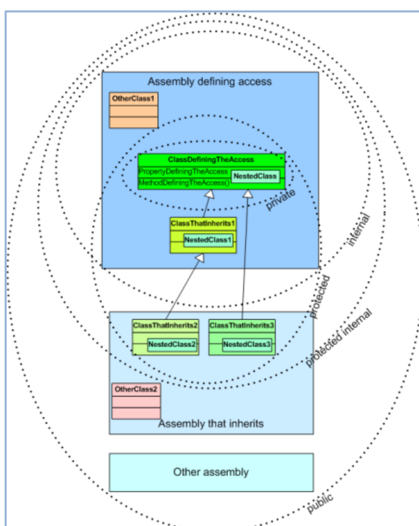
```
Variables Originales:
---------------------

N1 = 13
N2 = 18

---------------------

Resultado de la suma: 31
```

Resumamos: **En C#.NET** existen dos formas de pasarle parámetros a un método: **Por valor** y **por referencia**. La primera crea en memoria una copia exacta de la variable base y la segunda pasa un puntero o referencia conteniendo la dirección de memoria donde se encuentra alojada la variable original. La primera no modifica el valor base cuando se modifica la variable copia dentro del método y la segunda sí realiza tal modificación.

Ejercicios de aplicación:

- Escribir un programa que calcule el valor final de un producto considerando el Impuesto al **Valor Agregado** (**IVA**) que es de un **21%** sobre el precio base del producto. Para esto se necesitará un método que reciba por referencia el precio del producto y le aplique el **IVA** o sea, lo multiplique por **1,21**. Luego desde otro método, recibir por valor el precio final del producto e imprimirlo por consola.

6. Modificadores:



Hemos aprendido mucho sobre métodos, operadores y parámetros, pero aún nos falta abarcar un tema que no es menos importante: **Los modificadores de acceso**.

Es muy importante, como programadores, tener el mayor control posible sobre el código para así evitar errores de estructuración y prevenirnos de un comportamiento defectuoso o paralelo al que originalmente hemos ideado. Uno de los factores a tal fin es **la accesibilidad**.

Mediante accesibilidad podemos decidir desde qué parte del programa puede ser accedido un método o un tipo. La **MSDN** nos dice que los modificadores de acceso son palabras clave que especifican la

accesibilidad declarada de un miembro o un tipo. Estos son cuatro: **`Public`, `Protected`, `Internal`** y **`Private`**. Existe un quinto llamado **`Protected Internal`** que no es más que la combinación de estos dos.

Anteponiendo uno de estos modificadores a la declaración de un método o de un tipo, además de declararlo, estaríamos indicando su **dominio de accesibilidad**. Vamos ahora a estudiar cada uno de estos modificadores por separado.

`Public`: Un tipo o un método público, es decir, antecedido por el modificador public, es accesible desde cualquier parte del código. Ya sea desde dentro o fuera de la **clase** y desde dentro y fuera del **ensamblado**. Recordemos que como **.NET** es una plataforma de código pre-compilado, (no compila a código máquina sino a un lenguaje intermedio llamado **Common Intermediate Language** o simplemente **CIL**), no podemos hablar de ejecutables o de binarios, por tanto hablamos de ensamblados sea este un **.exe** o un **.dll**.

```
public void MetodoPublico()
{
    Console.WriteLine("Esto es un método público.");
}
```

`Protected`: Este modificador brinda acceso limitado a la clase contenedora o a los tipos derivados de esa clase. Como todavía no vimos **clases** ni **herencia**, no voy a entrar en demasiado detalle. Solo comentaré que un método o tipo designado con este modificador solo puede ser accedido desde la **clase** en donde se declaró o desde otra **clase** o tipo que **hereden** de ella. No te preocupes si no entendés bien esto, más adelante volveremos a explicarlo.

```
protected void MetodoProtegido()
{
    Console.WriteLine("Este es un método protegido");
}
```

`Internal`: Acceso limitado al ámbito del ensamblado actual. Este modificador limita al método o tipo al ensamblado en donde se encuentre contenido, no pudiendo ser accedido desde otro ensamblado externo como puede ser una dll.

```
internal void MetodoInterno()
{
    Console.WriteLine("Este es un método interno");
}
```

`Protected Internal`: Acceso limitado al **ensamblado** actual o los tipos derivados de la **clase** contenedora. Este modificador combina los dos anteriores; por tanto un método o un tipo que lo porte solo podrá ser accedido desde el **ensamblado** contenedor o los métodos o tipos derivados de la **clase** en donde fue declarado.

```
protected internal void MetodoProtegidoInterno()
{
    Console.WriteLine("Este es un método protegido interno");
}
```

`Private`: Acceso limitado al tipo contenedor. Es el más limitado de los modificadores. Indica que el método o tipo que lo porte solo es accesible desde su ámbito o **dominio de accesibilidad**. Por ejemplo, si

declaramos una variable como privada dentro de un método, ésta no podrá ser accedida desde fuera del mismo. Al igual que un método privado no podrá ser accedido desde fuera de la clase que lo contenga.

```
private void MetodoPrivado()
{
    Console.WriteLine("Este es un método privado");
}
```

Por ahora esto es todo en referencia a los modificadores de acceso. Es normal ver programadores novatos que al no comprender este tema optan por declarar todo como público, lo cual generalmente, cuándo el código crece, genera un sinnúmero de problemas difíciles de trazar y resolver de forma eficaz. Aunque este tema pueda resultar muy teórico y aburrido, es fundamental agarrarle la mano para poder escribir código limpio y entendible. Este taller, además de dar herramientas para poder desarrollar aplicaciones robustas en **C# .NET**, también pretende conducir al estudiante por la senda de las ***buenas prácticas de programación***. Esto es: Escribir código limpio, ordenado, prolijo y seguro.

A lo largo de los 13 años que llevo como programador, me ha tocado enfrentarme muchas veces con dos tipos de escenarios a la hora de entrar a trabajar a una empresa como programador. El primero de estos escenarios es cuando la empresa contrata un equipo de programadores para comenzar un desarrollo necesita. Acá no hay grandes inconvenientes, ya que al no haber nada hecho, podes tomarte la libertad de plantear el desarrollo a tu manera, o de forma consensuada con todo el equipo. El verdadero problema es el escenario número dos. Este se da cuando la empresa te contrata para integrarte a un equipo de desarrollo que ya viene trabajando hace tiempo. Puede ser porque renunció un miembro del equipo y hay que reemplazarlo, o porque la demanda de trabajo sobrepasa la capacidad actual del equipo y necesitan ampliarlo. En cualquier caso lo más probable es que terminemos enfrentándonos a un sistema ya empezado, con su código en una etapa avanzada, y si el programador anterior no hizo las cosas bien y no implementó conceptos de buenas prácticas de programación, creeme que te las vas a ver en colores. Lo peor que te puede pasar es tener que sentarte frente a un código inentendible, sin comentarios, sin documentación, y totalmente desprolijo. Perderás mucho tiempo entendiendo qué demonios quiso hacer el anterior programador antes de que puedas escribir tu primera línea de código. Esto sumado a la presión de la empresa por avanzar en el desarrollo te dejarán en el psicólogo al cabo de un tiempo. Y como no hay que hacer lo que no nos gusta que nos hagan, mejor hacer las cosas bien y aprender a programar de forma prolija, ordenada, comentando el código y documentándolo todo a nuestro paso.

Suficiente por ahora. Espero reencontrarlos próximamente en una tercera entrega. Gracias a Underc0de por el espacio. Happy Coding para todos!

CrazyKade

crazykade@mail.ru