

Practica 2

1. Qué líneas del siguiente código provocan conversiones *boxing* y *unboxing*.

```
char c1 = 'A';
string st1 = "A";
object o1 = c1;
object o2 = st1;
char c2 = (char)o1;
string st2 = (string)o2;
```

object o1 = c1; // Boxing: el valor de c1 (char) se almacena en un objeto (object)

char c2 = (char)o1; // Unboxing: el valor de o1 (object) se convierte a char

string st2 = (string)o2; // Unboxing: el valor de o2 (object) se convierte a string

Explicación: El boxing se produce cuando se convierte un valor de un tipo de valor (como char) en un objeto (como object), y el unboxing se produce cuando se convierte un objeto en un tipo de valor. En el código dado, el primer boxing se produce en la línea donde se asigna la variable c1 a la variable o1, ya que el valor de c1 (que es de tipo char) se almacena en un objeto (de tipo object).

Por otro lado, los dos unboxing se producen en las líneas donde se convierten las variables o1 y o2 (que son objetos) en los tipos char y string, respectivamente. Como los valores originales de estas variables son de tipos diferentes a los que se convierten, es necesario realizar el unboxing para obtener el valor original.

2. Sea el siguiente código:

```
object o1 = "A";
object o2 = o1;
o2 = "Z";
Console.WriteLine(o1 + " " + o2);
```

El tipo **object** es un tipo referencia, por lo tanto luego de la sentencia **o2 = o1** ambas variables están apuntando a la misma dirección. ¿Cómo explica entonces que el resultado en la consola no sea “**Z Z**”?

Aunque ambos objetos o1 y o2 apuntan inicialmente al mismo objeto en memoria que contiene el valor "A", en la línea o2 = "Z", se crea un nuevo objeto en memoria que contiene el valor "Z", y se actualiza la variable o2 para que apunte a este nuevo objeto.

Es importante destacar que los objetos de tipo string son inmutables en C#. Esto significa que, una vez creado un objeto de este tipo, su valor no puede ser modificado. En lugar de eso, cada vez que se realiza una operación que modifica el valor del objeto, se crea un nuevo objeto en memoria.

Por lo tanto, en la línea o2 = "Z", se crea un nuevo objeto string con el valor "Z" y se actualiza la variable o2 para que apunte a este nuevo objeto. La variable o1 sigue apuntando al objeto original que contiene el valor "A". Por lo tanto, al imprimir o1 + " " + o2, se muestra "A Z" en la consola.

En resumen, aunque ambas variables inicialmente apuntan al mismo objeto en memoria, al crear un nuevo objeto y actualizar la variable o2 para que apunte a este nuevo objeto, se separan las referencias a los dos objetos y se obtiene el resultado "A Z" al imprimir ambas variables.

3. Analizar la siguiente porción de código para calcular la sumatoria de 1 a 10. ¿Cuál es el error? ¿Qué hace realmente?

```
int sum = 0;
int i = 1;
while (i <= 10);
{
    sum += i++;
}
```

El error en este código es la presencia del punto y coma (;) después de la condición del ciclo while, lo que hace que el cuerpo del ciclo ({sum += i++;}) sea ignorado y que el ciclo se ejecute sin hacer nada. Por lo tanto, la variable sum no se actualiza y al final del ciclo, su valor sigue siendo cero.

Para corregir el código, se debe quitar el punto y coma después de la condición del ciclo while:

```
int sum = 0;
int i = 1;
while (i <= 10)
{
    sum += i++;
}
```

En este código corregido, el ciclo while se ejecutará mientras la variable i sea menor o igual a 10. Dentro del cuerpo del ciclo, se agrega el valor de i a la variable sum, y luego se incrementa i en 1 utilizando el operador de postincremento ++. Esto hace que la variable i se incremente en 1 después de agregar su valor a sum.

Al final del ciclo, la variable sum contendrá la suma de los números enteros del 1 al 10, que es 55.

4. ¿Cuál es la salida por consola si no se pasan argumentos por la línea de comandos?

```
Console.WriteLine(args == null);
Console.WriteLine(args.Length);
```

False
0

La primera línea imprime False porque la variable args es un arreglo de cadenas (string[]) que siempre se inicializa con una referencia a un arreglo vacío (no es null). Por lo tanto, la expresión args == null se evalúa a False.

La segunda línea imprime 0 porque la propiedad Length de un arreglo vacío es siempre cero. Como no se han pasado argumentos por la línea de comandos, el arreglo args está vacío y su propiedad Length es cero.

5. ¿Qué hace la instrucción

```
int[]? vector = new int[0];
```

¿asigna a la variable vector el valor **null**?

No, la instrucción int[]? vector = new int[0] no asigna un valor nulo (null) a la variable vector. La sintaxis int[]? indica que vector es una referencia a un arreglo de enteros que puede ser nula (nullable). Sin embargo, en este

caso, la instrucción crea un nuevo arreglo de enteros con cero elementos (`new int[0]`) y lo asigna a la variable vector.

En resumen, la variable vector no es nula, sino que hace referencia a un arreglo de enteros vacío.

6. Determinar qué hace el siguiente programa y explicar qué sucede si no se pasan parámetros cuando se invoca desde la línea de comandos.

```
Console.WriteLine("¡Hola {0}!", args[0]);
```

Utiliza la clase Console para imprimir en la consola un saludo personalizado. La instrucción `Console.WriteLine("¡Hola {0}!", args[0])`; utiliza una cadena de formato para imprimir el valor del primer argumento pasado por la línea de comandos (`args[0]`) en el lugar de `{0}`.

Si se invoca el programa sin pasar ningún argumento por la línea de comandos, se producirá una excepción de tipo `IndexOutOfRangeException`, ya que la variable args será un arreglo vacío y no habrá ningún elemento en la posición `args[0]` que se pueda utilizar como argumento para la cadena de formato. Esto causará que el programa falle y se detenga abruptamente.

Para evitar este error, es posible agregar una verificación para asegurarse de que haya al menos un argumento en el arreglo args antes de intentar utilizarlo.

7. Analizar el siguiente código. ¿Qué líneas producen error de compilación y por qué?

```
char c;
char? c2;
string? st;
c = "";
c = '';
c = null;
c2 = null;
c2 = (65 as char?);
st = "";
st = '';
st = null;
st = (char)65;
st = (string)65;
st = 47.89.ToString();
```

La línea `c = "";` produce un error de compilación porque la cadena vacía ("") no se puede asignar a una variable de tipo char, ya que un char es un solo carácter y una cadena es una colección de caracteres.

La línea `c = ";` produce un error de compilación porque un carácter debe estar entre comillas simples ('), pero en este caso, no se ha especificado ningún carácter.

La línea `c = null;` produce un error de compilación porque un valor null no se puede asignar a un tipo char no nullable.

La línea `c2 = null;` es válida, ya que `char?` es un tipo nullable y puede tener un valor null.

La línea `c2 = (65 as char?);` es válida, ya que 65 se puede asignar a un tipo `char?` utilizando la sintaxis `as`.

La línea `st = "";` es válida, ya que una cadena vacía ("") se puede asignar a una variable de tipo `string`?

La línea `st = ";` produce un error de compilación porque un carácter debe estar entre comillas simples ('), pero en este caso, no se ha especificado ningún carácter.

La línea `st = null;` es válida, ya que `string?` es un tipo nullable y puede tener un valor `null`.

La línea `st = (char)65;` es válida, pero asigna un carácter ('A') a una variable de tipo `string?`, lo cual no es lo que probablemente se espera.

La línea `st = (string)65;` produce un error de compilación porque no se puede convertir un valor de tipo `char` a `string` mediante una conversión explícita.

La línea `st = 47.89.ToString();` es válida, ya que `ToString()` es un método que convierte un número decimal en una cadena de caracteres. Sin embargo, se debe tener en cuenta que esto no es lo que probablemente se espera en este caso, ya que la variable `st` es de tipo `string`?

8. Escribir un programa que reciba una lista de nombres como parámetro e imprima por consola un saludo personalizado para cada uno de ellos.
 - a) utilizando la sentencia **for**
 - b) utilizando la sentencia **foreach**

```
using System;
```

```
class Program
{
    static void Main(string[] args)
    {
        string[] nombres = args;

        for (int i = 0; i < nombres.Length; i++)
        {
            Console.WriteLine("¡Hola " + nombres[i] + "! ¿Cómo estás?");
        }
    }
}
//-----
class Program
{
    static void Main(string[] args)
    {
        string[] nombres = args;

        foreach (string nombre in nombres)
        {
            Console.WriteLine("¡Hola " + nombre + "! ¿Cómo estás?");
        }
    }
}
```

9. Investigar acerca de la clase **StringBuilder** del espacio de nombre **System.Text** ¿En qué circunstancias es preferible utilizar **StringBuilder** en lugar de utilizar **string**? Implementar un caso de ejemplo en el que el rendimiento sea claramente superior utilizando **StringBuilder** en lugar de **string** y otro en el que no.

La clase **StringBuilder** en C# es una clase que pertenece al espacio de nombres **System.Text** y que se utiliza para construir y manipular cadenas de caracteres de manera eficiente. A diferencia de la clase **string**, que es inmutable y no se puede modificar después de su creación, la clase **StringBuilder** permite agregar, eliminar, insertar o reemplazar caracteres en una cadena existente sin tener que crear una nueva cadena en cada operación.

En general, se prefiere utilizar **StringBuilder** en lugar de **string** en las siguientes circunstancias:

Cuando se necesitan realizar múltiples operaciones de concatenación de cadenas, ya que en cada concatenación se crea una nueva cadena, lo que puede ser ineficiente en términos de rendimiento y consumo de memoria. En estos casos, es mejor utilizar **StringBuilder**, que permite realizar todas las operaciones en la misma instancia sin crear una nueva cadena en cada operación.

Cuando se necesitan manipular cadenas de caracteres de manera dinámica, es decir, agregar, eliminar o insertar caracteres en cualquier posición de la cadena. En estos casos, **StringBuilder** es más eficiente que crear una nueva cadena en cada operación, ya que se realiza en la misma instancia.

Cuando se trabaja con grandes cantidades de datos, ya que la creación de múltiples cadenas puede agotar la memoria disponible en el sistema, mientras que **StringBuilder** permite manipular la misma cadena sin crear copias.

A continuación, se presentan dos casos de ejemplo para ilustrar en qué circunstancias el rendimiento de **StringBuilder** es claramente superior al de **string** y en cuál no lo es.

Caso de ejemplo 1: Rendimiento superior de **StringBuilder**

Supongamos que se tiene que generar una cadena de caracteres que contenga 10000 números enteros separados por comas. En este caso, utilizar la clase **StringBuilder** sería más eficiente que utilizar la concatenación de cadenas con el operador **+**, ya que en cada concatenación se crearía una nueva cadena y se asignaría a una variable, lo que podría agotar la memoria disponible en el sistema. A continuación, se presenta un ejemplo de código que utiliza la clase **StringBuilder** para generar la cadena requerida:

```
StringBuilder sb = new StringBuilder();
for (int i = 0; i < 10000; i++)
{
    sb.Append(i.ToString());
    if (i < 9999)
    {
        sb.Append(",");
    }
}
string result = sb.ToString();
```

En este ejemplo, se crea una instancia de **StringBuilder** y se utiliza el método **Append** para agregar cada número entero a la cadena. También se agrega una coma después de cada número, excepto el último. Finalmente, se convierte la cadena **StringBuilder** en una cadena de caracteres utilizando el método **ToString**.

10. Investigar sobre el tipo **DateTime y usarlo para medir el tiempo de ejecución de los algoritmos implementados en el ejercicio anterior.**

El tipo DateTime en C# es un tipo de datos que representa una fecha y hora. Se encuentra en el espacio de nombres System y es una estructura que contiene información sobre el año, mes, día, hora, minuto, segundo y milisegundo.

La clase DateTime tiene muchos métodos útiles para manipular y comparar fechas y horas. Algunos de los métodos más comunes incluyen:

Now: devuelve la fecha y hora actual.

Parse: convierte una cadena en un objeto DateTime.

ToString: convierte un objeto DateTime en una cadena en un formato específico.

Add: agrega una cantidad específica de tiempo (días, horas, minutos, etc.) a un objeto DateTime.

Subtract: resta una cantidad específica de tiempo de un objeto DateTime.

DateTime también tiene propiedades como Year, Month, Day, Hour, Minute, Second y Millisecond para acceder a las diferentes partes de una fecha y hora.

Una de las formas en que DateTime puede ser útil en la programación es para medir el tiempo de ejecución de los algoritmos. Para hacer esto, puede crear un objeto DateTime antes y después de ejecutar el algoritmo y luego restar el segundo objeto DateTime del primero para obtener la cantidad de tiempo que tardó el algoritmo en ejecutarse.

A continuación, se muestra un ejemplo de cómo se puede medir el tiempo de ejecución de un algoritmo utilizando DateTime:

```
using System;

class Program
{
    static void Main(string[] args)
    {
        // Crear objeto DateTime antes de ejecutar el algoritmo
        DateTime startTime = DateTime.Now;

        // Ejecutar el algoritmo aquí
        for (int i = 0; i < 1000000; i++)
        {
            // Algoritmo que se quiere medir
            int j = i * i;
        }

        // Crear objeto DateTime después de ejecutar el algoritmo
        DateTime endTime = DateTime.Now;

        // Restar los objetos DateTime para obtener la duración del algoritmo
        TimeSpan duration = endTime - startTime;

        // Imprimir la duración del algoritmo
        Console.WriteLine("Duración del algoritmo: " + duration.TotalMilliseconds + " milisegundos");
    }
}
```

En este ejemplo, se utiliza DateTime.Now para crear un objeto DateTime antes y después de ejecutar un bucle que representa el algoritmo que se quiere medir. Luego, se resta el segundo objeto DateTime del primero para

obtener la duración del algoritmo en forma de un objeto TimeSpan. Finalmente, se imprime la duración del algoritmo en milisegundos.

11. ¿Para qué sirve el método **Split** de la clase **string**? Usarlo para escribir en la consola todas las palabras (una por línea) de una frase ingresada por consola por el usuario.

El método Split en C# es un método de la clase string que divide una cadena en subcadenas basadas en un carácter delimitador y devuelve un arreglo de estas subcadenas. El carácter delimitador se especifica como un argumento del método Split.

12. Comprobar el funcionamiento del siguiente programa y dibujar el estado de la pila y la memoria *heap* cuando la ejecución alcanza los puntos indicados (comentarios en el código)

```
using System.Text;

object[] v = new object[10];
v[0] = new StringBuilder("Net");
for (int i = 1; i < 10; i++)
{
    v[i] = v[i - 1];
}
(v[5] as StringBuilder).Insert(0, "Framework .");
foreach (StringBuilder s in v)
    Console.WriteLine(s);

//dibujar el estado de la pila y la mem. heap
//en este punto de la ejecución

v[5] = new StringBuilder("CSharp");
foreach (StringBuilder s in v)
    Console.WriteLine(s);

//dibujar el estado de la pila y la mem. heap
//en este punto de la ejecución
```

Antes de explicar el estado de la pila y la memoria heap en los puntos indicados en el código, es importante entender qué está haciendo el programa.

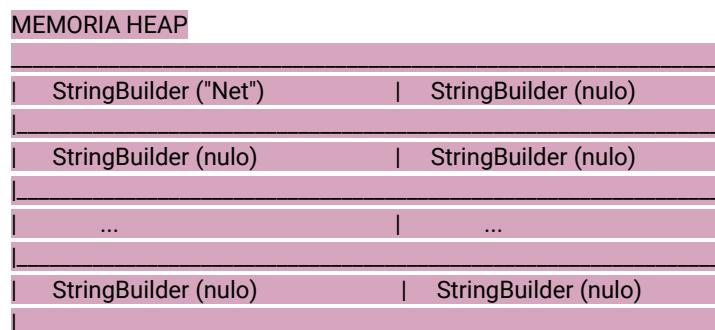
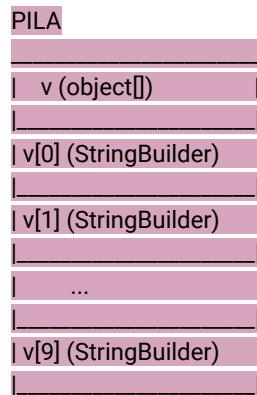
El programa crea un arreglo de objetos llamado v con una longitud de 10. En la primera posición del arreglo (v[0]), se crea un objeto de la clase StringBuilder con el valor "Net". Luego, en un bucle for, los elementos del arreglo desde la posición v[1] hasta v[9] se inicializan con el valor del elemento anterior (v[i] = v[i - 1]).

Después de esto, se llama al método Insert en la posición v[5] del arreglo ((v[5] as StringBuilder).Insert(0, "Framework .")) para insertar "Framework ." al comienzo del StringBuilder almacenado en esa posición.

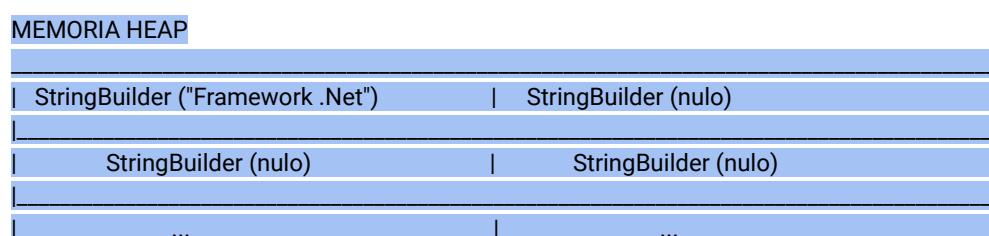
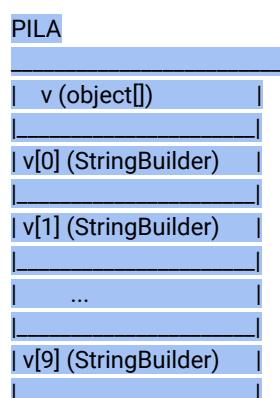
Finalmente, se reemplaza el valor en la posición v[5] con un nuevo objeto de la clase StringBuilder con el valor "CSharp", y se imprimen todos los objetos StringBuilder almacenados en el arreglo v.

Estado de la pila y la memoria heap:

Punto de ejecución antes del **método Insert**:



Punto de ejecución después del **método Insert**:



StringBuilder (nulo)	StringBuilder (nulo)

Punto de ejecución después de la **asignación de v[5]**:

PILA

v (object[])
v[0] (StringBuilder)
v[1] (StringBuilder)
...
v[9] (StringBuilder)

MEMORIA HEAP

StringBuilder ("Framework .Net")	StringBuilder ("CSharp")
StringBuilder (nulo)	StringBuilder (nulo)
...	...

13. Definir el tipo de datos enumerativo llamado **Meses** y utilizarlo para:

- a) Imprimir en la consola el nombre de cada uno de los meses en orden inverso (diciembre, noviembre, octubre ..., enero)
- c) Solicitar al usuario que ingrese un texto y responder si el texto tipeado corresponde al nombre de un mes

Nota: en todos los casos utilizar un **for** iterando sobre una variable de tipo **Meses**

```
using System;
```

```
enum Meses
{
    Enero,
    Febrero,
    Marzo,
    Abril,
    Mayo,
    Junio,
    Julio,
    Agosto,
    Septiembre,
    Octubre,
    Noviembre,
    Diciembre
}
```

```
class Program
{
```

```

static void Main(string[] args)
{
    for (int i = (int)Meses.Diciembre; i >= (int)Meses.Enero; i--)
    {
        Console.WriteLine(Enum.GetName(typeof(Meses), i));
    }

    Console.Write("Ingrese un texto: ");
    string texto = Console.ReadLine();

    bool esMes = Enum.TryParse(texto, out Meses mes);

    if (esMes)
    {
        Console.WriteLine("El texto ingresado corresponde al mes " + mes.ToString());
    }
    else
    {
        Console.WriteLine("El texto ingresado no corresponde a ningún mes");
    }
}
}

```

14. Implementar un programa que muestre todos los números primos entre 1 y un número natural dado (pasado al programa como argumento por la línea de comandos). Definir el método **bool EsPrimo(int n)** que devuelve **true** sólo si *n* es primo. Esta función debe comprobar si *n* es divisible por algún número entero entre 2 y la raíz cuadrada de *n*. (Nota: **Math.Sqrt(d)** devuelve la raíz cuadrada de *d*)

```

using System;

class Program
{
    static void Main(string[] args)
    {
        if (args.Length != 1)
        {
            Console.WriteLine("Debe pasar un argumento que sea un número natural.");
            return;
        }

        if (!int.TryParse(args[0], out int n) || n <= 0)
        {
            Console.WriteLine("El argumento debe ser un número natural.");
            return;
        }

        Console.WriteLine($"Números primos entre 1 y {n}:");
        for (int i = 2; i <= n; i++)
        {
            if (EsPrimo(i))
            {
                Console.WriteLine(i);
            }
        }
    }
}

```

```

static bool EsPrimo(int n)
{
    if (n < 2)
    {
        return false;
    }

    int tope = (int)Math.Sqrt(n);
    for (int i = 2; i <= tope; i++)
    {
        if (n % i == 0)
        {
            return false;
        }
    }

    return true;
}

```

Primero, se verifica que se haya pasado un argumento por línea de comandos y que este sea un número natural. Luego, se recorre un ciclo for desde 2 hasta n, y para cada número i se llama a la función EsPrimo para determinar si es primo o no. Si lo es, se imprime en la consola.

La función EsPrimo es bastante sencilla: si n es menor que 2, devuelve false ya que 1 no es considerado primo. Luego, se calcula el tope de un rango para la comprobación, que es la raíz cuadrada de n. A partir de ahí, se recorre un ciclo for desde 2 hasta tope, y se comprueba si n es divisible por i. Si lo es, se devuelve false. Si se llega al final del ciclo sin encontrar un divisor, n es primo y se devuelve true.

15. Escribir una función (método **int Fib(int n)**) que calcule el término *n* de la serie de Fibonacci.

Fib(n) = 1, si n <= 2
Fib(n) = Fib(n-1) + Fib(n-2), si n > 2

```

using System;

class Program
{
    static void Main(string[] args)
    {
        int n = 10;
        int fib = Fib(n);
        Console.WriteLine($"El término {n} de la serie de Fibonacci es {fib}.");
    }

    static int Fib(int n)
    {
        if (n <= 0)
        {
            return 0;
        }

        if (n == 1)
        {

```

```

        return 1;
    }

    int a = 0;
    int b = 1;
    for (int i = 2; i <= n; i++)
    {
        int c = a + b;
        a = b;
        b = c;
    }

    return b;
}
}

```

El programa utiliza la función Fib para calcular el término n de la serie de Fibonacci, y lo imprime en la consola. La función Fib es implementada mediante un ciclo for, en el que se calcula sucesivamente los términos de la serie. Para ello, se usa una variable a para almacenar el término anterior, y una variable b para almacenar el término actual. En cada iteración del ciclo, se calcula el siguiente término c como la suma de a y b , y se actualizan a y b para la siguiente iteración. Al final del ciclo, se devuelve b , que es el término n de la serie de Fibonacci.

16. Escribir una función (método **int Fac(int n)**) que calcule el factorial de un número n pasado al programa como parámetro por la línea de comando

- a) Definiendo una función no recursiva
- b) Definiendo una función recursiva
- c) idem a b) pero con *expression-bodied methods* (**Tip:** utilizar el operador condicional ternario)

17. Ídem. al ejercicio 16.a) y 16.b) pero devolviendo el resultado en un parámetro de salida

void Fac(int n, out int f)

18. Codificar el método **Swap** que recibe 2 parámetros enteros e intercambia sus valores. El cambio debe apreciarse en el método invocador.

```

static void Main(string[] args)
{
    int a = 5;
    int b = 10;

    Console.WriteLine($"Antes del intercambio: a = {a}, b = {b}");
    Swap(ref a, ref b);
    Console.WriteLine($"Después del intercambio: a = {a}, b = {b}");
}

static void Swap(ref int x, ref int y)
{
    int temp = x;
    x = y;
    y = temp;
}

```

En este ejemplo, el método Swap recibe dos parámetros ref de tipo int que representan los valores que se van a intercambiar. Dentro del método, se utiliza una variable temporal temp para almacenar el valor de x, luego x toma el valor de y, y finalmente y toma el valor de temp. De esta manera, los valores de x y y se intercambian.

En el método Main, se crea dos variables a y b y se les asigna valores iniciales. Se imprime el valor de estas variables antes del intercambio, se llama al método Swap pasando a y b por referencia y luego se imprime el valor de las variables después del intercambio. Como se puede ver, el método Swap efectivamente intercambia los valores de a y b.

19. Codificar el método **Imprimir** para que el siguiente código produzca la salida por consola que se observa. Considerar que el usuario del método **Imprimir** podría querer más adelante imprimir otros datos, posiblemente de otros tipos pasando una cantidad distinta de parámetros cada vez que invoque el método.

Tip: usar **params**

```
Imprimir(1, "casa", 'A', 3.4, DayOfWeek.Saturday);
Imprimir(1, 2, "tres");
Imprimir();
Imprimir("-----");
```

