

Seminario de lenguajes .NET

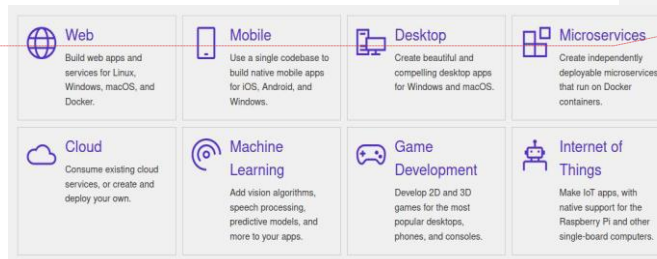
- ✓ Generalidades de .NET
 - Implementaciones de .NET
 - Historia de .NET
 - Common language runtime (CLR)
 - Microsoft intermediate language (MSIL)
 - Base classes library (BCL)
- ✓ Introducción a C#
 - Directiva using
 - Tipos integrados y operadores
 - Operadores aritméticos
 - Operadores relacionales
 - Operadores lógicos
 - Operadores de asignación
 - Estructuras de control
 - Interrupción de bucles
- ✓ Sistema de tipos
 - Common Type System (CTS)
 - Tipos de valor
 - Tipos de referencia
 - Conversión de tipos
 - Nullable value types
 - Sistema unificado de tipos
 - Boxing y unboxing
 - Arreglos
 - Clase string
 - String builder
 - Tipos enumerativos
 - Arreglo de 2 dimensiones
 - Palabra clave var
 - Tipos anónimos
- ✓ Métodos
 - Parámetros
 - Expression bodied methods
- ✓ Strings de formato compuesto y strings interpolados
 - Cadenas interpoladas
- ✓ Colecciones
 - ArrayList
 - Stack
 - Queue
 - HashTable
- ✓ Manejo de excepciones
 - Excepciones comunes
 - Propagación de excepciones
 - Lanzar una excepción
- ✓ Programación orientada a objetos
 - Miembros de una clase
 - Métodos de instancia
 - Operador null-coalescing
 - Miembros estáticos
 - Campos constantes (const)

- [Campo readonly](#)
- [Encapsulamiento](#)
- [Propiedades](#)
- [Descriptores de acceso con cuerpos de expresión](#)
- [Propiedades vs campos públicos](#)
- [Propiedades implementadas automáticamente](#)
- [Encadenamiento de métodos](#)
- [Herencia](#)
- [Invalidación de métodos](#)
- [Acceso a miembros de la clase base](#)
- [Constructores](#)
- [Modificadores de acceso](#)
- [Descriptores de acceso con distintos niveles de accesibilidad](#)
- [Invalidación de propiedades](#)
- [Clases abstractas](#)
- [Finalizadores \(destructores\)](#)
- [Polimorfismo](#)
- [Interfaz polimórfica](#)
- [Principio open/close](#)
- [Diseño ineficiente que no hace uso de polimorfismo](#)
- ✓ [Indizadores](#)
 - [Indizadores de objeto](#)
- ✓ [Método ToString\(\)](#)
- ✓ [Excepciones definidas por el usuario](#)
- ✓ [Herencia de campos estáticos](#)
- ✓ [Herencia de métodos estáticos](#)
- ✓ [Extensión de métodos](#)
- ✓ [Interfaces](#)
 - [Interfaces de la plataforma que se usan para la comparación](#)
 - [Interfaces para enumerar : "System.Collections.IEnumerable" y "System.Collections.IEnumerator"](#)
 - [Iteradores](#)
- ✓ [File system- Espacio de nombres System.IO](#)
 - [Archivos de texto](#)
 - [Interface IDisposable](#)
- ✓ [Delegados](#)
 - [Asignación de delegados](#)
 - [Pasando métodos como parámetros](#)
 - [Métodos anónimos](#)
- ✓ [Eventos](#)
 - [Eventos- Convenciones](#)
 - [Tipo Event Handler](#)
 - [Observaciones](#)
 - [Event](#)
- ✓ [Operador condicional NULL](#)

GENERALIDADES DE .NET

Es una plataforma de desarrollo gratuita, de código abierto y de uso general.

Soporta varios lenguajes de programación. Microsoft desarrolla activamente C#, Visual Basic y F#, pero también existen otros.



Comentado [SA1]: código abierto es el software cuyo código fuente y otros derechos que normalmente son exclusivos para quienes poseen los derechos de autor, son publicados bajo una licencia de código abierto o forman parte del dominio público

IMPLEMENTACIONES DE .NET

- ✓ .NET framework: implementación original, optimizado para aplicaciones de escritorio de Windows
- ✓ Mono: para entornos de ejecución pequeños como Android, macOS, iOS, tvOS, watchOS
- ✓ Plataforma Universal de Windows (UWP): para aplicaciones Windows e internet de las cosas
- ✓ .NET CORE: implementación multiplataforma de .NET

HISTORIA DE .NET

La versión 1.0 de .NET Framework fue lanzada oficialmente por Microsoft para su sistema operativo Windows en enero del 2002. Junto con .NET Framework, Microsoft liberó la primera versión del lenguaje C# y una nueva versión de Visual Basic. F# es un lenguaje funcional más reciente, fue desarrollado por Microsoft en 2005. .NET Framework es gratuito pero no es *open source*. La última versión es la 4.8 (última actualización abril de 2019)

La versión multiplataforma (Windows, macOS y Linux) y open-source de .NET se inició con el nombre de .NET Core (v 1.0 liberado en junio 2016). La última versión con el nombre .NET Core es la 3.1 (liberado en diciembre 2019). A partir de noviembre de 2020 cambió su nombre a .NET 5.0. La única rama que seguirá evolucionando en el futuro es la que se inicia con .NET 5.0

COMMON LANGUAGE RUNTIME (CLR)

El CLR es el motor de ejecución (runtime) de .NET. Es un entorno virtual independiente de la arquitectura de hardware en el que se ejecutan las aplicaciones escritas con cualquier lenguaje .NET

MICROSOFT INTERMEDIATE LANGUAGE (MSIL)

Cuando se compila una aplicación escrita en un lenguaje de .NET (VB, C# u otro de los soportados), el compilador genera código MSIL (también conocido como código CIL). MSIL es un lenguaje similar a un código ensamblador pero de más alto nivel, creado para un hipotético procesador virtual que no está atado a una arquitectura determinada.

De manera transparente el código MSIL se traduce al lenguaje nativo del procesador físico en tiempo de ejecución, por intermediación de un compilador bajo demanda "Just In Time" (JIT)

BASE CLASSES LIBRARY (BCL)

La BCL es la biblioteca de clases base de .NET que da soporte a infinidad de funcionalidades a través de las clases y otros tipos definidos en ella. Las clases en la BCL se organizan en espacios de nombres (namespaces) que agrupan a clases y a otros namespaces. La BCL incluye los tipos definidos en los espacios System.* y Microsoft.*

Por ejemplo, los tipos integrados están en el espacio de nombres System. En System.Collections (namespace Collections dentro del namespace System) encontramos clases que representan colecciones. Las clases relacionadas con el manejo de estructuras de datos XML se encuentra en el espacio de nombres System.Xml (namespace Xml dentro del namespace System). Las clases para manejar la comunicación entre servidor web y cliente están en el espacio de nombres System.Web

INTRODUCCIÓN A C#

DIRECTIVA USING

El nombre extenso de una clase lleva como prefijo el espacio de nombres en la que se ha definido. El nombre extenso de la clase Console definida en el espacio de nombres System es System.Console. La directiva using X al comienzo del archivo fuente permite omitir el prefijo X. para todos los miembros del espacio de nombres X. El alcance de la directiva using es el archivo fuente donde se especifica

TIPOS INTEGRADOS Y OPERADORES

Los tipos integrados son estructuras, salvo string y object que son clases. Como tipos de la plataforma, están definidos en el espacio de nombres System. Por lo tanto, si se utiliza la directiva using System es posible referirse al tipo System.Boolean simplemente como boolean. Sin embargo, para el caso de los tipos integrados, lo usual es utilizar el alias definido para el lenguaje correspondiente

TIPOS ENTEROS INTEGRADOS EN C#

Alias de C#	Intervalo	Tipo de .NET
Sbyte	-128 a 127	System.SByte
Byte	0 a 255	System.Byte
Short	-32 768 a 32767	System.Int16
Ushort	0 a 65 535	System.UInt16
Int	-2.147.483.648 a 2.147.483.647	System.Int32
UInt	De 0 a 4.294.967.295	System.UInt32
Long	-9.223.372.036.854.775.808 a 9.223.372.036.854.755.807	System.Int64
Ulong	0 a 18.446.744.073.709.551.615	System.UInt64

TIPOS DE PUNTO FLOTANTES INTEGRADOS EN C#

Alias en C#	Intervalos	Tipo de .NET	Precisión
Float	De $\pm 1.5 \times 10^{-45}$ a $\pm 3.4 \times 10^{38}$	System.Single	6 a 9 dígitos aprox
Double	De $\pm 5,0 \times 10^{-324}$ a $\pm 1,7 \times 10^{308}$	System.double	15 a 17 dígitos
Decimal	De $\pm 1,0 \times 10^{-28}$ a $\pm 7,9228 \times 10^{28}$	System.decimal	6 a 9 dígitos aprox.

OTROS TIPOS INTEGRADOS

Alias C#	Tipo de .NET
Char	System.char
Bool	System.boolean
String	System.string
Object	System.object
Dynamic	System.Object

C# es **case sensitive**

OPERADORES ARITMÉTICOS

Operador	Operación
+	Suma
-	Resta
*	Multiplicación
/	División
%	Residuo/ resto

OPERADORES RELACIONALES

Operador	Operación
<	Menor que
>	Mayor que
<=	Menor o igual que
>=	Mayor igual que
!=	Diferente de
==	Igual a

OPERADORES LÓGICOS

Operador	Operación
&	AND
	OR
!	NOT
^	XOR
&&	AND en cortocircuito
	OR en cortocircuito

OPERADORES DE ASIGNACIÓN

Operador	Operación
++	Incremento
--	Decremento
=	Asignación simple
*=	Multiplicación más asignación
/=	División más asignación
%=	Residuo más asignación
+=	Suma más asignación
-=	Resta más asignación

ESTRUCTURAS DE CONTROL

IF

CONDICIONAL TERNARIO

```

if (condición)
{
    //bloque que se ejecuta
    //si condición es verdadera
}
else
{
    //bloque que se ejecuta
    //si condición es verdadera
}

```

SWITCH

```

int n = 10;
string st = (n < 0) ? "negativo"

switch (expresion)
{
    case valor1:
        // lista de instrucciones para
        // cuando expresion == valor1
        break;
    case valor2:
        // lista de instrucciones para
        // cuando expresion == valor2
        break;
    default:

```

INTERRUPCIÓN DE BUCLES

- ✓ Break: el bucle termina inmediatamente
- ✓ Continue: termina el ciclo corriente inmediatamente
- ✓ Goto: permite saltar fuera del bucle (no es recomendable)
- ✓ Return: salta fuera del bucle y del método que lo contiene

SISTEMA DE TIPOS

COMMON TYPE SYSTEM

Define un conjunto de tipos orientado a objetos. Todo lenguaje de programación de .NET debe implementar los tipos definidos por el CTS. Los tipos de .NET pueden ser **tipos de valor** o **tipos de referencia**.

TIPOS DE VALOR

El espacio reservado para la variable en la pila de ejecución guarda directamente el valor asignado. Admite **ESTRUCTURAS** (char, bool, DateTime, tipos números, tipos punto flotante), **ENUMERACIONES**

Las variables de tipos de valor siempre contienen un valor válido para su tipo y por lo tanto no admiten el valor null (a excepción de los "nullable value types")

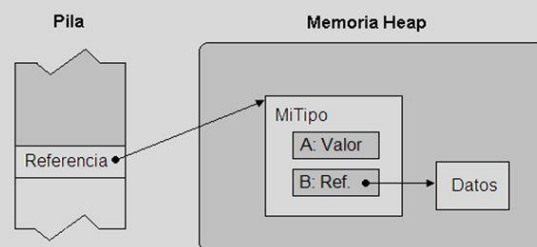
TIPOS DE REFERENCIA

El espacio reservado para la variable en la pila de ejecución guarda la dirección en la memoria heap donde está el valor asignado

Admite **CLASES**, **DELEGADOS** E **INTERFACES**

En particular **object** es un tipo de referencia y constituye la raíz de la jerarquía de tipos.

EL VALOR NULL



Las variables de tipos de referencia o bien contienen la referencia al objeto en la memoria heap, o bien poseen un valor especial nulo (palabra clave null)

CONVERSIÓN DE TIPOS

Hay distintos tipos de conversiones

- ✓ **Implícitas:** se realiza cuando la operación es segura, en otro caso se requiere del consentimiento del programador quien debe hacerse responsable de la seguridad de la operación
- ✓ **Explícitas:** requieren de un operador de conversión
- ✓ **Con tipos auxiliares:** para realizar conversiones entre tipos no compatibles (System.convert, Parse, toString)
- ✓ **Definidas por el usuario**

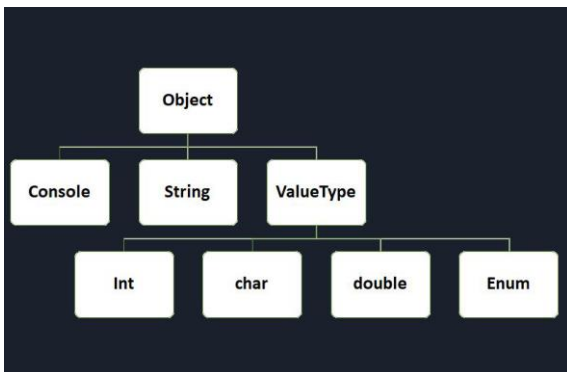
Además del operador de cast (...), para algunos casos también se puede usar el operador **as**. Cuando una conversión de tipo no puede llevarse a cabo el operador (...) provoca una excepción (error en tiempo de ejecución). Cuando una conversión de tipo no puede llevarse a cabo el operador as devuelve el valor null (no provoca una excepción). Por lo tanto as se utiliza sólo para tipos de referencia o tipos nullables

NULLABLES VALUE TYPES

Los tipos de valor que admiten valores nulos son útiles en algunos escenarios, por ejemplo un tipo numérico en una base de datos puede ser nulo

SISTEMA UNIFICADO DE TIPOS

Todos los tipos de datos derivan directa o indirectamente de un tipo base común: la clase System.Object. A diferencia de Java, en C# esto también es aplicable a los tipos de valor (conversiones **boxing y unboxing**)



Aunque C# es un lenguaje fuertemente tipado, debido a la jerarquía de tipos y a la relación "es un", las variables de tipo object admiten valores de cualquier tipo.

BOXING Y UNBOXING

Las conversiones boxing y unboxing permiten asignar variables de tipo de valor a variables de tipo de referencia y viceversa

- ✓ Cuando una variable de algún tipo de valor se asigna a una de tipo de referencia, se dice que se le ha aplicado la *conversión boxing*
- ✓ Cuando una variable de algún tipo de referencia se asigna a una de tipo de valor, se dice que se le ha aplicado la conversión unboxing.

ARREGLOS

Los arreglos son de tipo de referencia. Los arreglos pueden tener varias dimensiones. El número de dimensiones se denomina **rank**. El número total de elementos de un arreglo se llama longitud del arreglo **length**

CLASE STRING

Es un tipo de referencia, por lo tanto acepta el valor null. Sin embargo la comparación no es por dirección de memoria. Se ha redefinido el operador `==` para realizar una comparación lexicográfica. Tiene en cuenta mayúsculas y minúsculas. Los string son de **solo lectura**, se accede a los elementos [], el prime elemento tiene el índice 0

STRING BUILDER

El string es de lectura/escritura. Definida en el espacio de nombre System.text. Tiene métodos adicionales como *append, insert, remove, replace, etc.*

TIPOS ENUMERATIVOS

```
enum Tamaño
{
    chico, mediano, grande
}

Uso de enumeraciones
Tamaño t;
t = Tamaño.grande;
t = (Tamaño)0; //chico + vale
```

ARREGLO DE 2 DIMENSIONES

Matriz: `int[,] matriz = new int[,] { {1,2,3,4}, {5,6,7,8}, {9,10,11,12} };`

Arreglo de arreglos: `int[][] tabla = new int[2][];`

`tabla[0] = new int[2]; tabla[1] = new int[3];`

PALABRA CLAVE VAR

La palabra clave var en la declaración de una variable local con inicialización (`≠ null`) indica que el tipo de la misma es inferido por el compilador en función de la inicialización. Una vez inferido el tipo de una variable por el compilador queda fijo e inmutable (no es un tipo dinámico)

TIPOS ANÓNIMOS

La inferencia de tipos permite instanciar objetos de tipos anónimos. Una forma conveniente de encapsular un conjunto de propiedades de solo lectura en un solo objeto sin tener que definir explícitamente un tipo primero.

```
var x = new { Nombre = "Juan", Edad = 28 };
var y = new { Alto = 12.4, Ancho = 11, Largo = 20 };
Console.WriteLine("Nombre de x: " + x.Nombre);
Console.WriteLine("Ancho de y: " + y.Ancho);
```

TIPO DYNAMIC

Una variable declarada de tipo dynamic admite la asignación de elementos de distintos tipos durante la ejecución. El tipo dynamic funciona como si fuese el tipo object pero el compilador omite la verificación de tipos, simplemente supone que la operación es válida. Esto no nos previene de errores en tiempo de ejecución. Debido a la falta de verificación de tipos en las expresiones donde hay elementos de tipo Dynamic involucrados, tampoco son necesarias las conversiones explícitas.

MÉTODOS

Bloque con nombre de código ejecutable que puede invocarse desde diferentes partes del programa, e incluso desde otros programas

Si el método no devuelve ningún valor, se especifica **void** como tipo de retorno. En este caso return es opcional

PARÁMETROS

- ✓ De entrada/ por valor: recibe una copia del valor pasado como parámetro. Desde un método estático como es el caso de main, puede invocarse de forma directa a otro método estático de la misma clase
- ✓ De salida (out): se deben asignar dentro del cuerpo del método invocado antes de cualquier lectura. Es posible pasar parámetros de salida que sean variables no inicializadas
- ✓ De referencia: similar a los parámetros de salida, pero no es posible indicar el método pasando una variable no inicializada. El método invocado puede leer el valor del parámetro *ref* en cualquier momento pues la inicialización está garantizada por el invocador
- ✓ De entrada (in): el parámetro se pasa por referencia pero no puede modificarse dentro del método invocado

Params: permite que un método tome un número variable de argumentos. El tipo declarado del parámetro params debe ser un arreglo unidimensional

EXPRESSION BODIED METHODS

Para los casos en que el cuerpo de un método pueda escribirse como una sola expresión, es posible utilizar una sintaxis simplificada.

Ejemplo: `void Imprimir(string st) => Console.WriteLine(st);`

Esta sintaxis no está limitada a métodos que devuelven void, se puede utilizar con cualquier tipo de retorno.

STRINGS DE FORMATO COMPUESTO Y STRINGS INTERPOLADOS

`st = string.Format("Es un {0} año {1}", marca, modelo);` → esta es una cadena de formato compuesto. Es un string con marcadores de posición indizados. A los marcadores dentro del string de formato compuesto se los llama **elementos de formato**

CADENAS INTERPOLADAS

Las cadenas interpoladas también utilizan elementos de formato. Son más legible y cómodas de usar que los strings de formato compuesto. Las cadenas interpoladas llevan antepuesto el **símbolo \$**

Los elementos de formato en las cadenas interpoladas también admiten expresiones

`Console.WriteLine($"ancho = {ancho} y alto = {20}");`

`Console.WriteLine($"Es un {marca,7} año {modelo}");` → Alineación derecha completa con blancos de izquierda

`Console.WriteLine($"Es un {marca,-7} año {modelo}");` → Alineación izquierda

`Console.WriteLine($"Valor = { r:0.0}");` → máscaras de formato

El método `ToString()` definido en los tipos numéricos también acepta un parámetro que es una máscara de formato

COLECCIONES

Algunas funcionalidades no pueden resolverse con arreglos de manera conveniente. Como:

- ✓ Incrementar la longitud del arreglo
- ✓ Reducir la longitud del arreglo

- ✓ Insertar un elemento en cualquier posición
- ✓ Borrar un elemento reduciendo la longitud del arreglo
- ✓ Acceder a los elementos por medio de un índice no entero

Por esto, surgen las colecciones, que gestionan un conjunto de elementos pero lo hacen de una manera especial. Se pueden considerar arreglos especializados, hay varios tipos según la tarea. Los más importantes en System.collection encontramos:

ARRAY LIST

Similar a un vector de object pero que puedo redimensionar dinámicamente

- ✓ Add: (valor): añade el valor representado por valor
- ✓ AddRange: añade un conjunto de valores
- ✓ Insert (Posicion, valor): inserta el valor en una posición determinada desplazando el resto de valores
- ✓ InsertRange(Posicion, colección): inserta un conjunto de valores a partir de una posición determinada
- ✓ SerRange(Posicion, Colección): sobrescribe elementos en un array con los valores de la colección, comenzando en la posición Posicion.

La capacidad es el espacio reservado para alojar a los elementos del ArrayList. Count es el número total de elementos que efectivamente contiene el ArrayList. Si se agregan elementos al ArrayList continuamente, la estrategia de aumentar la capacidad de este solo lo necesario es muy mala porque el rendimiento cae abruptamente. La capacidad se duplica cada vez que se necesita reservar más espacio en el ArrayList

STACK

Las pilas están implementadas como objetos. Cuentan con métodos especializados

- ✓ Puch(object o): coloca el objeto indicado en la cima de la pila
- ✓ Object Pop(): devuelve el elemento de la cima de la pila y lo saca
- ✓ Object Peek(): devuelve el elemento de la cima de la pila pero sin sacarlo de ella

QUEUE

Las colas están implementadas como objetos. Permiten:

- ✓ Enqueue(object o): coloca el objeto indicado al final de la cola
- ✓ Object Dequeue(): devuelve el primer objeto de la cola y lo saca de ella
- ✓ Object peek(): devuelve el primer objeto de la cola pero no lo saca

HASHTABLE

Colección de elementos, acceso a los valores por la clave. Representa una colección de pares clave/valor que se organizan en función del código hash de la clave. El acceso a los elementos de la colección se realiza a través de la clave que puede ser de cualquier tipo

```
Hashtable ht = new Hashtable();
```

```
//Agregando elementos
```

```
ht["un número" ] = 7;
```

```
ht[2.23] = "dos con 23" ;
```

```
ht[DayOfWeek.Friday] = 'F';
```

```
//Accediendo a los elementos
```

```
Console.WriteLine(ht["un número" ]); 7
```

```
Console.WriteLine(ht[2.23]); dos con 23
```

```
Console.WriteLine(ht[DayOfWeek.Friday]); F
```

```
if (ht[3] == null)
```

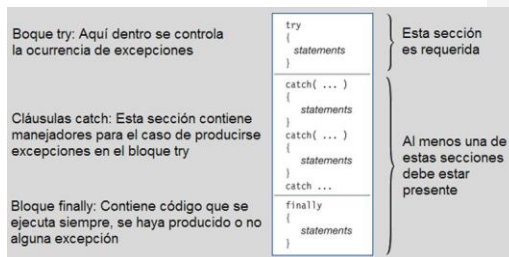
```
{
Console.WriteLine("La clave 3 no existe" );      La clave 3 no existe
}
```

MANEJO DE EXCEPCIONES

Las excepciones son errores en tiempo de ejecución. Ejemplos de excepciones: Intentar dividir por cero, escribir un archivo de sólo lectura, referencias a null, acceder a un arreglo con un índice fuera del rango válido, etc.

EXCEPCIONES COMÚNES

- ✓ DivideByZeroException
- ✓ OverflowException
- ✓ NullReferenceException
- ✓ IndexOutOfRangeException
- ✓ IOException
- ✓ InvalidCastException



El bloque finally se ejecuta SIEMPRE antes de finalizar el try/catch independientemente de la ejecución o no de alguna cláusula catch. El bloque finally se ejecuta aún si se alcanza una sentencia return en el bloque try o alguno del bloque catch

PROPAGACIÓN DE EXCEPCIONES

Si Metodo1 invoca a Metodo2 y dentro de este último se produce una excepción que no es manejada, ésta se propaga a Metodo1. Desde la perspectiva de Metodo1, la invocación a Metodo2 es la instrucción que genera la excepción

LANZAR UNA EXCEPCIÓN

En ocasiones vamos a querer que nuestro código lance excepciones. Para ello se utiliza el operador Throw. Uso: `throw e` (Lanza la excepción e, siendo e un objeto de una clase derivada de System.Exception). `throw` (Dentro de un bloque catch, relanza la excepción corriente).

Ejemplo: Se requiere codificar el método imprimir que reciba un string como parámetro. Se lanzará la excepción ArgumentNullException en caso que el argumento sea null

```
static void Imprimir(string st)
{
if (st == null) // Si st es null se crea un objeto ArgumentNullException y se lanza esta excepción
{
throw new ArgumentNullException("st");
}
Console.WriteLine(st);
}
```

Es posible crear nuestros propios tipos de excepciones, pero también podemos lanzar una excepción genérica con un mensaje personalizado de la siguiente manera:

```
throw new Exception("msg personalizado");
```

PROGRAMACIÓN ORIENTADA A OBJETOS

Es una manera de construir Software. Es un paradigma de programación. Propone resolver problemas de la realidad a través de identificar objetos y relaciones de colaboración entre ellos. El objeto y el mensaje son sus elementos fundamentales. La POO en .Net está basada en las clases. Una clase describe el comportamiento (métodos) y los atributos (campos) de los objetos que serán instanciados a partir de ella.

Todos los métodos que definimos dentro de una clase son miembros de esa clase

MIEMBROS DE UNA CLASE

Los miembros de una clase pueden ser:

- ✓ **De instancia:** pertenecen al objeto. Pueden ser campos, métodos, constructores, constantes (se definen como miembros de instancia pero se usan como miembros estáticos), propiedades, indizadores, finalizadores, eventos, operadores, tipos anidados)
- ✓ **Pertenecen a la clase**

Los miembros de una clase son privados por defecto.

USO DE THIS

Dentro de un constructor o método de instancia, la palabra clave this hace referencia a la instancia (el propio objeto) que está ejecutando ese código. Puede ser útil para diferenciar el nombre de un campo de una variable local o parámetro con el mismo nombre

CAMPO DE INSTANCIA

Un campo o variable de instancia es un miembro de datos de una clase. Cada objeto instanciado de esa clase tendrá su propio campo de instancia con un propio valor (posiblemente distinto al valor que tengan en dicho campo otros objetos de la misma clase)

MÉTODOS DE INSTANCIA

Los métodos de instancia permiten manipular los datos almacenados en los objetos. Los métodos de instancia implementan el comportamiento de los objetos. Dentro de los métodos de instancia se pueden acceder a todos los campos del objeto, incluidos los privados

CONSTRUCTORES DE INSTANCIA

Un constructor de instancia es un método especial que contiene código que se ejecuta en el momento de la instanciación de un objeto. Habitualmente se utilizan para establecer el estado del nuevo objeto por medio del pasaje de argumentos

Si no se define un constructor explícitamente, el compilador agrega uno sin parámetros y con cuerpo vacío. Si se define un constructor explícitamente, el compilador ya no incluye el constructor por defecto

SOBRECARGA

Es posible tener más de un constructor en cada clase (sobrecarga de constructores) siempre que difieran en alguno de los siguientes puntos: la cantidad de parámetros, el tipo y el orden de los parámetros, los modificadores de los parámetros

En el encabezado de un constructor se puede invocar a otro constructor de la misma clase empleando la sintaxis :this. Este constructor invocado se ejecuta antes que las instrucciones del cuerpo del constructor invocador.

`Public Auto(String marca): this(marca, DateTime.Now.Year){}`. El cuerpo está vacío, todo se resuelve en la invocación al otro constructor

Los métodos también pueden ser sobrecargados. Para sobrecargar los métodos valen las mismas consideraciones que en el caso de los constructores. El valor de retorno NO puede utilizarse como única diferencia para permitir una sobrecarga

INVOCACIÓN A MÉTODOS Y CONSTRUCTORES

Los métodos (aunque devuelvan valores) y los constructores de objetos (expresiones con operador new) pueden usarse como una instrucción, es decir, no se requiere asignar el valor devuelto, en todo caso, dicho valor se pierde.

¿? OPERADOR NULL-COALESCE (OPERADOR DE FUSIÓN NULA)

`A = b ?? C; → a = (b != null) ? b : c;`

MIEMBROS ESTÁTICOS

Los miembros estáticos son miembros que pertenecen a la propia clase (o tipo) en lugar de a un objeto (instancia) específico de la misma. Para declararlos se utiliza el modificador static

CAMPOS ESTÁTICOS

Un campo estático de una clase es una variable accesible a través de la clase en la que fue definido (NO a través de las instancias). Es una única variable compartida por todas las instancias de esa clase, incluso por objetos de otras clases en caso de no ser privada.

Los miembros estáticos pueden accederse desde la propia clase sin anteponer el nombre de la misma

CONSTRUCTORES ESTÁTICOS

Un constructor estático se declara como uno de instancia pero con el modificador static. No se puede invocar explícitamente. Es invocado por el runtime de .Net una única vez cuando se carga la clase, por lo tanto: no pueden tener parámetros ni modificadores de acceso, no pueden sobrecargarse (sólo puede definirse un constructor estático por clase)

El runtime de .NET no garantiza cuándo se ejecutará un constructor estático, ni en qué orden se ejecutarán los constructores estáticos de diferentes clases. Sin embargo, lo que está garantizado es que el constructor estático se ejecutará antes de que nuestro código haga referencia a la clase

En C #, el constructor estático generalmente se ejecuta inmediatamente antes de la primera llamada a cualquier miembro de la clase. Es posible tener un constructor estático y un constructor de instancia ambos sin parámetros definidos en la misma clase. Si a los campos estáticos se les han dado valores predeterminados, estos se asignan antes de que se llame al constructor estático

CLASES ESTÁTICAS

Las clases estáticas llevan el modificador static en su declaración. Sólo pueden poseer miembros estáticos. No es posible instanciar objetos de una clase estática. A menudo agrupan un conjunto de utilidades y datos relacionados (utility class). Ejemplos de clases estáticas: Console, File, Directory, Math, etc.

UTILITY CLASSES

Las clases estáticas por lo general constituyen "clases de utilidad", agrupando cierta funcionalidad que se expone completamente como miembros de nivel de clases, como math

CAMPOS CONSTANTES (CONST)

Son valores inmutables que no cambian durante la vida del programa. Se declaran con el modificador const. Deben inicializarse cuando se declaran. Son siempre implícitamente estáticas. Sin embargo no se usa el modificador static.

La expresión que se asigna a una constante es computada por el compilador, por lo tanto: es una expresión simple, no puede referir a ninguna variable (todas las variables, aún las estáticas se inicializan en tiempo de ejecución), no puede implicar la ejecución de código del programa, solo los tipos integrados de C# (excluido object) pueden declararse como const.

Los tipos integrados en C# son: bool, byte, sbyte, char, decimal, double, float, int, uint, long, ulong, short, ushort, object y string. Por lo tanto pueden ser constante todos los tipos numéricos, char, bool y string

CAMPOS READONLY

Con los campos readonly se obtiene un efecto similar al que tienen los campos constantes pero sin sus restricciones. Se identifican con el modificador readonly, sólo pueden asignarse en su declaración o dentro de un constructor (los estáticos sólo en el constructor estático). Se asignan en tiempo de ejecución, como cualquier variable, por lo tanto no están restringidos a un conjunto de tipos u operaciones simples como en el caso de las constantes

ENCAPSULAMIENTO

El encapsulamiento es uno de los pilares de la programación orientada a objetos. Es la capacidad del lenguaje para ocultar detalles de implementación hacia fuera del objeto, en estrecha relación con la noción de encapsulamiento está la idea de la protección de datos. Idealmente, el estado de los objetos debería especificarse usando campos privados.

Para acceder a los campos privados usamos *getters* y *setters* pero no es lo indicado, se deben usar **propiedades**

PROPIEDADES

Una propiedad integra los conceptos de campo y método al mismo tiempo. Externamente se asigna y lee como si fuese un campo. Internamente se codifican dos bloques de código:

- ✓ bloque get: se ejecuta cuando se lee la propiedad
- ✓ bloque set: se ejecuta cuando se escribe la propiedad.

A los bloques get y set se los llama descriptores de acceso (accessors en inglés)

```
public double Lado
{
    get
    {
        <código para leer el
        valor de la propiedad>
    }
    set
```

Una propiedad que implementa sólo el bloque get es una propiedad de sólo lectura. Una propiedad que implementa sólo el bloque set es una propiedad de sólo escritura. No es aconsejable el uso de propiedades de sólo escritura. Si la intención es desencadenar algún efecto secundario cuando se asigna el valor, es preferible usar un método en lugar de una propiedad.

DESCRIPTORES DE ACCESO CON CUERPOS DE EXPRESIÓN

Si el cuerpo de un descriptor de acceso consta de una sola expresión puede utilizarse la sintaxis alternativa (expression bodied member)

PROPIEDADES VS CAMPOS PUBLICOS

Las propiedades publicas siempre son preferibles a los campos públicos porque, al ser miembros de funciones, pueden procesar la entrada y la salida lo que permite establecer controles sobre los valores de la propiedad.

Si la clase que modificamos es utilizada por otro programa, al cambiar un campo por una propiedad debemos volver a compilar también el otro programa. Esto no sucede si desde el principio se codifica una propiedad. Al cambiar su implementación no es necesario recompilar otros programas que acceden a ella

PROPIEDADES IMPLEMENTADAS AUTOMÁTICAMENTE

Para facilitar la tarea del programador C# 3.0 introdujo las propiedades auto-implementadas. Con ellas es posible declarar la propiedad sin declarar el campo asociado, el compilador crea un campo oculto, no está accesible para el programador, que asocia a la propiedad auto-implementada.

No hay excusa para seguir utilizando campos públicos. Solo agregando (get;set;) los convertimos en propiedades auto implementadas

ENCADENAMIENTO DE MÉTODOS

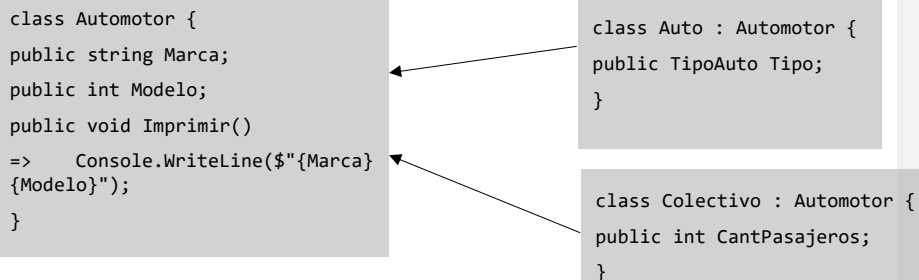
El encadenamiento de métodos es una técnica muy útil que permite invocar múltiples métodos en una sentencia.

HERENCIA

La herencia permite crear clases que reutilizan, extienden y modifican el comportamiento definido en otras clases. La clase cuyos miembros se heredan se denomina **clase base** y las clases que heredan esos miembros se denominan **clases derivadas**. Una clase derivada sólo puede tener una clase base directa, pero la herencia es transitiva.

DERIVACIÓN DE CLASES

Claramente las clases Auto y Colectivo comparten tanto atributos como comportamiento. Es posible por lo tanto generalizar el diseño colocando las características comunes en una superclase que llamaremos Automotor



```
class Automotor {
    public string Marca;
    public int Modelo;
    public void Imprimir()
    => Console.WriteLine($"{Marca} {Modelo}");
}

class Auto : Automotor {
    public TipoAuto Tipo;
}

class Colectivo : Automotor {
    public int CantPasajeros;
}
```

Una clase derivada obtiene implícitamente todos los miembros de la clase base, salvo sus constructores y sus finalizadores. Las clases Auto y Colectivo del ejemplo derivan de la clase Automotor, por lo tanto un Auto es un Automotor y un Colectivo también es un Automotor.

Las clases Auto y Colectivo también heredan el método Imprimir() definido en Automotor. Sin embargo, el método Imprimir() resulta poco útil al no poder acceder a las variables específicas de Auto y Colectivo (Tipo y CantPasajeros respectivamente). Solución: *Sobrescribir* (invalidar) el método Imprimir() en cada una de las subclases para que tanto autos como colectivos se impriman de forma más adecuada.

INVALIDACIÓN DE MÉTODOS

Las clases derivadas pueden **invalidar** los métodos heredados para proporcionar una implementación alternativa. Para poder invalidar un método, el método de la clase base debe marcarse con la palabra **virtual**

```
class Automotor {
    public string Marca;
    public int Modelo;
    public virtual void Imprimir()
    => Console.WriteLine($"{Marca} {Modelo}");
}
```

El modificador **override** indica que se está invalidando el método de la clase base. Un método override, al igual que virtual, también puede ser invalidado en una clase derivada

virtual is used to modify a method, property, indexer, or event declared in the base class and allow it to be overridden in the derived class.

override is used to extend or modify a virtual/abstract method, property, indexer, or event of base class into derived class.

ACCESO A MIEMBROS DE LA CLASE BASE

La palabra clave base se utiliza para obtener acceso a los miembros de la clase base (la superclase) desde una clase derivada. Se utiliza en dos situaciones:

Para Invocar a un método (o propiedad) de la clase base

Para indicar a qué constructor de la clase base se debe llamar al crear instancias de la clase derivada.

Sería como el *.super* en Java

CONSTRUCTORES

Los constructores **no se heredan**, sin embargo, debemos estar atentos cuando los definimos en una jerarquía de clases

```
Public auto():base(){ }
```

En lugar de agregar el constructor sin argumentos en la clase Automotor se pueden definir constructores adecuados en las clases Auto y Colectivo que invoquen al constructor de dos argumentos de la clase automotor de la siguiente manera:

```
Public Auto(string marca, int modelo, TipoAuto tipo): base(marca,modelo){
    This.Tipo=tipo
}
```

MODIFICADORES DE ACCESO

Si bien las clases derivadas heredan todos los miembros de una clase base (a excepción de los constructores y finalizadores), que dichos miembros estén o no visibles depende de su **accesibilidad**

```
public override void Imprimir()
=> Console.WriteLine($"{Marca} {Modelo} ({CantPasajeros} pasajeros)");
```

Surgirá un error de compilación. Los miembros privados solo son visibles en la clase donde se define y aunque se hereda, no puede accederse desde el código de las clases derivadas. Sin embargo, si se necesita extender el acceso de un miembro a las clases derivadas, se debe marcar como **protected**

Los miembros de las clases pueden declararse como públicos, internos, protegidos o privados.

- ✓ **Públicos:** precedidos por el modificador de acceso public. Pueden accederse desde cualquier clase de cualquier ensamblado (compilado ejecutable .exe o una biblioteca de clases DLL) que compone una aplicación.
- ✓ **Internos:** precedido por el modificador de acceso *internal*. Pueden accederse solo desde las clases en el mismo ensamblado
- ✓ **Protegidos:** precedidos por el modificador de acceso *protected*. Sólo pueden accederse desde la propia clase o desde sus clases derivadas (estén o no en el mismo ensamblado)
- ✓ **Privados:** precedidos por el modificador de acceso *private* (por defecto para los miembros de una clase). Sólo pueden accederse desde la propia clase

```
class Automotor
{
    protected string Marca;
    protected int Modelo;
    public Automotor(string marca, int modelo)
    {
        Marca = marca;
        Modelo = modelo;
    }
}
```


Además existen las siguientes combinaciones de modificadores de acceso:

- ✓ **Protected terminal:** pueden accederse desde cualquier código de ensamblado, o desde una clase derivada de cualquier ensamblado
- ✓ **Private protected:** solo pueden accederse desde la propia clase o desde sus clases derivadas en el mismo ensamblado

CLASES

- ✓ **Públicas:** precedidas por el modificador *public*
- ✓ **Internas:** precedidas por el modificador de acceso *terminal* o no contienen el modificador

No se pueden definir clases con el modificador *private* o *protected* a menos que sea una clase anidada dentro de otra

PROPIEDADES

```
class Automotor
{
public string Marca { get; }
private int _modelo;
public int Modelo
{
get => _modelo;
set => _modelo = (value < 2005) ? 2005 : value;
}
```

Se desea controlar que el modelo sea mayor o igual a 2005



RESTRICCIONES

Las clases no pueden ser más accesibles que su **clase base**. Las constantes, campos, propiedades e índices no pueden ser más accesibles que sus respectivos tipos. Los métodos, constructores e índices no pueden ser más accesibles que los tipos de sus parámetros, índices o valor de retorno

Si defino una clase como pública, sus propiedades deben serlo también porque si no existe una contradicción

DESCRIPTORES DE ACCESO CON DISTINTOS NIVELES DE ACCESIBILIDAD

RESTRINGIR LA ESCRITURA DE LA PROPIEDAD MODELO PARA TODAS LAS CLASES DISTINTAS DE AUTOMOTOR Y DERIVADAS

```
class Automotor
{
    public string Marca { get; }
    private int _modelo;
    public int Modelo
    {
        get => _modelo;
```

El modificador de acceso solo afecta al descriptor *set*, debe ser más restrictivo que el modificador de la propiedad

Solo puede ser escrita en Automotor o sus jerarquías derivadas

INVALIDACIÓN DE PROPIEDADES

Las propiedades, al igual que los métodos, pueden ser redefinidas en las clases derivadas. Para ello, al igual que los métodos, la propiedad debe ser marcada como una propiedad virtual mediante la palabra clave *virtual*

Esto permite que las clases derivadas invaliden el comportamiento de la propiedad mediante la palabra clave ***override***

```
class Colectivo : Automotor
{
    public int CantPasajeros;
    public override int Modelo
    {
        protected set =>
        base.Modelo = (value < 2015) ? 2015 : value;
    }
}
```

```
Console.WriteLine(a.Marca + " " +
a.Modelo);
Console.WriteLine(c.Marca + " " +
```

Imprimirá "Ford 2005 // Mercedes 2015"

Cualquier intento de asignación de *a.Marca*, *a.Modelo* provocará error de compilación

CLASES ABSTRACTAS

El propósito de una clase abstracta es proporcionar una definición común de una clase base que hereden múltiples clases derivadas. No se pueden crear instancias de una clase abstracta. Las clases abstractas se señalan con el modificador *abstract*.

Una clase abstracta puede tener métodos, propiedades, indicadores y eventos abstractos. Una clase abstracta también pueden tener miembros no abstractos. Los miembros abstractos no tienen implementación, se escriben sin el cuerpo, y llevan el modificador *abstract*

El compilador dará error cuando:

- ✓ Se declare un miembro abstracto en una clase no abstracta
- ✓ Al intentar instanciar un objeto de una clase abstracta
- ✓ Si no se implementan los métodos y las propiedades abstractos de la clase base en una clase derivada no abstracta

Ejemplo:

```
abstract class Automotor
{
public abstract void HacerMantenimiento();
public abstract DateTime FechaService { get; set; }
public abstract double PrecioDeVenta { get; }
public string Marca { get; }
private int _modelo;
```

Auto y Colectivo
deben implementar
los métodos
abstractos

FINALIZADORES (DESTRUCTORES)

Los finalizadores (también conocidos como destructores) se usan para "hacer limpieza" cuando el recolector de elementos no utilizados (garbage collector) libera memoria. El garbage collector comprueba periódicamente si hay objetos no referenciados por ninguna aplicación. En tal caso llama al finalizador (si existe) y libera la memoria que ocupaba el objeto. Se puede forzar la recolección con `GC.Collect()` pero en general debe evitarse por razones de rendimiento.

En las aplicaciones de .NET Framework (pero no en las de .NET Core), cuando se cierra el programa también se llama a los finalizadores. No se puede llamar a los finalizadores, se invocan automáticamente. Puede haber un solo finalizador por clase, no puede tener ni parámetros ni modificadores. Los finalizadores no se heredan.

```
class MiClase
{
~MiClase() // Finalizador o destructor
{
// código de limpieza...
}
```

Luego de ejecutarse el código de un finalizador se invoca implícitamente al finalizador de su clase base. Es decir que se ejecutan en cadena todos los finalizadores de manera recursiva desde la clase más a la menos derivada.

Los finalizadores pueden ser útiles para liberar objetos que incluyen recursos no administrados, el resto los gestiona el garbage collector. Los tipos más comunes de recurso no administrado son objetos que contienen recursos del sistema operativo, como archivos, ventanas, conexiones de red o conexiones de base de datos

POLIMORFISMO

El polimorfismo suele considerarse el tercer pilar de la programación orientada a objetos, después del encapsulamiento y la herencia. Un objeto polimórfico en tiempo de ejecución puede adoptar la forma de un tipo no idéntico a su tipo declarado. La relación "es un" asociada a la herencia permite tener objetos polimórficos.

```
object o;
o = 1;
o = "ABC";
Automotor a;
a = new Auto("Ford", 2000, TipoAuto Deportivo);
```

O y **a** son objetos polimórficos. En tiempo de ejecución va a ocurrir polimorfismo (van a adoptar distintas formas de tipos)

MÉTODOS VIRTUALES Y ENLACE DINÁMICO

Los miembros virtuales, la invalidación y el enlace dinámico permiten aprovechar el polimorfismo

```
a.Imprimir();
```

Imprimirá "depende de quién sea a". El método que se invocará en la expresión "a.Imprimir();" NO se enlaza estáticamente en tiempo de compilación. El CLR busca en tiempo de ejecución, el tipo específico del objeto a, e invoca la invalidación correcta de Imprimir() (enlace dinámico)

El polimorfismo permite que un mismo mensaje enviado a un objeto provoque comportamientos distintos dependiendo de quién sea el objeto. Si a es Auto, se imprimirá como Auto, si es un colectivo, se imprime como colectivo.

INTERFAZ POLIMÓRFICA

La interfaz polimórfica de una clase base es el conjunto de sus métodos virtuales y eventualmente abstractos. Esto es más interesante de lo que parece a primera vista, ya que permite crear aplicaciones de software fácilmente extensibles y flexibles. Un diseño adecuado de la jerarquía de clases permitirá tomar ventaja del polimorfismo.

Ejemplo: Supongamos que tenemos la clase estática (utility class) ImprimidorAutomotores dedicada a la impresión de automotores, que usamos de la siguiente manera:

Gracias al polimorfismo, el código de la utility class es muy sencillo. El programador no necesita tener en cuenta la manera en que se imprime cada subtipo de automotor

```
static void Main(string[] args)
{
    Automotor[] vector = new Automotor[] {
        new Auto("Ford", 2015, TipoAuto.Deportivo),
        new Colectivo("Mercedes", 2019, 20),
        new Colectivo("Mercedes", 2018, 30),
        new Auto("Nissan", 2020, TipoAuto.Familiar)
    }
```

```
static class ImprimidorAutomotores
{
    public static void Imprimir(Automotor[] vector)
    {
        foreach (Automotor a in vector)
        {
            a.Imprimir();
        }
    }
}
```

Ejemplo: Supongamos que el sistema está en producción y algún tiempo después se solicita una modificación importante. El sistema ahora debe también trabajar con un nuevo tipo de automotor: las motos. La tarea entonces consiste en agregar la clase Moto derivada de Automotor y sobrescribir (invalidar) convenientemente los miembros necesarios de su interfaz polimórfica

```
class Moto : Automotor
{
    public Moto(string marca, int modelo)
    {
        base(marca, modelo) { }
    }
    public override void Imprimir()
    {
        Console.WriteLine("Moto ");
        base.Imprimir();
    }
}
```

```
static void Main(string[] args)
{
    Automotor[] vector = new Automotor[] {
        new Auto("Ford", 2015, TipoAuto.Deportivo),
        new Colectivo("Mercedes", 2019, 20),
        new Colectivo("Mercedes", 2018, 30),
        new Moto("Gilera", 2017)
    };
}
```

Gracias al polimorfismo no tuvimos que modificar la clase ImprimidorAutomotores

PRINCIPIO OPEN/CLOSE

Establece que una entidad de software debe ser abierta para su extensión, pero cerrada para su modificación. El polimorfismo contribuye a que podamos cumplir con este principio.

Si nuestro código requiere consultar por el tipo de un objeto, puede ser una señal de un diseño ineficiente. Ejemplo: si `a` es una instancia de `Auto`, retornará `true`

DISEÑO INEFICIENTE QUE NO HACE USO DE POLIMORFISMO

Se puede considerar ineficiente si

- ✓ No definimos la clase `Automotor` con su interfaz polimórfica
- ✓ `Auto` y `Colectivo` derivan directamente de `object`
- ✓ Los métodos `imprimir()` de `Auto` y `Colectivo` no están relacionados entre sí, incluso sus nombres podrían no coincidir
- ✓ `Imprimir()` de la clase `ImprimidorAutomotores` no puede tomar ventaja del polimorfismo

INDIZADORES

Ahora se desea acceder a los miembros de una familia a través de un índice (como si se tratase de una colección).

Un indizador es una definición de cómo aplicar el operador (`[]`) a los objetos de una clase. A diferencia de los arreglos, los índices que se les pase entre corchetes no están limitados a los enteros, pudiéndose definir varios indizadores en una misma clase siempre y cuando cada uno tome un número o tipo de índices diferente (sobrecarga). Los indizadores son sólo de instancia, no pueden definirse indizadores estáticos

```
public Persona this[<lista de índices>]
{
    get
    {
        código que retorna el elemento
        según la <lista de índices>
    }
    set
    {
        código que establece el elemento
        según la <lista de índices>
    }
}
```

INDIZADORES DE OBJETO

Los inicializadores de objeto permiten asignar valores a cualquier campo o propiedad accesible de un objeto en el momento de su creación. Un inicializador consiste en una lista de elementos separada por comas encerradas entre llaves `{ }` que sigue a la invocación del constructor. Cada miembro de la lista mapea con un campo o propiedad pública del objeto al que le asigna un valor.

MÉTODO TOSTRING()

Uno de los métodos de la interfaz polimórfica de `object` es: `public virtual string ToString();`

La implementación genérica en `object` simplemente devuelve el nombre del tipo del objeto que recibe el mensaje, es lo que muestra el método `WriteLine()` de la clase `Console`.

Por ejemplo si `a` es una instancia de `Auto`, la instrucción `Console.WriteLine(a)` imprime en la consola `Teoria6.Auto`. Gracias al polimorfismo, no es necesario modificar el método `WriteLine()` de la clase `Console` para cambiar este comportamiento. Podemos hacerlo invalidando el método `ToString()` de la clase `Auto` (principio Open/Close).

EXCEPCIONES DEFINIDAS POR EL USUARIO

.NET proporciona una jerarquía de clases de excepciones derivadas en última instancia de la clase base `Exception`. También podemos crear nuestras propias excepciones derivando clases de `Exception`. La convención es terminar el nombre de clase con la palabra "Exception" e implementar los tres constructores comunes

```
public class TanqueCapacidadExcedidaException : Exception {
    public TanqueCapacidadExcedidaException() { }
    public TanqueCapacidadExcedidaException(string message)
        : base(message) { }
    public TanqueCapacidadExcedidaException(string message,
```

Se usa cuando en un manejador de excepción (bloque catch) se lanza otra excepción pero en inner se mantiene la referencia a la original

HERENCIA DE CAMPOS ESTÁTICOS

Los campos estáticos también se heredan. Sin embargo, a diferencia de los campos de instancia, la clase derivada no obtiene su propio campo sino que accede al mismo campo que su clase base

Cuando heredo campos, no obtengo dos variables, si no que tengo dos accesos a la misma variable

Es posible volver a definir un campo estático en una clase derivada. De esta forma se dice que el nuevo campo oculta al campo de la clase base. Se debe utilizar el modificador *new* para evitar un Warning del compilador y ahora tengo 2 variables

HERENCIA DE MÉTODOS ESTÁTICOS

Los métodos estáticos también se heredan pero no pueden invalidarse (no puedo declarar como virtual o override un método estático pero sí pueden ser ocultados). Marcar un método estático como virtual, override o abstract (ver más adelante) provoca error de compilación. Los métodos estáticos pueden ser ocultados en las clases derivadas.

Utilizo un new

EXTENSIÓN DE MÉTODOS

Implemento métodos en una clase aunque no estén definidas en la clase, por lo tanto conviene agruparlas en una clase estática

La herencia permite extender el funcionamiento de clases existentes, sin embargo no siempre está disponible. No pueden derivarse clases selladas (marcadas con el modificador *sealed*) ni tampoco estructuras (por ejemplo los tipos numéricos). Los métodos de extensión permiten agregar funcionalidad a los tipos incluso en los casos en que la herencia no está permitida.

Los métodos de extensión son métodos con los que se extiende a un tipo pero se definen realmente como miembros en otro tipo (una clase estática). Supongamos que queremos contar con un conjunto extra de funciones para trabajar con tipos *int*. Una buena idea es definir una clase estática que agrupe las nuevas funciones.

Para llamar a estos métodos, debo poner *this* y el tipo al principio de los parámetros de los métodos extendidos, brinda azúcar sintáctica

Si la clase estática con los métodos de extensión se encuentra definida en otro espacio de nombres, estos métodos estarán disponibles únicamente cuando el espacio de nombres se importe explícitamente con la directiva *using*

INTERFACES

La interfaz polimórfica establecida por una clase base solo es aprovechada por los tipos derivados. Sin embargo, en sistemas de software más grandes, es común desarrollar múltiples jerarquías de clases que no tienen un padre común más allá de `System.Object`. Para esto, implementamos las interfaces

Las interfaces son un tipo referencia que especifica un conjunto de funciones sin implementarlas

Pueden especificar métodos, propiedades, indizadores y eventos de instancia. En lugar del código que los implementa, llevan un `;` al final. Por convención comienzan con la *I* (i latina mayúscula)

Una interface no puede contener campos de instancia, constructores ni finalizadores

DECLARACIÓN DE INTERFACE

```
public interface IMiInterface
{
    public void UnMetodo();
}
```

Las clases derivan de otras clases y opcionalmente implementan una o más interfaces. Si una clase implementa una interface debe implementar todos los miembros de la interface que no tienen implementación determinada. Si una clase deriva de otra clase y además implementa algunas interfaces, la clase debe ser la primera en la lista después de los dos puntos. Entonces al declarar una clase se hace de la siguiente manera:

```
class Rombo : Figura, IImprimible, IAgrandable
{
    . . . Donde Figura es la clase base y IImprimible y IAgrandable son interfaces
}
```

UTILIZACIÓN DE INTERFACES

Es posible definir y usar variables de tipo interface. Entonces:

```
Rombo r1 = new Rombo();
Figura r2 = new Rombo();
IAgrandable r3 = new Rombo();
IImprimible r4 = new Rombo();
```

Las interfaces son tipos de referencias, por lo tanto es posible usar el operador `as` y `puedo`

```
IImprimible[] vector = new IImprimible[10];
```

Acá no instanciamos ningún objeto `IImprimible`, los elementos que agreguemos al vector tendrán que implementar la interface `IImprimible`.

HERENCIA EN INTERFACES

Las interfaces pueden heredar de múltiples interfaces

```
interface IInterface1 {  
    void Metodo1();  
}  
  
interface IInterface2 {  
    void Metodo2();  
}  
  
interface IInterface3: IInterface1, IInterface2 {  
    void Metodo3();  
}  
  
class A : IInterface3 {  
    . . .  
}
```


La clase A deberá implementar metodo1(), metodo2() y metodo3().

IMPLEMENTACIÓN EXPLÍCITA DE INTERFACES

Es posible que los métodos de igual nombre difieran semánticamente por lo tanto usamos la implementación explícita. Donde:

```
class A : IInterface1, IInterface2
{
    void IInterface1.Metodo() =>
    Console.WriteLine("método de Interface1");
    void IInterface2.Metodo() =>
    Console.WriteLine("método de Interface2");
    public void Metodo() =>
    Console.WriteLine("método a nivel de la clase");
```

IMPLEMENTACIÓN EXPLÍCITA DE MIEMBROS DE INTERFACES

Cuando hay implementaciones explícitas de miembros de interface, la implementación a nivel clase está permitida pero no es requerida. Por lo tanto se tienen los siguientes escenarios:

- ✓ Una implementación a nivel clase
- ✓ Una explícita de interface
- ✓ Ambas, una implementación explícita de interface y otra a nivel clase

INTERFACES DE LA PLATAFORMA USADAS PARA LA COMPARACIÓN

ICOMPARABLE

```
ArrayList lista =
new ArrayList() { 27, 5, 100, -1, 3 };
lista.Sort();
foreach (int i in lista)
```

```
{
    Console.WriteLine(i);
```

El método Sort() de ArrayList funciona correctamente porque todos los elementos de la lista son comparables entre sí porque implementan la interface Icomparable. Este método puede ordenar elementos de cualquier tipo, solo se necesita que le enseñemos a compararse, implementando IComparable

```
namespace System
```

```
{
    // Summary:
    // Defines a generalized type-specific comparison method that a value type or class
    // implements to order or sort its instances.
    public interface IComparable
    {
        // Compares the current instance with another object of the same type and returns
```

```
// an integer that indicates whether the current instance precedes, follows, or
// occurs in the same position in the sort order as the other object.
```

```
int CompareTo(object obj);
}
}
```

VALORES DE RETORNO DEL COMPARATOR

(<0) si this está antes que object

(=0) si this ocupa la misma posición que obj

(>0) si this está después que obj

```
public int CompareTo(object obj)
{
    string st1 = this.Nombre;
    string st2 = (obj as Empleado).Nombre;
    return st1.CompareTo(st2);
}
```

Si queremos otro criterio de orden, podemos usar una sobrecarga del método Sort que espera como argumento un objeto comparador que debe implementar la interface IComparer

```
namespace System.Collections
```

```
{
```

```
//
```

```
// Summary:
```

```
// Exposes a method that compares two objects.
```

```
public interface IComparer
```

```
{
```

```
//
```

```
// Summary:
```

```
// Compares two objects and returns a value indicating whether one is less than,
```

```
// equal to, or greater than the other.
```

```
//
```

```
int Compare(object x, object y);
```

```
}
```

Entonces

```
//Definir la clase CompararPorLegajo e implementar la interface IComparer
```

```
class ComparadorPorLegajo : IComparer
```

```
{
```

```
public int Compare(object x, object y)
```

```
{
```

```
Empleado e1 = x as Empleado;
```

```
Empleado e2 = y as Empleado;
```

```
return e1.Legajo.CompareTo(e2.Legajo);
```

```
}
```

```
}
```

```
//Luego
```

```
ArrayList lista = new ArrayList() {
```

```
new Empleado("Juan") {Legajo=79},
```

```
new Empleado("Adriana") {Legajo=123},
```

```
new Empleado("Diego") {Legajo=12}
```

```
};
```

```
lista.Sort(new ComparadorPorLegajo());
```

```
foreach (Empleado e in lista)
```

```
{
```

```
e.Imprimir();
```

```
}
```

INTERFACES PARA ENUMERAR: "System.Collections.IEnumerable" y "System.Collections.IEnumerator"

```
string[] vector = new string[] { "uno", "dos", "tres" };
```

```
foreach(string st in vector)
```

Un tipo es numerable si implementa la interface System.Collections.IEnumerable

```
namespace System.Collections
```

```
{
```

```
    public interface IEnumerable
```

```
    {
```

```
        // Returns an enumerator that
```

```
        // iterates through a collection.
```

```
        IEnumerator GetEnumerator(); → devuelve un objeto de tipo interface, es decir algún tipo que implementa la interface IEnumerator
```

```
    }
```

```
}
```

ENUMERADOR

Es un objeto que puede devolver los elementos de una colección, uno por uno, en orden según se lo solicite. Un enumerador "conoce" el orden de los elementos y realiza un seguimiento de dónde está en la secuencia. Luego devuelve el elemento actual cuando se solicita. Un enumerador debe implementar la interface System.Collection.IEnumerator

```
namespace System.Collections
```

```
{
```

```
    public interface IEnumerator
```

```
    {
```

```
        // Gets the current element in the current position.
```

```
        object Current { get; }
```

```
        // Advances the enumerator to the next element
```

```
        // Returns true if the enumerator was successfully advanced
```

```
        bool MoveNext();
```

```
        // Sets the enumerator before the first element
```

```
        void Reset();
```

```
}
}
```

Se requiere codificar una clase que implemente la interface `System.Collections.IEnumerator` para enumerar los nombres de las estaciones del año comenzando por “verano”

ITERADORES

Los iteradores constituyen una forma mucho más simple de crear enumeradores y enumerables. Usan la sentencia `yield`.

`Yield return` → devuelve un elemento de una colección y mueve la posición al siguiente elemento

`Yield break` → detiene la iteración

Un bloque iterador es un bloque de código que contiene una o más sentencias `yield`. Puede contener múltiples sentencias `yield return` o `yield break` pero no se permiten sentencias `return`

El tipo de retorno de un bloque iterador debe declararse `IEnumerator` o `IEnumerable`/

Un iterador produce un enumerador y no una lista de elementos. Este enumerador es invocado por la instrucción `foreach`. Esto permite iterar a través de grandes cantidades de datos sin leer todos los datos en memoria de una vez.

El iterador no es un método que se va a ejecutar desde la primera a última instrucción

FILE SYSTEM- ESPACIO DE NOMBRES SYSTEM.IO

La BCL incluye todo un espacio de nombres llamados `System.IO` especialmente orientado al trabajo con archivos. Entre las clases más usadas están:

PATH

Incluye un conjunto de miembros estáticos diseñados para realizar cómodamente las operaciones más frecuentes relacionadas con rutas y nombres de archivos. Con los campos públicos **`VolumeSeparatorChar`, `DirectorySeparatorChar`, `AltDirectorySeparatorChar` y `PathSeparator`** se obtiene el carácter específico de la plataforma que se usa para separar unidades, carpetas y archivos y el separador de múltiples rutas. EN Windows son: `:\`;

Ejemplo:

```
string archivo = "/Documentos/notas.txt";
Console.WriteLine(Path.GetFullPath(archivo));
Console.WriteLine(Path.GetFileName(archivo));
Console.WriteLine(Path.GetExtension(archivo));
Console.WriteLine(Path.GetDirectoryName(archivo));
Console.WriteLine(Path.ChangeExtension(archivo, "doc"));
Console.WriteLine(Path.GetFileNameWithoutExtension(archivo));
Console.WriteLine(Path.GetTempPath());
```

Mostrará:

```
C:\Documentos\notas.txt
```

```

notas.txt
.txt
\Documentos
/Documentos/notas.doc
notas
C:\Users\Leo\AppData\Local\Temp\

```

LAS CLASES DIRECTORYINFO, FILEINFO, DIRECTORY Y FILE

Para trabajar con archivos se usan objetos de la clase FileInfo y para trabajar con directorios objetos de la clase DirectoryInfo. Las clases File y Directory que solo tienen métodos estáticos son útiles para realizar tareas sencillas. No requieren la creación de ningún objeto pero son menos poderosas y eficientes

Ejemplo:

```

string stDir = Environment.CurrentDirectory;
DirectoryInfo dirInfo = new DirectoryInfo(stDir);
FileInfo[] archivos = dirInfo.GetFiles();
foreach (FileInfo archivo in archivos)
{
    string st = $"{archivo.Name} {archivo.Length} bytes";
    Console.WriteLine(st);
}
}

```

DirectoryInfo y FileInfo ejemplo

Imprimirá:

- 1) Program.cs 541 bytes
- 2) Teoria7.csproj 178 bytes

ARCHIVOS DE TEXTO

El trabajo con archivos en .NET está ligado al concepto de stream o flujo de datos, que consiste en tratar su contenido como una secuencia ordenada de datos. El concepto de stream es aplicable también a otros tipos de almacenes de información tales como conexiones de red o buffers de memoria. La BCL proporciona las clases StreamReader y StreamWriter. Los objetos de estas clases facilitan la lectura y escritura de archivos de textos

STREAMREADER

Para facilitar la lectura de flujos de texto StreamReader ofrece una familia de métodos que permiten leer sus caracteres de diferentes formas

- ✓ De uno en uno: el método `int Read()` devuelve el próximo carácter del flujo. Tras cada lectura la posición actual en el flujo se mueve un carácter hacia adelante
- ✓ Por líneas: el método `string ReadLine()` devuelve la siguiente línea del flujo (y avanza la posición en el flujo). Una línea de texto es cualquier secuencia de caracteres terminada en `\n`, `\y` o `\r\n`.

- ✓ Por completo: `string ReadToEnd()`, que nos devuelve una cadena con todo el texto que hubiese desde la posición actual hasta el final (y avanza hasta el final del flujo)

STREAMWRITTER

Ofrece métodos que permiten:

- ✓ Escribir cadenas de texto: el método `Write()` escribe cualquier cadena de texto en el destino que tenga asociado. Pueden usarse formatos compuestos
- ✓ Escribir líneas de texto: el método `WriteLine()` funciona igual que `Write()` pero añade un indicador de fin de línea. También pueden usarse formatos compuestos

Dado que el indicador de fin de línea depende de cada sistema operativo, `StreamWriter` dispone de una propiedad `string NewLine` mediante la que puede configurarse este indicador. Su valor por defecto en Windows es `\r\n`.

Ejemplo:

```
StreamReader sr = new StreamReader("fuente.txt");
StreamWriter sw = new StreamWriter("destino.txt");
string linea;
while (!sr.EndOfStream)
{
    linea = sr.ReadLine();
    sw.WriteLine(linea);
}
```

`sr.Close(); sw.Close();` → el método `close()` libera los recursos de manera explícita, invocando un método `Dispose()`

Si queremos saber si es necesario liberar recursos explícitamente debemos verificar si implementa la interface `IDisposable`.

En C# la alternativa recomendada al uso de finalizadores, es implementar la interfaz `System.IDisposable` que posee un único método:

```
public interface IDisposable
{
    void Dispose();
}
```

INTERFACE IDISPOSABLE

El cual define un mecanismo determinista para liberar recursos no administrados y evita los problemas relacionados con el recolector de basura inherentes a los finalizadores

Cuando se termina de usar un objeto que implementa `IDisposable`, se debe invocar el método `Dispose()` del objeto. Hay 2 maneras de hacerlo:

- ✓ Mediante un bloque `try/finally`
- ✓ Mediante la instrucción `using`

```
TipoDisposable recurso = new TipoDisposable(...)
try {
    bloque de sentencias
} finally {
    if (recurso != null) recurso.Dispose();
}
```

INSTRUCCIÓN USING

En una instrucción using se pueden instanciar más de un objeto del mismo tipo:

```
using (StreamReader f1 = new StreamReader("file1.txt"),
f2 = new StreamReader("file2.txt"))
{
    ...
}
```

Si se trata de distintos tipos los using se pueden anidar, como se observa:

```
using (StreamReader sr = new StreamReader("fuente.txt")){
    using (StreamWriter sw = new StreamWriter("destino.txt")){
        sw.Write(sr.ReadToEnd());
    }
}
catch (Exception e){
    Console.WriteLine(e.Message);
}
```

DELEGADOS

Es un tipo especial de clase cuyos objetos almacenan **referencias a uno o más métodos** de manera de poder ejecutar en cadena esos métodos. Permiten pasar **métodos como parámetros** a otros métodos y proporcionan un mecanismo para implementar eventos.

Las variables que admiten métodos son de algún tipo **delegado**

Para definir un tipo de delegado, se usa una sintaxis similar a la definición de una firma de método. Solo hace falta agregar la palabra *delegate*

```
delegate int FuncionEntera(int n);
```

El compilador genera una clase derivada de System.Delegate que coincide con la firma usada

Ejemplo:

```
delegate int FuncionEntera(int n);
class Program
```



```

{
static void Main(string[] args)
{
FuncionEntera f;
f = SumaUno;
Console.WriteLine(f(10)); → se invoca SumaUno por medio de f
f = SumaDos;
Console.WriteLine(f(10)); → se invoca SumaDos por medio de f
}
static int SumaUno(int n) => n + 1;
static int SumaDos(int n) => n + 2;
}

```

También se pueden invocar los métodos en los delegados de forma explícita usando el método Invoke

ASIGNACIÓN DE DELEGADOS

Las variables de tipo delegado pueden asignarse directamente con el nombre del método o con su correspondiente constructor pasando el método como parámetro

F= SumaUno ↔ f= new FuncionEntera(SumaUno);

PASANDO METODOS COMO PARAMETROS

static void Aplicar(int[] v, FuncionEntera f) → recibe como parámetros un vector y una función

```

{
for (int i = 0; i < v.Length; i++)
{
v[i] = f(v[i]); → aplica la función f a cada uno de los elementos del vector v
}
}

```

MÉTODOS ANONIMOS

En ocasiones los métodos solo se utilizan para crear una instancia de un delegado. Los métodos anónimos permiten prescindir del método con nombre definido por separado. Un método anónimo es un método que se declara en línea, en el momento de crear una instancia de un delegado

La sintaxis de un método anónimo incluye:

- 1) La palabra clave delegate
- 2) La lista de parámetros (si son necesarios)
- 3) El bloque de sentencias con la implementación del método

Entonces:

Delegate (parámetros) {implementación};

Los métodos anónimos pueden acceder a sus variables locales y a las definidas en el entorno que lo rodea.

Se pueden transformar en una expresión lambda haciendo lo siguiente:

- 1) Eliminar la palabra clave delegado

2) Colocar el operador lambda => entre la lista de parámetros y el cuerpo del método anónimo

Entonces:

```
f= delegate (int n) {return n*2};
```

```
f=(int n)=> {return n*2};;
```

Pero aún es posible otras simplificaciones sintácticas:

Si no existen parámetros red, in o out, el tipo de los parámetros puede omitirse:

```
F=(n) => {return n*2};;
```

Si hay un único parámetro, pueden omitirse los paréntesis

```
f= n => {return n*2};;
```

Si el bloque de instrucciones es solo una expresión de retorno, puede reemplazarse todo el bloque por la expresión de retorno:

```
f= n => n*2;
```

Nota: si el delegado no tiene parámetros se deben usar paréntesis vacíos:

```
linea= () => Console.WriteLine();
```

Un delegado puede llamar a más de un método cuando se invoca, se conoce como multidifusión.

La instrucción `.GetInvocationList()` devuelve un arreglo de objetos `Delegate` que corresponden a la lista de delegados encolados.

Action es un tipo delegado predefinido en la BCL.

EVENTOS

Cuando ocurre algo importante, un objeto puede notificar el evento a otras clases u objetos. La clase que produce el evento recibe el nombre de **editor** y las clases que están interesadas en conocer la ocurrencia del evento se denominan **suscriptores**. Para que un suscriptor sea notificado, necesita estar suscripto al evento.

El **editor** determina **cuándo** se produce un evento; los **suscriptores** codifican en un método (manejador del evento) lo que harán cuando se produzca ese evento.

Un **evento** puede tener **varios suscriptores**. Un **suscriptor** puede manejar varios eventos de **varios editores**. Nunca se provocan eventos que no tienen suscriptores.

Ejemplo:

Dada la clase Program conoce y va a poner a trabajar un objeto t de tipo Trabajador. El objeto t no conoce a la clase Program. Pero Program se tiene que enterar cuando t termina de trabajar pero queremos mantener el bajo acoplamiento. Entonces Program se suscribe al evento TrabajoFinalizado de t. Cuando Program hace trabajar a t, Program es notificado cuando t termina su produce el evento TrabajoFinalizado pero realmente no conoce quién o quiénes se están notificando

```

class Program {
    public static void Main() {
        Trabajador t = new Trabajador();
        t.TrabajoFinalizado = ManejadorDelEvento;
        t.Trabajar();
    }
    private static void ManejadorDelEvento() → TrabajoFinalizado es un campo publico de tipo delegado de t
    => Console.WriteLine("trabajo finalizado");
}

```

La clase Program se suscribe al evento TrabajoFinalizado del objeto t, asignando su propio método ManejadorDelEvento para manejar dicho evento

```

class Trabajador {
    public Action TrabajoFinalizado;
    public void Trabajar() {
        Console.WriteLine("trabajador trabajando...");
        // hace algún trabajo útil
        if (TrabajoFinalizado != null) { → **
            TrabajoFinalizado();
        }
    }
}

** Aquí se produce el evento invocando la lista de métodos encolados en el delegado. Si no se ha encolado ningún método la variable tiene el valor null.

```

EVENTOS- CONVENCIONES

Para los nombres de los eventos se recomiendan verbos en gerundio (ej: iniciandoTrabajo) o participio (ej: TrabajoFinalizado) según se produzcan antes o después del hecho de significación

Los delegados usados para invocar a los manejadores de eventos deben tener 2 argumentos: uno de tipo object que contendrá al objeto que genera el evento y otro de tipo EventArgs para pasar argumentos. Además su tipo de retorno debe ser void.

TIPO EVENT HANDLER

El tipo delegado EventHandler se usa para el caso de un evento que no requiere pasar datos como parámetros cuando se invoque el delegado

public delegate void EventHandler (object sender, EventArgs e); → es una clase vacía, no lleva datos pero constituye la clase base de todas las que se usan para pasar argumentos

- ✓ Es deseable que los nombres que se usen compartan una raíz común

- ✓ Por ejemplo si defino un evento CapacidadExcedida: la clase para apsar los argumentos se debería denominar CapacidadExcedidaEventArgs y el delegado asociado al evento, se debería denominar CapacidadExcedidaEventHandler

Ejemplo:

Si una clase produce el evento TrabajoFinalizado, deberíamos definir los siguientes tipos:

```
class TrabajoFinalizadoEventArgs : EventArgs
{
    public TimeSpan TiempoConsumido { get; set; }
}
delegate void TrabajoFinalizadoEventHandler(
    object sender,
    TrabajoFinalizadoEventArgs e);
```

Tarea: se requiere codificar una clase Trabajador, con un método publico Trabajar que producto un evento TrabajoFinalizado una vez concluida su tarea. Debe además comunicar el tiempo insumido en la ejecución del trabajo

```
class TrabajoFinalizadoEventArgs : EventArgs
{
    public TimeSpan TiempoConsumido { get; set; }
}
delegate void TrabajoFinalizadoEventHandler(
    object sender,
    TrabajoFinalizadoEventArgs e);
class Trabajador
{
    public TrabajoFinalizadoEventHandler TrabajoFinalizado;
    public void Trabajar()
    {
        Console.WriteLine("Trabajador trabajando...");
        DateTime tInicial = DateTime.Now;
        //Pierdo tiempo simulando el trabajo
        for (int i = 1; i < 100_000_000; i++) ;
        TimeSpan lapso = DateTime.Now - tInicial;
        if (TrabajoFinalizado != null)
        {
            TrabajoFinalizadoEventArgs e;
            e = new TrabajoFinalizadoEventArgs();
            e.TiempoConsumido = lapso;
            TrabajoFinalizado(this, e);
        }
    }
}
```

```

}
}
class Program
{
    public static void Main()
    {
        Trabajador t = new Trabajador();
        t.TrabajoFinalizado = t_TrabajoFinalizado;
        t.Trabajar();
    }
    private static void t_TrabajoFinalizado(object sender, TrabajoFinalizadoEventArgs e)
    {
        string st = "Trabajo terminado en ";
        st += $"{e.TiempoConsumido.TotalMilliseconds} ms.";
        Console.WriteLine(st);
    }
}

```

OBSERVACIONES

Gracias a la capacidad de multidifusión de los delegados, es posible que varias entidades se suscriban a un mismo evento, solo tienen que conocer al que lo genera para encolar su propio manejador.

Cada uno de los suscriptores debería suscribirse al evento usando el operador += para encolar su manejador sin eliminar los otros. Pero no podemos garantizarlo porque dejamos público el campo delegado que representa al evento

EVENT

Un evento será un miembro definido con la palabra clave Event. Así como una propiedad controla el acceso a un campo de una clase u objeto, un evento lo hace con respecto a campos de tipo delegados, permitiendo ejecutar código cada vez que se añade o elimina un método del campo delegado. A diferencia de los delegados, a los eventos solo se le pueden aplicar dos operaciones: += y -=

Sintaxis:

```

public event <TipoDelegado> NombreDelEvento
{
    add
    {
        <código add> → 1)
    }
    remove
    {
        <código remove> 2)
    }
}

```

```
}
```

- 1) Código que se ejecutará cuando desde afuera se haga un +=. En este bloque la variable implícita value contiene el delegado que se desea encolar
- 2) Código que se ejecutara cuando desde afuera se haga un -=. En este bloque la variable implícita value contiene el delegado que se desea encolar

En ocasiones no es necesario establecer control alguno en los descriptores de acceso add y remove. Pasa estos casos C# provee una notación abreviada

```
Public evento EventHandler TrabajoFinalizado;
```

El compilador crea un campo privado de tipo EventHandler e implementa los descriptores de acceso add y remove para suscribirse y anular la suscripción al evento

OPERADOR CONDICIONAL NULL

Si la variable persona es null, en lugar de generar una excepción NullReferenceException, se cortocircuita y devuelve null. A menudo se usa con el operador ??

```
String nombre= persona?.Nombre;
```

```
String nombre= persona?.Nombre ?? "indefinido";
```

También se usa para invocar métodos de forma condicional. El uso más común es invocar de forma segura un delegado

```
This.SomethingHappened?.Invoke(this, new EventArgs());
```