



# .NET

## Teoría 2

# Sistema de Tipos



## Common Type System (CTS)

- Define un conjunto común 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



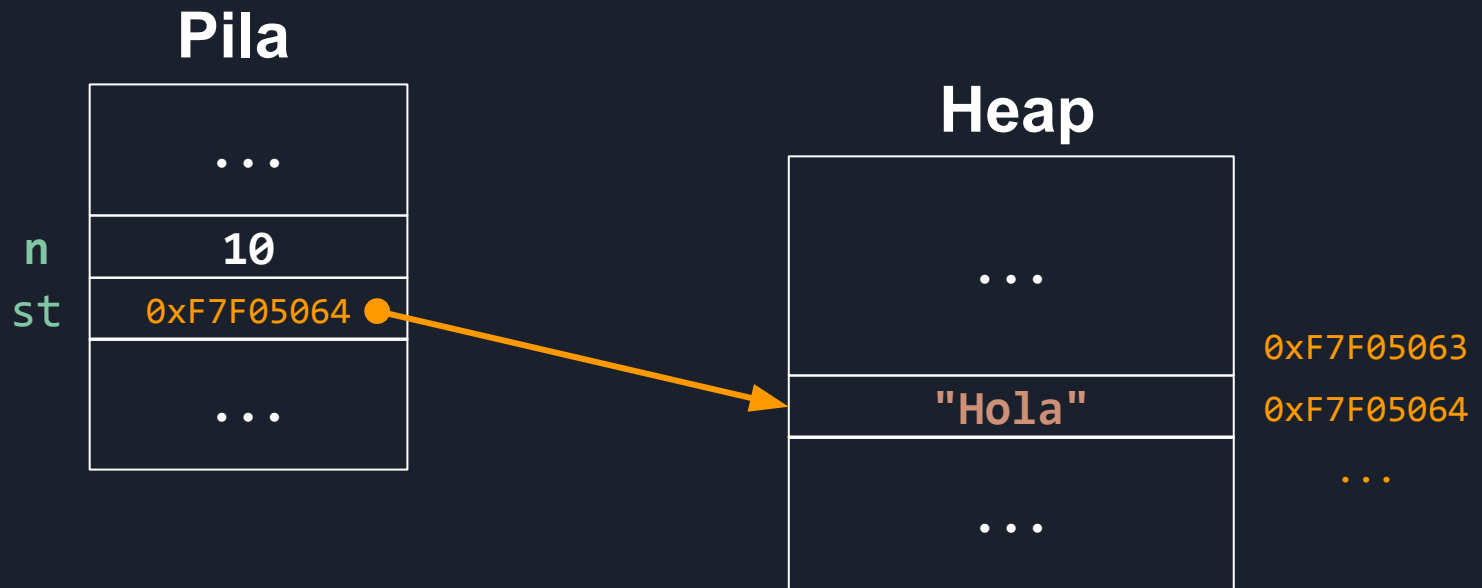
# Sistema de tipos

- **Tipos de valor:** El espacio reservado para la variable en la pila de ejecución guarda directamente el valor asignado.
- **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

## Sistema de tipos

## CÓDIGO

```
...  
int n = 10;           // int es un tipo de valor  
string st = "Hola";  //string es un tipo de referencia  
...
```



## Sistema de tipos

## CÓDIGO

```
...  
int n = 10;  
string st = "Hola";  
n = 11;  
...
```

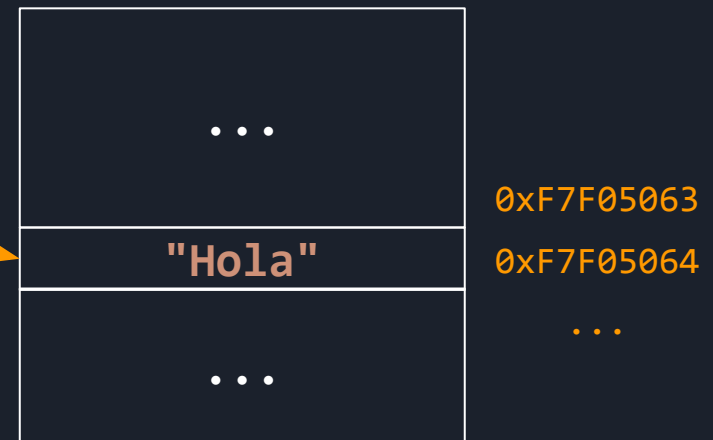


Asignar un nuevo valor a la variable **n** (tipo de valor) implica modificar la memoria de la pila asociada a esta la variable **n** escribiéndose allí el nuevo valor

## Pila



## Heap



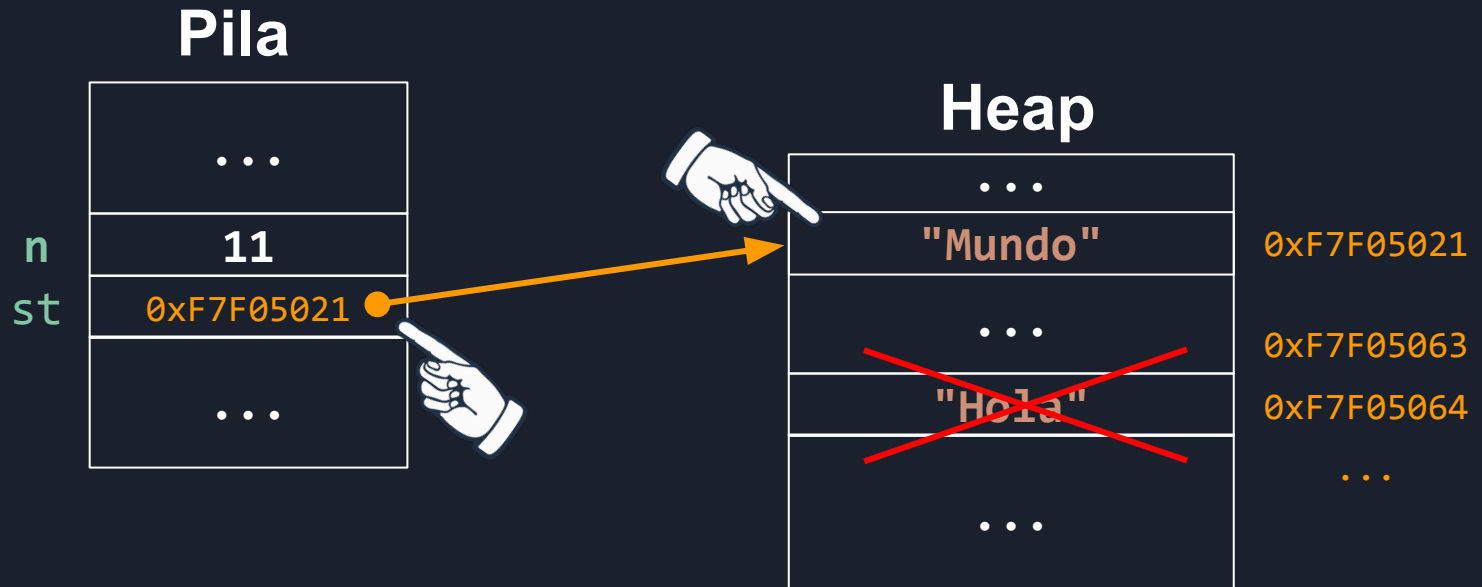
## Sistema de tipos

## CÓDIGO

```
...  
int n = 10;  
string st = "Hola";  
n = 11;  
st = "Mundo";  
...
```



Asignar un nuevo valor a la variable **st** (tipo de referencia) implica conseguir espacio libre en la heap, colocar el nuevo valor en la nueva dirección escogida y modificar la memoria de la pila asociada a la variable **st** escribiéndose allí la nueva dirección



## Sistema de tipos

- **Common Type System** admite las cinco categorías de tipos siguientes:

- Estructuras
- Enumeraciones
- Clases
- Delegados
- Interfaces

Tipos de Valor

Tipos de Referencia



## Tipos de valor en C#

- Estructuras
  - `char`
  - `bool`
  - Tipos numéricos
    - Tipos enteros (`sbyte`, `byte`, `short`, `int` ...)
    - Tipos de punto flotantes (`float`, `double` y `decimal`)
  - Gran cantidad de estructuras definidas en la BCL (`DateTime`, `TimeSpan`, `Guid` etc. )
  - Estructuras definidas por el usuario
- Enumeraciones

Todos los tipos integrados (con excepción de `string` `object` y `dynamic`) son estructuras

# Tipos de referencia en C#

- Clases
- Delegados
- Interfaces

En particular `object` es un tipo de `referencia` y constituye la `raíz` de la jerarquía de tipos (`Sistema unificado de tipos`).

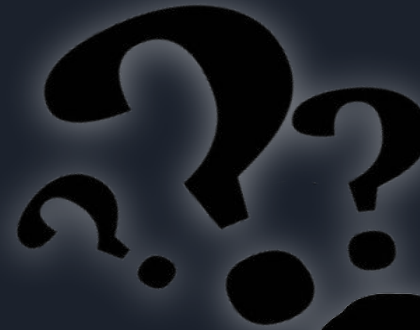
## Sistema de Tipos - Conversión de tipos

- Conversiones implícitas
- Conversiones explícitas, requieren un operador de conversión.
- Conversiones con tipos auxiliares: para realizar conversiones entre tipos no compatibles.
  - La clase `System.Convert`
  - Los métodos `Parse` de los tipos numéricos
  - El método `ToString` redefinible en todos los tipos
- Conversiones definidas por el usuario

## Conversiones de tipo numéricas

El siguiente código tiene algunos errores

```
byte b = 10;  
double x = 12.25;  
int i = b;  
double y = i;  
short j = i;  
i = x;
```



¿Donde están  
los errores?

## Conversiones de tipo numéricas

El siguiente código tiene algunos errores

```
byte b = 10;  
double x = 12.25;  
int i = b;  
double y = i;  
short j = i;  
-----  
i = x;  
-----
```

OK Conversión implícita  
de **byte** a **int**

OK Conversión implícita  
de **int** a **double**

ERROR DE COMPILACIÓN  
En estos dos casos no es posible  
la conversión implícita

## Conversiones de tipo numéricas

Corrigiendo dichos errores

```
byte b = 10;  
double x = 12.25;  
int i = b;  
double y = i;  
short j = (short)i;  
i = (int)x;
```

CONVERSIÓN EXPLÍCITA  
utilizando una expresión cast  
**i** se asigna con valor 12



## Conversiones de tipo

En general, la conversión de tipo implícita se realiza cuando la operación es segura.

En otro caso, se requiere el consentimiento del programador quien debe hacerse responsable de la seguridad de la operación



## Atención !



Las conversiones de `int`, `uint`, `long` o `ulong` a `float` y de `long` o `ulong` a `double` pueden producir una pérdida de precisión, pero no una de magnitud.

Ejemplo:

```
int i = 1_000_000_321;
```


```
float f = i;
```

f queda asigna con  
1\_000\_000\_000

En los literales numéricos el guión bajo “\_” es ignorado. Es útil para hacer más legible los números de muchas cifras.



## Conversiones de tipo con clases auxiliares



```
int i = int.Parse("321");
double d = int.Parse("321,34");
d = double.Parse("321,45");
string st = i.ToString();
st = 27.654.ToString();
DateTime fecha = DateTime.Parse("23/3/2012");
i = (int>true;
i = Convert.ToInt32(true);
```

Error en ejecución  
(`FormatException`)

Error de compilación  
Esta conversión  
explícita no está  
definida

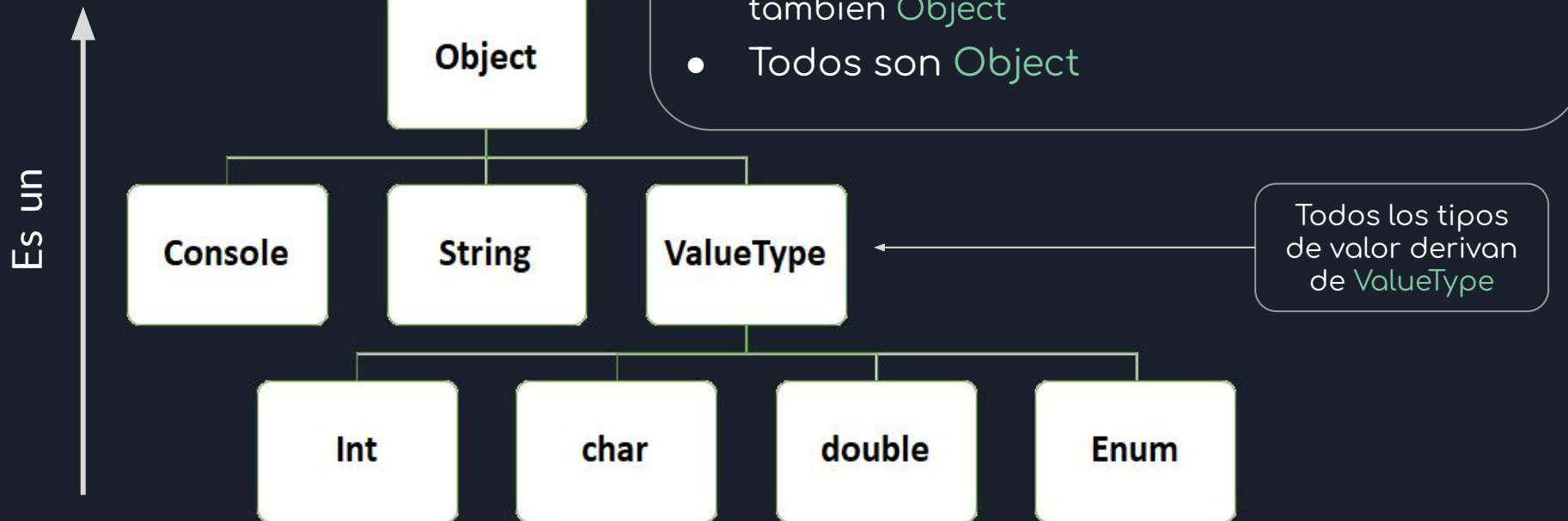
Esto sí funciona, en este caso se asigna 1 a i  
La clase `Convert` puede realizar una gran  
cantidad de conversiones



## Sistema unificado de tipos

- Todos los tipos de datos **derivan** directa o indirectamente de un tipo base común: la clase **System.Object**
- Esto también es aplicable a los **tipos de valor** (conversiones boxing y unboxing )

## Sistema unificado de tipos

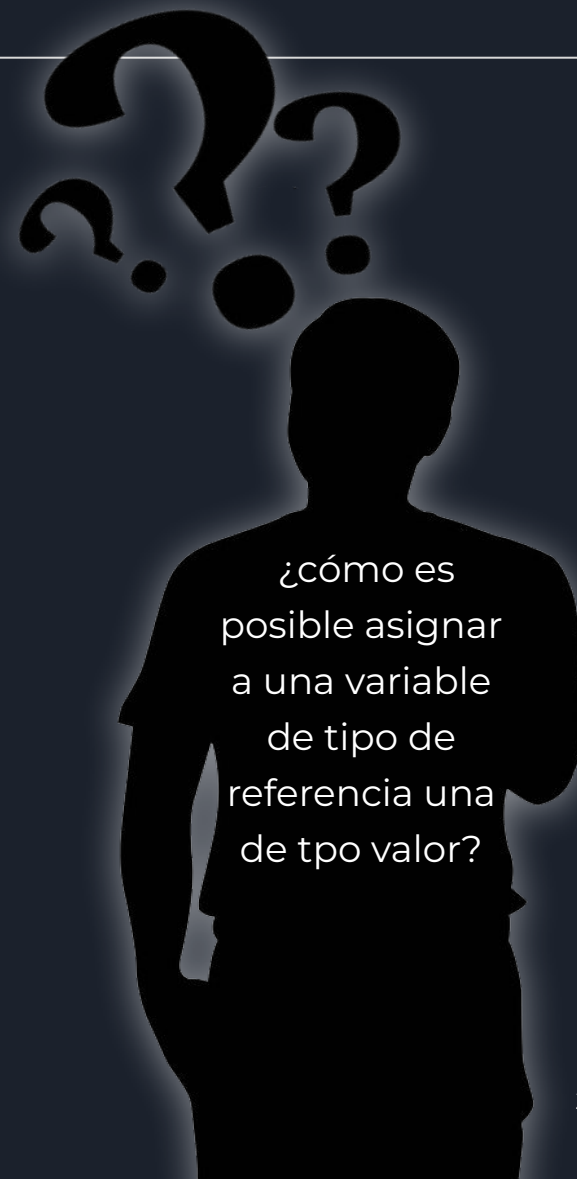


## Consecuencia de tener un Sistema unificado de tipos

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.

Por ejemplo, el siguiente fragmento de código es válido:

```
int i = 123;  
object o = i;
```

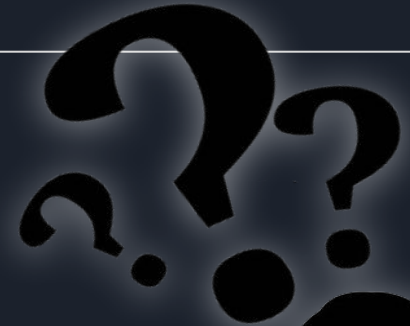


¿cómo es posible asignar a una variable de tipo de referencia una de tpo valor?

# Consecuencia de tener un Sistema unificado de tipos



Las conversiones boxing y unboxing lo hacen posible



¿cómo es posible asignar a una variable de tipo de referencia una de tpo valor?

## Boxing y Unboxing

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

```
object o;  
int i = 123;  
.  
.  
.  
o = i;  
.  
.  
.  
int j = (int)o;
```

## Boxing y Unboxing

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

```
object o;
```

```
int i = 123;
```

```
. . .
```

```
o = i;
```

```
. . .
```

```
int j = (int)o;
```

*boxing*

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*.

*unboxing*

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*.

## Boxing y Unboxing

Estado de la pila y la memoria heap previos al boxing y unboxing para el siguiente fragmento de código

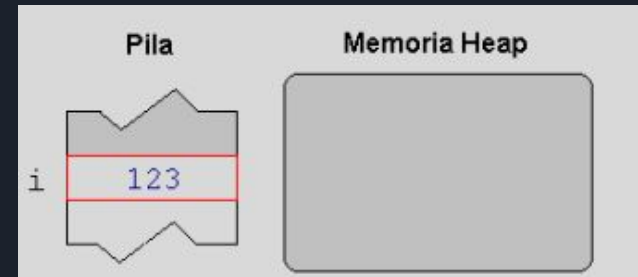
```
object o;  
int i = 123;
```

• • •

```
o = i;
```

• • •

```
int j = (int)o;
```





## Boxing

Se “encaja” el valor de la variable *i* en un *objeto* en la *heap* y la *referencia* es guardada en la variable *o*

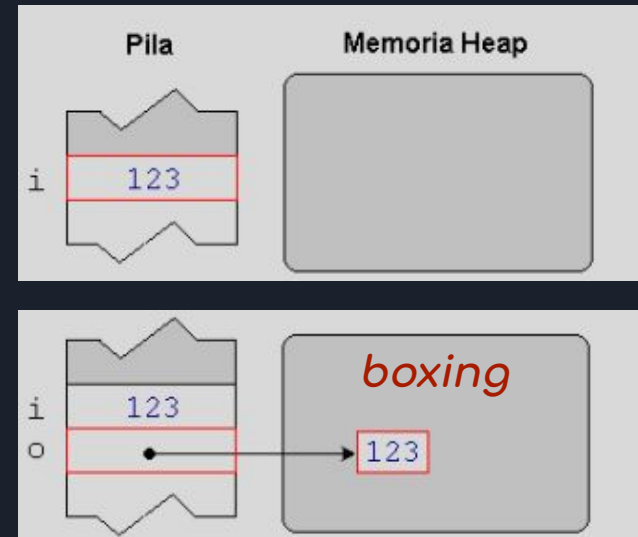
```
object o;  
int i = 123;
```

• • •

```
o = i;
```

• • •

```
int j = (int)o;
```



# Boxing

Se “encaja” el valor de la variable *i* en un **objeto** en la **heap** y la **referencia** es guardada en la variable *o*

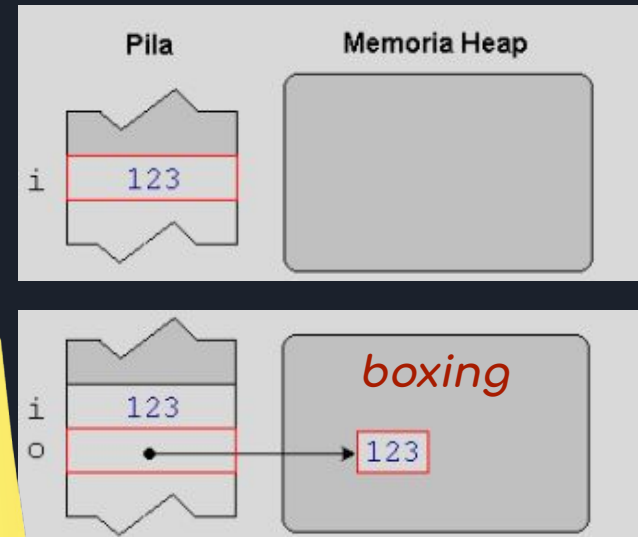
```
object o;  
int i = 123;
```

```
• • •  
o = i;
```

```
• • •
```

```
int j = (
```

**NOTA:** Si se asignara a la variable *o* un literal de algún tipo de valor, por ejemplo *o = 123* también provocaría una conversión boxing.



# Unboxing

Se “desencaja” el valor referenciado por la variable `o` y se asigna a la variable `j`

```
object o;
```

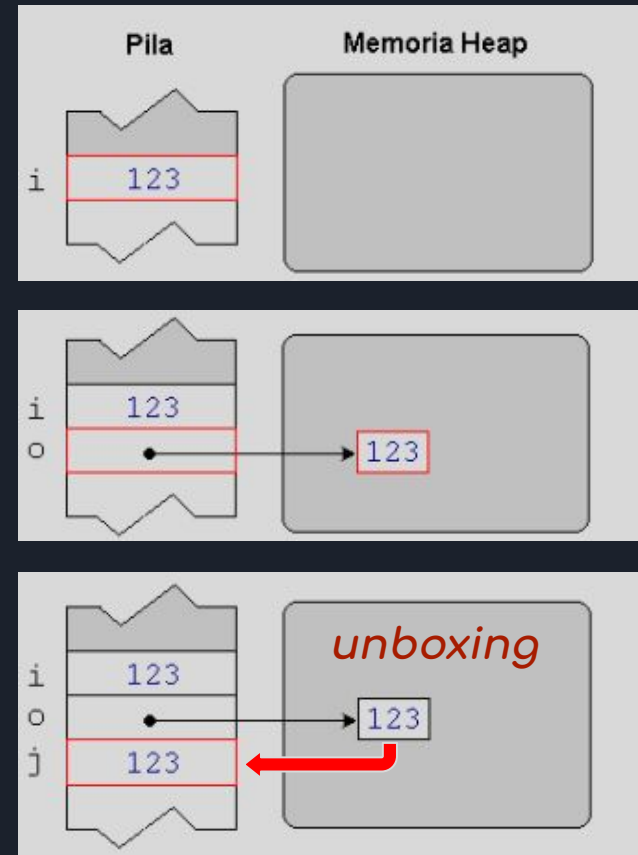
```
int i = 123;
```

```
• • •
```

```
o = i;
```

```
• • •
```

```
int j = (int)o;
```



## Consecuencias del Sistema unificado de tipos

Si un `int` es también un `object` entonces lo que podemos hacer con un `object` también lo podemos hacer con un `int` (lo inverso no es cierto)

Los métodos `ToString()` y `GetType()` están definidos en la clase `object`, por lo tanto todos los objetos de cualquier tipo podrán invocar estos dos métodos.

`7.ToString();`

Devuelve un `string` con la representación del objeto que lo invoca, en este caso `"7"`

`"casa".GetType();`

Devuelve el `tipo exacto` (el más específico) del objeto que lo invoca, en este caso `string`



### Poniendo en práctica



1. Abrir una consola del sistema operativo
2. Cambiar a la carpeta `proyectosDotnet`
3. Crear la aplicación de consola `Teoria2`
4. Abrir `Visual Studio Code` sobre este proyecto



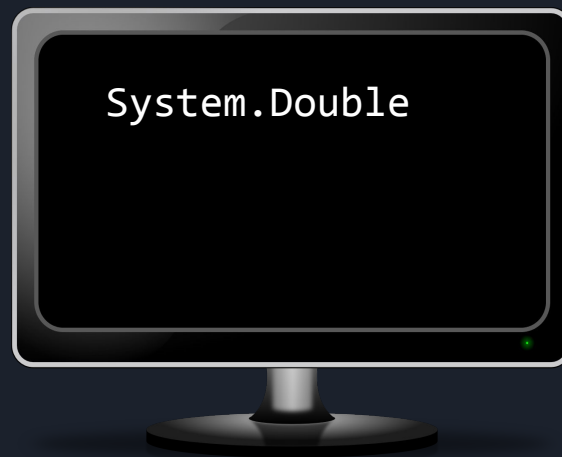
# Probar el siguiente código

```
object obj = 7.3; // obj apunta a un valor de tipo double  
Console.WriteLine(obj.GetType());
```



### Salida por la consola

```
object obj = 7.3; // obj apunta a un valor de tipo double  
Console.WriteLine(obj.GetType());
```





Agregar las líneas resaltadas y volver a ejecutar

```
object obj = 7.3; // obj apunta a un valor de tipo double  
Console.WriteLine(obj.GetType());  
obj = "Casa"; // ahora de tipo string  
Console.WriteLine(obj.GetType());
```

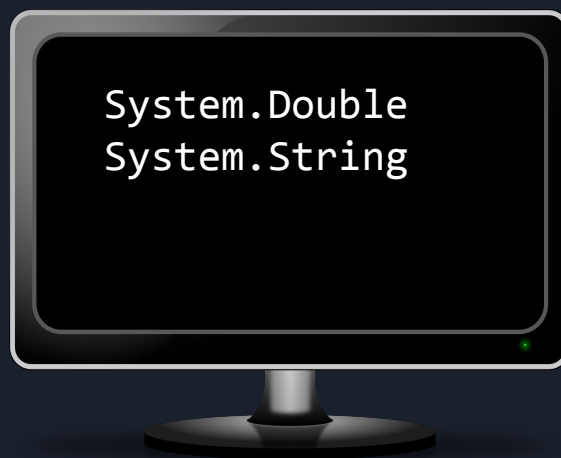






### Salida por la consola

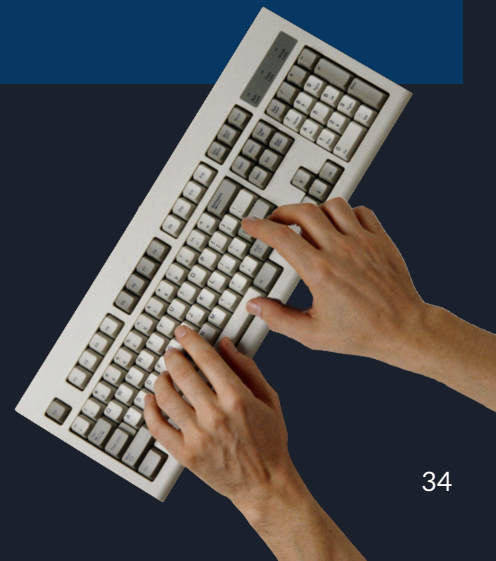
```
object obj = 7.3; // obj apunta a un valor de tipo double
Console.WriteLine(obj.GetType());
obj = "Casa"; // ahora de tipo string
Console.WriteLine(obj.GetType());
```





Agregar las líneas resaltadas y volver a ejecutar

```
object obj = 7.3; // obj apunta a un valor de tipo double
Console.WriteLine(obj.GetType());
obj = "Casa"; // ahora de tipo string
Console.WriteLine(obj.GetType());
obj = 4; // ahora de tipo int
Console.WriteLine(obj.GetType());
```





### Salida por la consola

```
object obj = 7.3; // obj apunta a un valor de tipo double
Console.WriteLine(obj.GetType());
obj = "Casa"; // ahora de tipo string
Console.WriteLine(obj.GetType());
obj = 4; // ahora de tipo int
Console.WriteLine(obj.GetType());
```

A black computer monitor with a silver stand, displaying the output of the code. The screen shows three lines of text: 'System.Double', 'System.String', and 'System.Int32'.

```
System.Double
System.String
System.Int32
```



# Eliminar las invocaciones a GetType y volver a ejecutar

```
object obj = 7.3; // obj apunta a un valor de tipo double
```

```
Console.WriteLine(obj);
```

```
obj = "Casa"; // ahora de tipo string
```

```
Console.WriteLine(obj);
```

```
obj = 4; // ahora de tipo int
```

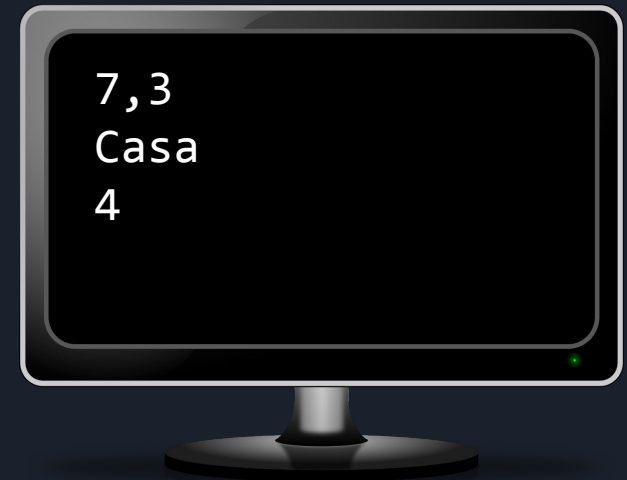
```
Console.WriteLine(obj);
```





### Salida por la consola

```
object obj = 7.3; // obj apunta a un valor de tipo double
Console.WriteLine(obj);
obj = "Casa"; // ahora de tipo string
Console.WriteLine(obj);
obj = 4; // ahora de tipo int
Console.WriteLine(obj);
```



¿Cómo funciona el método `WriteLine` de la clase `Console`?

```
Console.WriteLine(obj);
```

Haciendo una simplificación podemos pensar que `WriteLine` recibe como parámetro un objeto de cualquier tipo, se invoca `obj.ToString()` y el resultado devuelto se imprime en la pantalla.



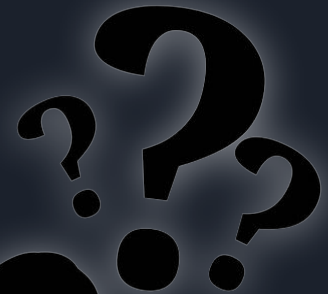
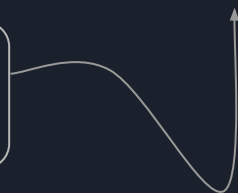


## Probar el siguiente código



```
object obj = 7.3; // obj apunta a un valor de tipo double
Console.WriteLine(obj);
obj = "Casa"; // ahora de tipo string
Console.WriteLine(obj);
obj = 4; // ahora de tipo int
Console.WriteLine(obj + 1);
```

Intentar esta  
suma



¿Cual es el  
problema?



Probar el siguiente código

```
object obj = 7.3; // obj apunta  
Console.WriteLine(obj);  
obj = "Casa"; // ahora de tipo  
Console.WriteLine(obj);  
obj = 4; // ahora de tipo int  
Console.WriteLine(obj + 1);
```

El operador **+** no está  
definido para el caso  
en que uno de sus  
operandos sea un  
**object** y el otro un **int**  
¿Cómo se arregla?





## Probar el siguiente código



```
object obj = 7.3; // obj apunta a un valor de tipo double
Console.WriteLine(obj);
obj = "Casa"; // ahora de tipo string
Console.WriteLine(obj);
obj = 4; // ahora de tipo int
Console.WriteLine((int)obj + 1);
```

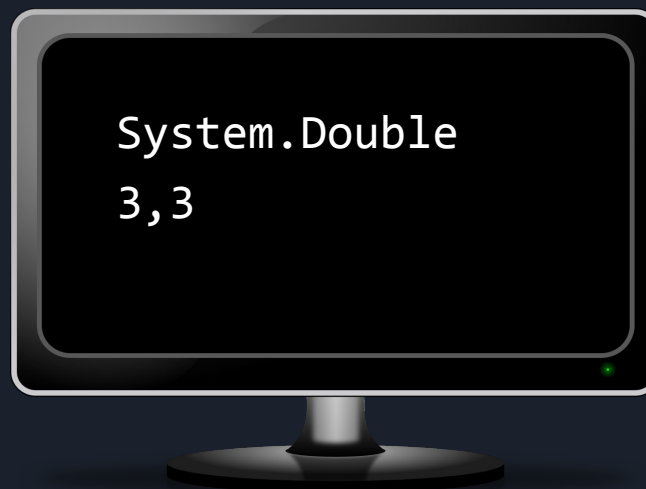
Conversión explícita del  
contenido de la variable  
`obj`



Atención. El operador `+` está sobrecargado

```
object obj = 1 + 2.3;  
Console.WriteLine(obj.GetType());  
Console.WriteLine(obj);
```

Sumando un `int` con  
un `double`

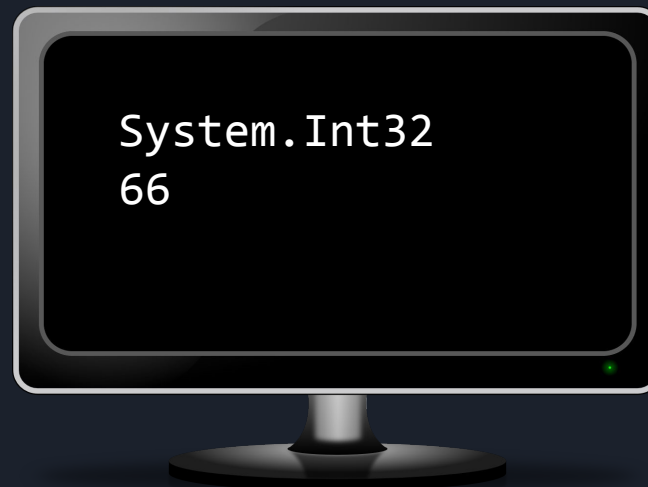




Atención. El operador `+` está sobrecargado

```
object obj = 1 + 'A';  
Console.WriteLine(obj.GetType());  
Console.WriteLine(obj);
```

Sumando un `int` con  
un `char`

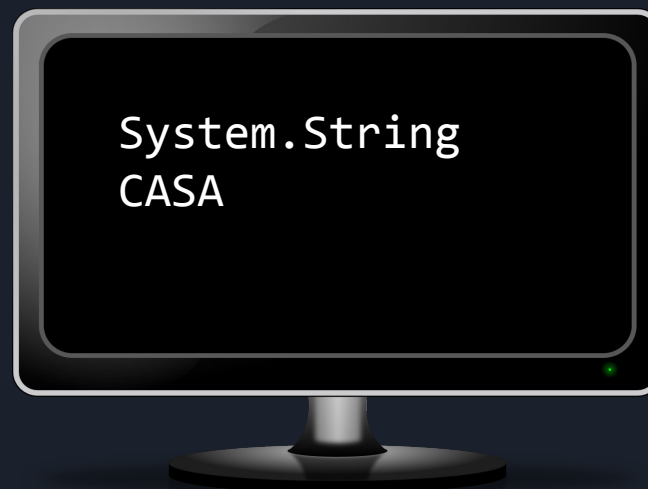




Atención. El operador `+` está sobrecargado

```
object obj = "CAS" + 'A';  
Console.WriteLine(obj.GetType());  
Console.WriteLine(obj);
```

Sumando un `string`  
con un `char`

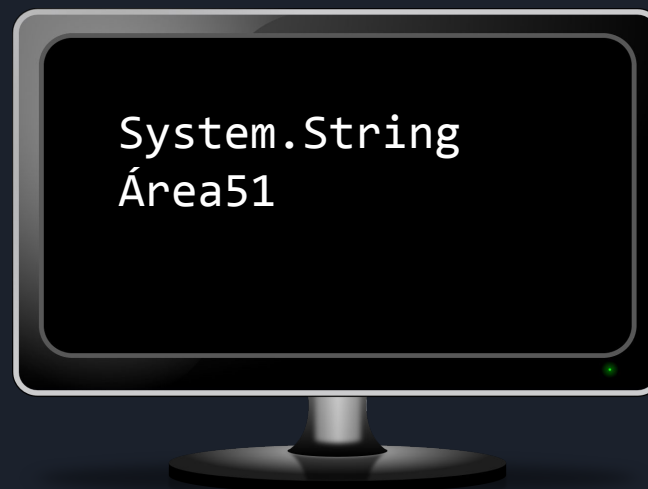




Atención. El operador `+` está sobrecargado

```
object obj = "Área" + 51;  
Console.WriteLine(obj.GetType());  
Console.WriteLine(obj);
```

Sumando un `string`  
con un `int`



## Responder

¿Cuál es el resultado de las siguientes operaciones?

"Área" + 5 + 1

5 + 1 + "Área"




## Responder

¿Cuál es el resultado de las siguientes operaciones?

"Área" + 5 + 1 → "Área51"

5 + 1 + "Área" → "6Área"

# Sistema de Tipos - El valor **null**



Si queremos que una variable de un tipo **T** admita, además de todos los valores propios de **T**, el valor especial **null** (es decir que sea nullable) debemos declararla de tipo **T?**

Generalmente asociamos el valor **null** con “ausencia de valor”, situación relativamente frecuente en algunos contextos como puede ser el de las bases de datos



# Sistema de Tipos - El valor `null`

## Ejemplo asignación valor `null`

```
int? i1 = null;
```

OK

```
string? st1 = null;
```

OK

```
int i2 = null;
```

Asignar `null` a una variable de un tipo de valor no nullable provoca un `error` de compilación

```
string st2 = null;
```

Asignar `null` a una variable de un tipo de referencia no nullable provoca sólo un `warning` del compilador

## El valor `null` en tipos de referencia

Para los diseñadores del lenguaje fue fácil implementar `null` en tipos de referencia, simplemente se utiliza una dirección inválida en la pila, típicamente la dirección cero

CODIGO

```
...  
string? st1 = null;  
string? st2 = "Hola";  
...
```



## El valor `null` en tipos de valor

Dado que cualquier secuencia de bits asociada en la pila a una variable de tipo de valor representará un dato válido, para implementar un tipo de valor que admita `null` se utilizó una estructura con dos propiedades (`HasValue` y `Value`)

CODIGO

```
...
int? i1 = 10;
int? i2 = null;
...
```

```
i1 { i1.HasValue
     i1.Value
i2 { i2.HasValue
     i2.Value
```

### Pila

...
true
10
false
0
...

## Usando tipos de valor que admiten null

```
int i = 1;
```

```
int? j = i;
```

Conversión Implícita

```
i = (int)j;
```

Es necesario "castear"  
Conversión explícita

```
j = null;
```

```
 i = (int)j;
```

Error en tiempo de ejecución  
porque *j* es *null*

```
i = j.HasValue ? j.Value : -1;
```

```
i = j != null ? (int)j : -1;
```

```
i = j ?? -1;
```

Las tres formas son correctas y equivalentes: Se asigna el valor de *j* sólo en caso de ser distinto de *null*, en otro caso se asigna -1. La forma más simple es utilizar el operador *??* llamado *operador null coaleasing*

## Usando tipos de referencias que admiten null

```
string st1 = "casa";
```

OK

```
string? st2 = st1;
```

```
st1 = st2;
```

Si el compilador no puede deducir que `st2` es distinto de `null` nos arroja un warning

```
st1 = st2 != null ? st2 : "default";
```

```
st1 = st2 ?? "default";
```

Ambas formas son correctas y hacen exactamente lo mismo., Se asigna el valor de `st2` sólo en caso de ser distinto de `null`, en otro caso se asigna "default".

## Operador is

- Con el operador `is` podemos consultar por el tipo de una expresión o contenido de una variable. Devuelve un bool.
- Semántica del operador `is` :

```
object o = 1;  
Console.WriteLine(o is int);  
Console.WriteLine(o is ValueType);  
Console.WriteLine(o is object);  
Console.WriteLine(o is string);  
Console.WriteLine(3 * 1.1 is double);
```

True

True

True

False

True



Con los tipos que admiten null se puede utilizar el operador de conversión “as”

- El operador `as`, al igual que una `expresión cast`, se utiliza para realizar una conversión explícita hacia un `tipo que admite null`.
- Cuando no se puede llevar a cabo la conversión el operador `as` devuelve `null` mientras que una `expresión cast` lanza una excepción.
- Si `E` es una expresión que devuelve un valor y `T` es el nombre de un tipo que admite null, entonces `E as T` produce el mismo resultado que:  
$$E \text{ is } T ? (T)(E) : \text{null}$$

Con los tipos que admiten null se puede utilizar el operador de conversión “as”

```
object obj = "casa";
```

```
string? st = (string)obj;
```

OK

```
obj = 12;
```

```
st = obj as string;
```

st recibe el valor null  
porque no se puede  
convertir un entero en un  
string

```
st = (string)obj;
```

Provoca error en tiempo de  
ejecución  
(InvalidCastException)





# Sistema de Tipos - Variables locales



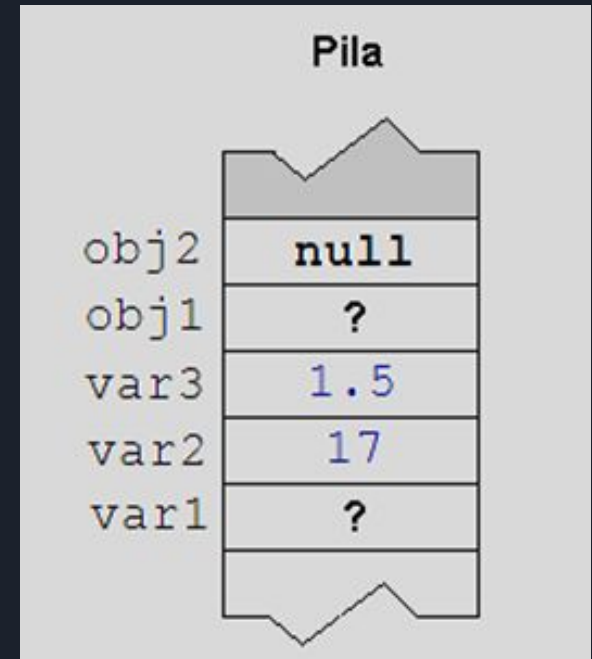
Las **variables locales** son las que se declaran dentro de los métodos.

Las variables que venimos utilizando son todas locales porque forman parte del método **Main** (el compilador genera este método a partir de las instrucciones de nivel superior)

## Sistema de Tipos - Variables locales

- Variables locales - Pila resultante

```
int var1;  
int var2 = 17;  
double var3 = 1.5;  
object obj1;  
object? obj2 = null;  
Console.WriteLine(var1);
```



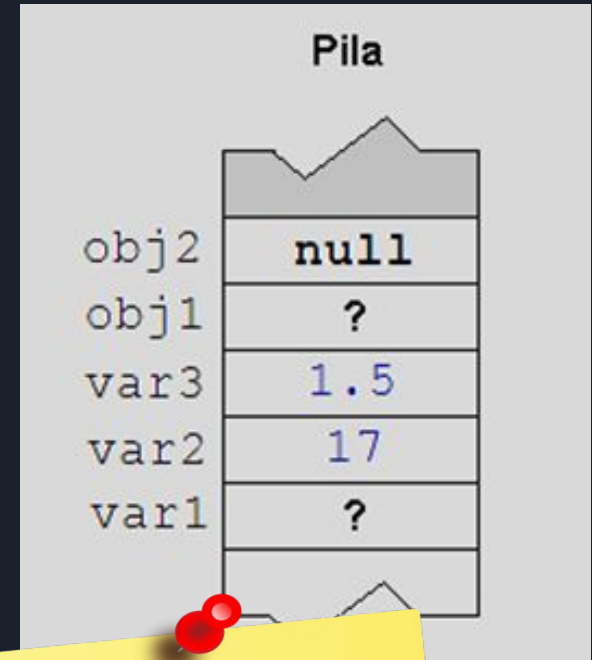
- Las **variables locales** sin inicializar poseen un valor indefinido. El compilador es capaz de determinar si existe un intento de lectura de una variable local aún no inicializada.

# Sistema de Tipos - Variables locales

- Variables locales - Pila resultante

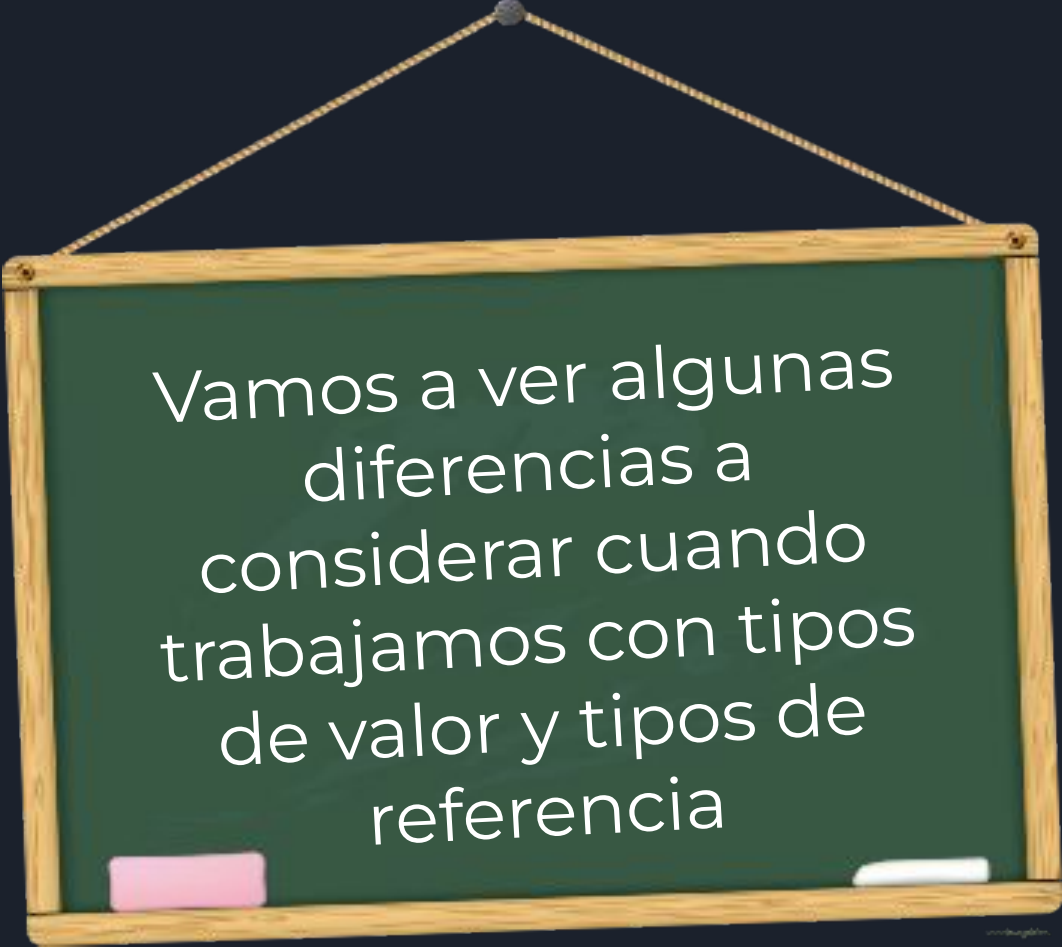
```
int var1;  
int var2 = 17;  
double var3 = 1.5;  
object obj1;  
object? obj2 = null;  
Console.WriteLine(var1);
```

Error de compilación  
Uso de la variable local no  
asignada 'var1'



El compilador le sigue la pista a las variables locales no permitiendo su lectura antes de su inicialización

## Diferencias entre tipos de valor y de referencia



Vamos a ver algunas  
diferencias a  
considerar cuando  
trabajamos con tipos  
de valor y tipos de  
referencia

### Codificar y ejecutar

```
char c1 = 'A';  
char c2 = 'A';  
Console.WriteLine(c1 == c2);
```



### Codificar y ejecutar

```
char c1 = 'A';  
char c2 = 'A';  
Console.WriteLine(c1 == c2);
```



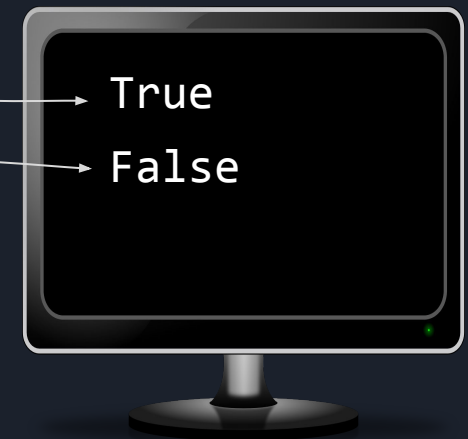
 Agregar las líneas resaltadas y volver y ejecutar

```
char c1 = 'A';  
char c2 = 'A';  
Console.WriteLine(c1 == c2);  
object o1 = 'A';  
object o2 = 'A';  
Console.WriteLine(o1 == o2);
```



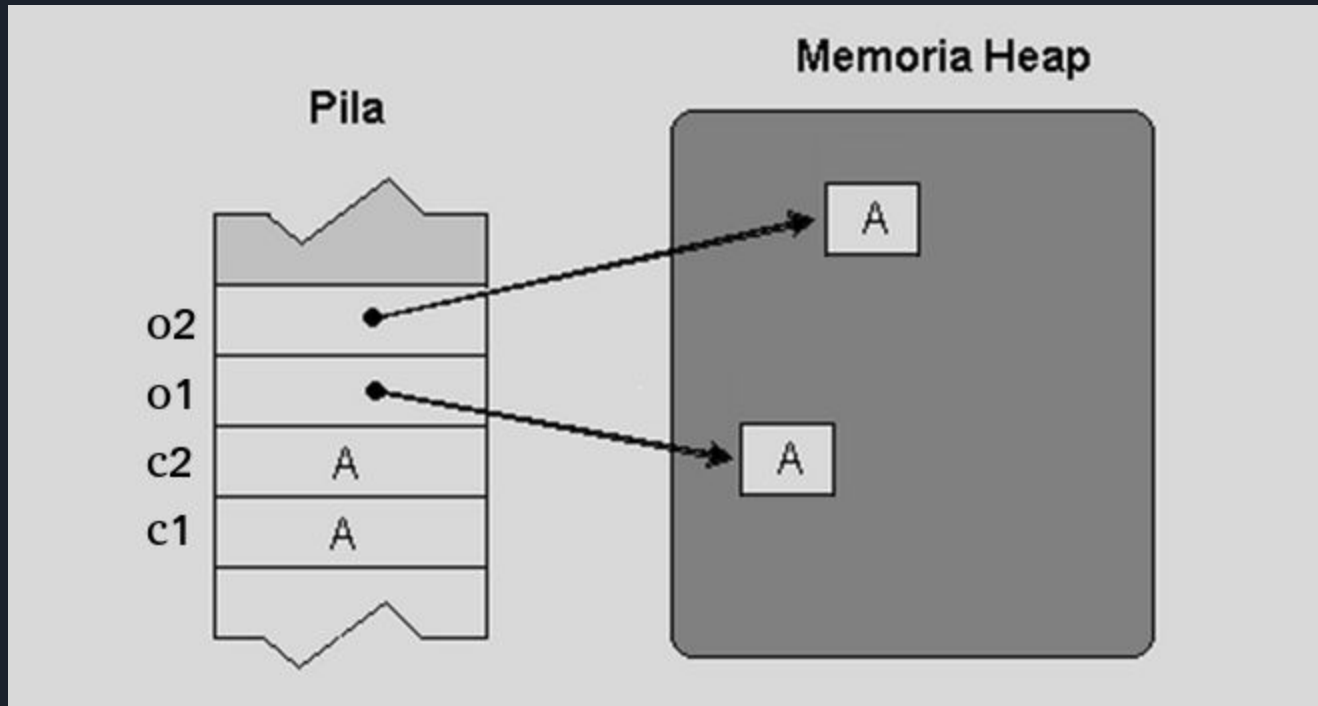
 Agregar las líneas resaltadas y volver y ejecutar

```
char c1 = 'A';  
char c2 = 'A';  
Console.WriteLine(c1 == c2);  
object o1 = 'A';  
object o2 = 'A';  
Console.WriteLine(o1 == o2);
```





## Sistema de Tipos – Diferencias entre tipos de valor y de referencia



La comparación por igualdad de `o1` y `o2` resulta falsa puesto que, por tratarse de tipos de referencia, no se compara el contenido sino las referencias



Modificar las líneas resaltadas y volver y ejecutar

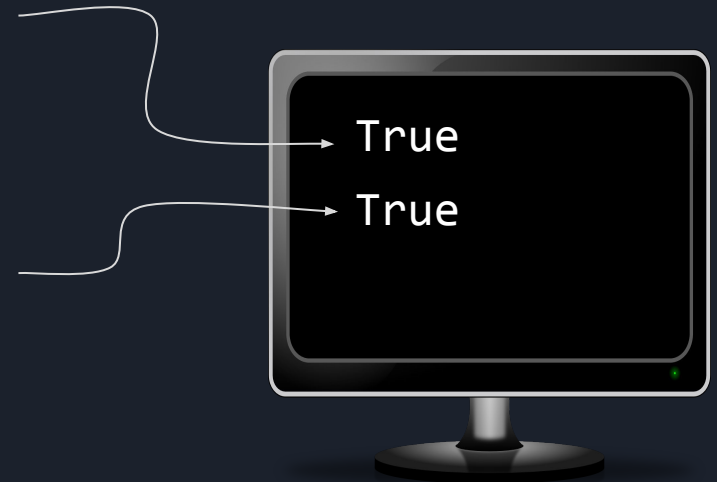
```
char c1 = 'A';  
char c2 = c1;  
Console.WriteLine(c1 == c2);  
object o1 = 'A';  
object o2 = o1;  
Console.WriteLine(o1 == o2);
```



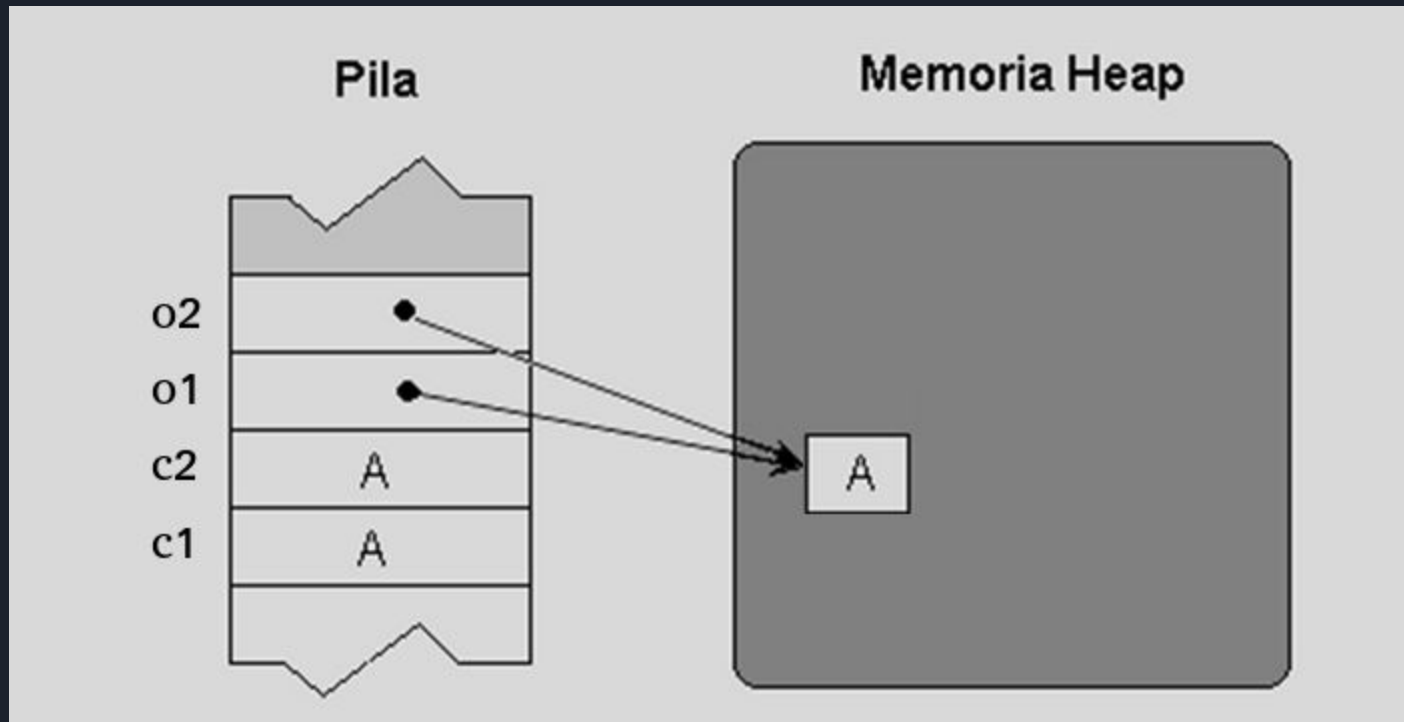


# Modificar las líneas resaltadas y volver y ejecutar

```
char c1 = 'A';  
char c2 = c1;  
Console.WriteLine(c1 == c2);  
object o1 = 'A';  
object o2 = o1;  
Console.WriteLine(o1 == o2);
```



## Sistema de Tipos – Diferencias entre tipos de valor y de referencia



La comparación por igualdad de `o1` y `o2` resulta verdadera puesto que ambas variables poseen la misma referencia (apuntan al mismo objeto)



# Sistema de Tipos - Arreglos

- Los arreglos son de **tipo de referencia**.
- Los arreglos pueden tener varias dimensiones (vector, matriz, tensor, etc.) el número de dimensiones se denomina **Rank**
- El número total de elementos de un arreglo se llama longitud del arreglo (**Length**)

## Sistema de Tipos - Arreglos

### Arreglos de una dimensión (vectores). Ejemplo

Declara un vector

```
int[] vector1;
```

Instancia un vector de 200  
elementos de tipo entero  
(se aloca en la heap)

```
vector1 = new int[200];
```

Declara e instancia un  
vector de 100 elementos  
enteros

```
int[] vector2 = new int[100];
```

```
int[] vector3 = new int[4] { 5, 1, 4, 0 };
```

Declara, instancia e inicializa un vector  
con 4 elementos enteros

## Sistema de Tipos - Arreglos

Cuando declaramos e inicializamos un vector lo podemos hacer de distintas formas:

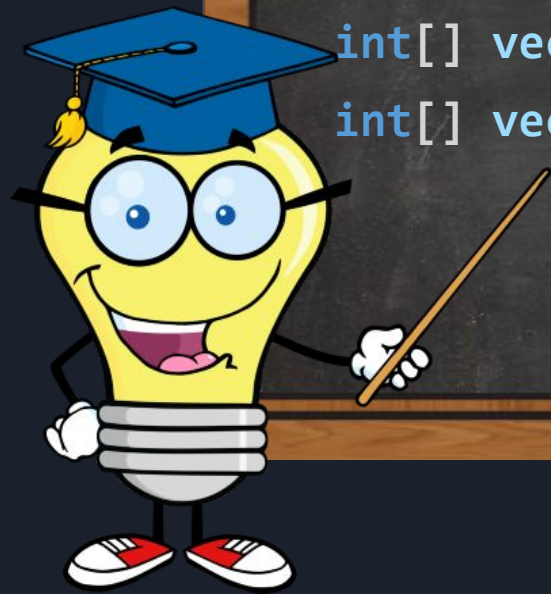
```
int[] vector1 = new int[3] { 5, 1, 4 };
```

```
int[] vector2 = new int[] { 5, 1, 4 };
```

```
int[] vector3 = { 5, 1, 4 };
```

```
int[] vector4 = [ 5, 1, 4 ];
```

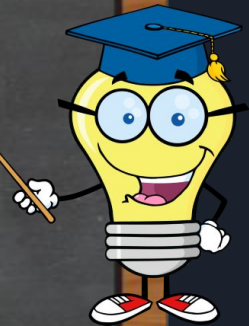
A partir de C# versión 12



## Sistema de Tipos - Arreglos

```
int[] vector4 = [ 5, 1, 4 ];
```

Expresión de colección



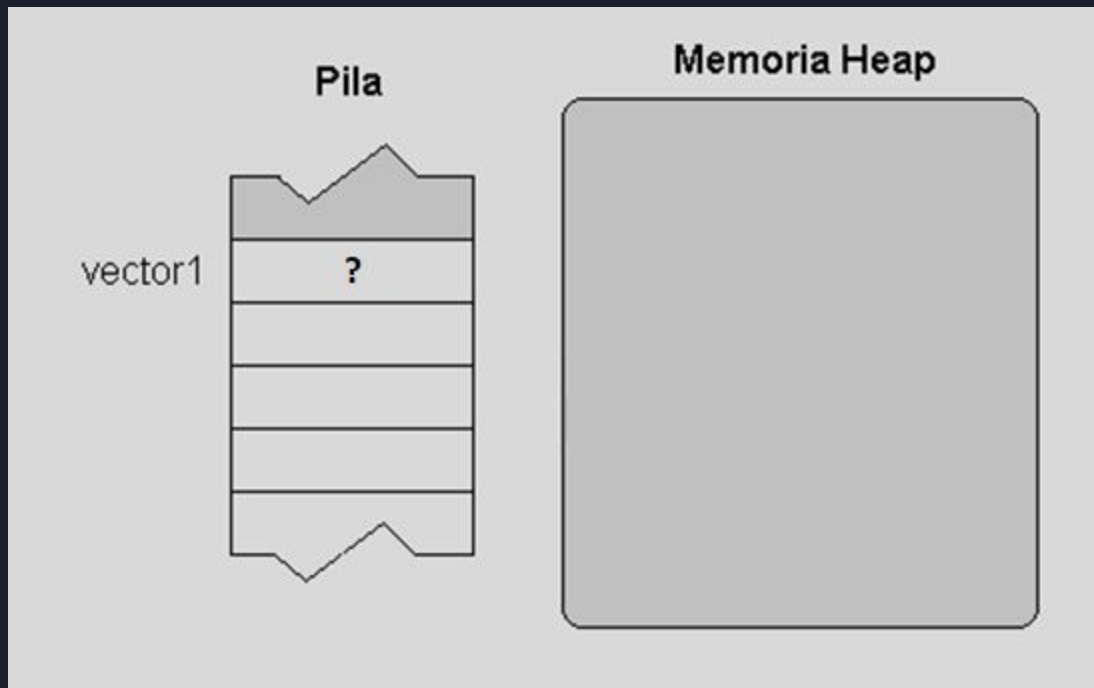
Las expresiones de colección fueron incorporadas en la versión 12 de C# con la liberación de .NET 8.

Se trata de una nueva característica que también puede utilizarse con otras colecciones de la plataforma.



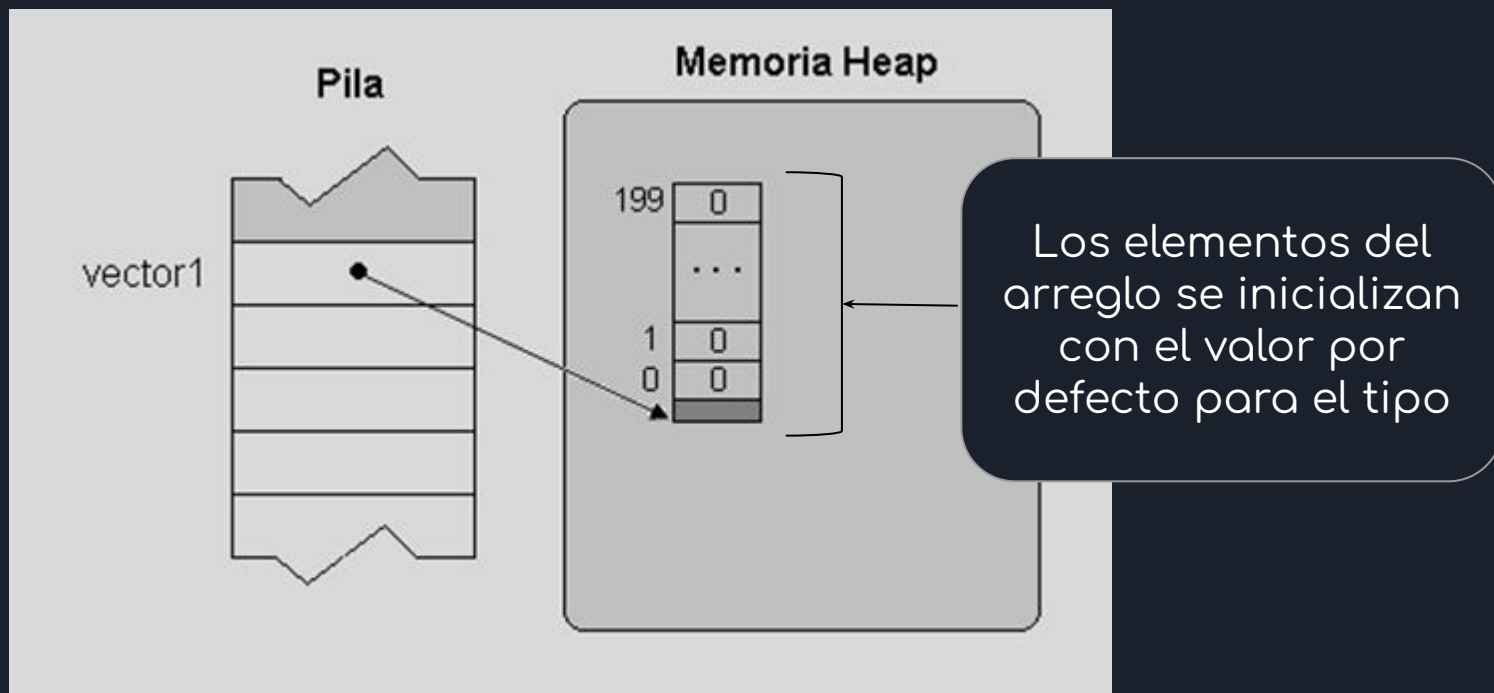
## Sistema de Tipos - Arreglos

```
int[] vector1;
```



## Sistema de Tipos - Arreglos

```
int[] vector1;  
vector1 = new int[200];
```





## Sistema de Tipos - Arreglos

Cuando se instancia un arreglo, el tamaño puede especificarse por medio de una variable

```
int tam = 5;  
char[] vocal = new char[tam];
```

Acceso a los elementos con operador [ ]

```
vocal[1] = 'E';
```

# Sistema de Tipos - Arreglos

El primer elemento ocupa la posición 0 (cero)

```
vocal[0] = 'A';
```

Último elemento:

```
vocal[vocal.Length - 1] = 'U';
```



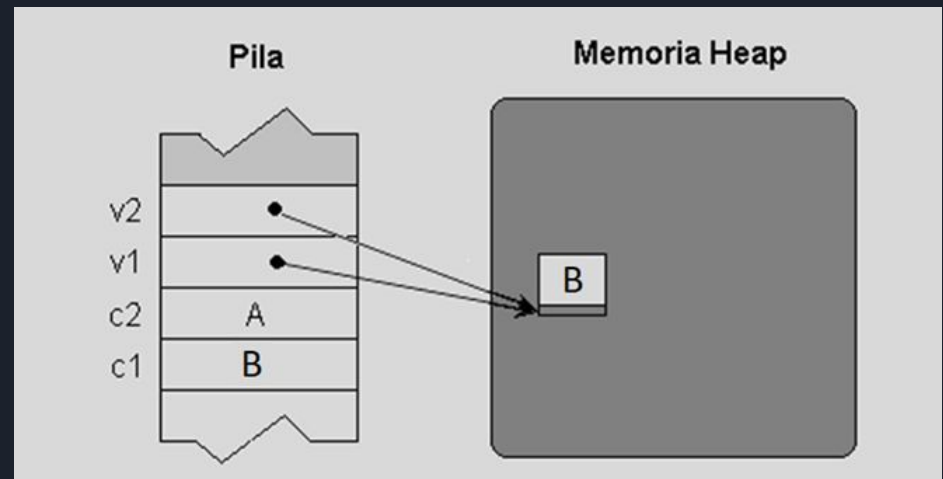
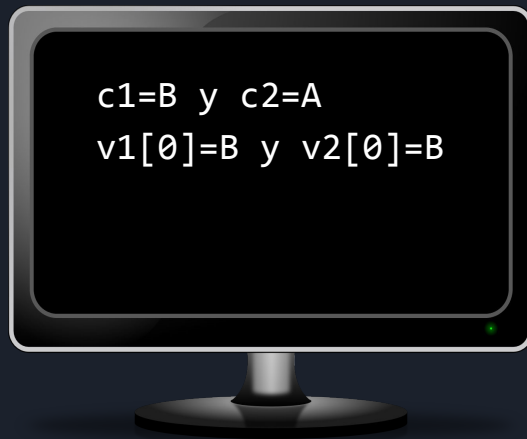
Responder sobre el siguiente código

```
char c1 = 'A';  
char c2 = c1; //se copia el valor 'A'  
c1 = 'B';  
Console.WriteLine("c1=" + c1 + " y c2=" + c2);  
  
char[] v1 = ['A'];  
char[] v2 = v1; //se copia la referencia  
v1[0] = 'B';  
Console.WriteLine("v1[0]=" + v1[0] + " y v2[0]=" + v2[0]);
```

¿Cuál es la salida  
por consola?



## Respuesta



Dado que `v1` y `v2` son en realidad el mismo objeto, el efecto de asignar `v1[0]` es el mismo de asignar `v2[0]`

## Instrucción foreach

Uso de `foreach`. Útil cuando se pretende recorrer el arreglo completo

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

```
foreach (string st in vector)
{
    Console.WriteLine(st);
}
```

tipo de los  
elementos

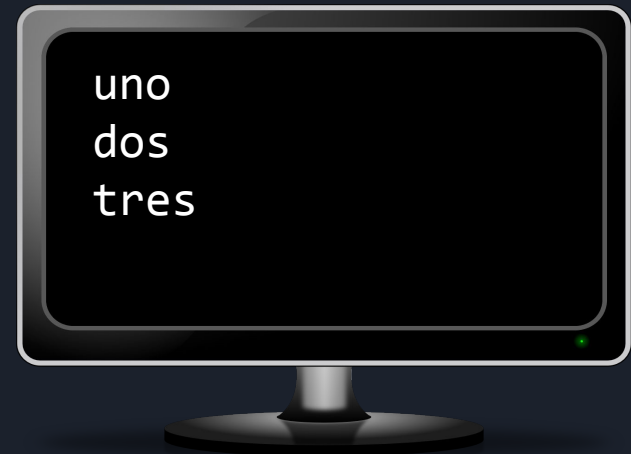
Variable que va  
a asignarse con  
cada uno de los  
elementos de la  
colección

Restricción: La variable de iteración  
no puede ser asignada en el cuerpo  
del foreach

## Instrucción foreach

Uso de `foreach`. Útil cuando se pretende recorrer el arreglo completo

```
string[] vector = ["uno", "dos", "tres"];  
  
foreach (string st in vector)  
{  
    Console.WriteLine(st);  
}
```





## Sistema de Tipos - La clase `String`

Secuencia de caracteres

```
string? st1 = "es un string";  
st1 = "";  
st1 = null;
```

Es un tipo de referencia

- 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

## Sistema de Tipos - La clase `String`

- Los string son de `inmutables` (no se pueden modificar caracteres individuales)
- Acceso a los elementos: `[]`
- Primer elemento: índice cero

```
string st = "Hola";
```

```
char c = st[0];
```

```
st[1]='O';
```

Válido. En la variable `c` queda asignado el char `'H'`

Error de compilación:  
los string son de sólo lectura

```
string st = "Hola";
```

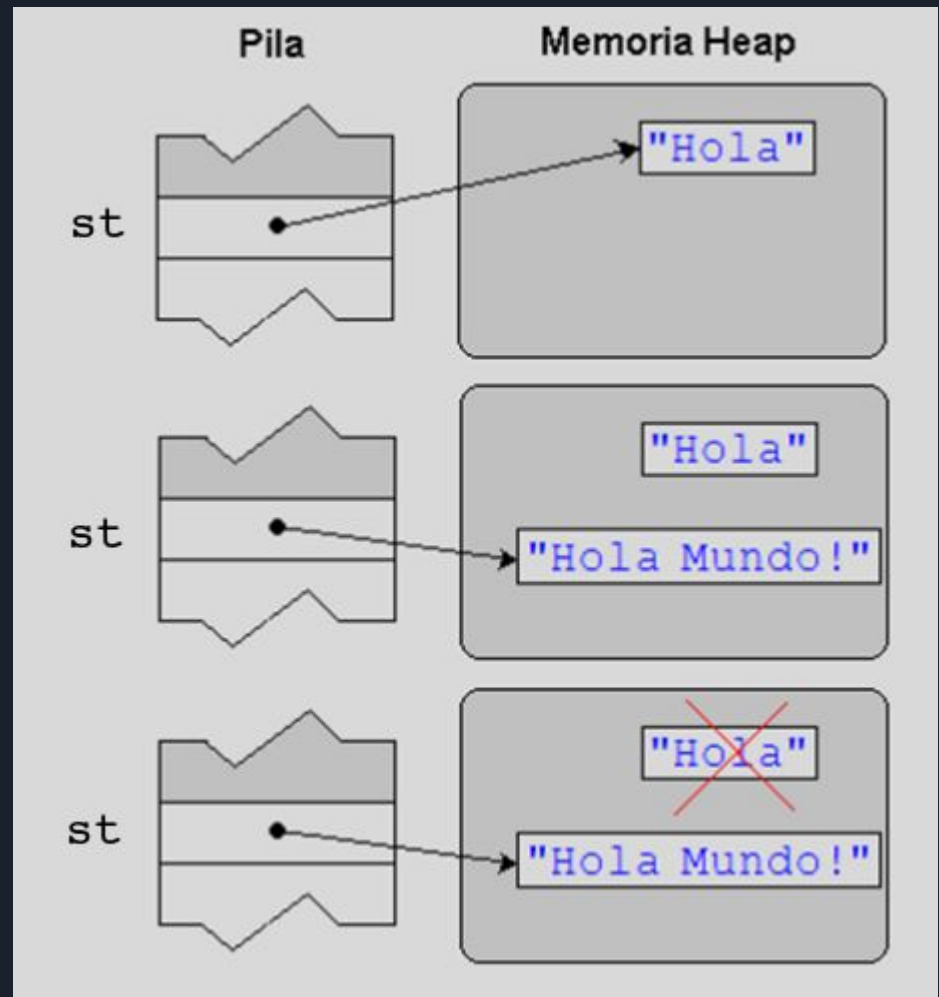
```
st = st + " Mundo!";
```

Correcto: se está creando  
un nuevo string

Sistema de Tipos - La clase `String`

```
string st = "Hola";
```

```
st = st + " Mundo!";
```



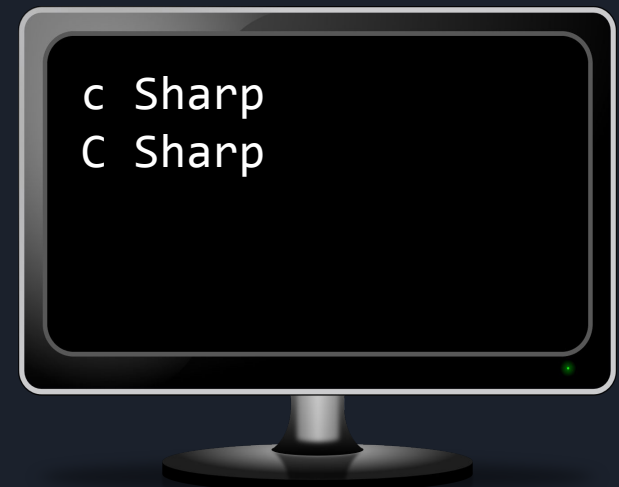
## Sistema de Tipos - La clase `StringBuilder`

- Símil a un `string` de lectura/escritura
- Definida en el espacio de nombre `System.Text`
- Métodos adicionales
  - `Append`
  - `Insert`
  - `Remove`
  - `Replace`
  - `etc.`

## Sistema de Tipos - La clase `StringBuilder` Ejemplo

```
using System.Text;

StringBuilder stb;
stb = new StringBuilder("c Sharp");
Console.WriteLine(stb);
stb[0] = 'C';
Console.WriteLine(stb);
```



## Tipos enumerativos

### Definición de enumeraciones

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

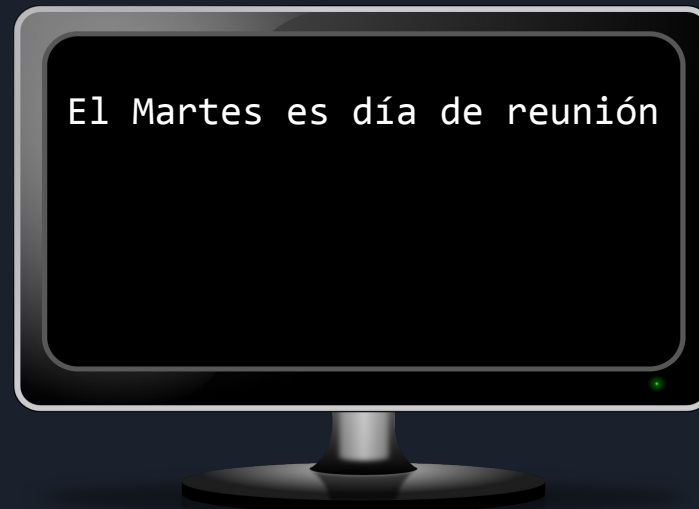
### Uso de enumeraciones

```
Tamaño t;
t = Tamaño.grande;
t = (Tamaño)0; //ahora t vale Tamaño.chico
```

## Tipos enumerativos - ejemplo

```
DiaDeSemana diaDeReunion = DiaDeSemana.Martes;  
for (DiaDeSemana d = DiaDeSemana.Lunes; d <= DiaDeSemana.Viernes; d++)  
{  
    if (d == diaDeReunion)  
    {  
        Console.WriteLine("El " + d + " es día de reunión");  
    }  
}
```

```
enum DiaDeSemana  
{  
    Domingo, Lunes, Martes,  
    Miércoles, Jueves,  
    Viernes, Sábado  
}
```



## Tipos enumerativos - ejemplo

Instrucciones de nivel superior

```
DiaDeSemana diaDeReunion = DiaDeSemana.Martes;  
for (DiaDeSemana d = DiaDeSemana.Lunes; d <= DiaDeSemana.Viernes; d++)  
{  
    if (d == diaDeReunion)  
    {  
        Console.WriteLine("El " + d + " es día de reunión");  
    }  
}
```

Declaración de tipo

```
enum DiaDeSemana  
{  
    Domingo, Lunes, Martes,  
    Miércoles, Jueves,  
    Viernes, Sábado  
}
```

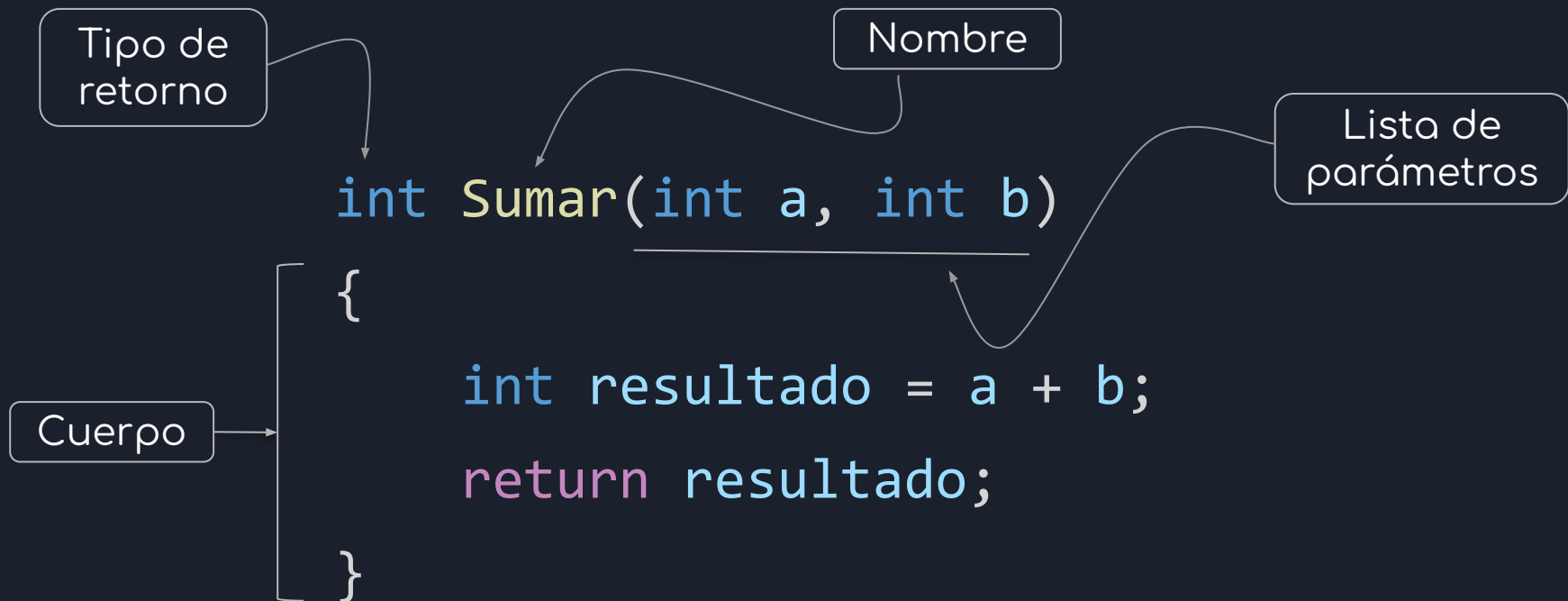
Las instrucciones de nivel superior deben preceder a las declaraciones de tipos  
Sin embargo es aconsejable dejar en el archivo Program.cs sólo instrucciones de nivel superior y declarar cada tipo en su propio archivo fuente



# Métodos

# Métodos

**Método:** Bloque con nombre de código ejecutable. Puede invocarse desde diferentes partes del programa, e incluso desde otros programas



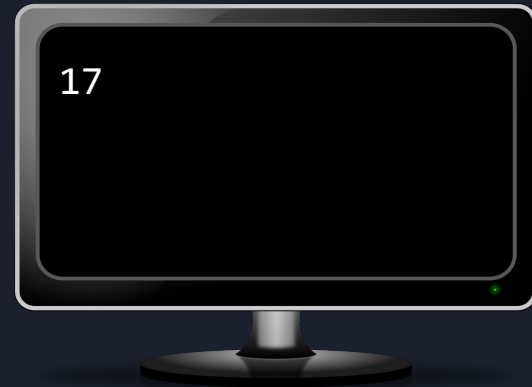
# Métodos

```
int valor = 15;  
int resultado = Sumar(valor, 2);  
Console.WriteLine(resultado);
```

```
int Sumar(int a, int b)  
{  
    int resultado = a + b;  
    return resultado;  
}
```



¿Cuál es la salida  
por consola?



# Métodos

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

Tipo de  
retorno

```
void Imprimir(string st)
{
    Console.WriteLine(st);
    return;
}
```

Se puede omitir  
porque el tipo  
de retorno es  
void

## Pasaje de parámetros

**Pasaje de parámetros por valor:** Recibe una copia del valor pasado como parámetro.

Ejemplo:

```
int entero = 10;  
Imprimir(entero);  
Console.WriteLine(entero);
```

```
void Imprimir(int n)  
{  
    n = 100;  
    Console.WriteLine(n);  
}
```

Parámetro de  
entrada



¿Cuál es la salida  
por consola?



## Pasaje de parámetros

### Pasaje de parámetros por referencia (`ref`)

```
int sumaAcumulada = 10;  
SumarA(ref sumaAcumulada, 5);  
Console.WriteLine(sumaAcumulada);
```

```
void SumarA(ref int acum, int valor)  
{  
    Console.WriteLine("Se sumará " + valor + " a " + acum);  
    acum += valor;  
}
```

- En la invocación también se debe utilizar `ref`
- la variable pasada por `ref` debe estar inicializada

Es seguro leer el parámetro `acum` porque el invocador está obligado a inicializarlo

## Pasaje de parámetros

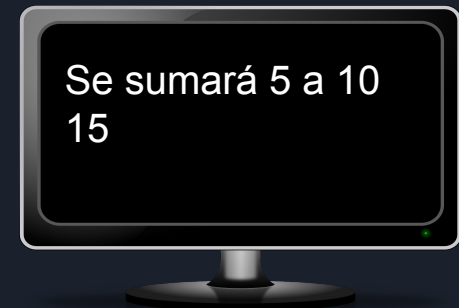
### Pasaje de parámetros por referencia (`ref`)

```
int sumaAcumulada = 10;  
SumarA(ref sumaAcumulada, 5);  
Console.WriteLine(sumaAcumulada);
```

```
void SumarA(ref int acum, int valor)  
{  
    Console.WriteLine("Se sumará " + valor + " a " + acum);  
    acum += valor;  
}
```



¿Cuál es la salida  
por consola?



# Pasaje de parámetros

## Pasaje de parámetros de salida (out)

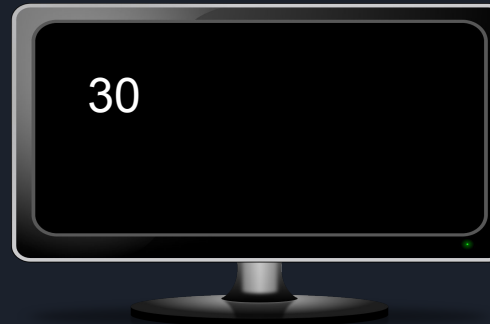
Ejemplo:

```
int r;  
Sumar(10, 20, out r);  
Console.WriteLine(r);
```

- La variable `r` puede estar sin inicializar
- En la invocación también se debe utilizar `out`

```
void Sumar(int a, int b, out int resultado)  
{  
    resultado = a + b;  
}
```

El método invocado es el responsable de establecer el valor del parámetro de salida

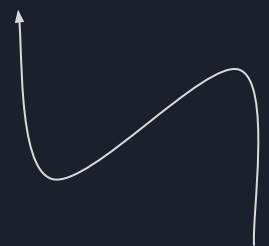




## Pasaje de parámetros

### Pasaje de parámetros de salida (out)

```
void Sumar(int a, int b, out int resultado)
{
    Console.WriteLine(resultado);
    resultado = a + b;
}
```

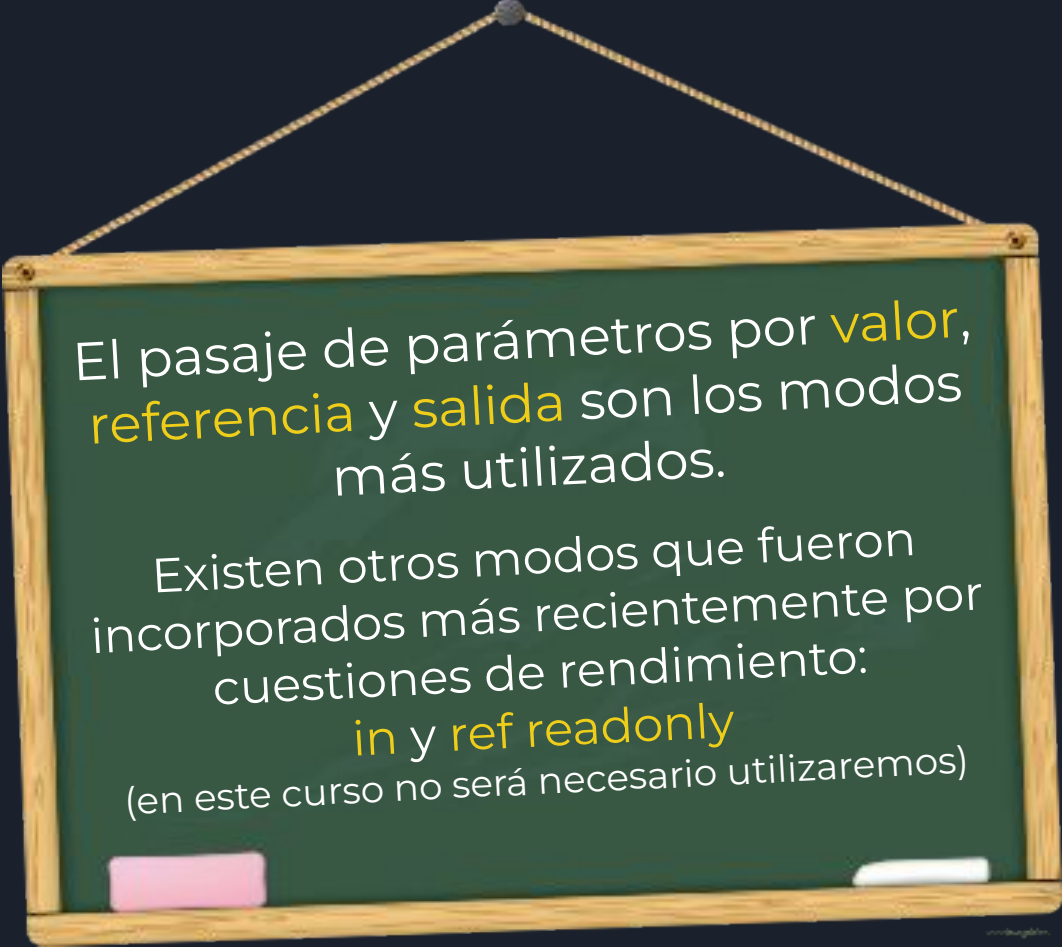


#### ERROR DE COMPILACIÓN

Uso del parámetro out sin asignar

El método invocado no puede leer un parámetro **out** antes de establecerle su valor, que además está obligado a hacerlo

# Pasaje de parámetros



El pasaje de parámetros por **valor**, **referencia** y **salida** son los modos más utilizados.

Existen otros modos que fueron incorporados más recientemente por cuestiones de rendimiento:

**in** y **ref readonly**  
(en este curso no será necesario utilizaremos)



## Pasaje de parámetros

Pasaje de parámetros de entrada (*in*): El parámetro se pasa por referencia pero no puede modificarse dentro del método invocado

```
int n = 10;  
Imprimir(in n);
```

El valor de *a* no puede modificarse, es de sólo lectura

```
static void Imprimir(in int a)  
{  
    Console.WriteLine(a);  
}
```

Nota:  
Útil por razones de rendimiento. Se pasa un parámetro por referencia para evitar la copia pero asegurando la no modificación del valor en el método invocado



## Pasaje de parámetros

Parámetros `ref readonly`: Con la misma intención que los parámetros de entrada, en `C#12` se introdujo el modificar `ref readonly`

```
int n = 10;  
Imprimir(in n);  
Imprimir(ref n);
```

Se puede invocar con  
`ref` o con `in`

```
static void Imprimir(ref readonly int a)  
{  
    Console.WriteLine(a);  
}
```

El valor de `a` no puede modificarse,  
es de `sólo lectura`



## Pasaje de parámetros

En realidad se admiten varias formas de invocar a métodos con parámetros **in** y **ref readonly**. Cambia los warnings del compilador

```
int n = 10;  
Imprimir(33);  
Imprimir(n);  
Imprimir(in n);  
Imprimir(ref n);
```

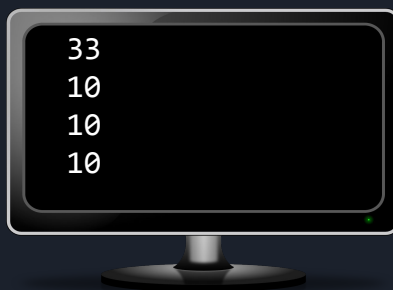
**in**

```
static void Imprimir(in int a)  
{  
    Console.WriteLine(a);  
}
```

```
int n = 10;  
Imprimir(33);  
Imprimir(n);  
Imprimir(in n);  
Imprimir(ref n);
```

**ref readonly**

```
static void Imprimir(ref readonly int a)  
{  
    Console.WriteLine(a);  
}
```



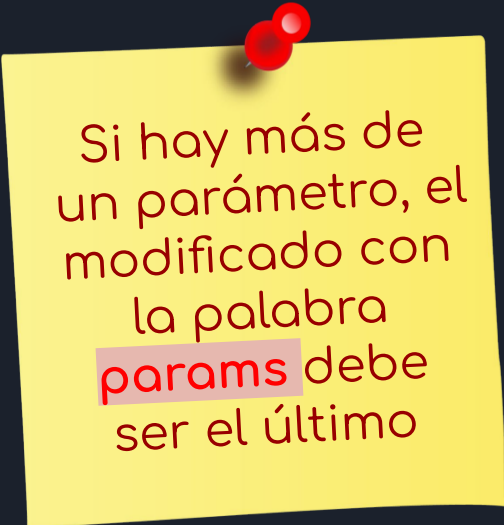
## Pasaje de parámetros

Uso de la palabra clave `params` :

Permite que un método tome un número variable de argumentos. El tipo declarado del parámetro `params` debe ser una arreglo unidimensional

Ejemplo:

```
void Imprimir(params int[] vector)
{
    foreach (int i in vector)
    {
        Console.Write(i + " ");
    }
    Console.WriteLine("Ok");
}
```



Si hay más de un parámetro, el modificado con la palabra `params` debe ser el último

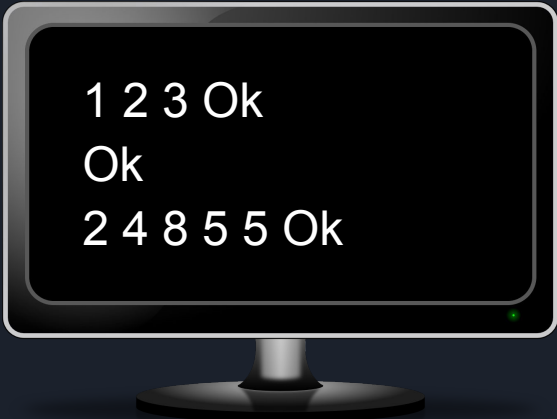
## Pasaje de parámetros

Uso de la palabra clave params :

```
int[] vector = { 1, 2, 3 };  
Imprimir(vector);  
Imprimir();  
Imprimir(2, 4, 8, 5, 5);
```

Se puede pasar un vector  
de enteros o una lista de  
ceros o más enteros

```
static void Imprimir(params int[] vector)  
{  
    foreach (int i in vector)  
    {  
        Console.Write(i + " ");  
    }  
    Console.WriteLine("Ok");  
}
```



1 2 3 Ok  
Ok  
2 4 8 5 5 Ok

### Métodos con forma de expresión (*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);
}
```

Puede escribirse como:

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



### Métodos con forma de expresión (*expression-bodied methods*)

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

#### Ejemplo

```
int Suma(int a, int b)
{
    return a + b;
}
```

Observar que no va la  
sentencia `return`

Puede escribirse como:

```
int Suma(int a, int b) => a + b;
```

# Pasaje de parámetros por la línea de comandos

Recordemos que al utilizar instrucciones de nivel superior, en realidad estamos codificando el cuerpo del método **Main** por el que comienza la ejecución de nuestra aplicación



# Pasaje de parámetros por la línea de comandos

Program.cs

```
Console.WriteLine("OK");
```

Es equivalente

Program.cs

```
class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine("Ok");
    }
}
```



# Pasaje de parámetros por la línea de comandos

Program.cs

```
Console.WriteLine("OK");
```

Es equivalente

Program.cs

```
class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine("Ok");
    }
}
```

En el vector args  
recibimos los  
parámetros  
pasados a la  
aplicación por la  
línea de comandos  
pasados

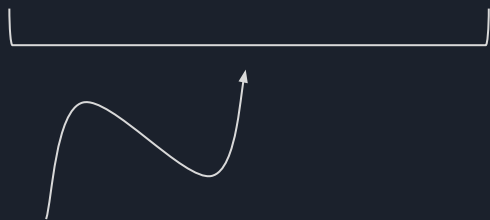


Observar el siguiente código:

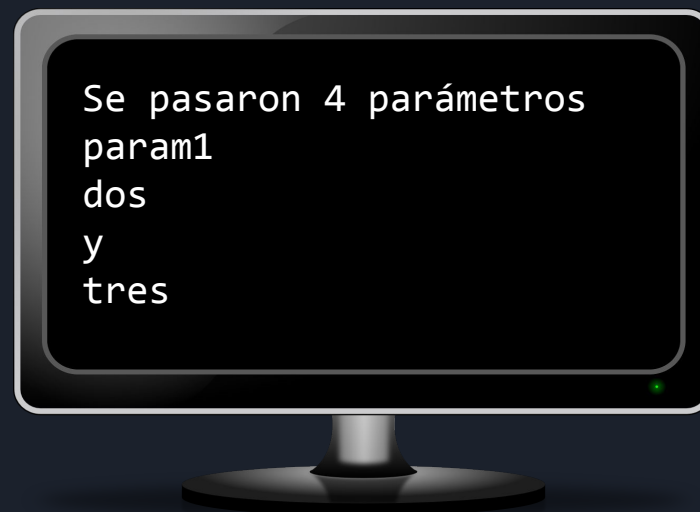
```
Console.WriteLine("Se pasaron " + args.Length + " parámetros");  
foreach (string st in args)  
{  
    Console.WriteLine(st);  
}
```

Podemos compilar, ejecutar y pasar parámetros de la siguiente manera:

```
dotnet run param1 dos y tres
```



Parámetros pasados a la  
aplicación por la línea de  
comandos

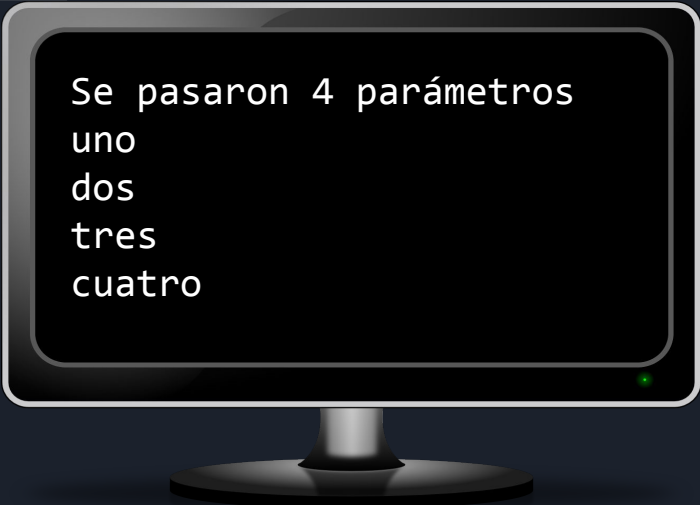


Una vez compilado, desde una terminal puede invocarse directamente el ejecutable generado

```
./bin/Debug/net8.0/Teoria2 uno dos tres cuatro
```

Ejecutable generado  
durante la compilación

Parámetros pasados a la aplicación por  
la línea de comandos

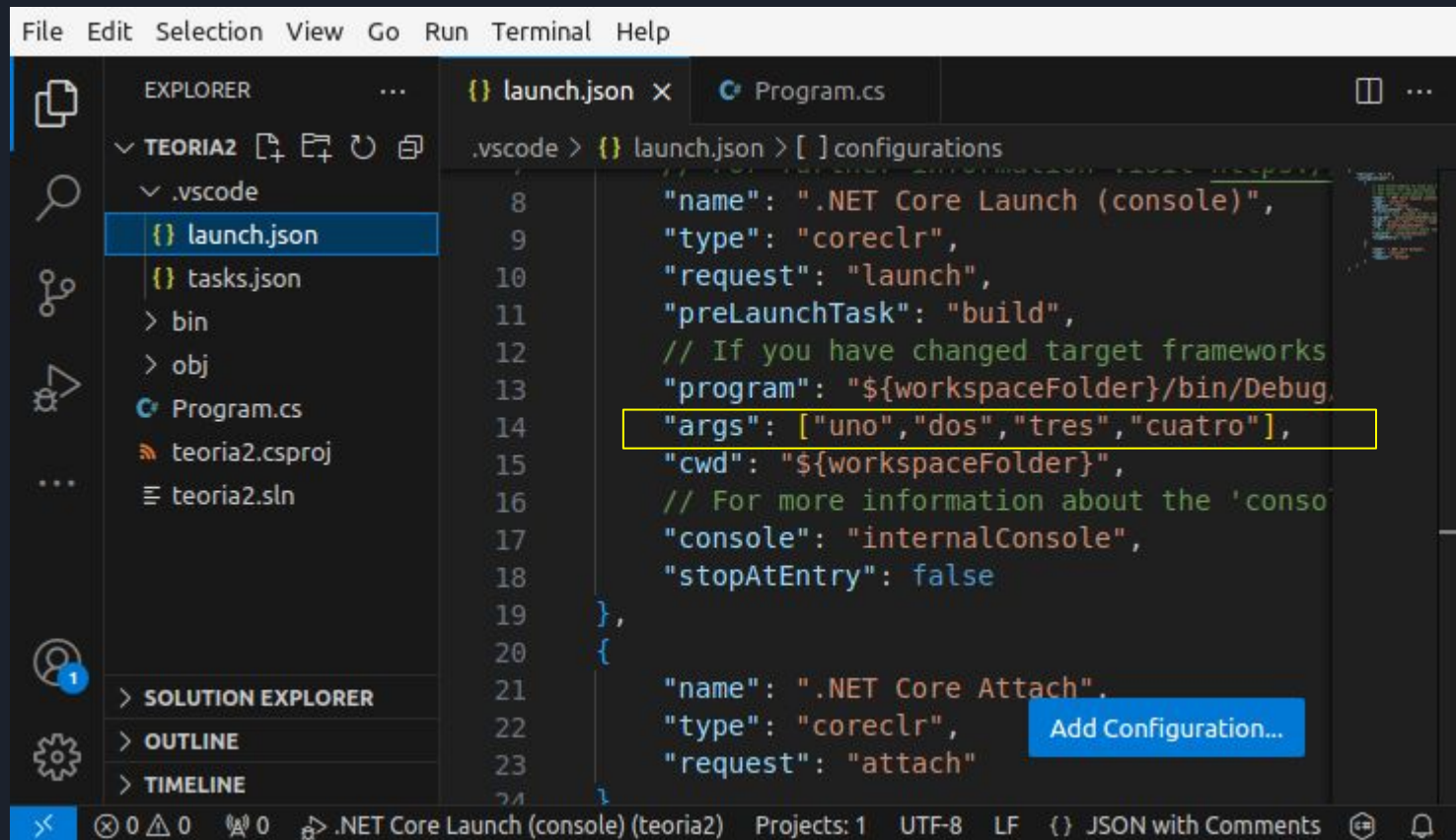


Se pasaron 4 parámetros  
uno  
dos  
tres  
cuatro



# Pasaje de parámetros por la línea de comandos

Para facilitar la compilación y ejecución con argumentos pasados por la línea de comandos, Visual Studio Code permite definirlos en el archivo `launch.json`



The screenshot shows the Visual Studio Code interface. In the Explorer on the left, the `launch.json` file is selected under the `.vscode` folder. The main editor displays the contents of `launch.json`, which is a JSON configuration for debugging. The `args` property is highlighted with a yellow box, showing an array of strings: `["uno", "dos", "tres", "cuatro"]`. A blue button labeled "Add Configuration..." is visible in the bottom right corner of the editor. The Run and Debug console at the bottom shows the command `.NET Core Launch (console) (teoria2)` and the status "Projects: 1 UTF-8 LF {} JSON with Comments".

```
File Edit Selection View Go Run Terminal Help

EXPLORER
TEORIA2
  .vscode
    {} launch.json
    {} tasks.json
  bin
  obj
  Program.cs
  teoria2.csproj
  teoria2.sln

SOLUTION EXPLORER
OUTLINE
TIMELINE

{} launch.json x Program.cs

.vscode > {} launch.json > [ ] configurations
// For more information about the 'conso
"args": ["uno", "dos", "tres", "cuatro"],
"cwd": "${workspaceFolder}",
"console": "internalConsole",
"stopAtEntry": false
},
{
  "name": ".NET Core Attach",
  "type": "coreclr",
  "request": "attach"
}
```

Fin de la teoría 2



# Práctica sobre la teoría 2

1) 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**”?

2) 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;
```

3) ¿Qué diferencias existen entre las conversiones de tipo implícitas y explícitas?

4) Resolver los errores de compilación en el siguiente fragmento de código. Utilizar el operador **as** cuando sea posible.

```
object o = "Hola Mundo!";  
string st = o;  
int i = 12;  
byte b = i;  
double d = i;  
float f = d;  
o = i;  
i = o + 1;
```

5) 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;
```

6) Supongamos que Program.cs sólo tiene las siguientes dos líneas:

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

¿Por qué no compila?

7) ¿Cuál es la salida por consola del siguiente fragmento de código? ¿Por qué la tercera y sexta línea producen resultados diferentes?

```
char c1 = 'A';  
char c2 = 'A';  
Console.WriteLine(c1 == c2);  
object o1 = c1;  
object o2 = c2;  
Console.WriteLine(o1 == o2);
```

8) 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.

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

10) 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
```

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.

12) Definir el tipo de datos enumerativo llamado **Meses** y utilizarlo para:

- Imprimir en la consola el nombre de cada uno de los meses en orden inverso (diciembre, noviembre, octubre ..., enero)
- 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**

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

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

14) ¿Qué hace la instrucción? ¿Asigna a la variable vector el valor **null**?

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

15) 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]);
```

16) Escribir un programa que reciba una lista de nombres como parámetro por la línea de comandos e imprima por consola un saludo personalizado para cada uno de ellos.

- a) Utilizando la sentencia **for**
- b) Utilizando la sentencia **foreach**

17) 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**)

18) 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)

19) Idem. al ejercicio 18.a) y 18.b) pero devolviendo el resultado en un parámetro de salida **void Fac(int n, out int f)**

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

21) 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("-----");
```



```
1 casa A 3,4 Saturday  
1 2 tres  
-----
```