



.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

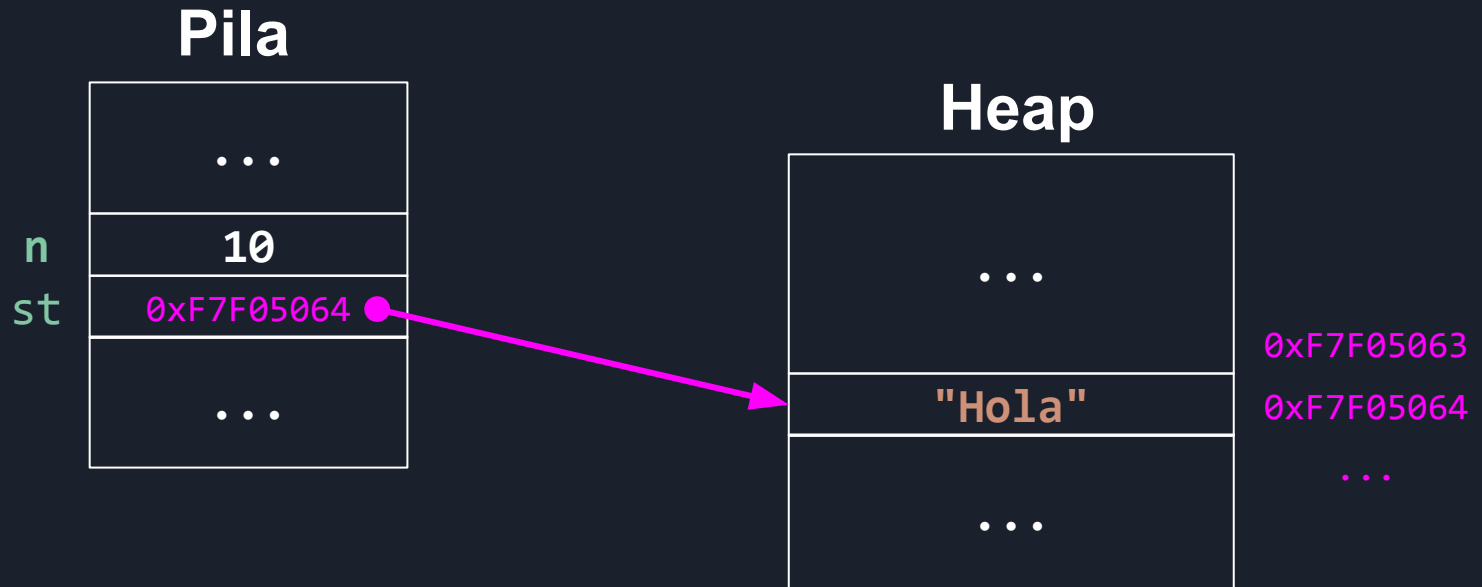
Sistema de tipos

- **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

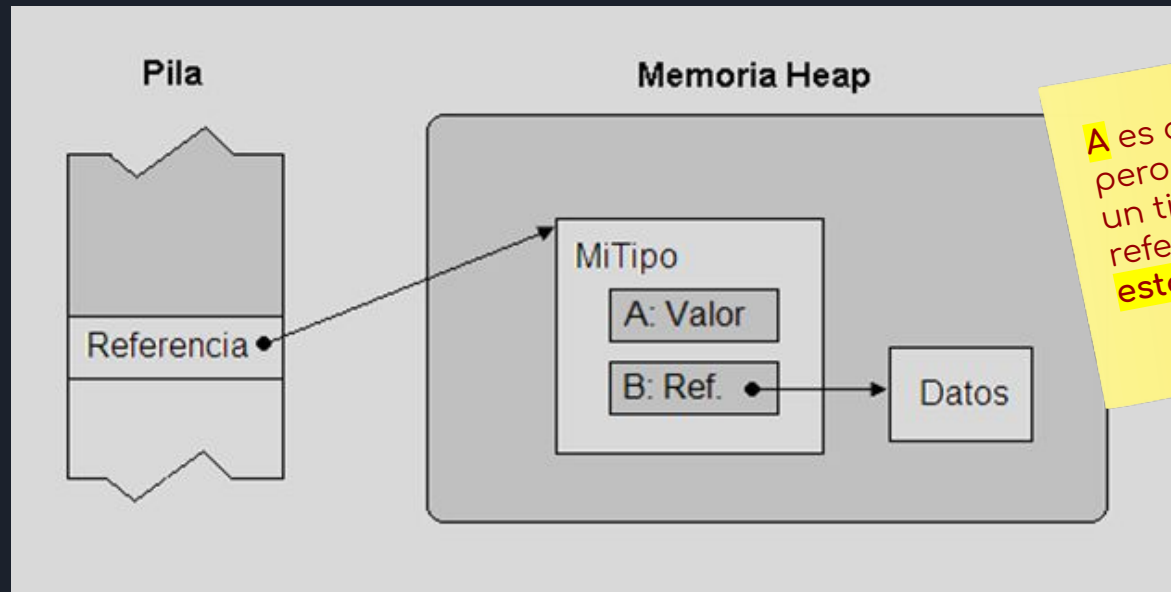
CÓDIGO

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



Sistema de tipos

- La porción de datos de un tipo de referencia siempre se guarda en la memoria heap
- Los datos de un tipo de valor pueden guardarse en la pila o en la memoria heap dependiendo de las circunstancias



A es de tipo de valor pero forma parte de un tipo que es referencia, por eso está en la Heap

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`

- `System.DateTime`

- Tipos numéricos

- Tipos enteros (`sbyte`, `byte`, `short`, `int` ...)


- Tipos de punto flotantes (`float` y `double`)

- `decimal`


- Estructuras definidas por el usuario

- Enumeraciones

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



No es un tipo integrado pero es muy utilizado



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 - 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`)
- 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 “`nullables value types`” -ver más adelante en este documento-).

Sistema de Tipos - el valor `null`

Ejemplo asignación valor `null`

```
int i;
```

```
object obj;
```

```
obj = null;
```

Válido

```
i = null;
```

Inválido
(error de compilación)

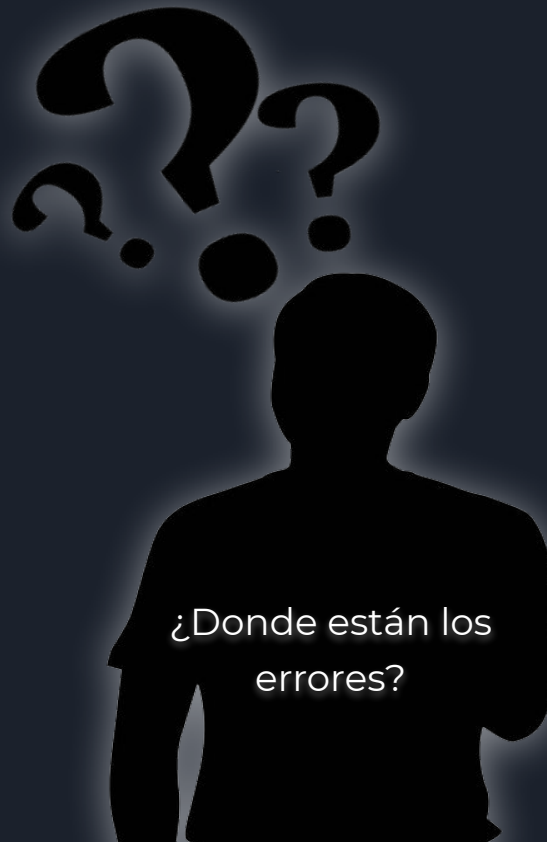
Sistema de Tipos - Conversión de tipos

- 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

```
public static void Main(string[] args)
{
    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

```
public static void Main(string[] args)
{
    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

```
public static void Main(string[] args)
{
    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 implícitas (de ampliación)

Desde	Hacia
sbyte	short, int, long, float, double, decimal
byte	short, ushort, int, uint, long, ulong, float, double, decimal
short	int, long, float, double, decimal
ushort	int, uint, long, ulong, float, double, decimal
int	long, float, double, decimal
uint	long, ulong, float, double, decimal
long	float, double, decimal
ulong	float, double, decimal
char	ushort, int, uint, long, ulong, float, double, decimal
float	double

Conversiones explícitas (de restricción)

Desde	Hacia
sbyte	byte, ushort, uint, ulong, char
byte	sbyte, char
short	sbyte, byte, ushort, uint, ulong, char
ushort	sbyte, byte, short, char
int	sbyte, byte, short, ushort, uint, ulong, char
uint	sbyte, byte, short, ushort, int, char
long	sbyte, byte, short, ushort, int, uint, ulong, char
ulong	sbyte, byte, short, ushort, int, uint, long, char
char	sbyte, byte, short
float	sbyte, byte, short, ushort, int, uint, long, ulong, char, decimal
double	sbyte, byte, short, ushort, int, uint, long, ulong, char, float, decimal
decimal	sbyte, byte, short, ushort, int, uint, long, ulong, char, float, double

Puede haber
pérdida de
información o
incluso
excepciones

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 (no hay dato)
- Para declarar un tipo **nullable** se debe colocar el símbolo **?** después del tipo correspondiente

```
int x1 = null;
```

ERROR

```
int? x2 = null;
```

Válido
x2 es un entero
nullable

Tipos nullables

```
int x1 = 1; int? x2 = null;
```

```
int? x3 = x1;
```

Conversión Implícita

```
int x4 = (int)x3;
```

Es necesario "castear"
Conversión explícita

```
int x5 = (int)x2;
```

Error en tiempo de ejecución porque
x2 es *null*

```
int x6 = x2.HasValue ? x2.Value : -1;
```

```
int x7 = x3 != null ? (int)x3 : -1;
```

Ambas formas son correctas: Primero consultar si la variable está asignada con un valor válido, en dicho caso acceder a ese valor

Conversiones de tipo explícitas operadores “(...)” “as”

- 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**

Conversiones de tipo explícitas operadores “(...)” “as”

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

Las variables de tipo `object` admiten valores de cualquier tipo (lo veremos en detalle más adelante en esta teoría)

```
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`)

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)

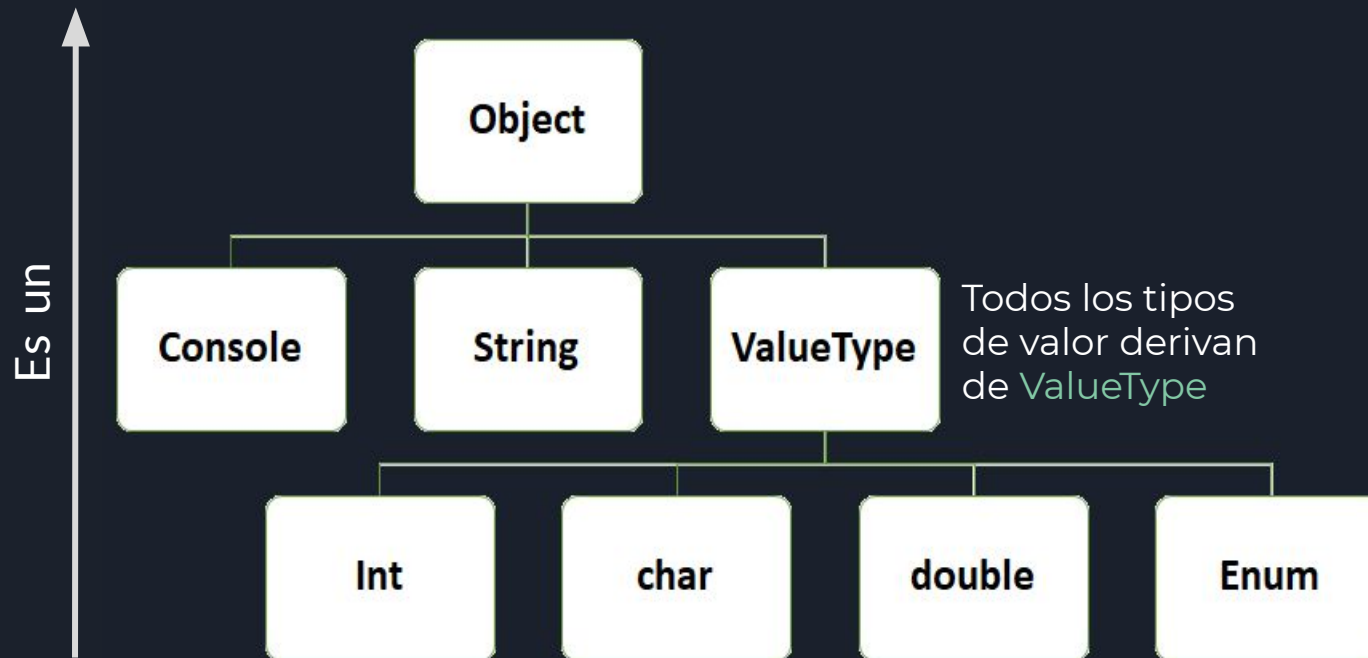
Error de compilación

OK
Se asigna 1 a i

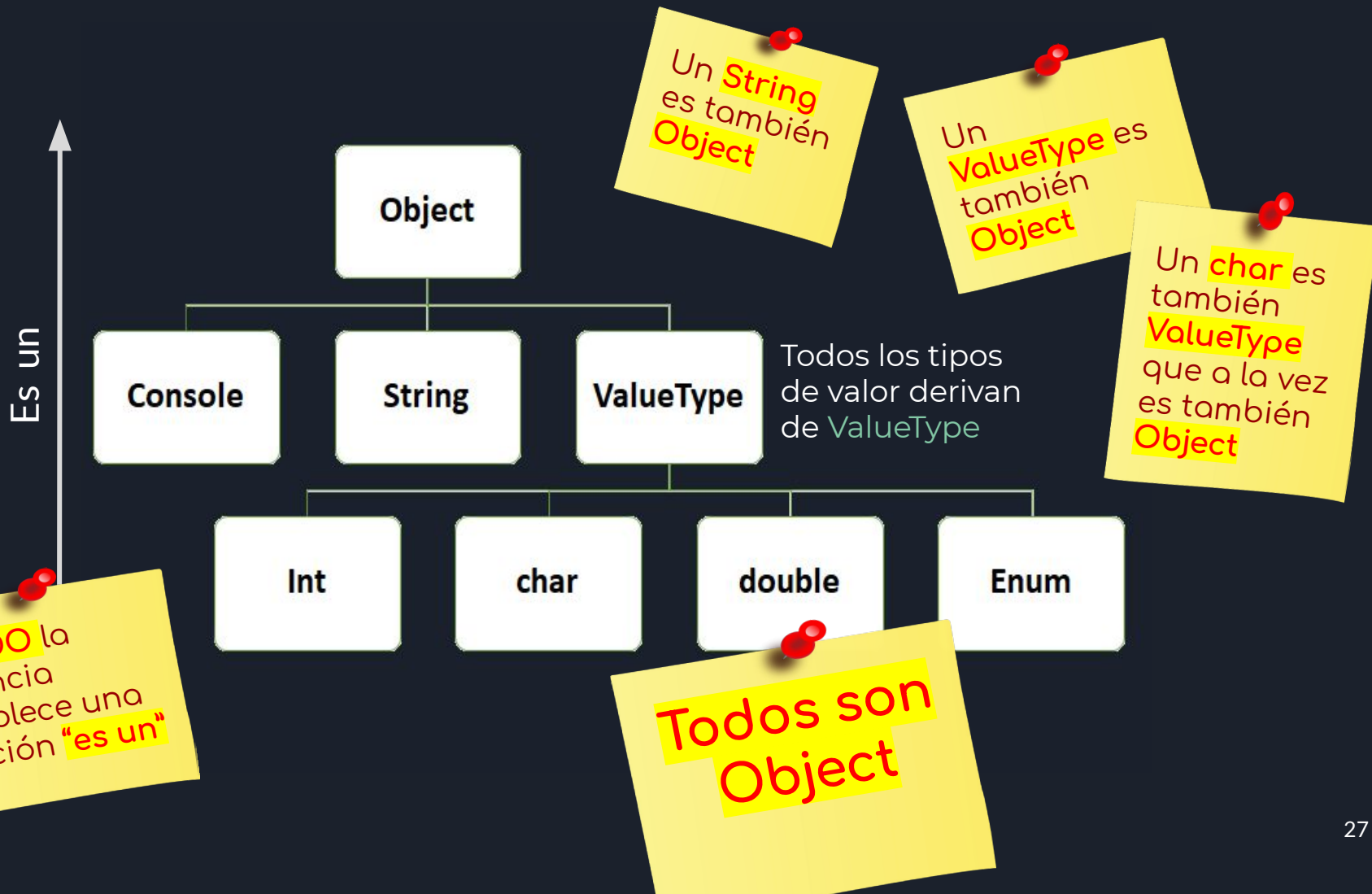
Sistema unificado de tipos

- 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)

Sistema unificado de tipos



Sistema unificado de tipos

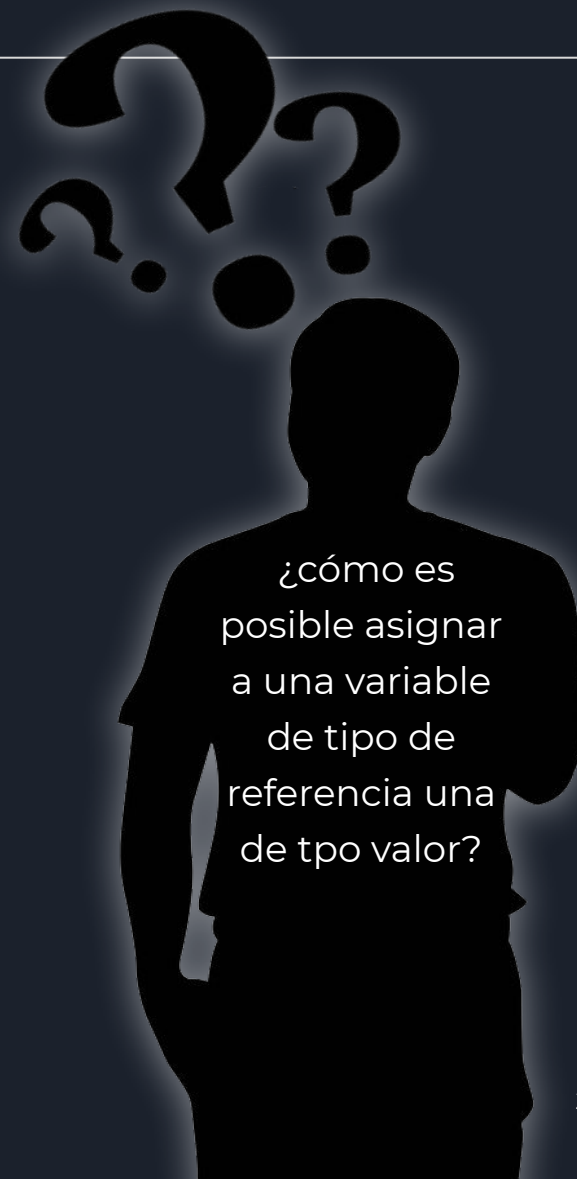


Consecuencia de tener un Sistema unificado de tipos

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;
```

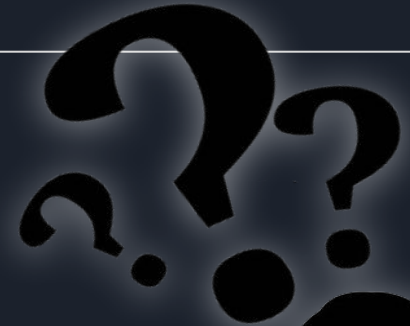


¿cómo es posible asignar a una variable de tipo de referencia una de tpo valor?

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;
```

boxing

```
o = i;
```

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

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

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

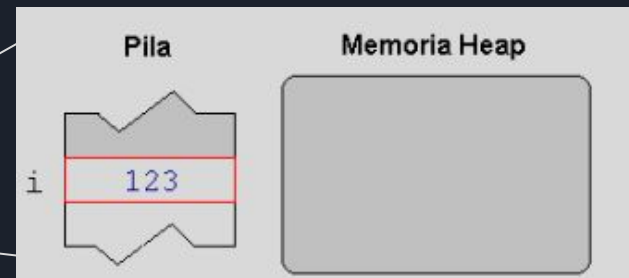
```
object o;  
int i = 123;
```

. . .

```
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*

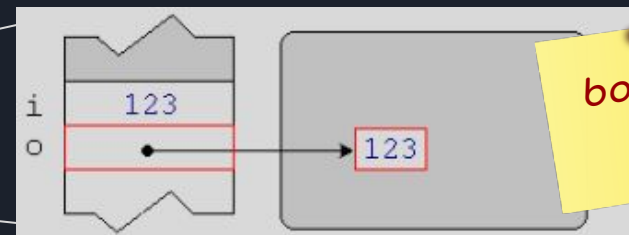
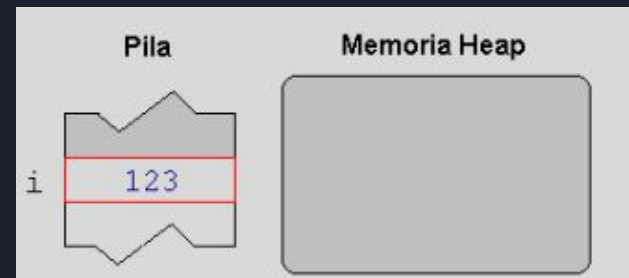
```
object o;  
int i = 123;
```

• • •

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

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;
```

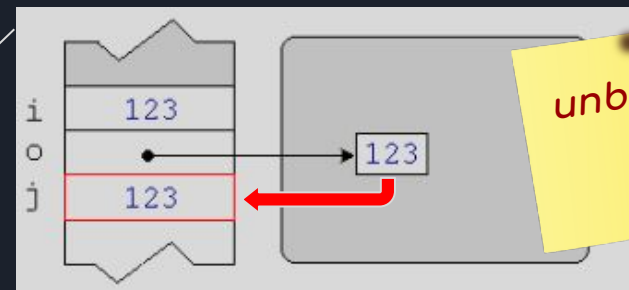
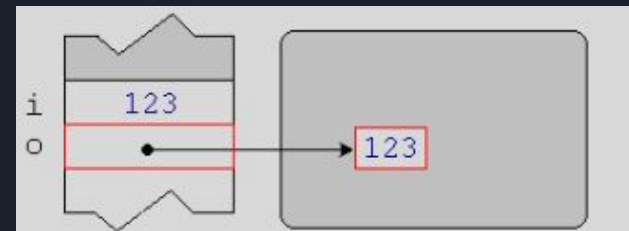
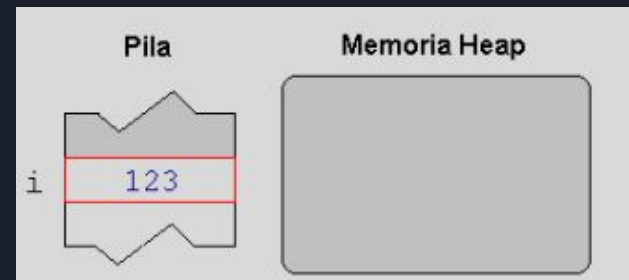
```
int i = 123;
```

```
• • •
```

```
o = i;
```

```
• • •
```

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



unboxing

Consecuencias del Sistema unificado de tipos

Si un `int` es también un `object` implica que todo lo que puede hacer un `object` también lo puede hacer 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

```
teoria2 : bash — Konsole
Archivo  Editar  Ver  Marcadores  Preferencias  Ayuda
leo@leo-desktop:~$ cd proyectosDotnet/
leo@leo-desktop:~/proyectosDotnet$ dotnet new console -o teoria2
The template "Console Application" was created successfully.

Processing post-creation actions...
Running 'dotnet restore' on teoria2/teoria2.csproj...
  Restauración realizada en 116,08 ms para /home/leo/proyectosDotnet/teoria2.csproj.

Restore succeeded.

leo@leo-desktop:~/proyectosDotnet$ cd teoria2
leo@leo-desktop:~/proyectosDotnet/teoria2$ code .
```



Probar el siguiente código

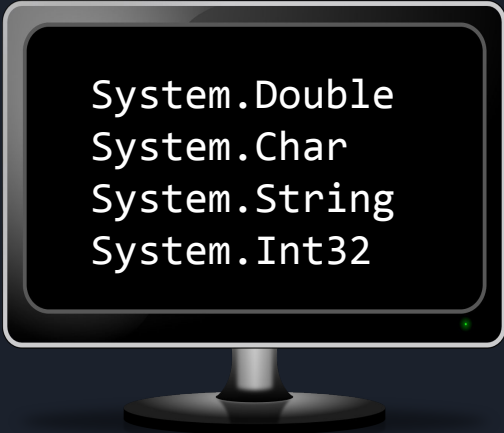
```
using System;
namespace Teoria2
{
    class Program
    {
        static void Main(string[] args)
        {
            object obj = 7.3; // obj apunta a un valor de tipo double
            Console.WriteLine(obj.GetType());
            obj = 'A'; // ahora de tipo char
            Console.WriteLine(obj.GetType());
            obj = "Casa"; // ahora de tipo string
            Console.WriteLine(obj.GetType());
            obj = 4; // ahora de tipo int
            Console.WriteLine(obj.GetType());
        }
    }
}
```





Salida por la consola

```
using System;
namespace Teoria2
{
    class Program
    {
        static void Main(string[] args)
        {
            object obj = 7.3; // obj apunta a un valor de tipo double
            Console.WriteLine(obj.GetType());
            obj = 'A'; // ahora de tipo char
            Console.WriteLine(obj.GetType());
            obj = "Casa"; // ahora de tipo string
            Console.WriteLine(obj.GetType());
            obj = 4; // ahora de tipo int
            Console.WriteLine(obj.GetType());
        }
    }
}
```



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



Probar el siguiente código

```
using System;
namespace Teoria2
{
    class Program
    {
        static void Main(string[] args)
        {
            object obj = 7.3; // obj apunta a un valor de tipo double
            Console.WriteLine(obj);
            obj = 'A'; // ahora de tipo char
            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

```
using System;
namespace Teoria2
{
    class Program
    {
        static void Main(string[] args)
        {
            object obj = 7.3; // obj apunta a un valor de tipo double
            Console.WriteLine(obj);
            obj = 'A'; // ahora de tipo char
            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);
```

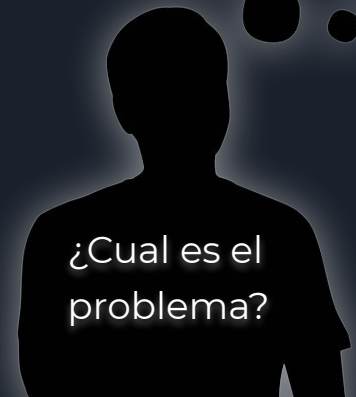
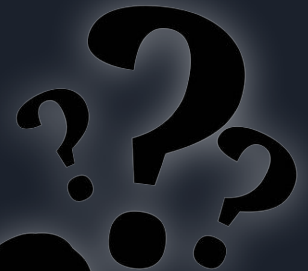
`WriteLine` recibe como parámetro un objeto de cualquier tipo, se invoca `obj.ToString()` y el resultado devuelto se imprime en la pantalla



Probar el siguiente código

```
using System;
namespace Teoria2
{
    class Program
    {
        static void Main(string[] args)
        {
            object obj = 7.3; // obj apunta a un valor de tipo double
            Console.WriteLine(obj);
            obj = 'A'; // ahora de tipo char
            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

```
using System;
namespace Teoria2
{
    class Program
    {
        static void Main(string[] args)
        {
            object obj = 7.3; // ahora de tipo double
            Console.WriteLine(obj);
            obj = 'A'; // ahora de tipo char
            Console.WriteLine(obj);
            obj = "Casa"; // ahora de tipo string
            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



```
using System;
namespace Teoria2
{
    class Program
    {
        static void Main(string[] args)
        {
            object obj = 7.3; // obj apunta a un valor de tipo double
            Console.WriteLine(obj);
            obj = 'A'; // ahora de tipo char
            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`

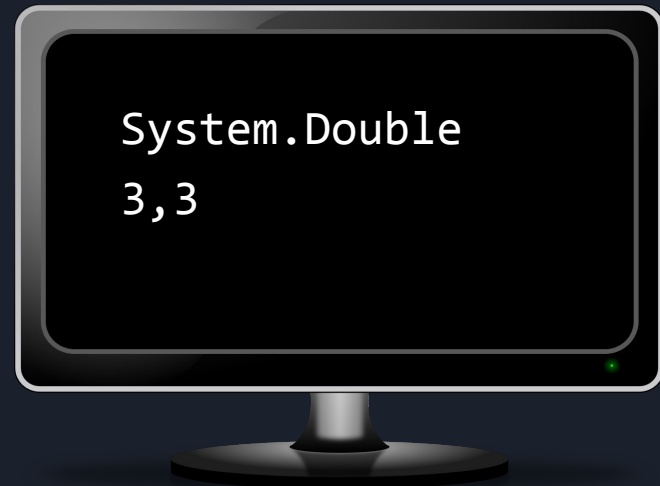


Atención. El operador `+` está sobrecargado



```
public static void Main(string[] args)
{
    object obj = 1 + 2.3;
    Console.WriteLine(obj.GetType());
    Console.WriteLine(obj);
}
```

Sumando un `int` con
un `double`



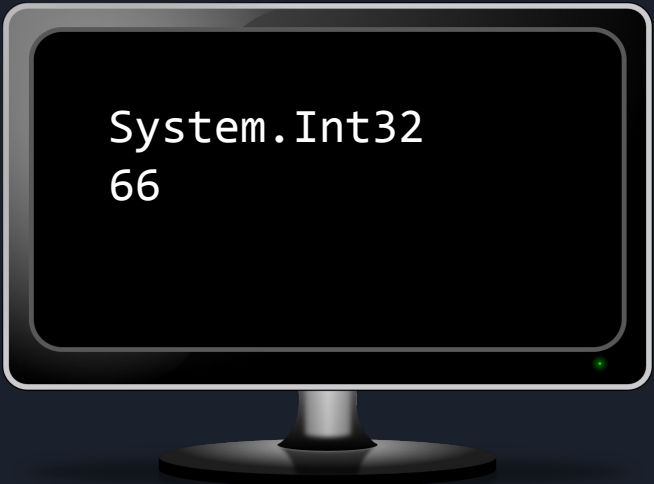


Atención. El operador `+` está sobrecargado



```
public static void Main(string[] args)
{
    object obj = 1 + 'A';
    Console.WriteLine(obj.GetType());
    Console.WriteLine(obj);
}
```

Sumando un `int` con
un `char`



```
System.Int32
66
```

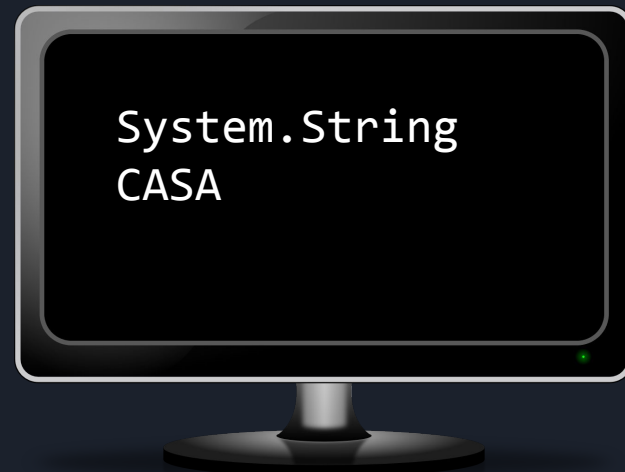


Atención. El operador `+` está sobrecargado



```
public static void Main(string[] args)
{
    object obj = "CAS" + 'A';
    Console.WriteLine(obj.GetType());
    Console.WriteLine(obj);
}
```

Sumando un `string`
con un `char`



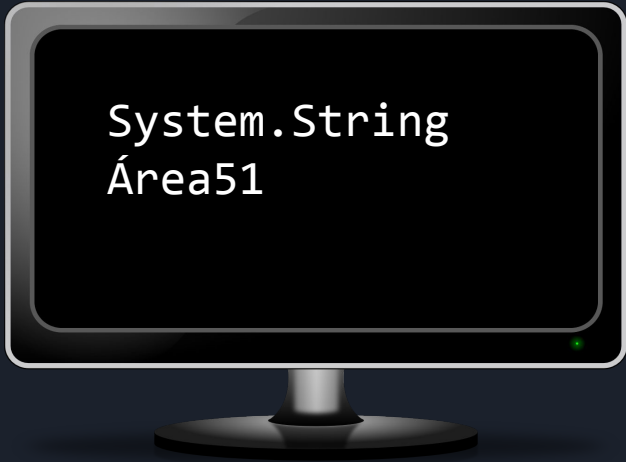


Atención. El operador `+` está sobrecargado



```
public static void Main(string[] args)
{
    object obj = "Área" + 51;
    Console.WriteLine(obj.GetType());
    Console.WriteLine(obj);
}
```

Sumando un `string`
con un `int`



```
System.String
Área51
```

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"`

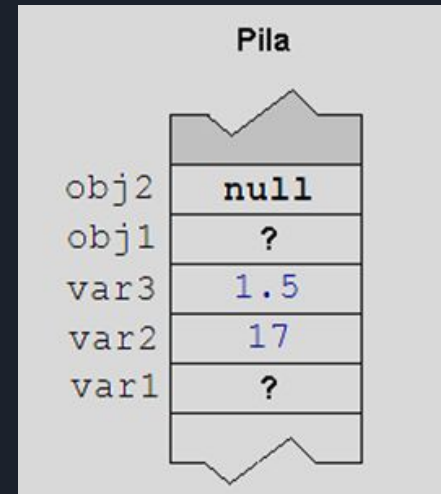
`"6Área"`



Sistema de Tipos

- Declaración de **variables locales** - Pila resultante

```
static void Main(String[] args)
{
    int var1;
    int var2 = 17;
    double var3 = 1.5;
    object obj1;
    object obj2 = null;
}
```



- 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

- Declaración de **variables locales**

```
using System;
class Ejemplo
{
    static void Main(String[] args)
    {
        int i;
        Console.WriteLine(i);
    }
}
```

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

Sistema de Tipos

- Declaración de variables locales

```
using System;
class Ejemplo
{
    static void Main(S
    {
        int i;
        Console.WriteLine(i);
    }
}
```

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

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



Sistema de Tipos - Arreglos I

- 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 I

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

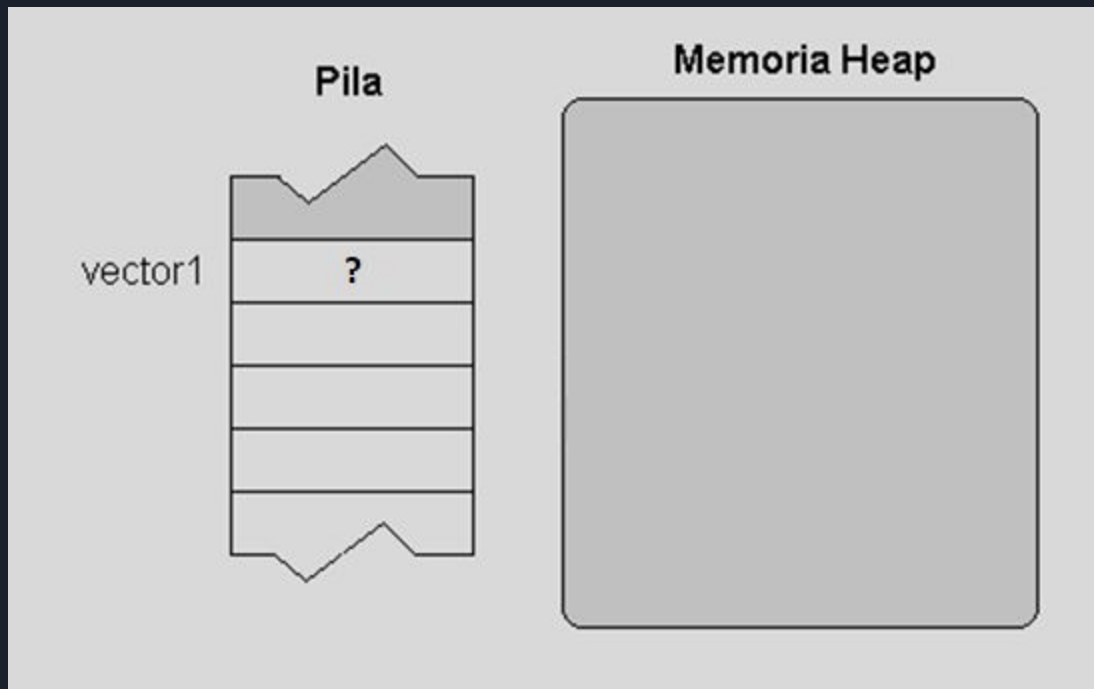
```
int[] vector2 = new int[100];
```

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

Declara, instancia e inicializa un vector con
4 elementos enteros

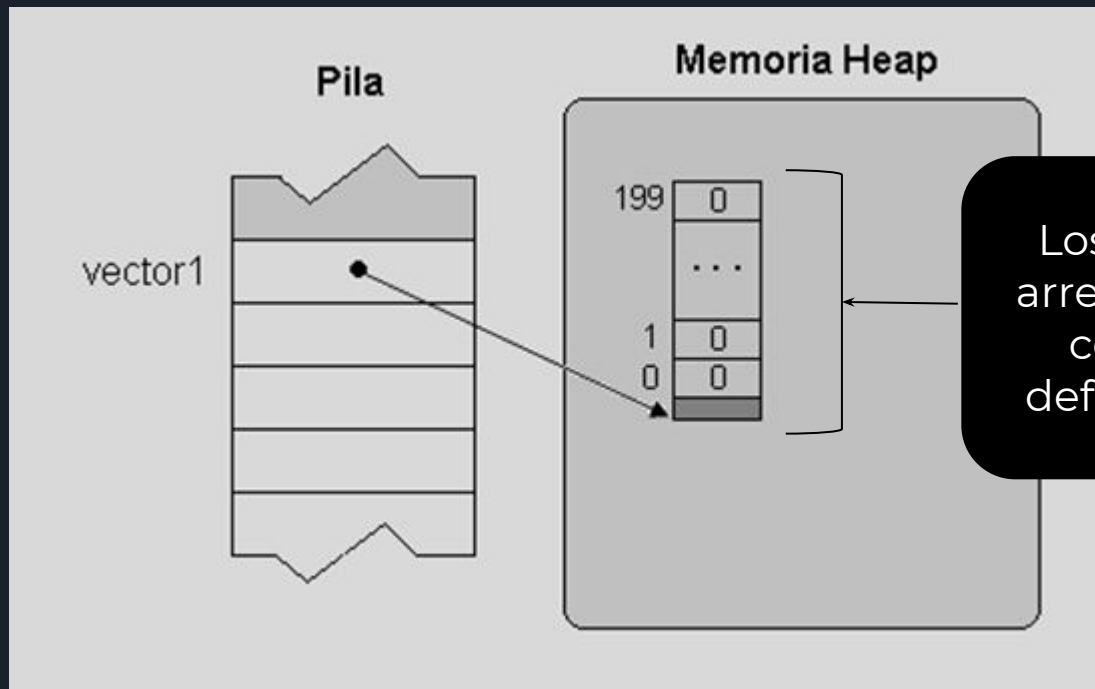
Sistema de Tipos - Arreglos I

```
int[] vector1;
```



Sistema de Tipos - Arreglos I

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



Los elementos del arreglo se inicializan con el valor por defecto para el tipo



Sistema de Tipos - Arreglos I

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';
```

El primer elemento ocupa la posición 0

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

Último elemento:

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

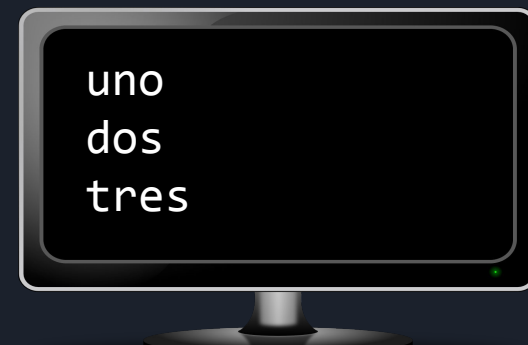
Sistema de Tipos - Arreglos I

Estructura de control `foreach`

```
foreach (<tipoElem> <elem> in <colección>)  
    <instrucción o bloque de instrucciones>
```

Restricción: La variable de iteración `<elem>` no puede ser asignada en el cuerpo del `foreach`

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

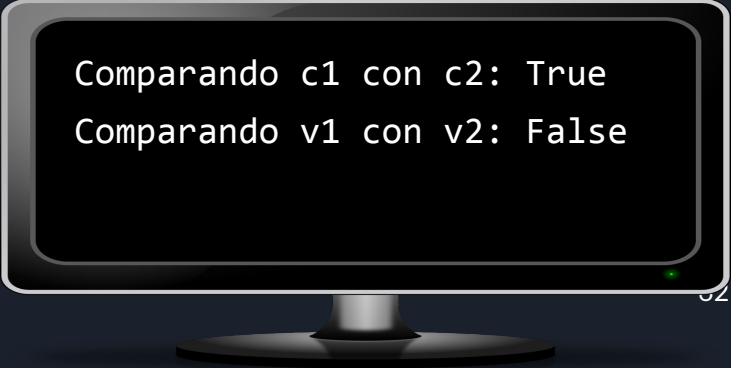


Sistema de Tipos – Diferencias entre tipos de valor y de referencia

```
char c1 = 'A';  
char c2 = 'A';  
Console.Write("Comparando c1 con c2: ");  
Console.WriteLine(c1 == c2);  
char[] v1 = new char[] { 'A' }; //vector de un  
elemento  
char[] v2 = new char[] { 'A' }; //vector de un  
elemento  
Console.Write("Comparando v1 con v2: ");  
Console.WriteLine(v1 == v2);
```

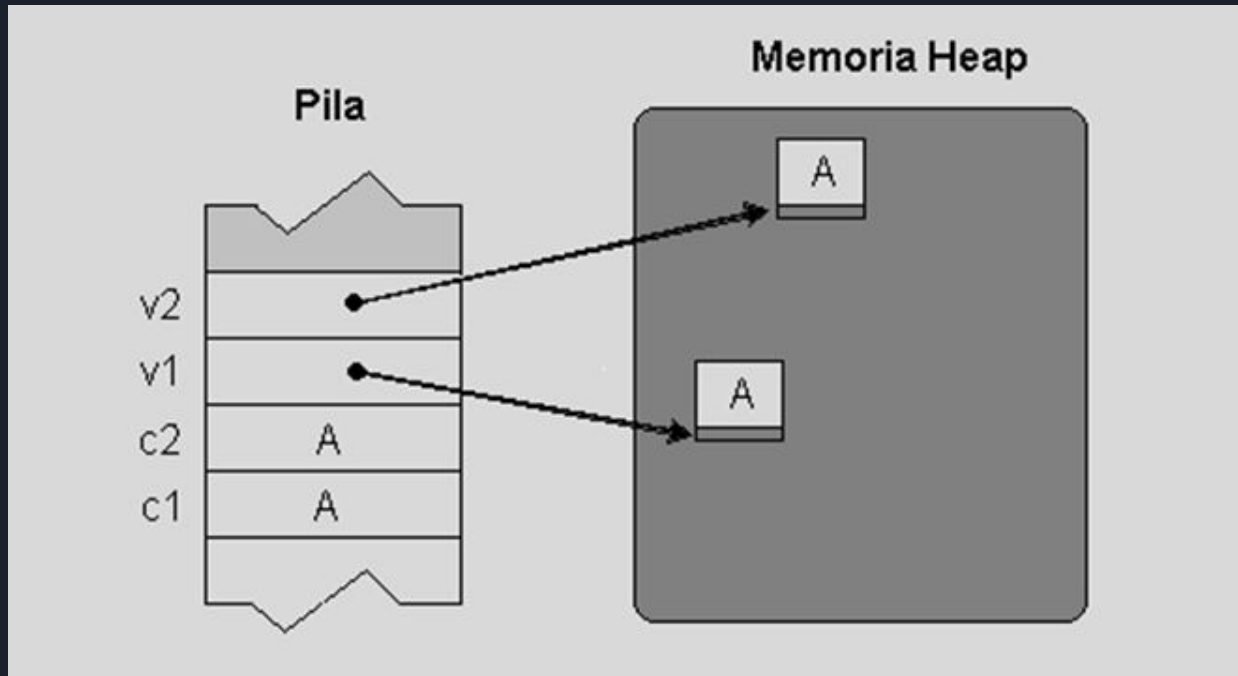
Sistema de Tipos – Diferencias entre tipos de valor y de referencia

```
char c1 = 'A';  
char c2 = 'A';  
Console.Write("Comparando c1 con c2: ");  
Console.WriteLine(c1 == c2);  
char[] v1 = new char[] { 'A' }; //vector de un  
elemento  
char[] v2 = new char[] { 'A' }; //vector de un  
elemento  
Console.Write("Comparando v1 con v2: ");  
Console.WriteLine(v1 == v2);
```



```
Comparando c1 con c2: True  
Comparando v1 con v2: False
```

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



La comparación por igualdad de `v1` y `v2` resulta falsa puesto que, por tratarse de tipos de referencia, no se compara el contenido sino las referencias

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

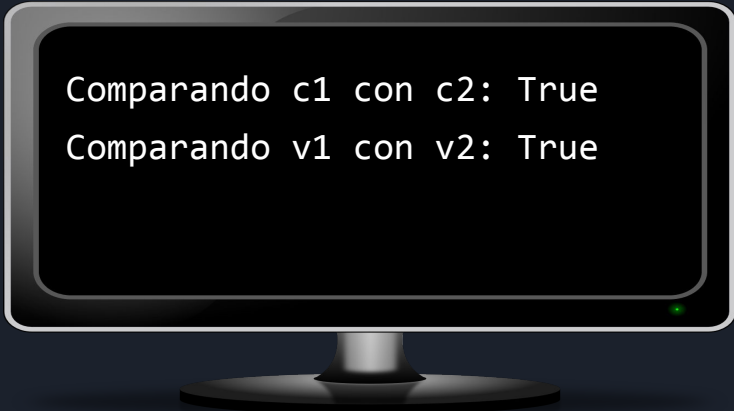
```
char c1 = 'A';  
char c2 = c1; //copia el valor 'A'  
Console.Write("Comparando c1 con c2: ");  
Console.WriteLine(c1 == c2);
```

```
char[] v1 = new char[] { 'A' };  
char[] v2 = v1; //copia la referencia  
Console.Write("Comparando v1 con v2: ");  
Console.WriteLine(v1 == v2);
```


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

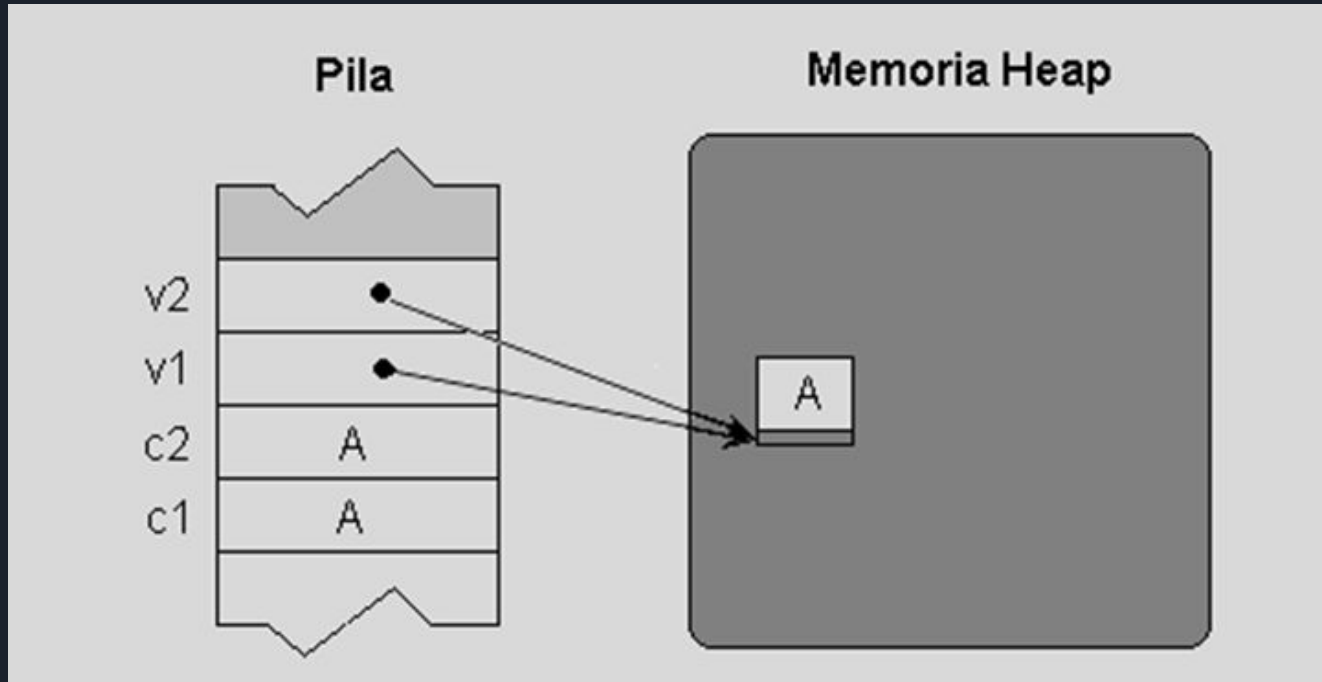
```
char c1 = 'A';  
char c2 = c1; //copia el valor 'A'  
Console.Write("Comparando c1 con c2: ");  
Console.WriteLine(c1 == c2);
```

```
char[] v1 = new char[] { 'A' };  
char[] v2 = v1; //copia la referencia  
Console.Write("Comparando v1 con v2: ");  
Console.WriteLine(v1 == v2);
```



```
Comparando c1 con c2: True  
Comparando v1 con v2: True
```

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



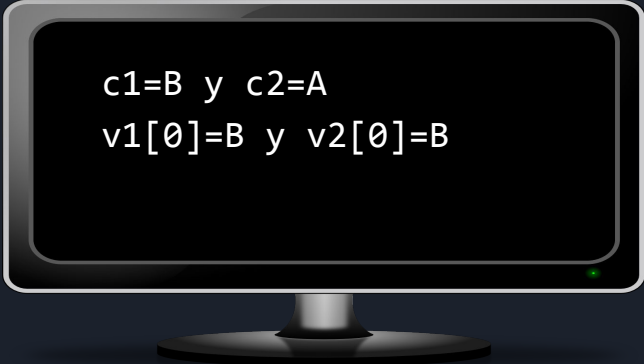
La comparación por igualdad de **v1** y **v2** resulta verdadera puesto que ambas variables poseen la misma referencia (apuntan al mismo objeto)

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

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

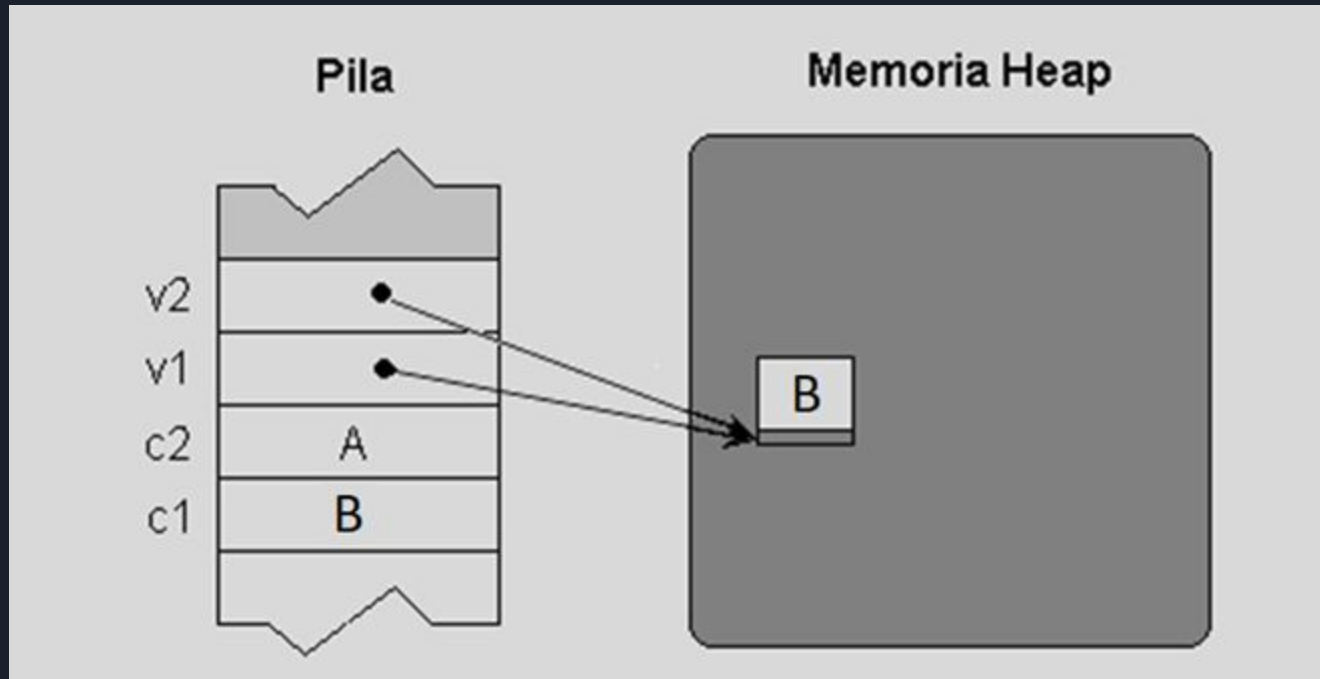
Sistema de Tipos – Diferencias entre tipos de valor y de referencia

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



```
c1=B y c2=A  
v1[0]=B y v2[0]=B
```

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



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

Sistema de Tipos - La clase `String`

Secuencia de caracteres

```
string st1 = "es un string";  
string st2 = "";  
string st3 = null;
```

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

Sistema de Tipos - La clase `String`

- Los string son de **sólo lectura** (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 cad = "Hola";
```

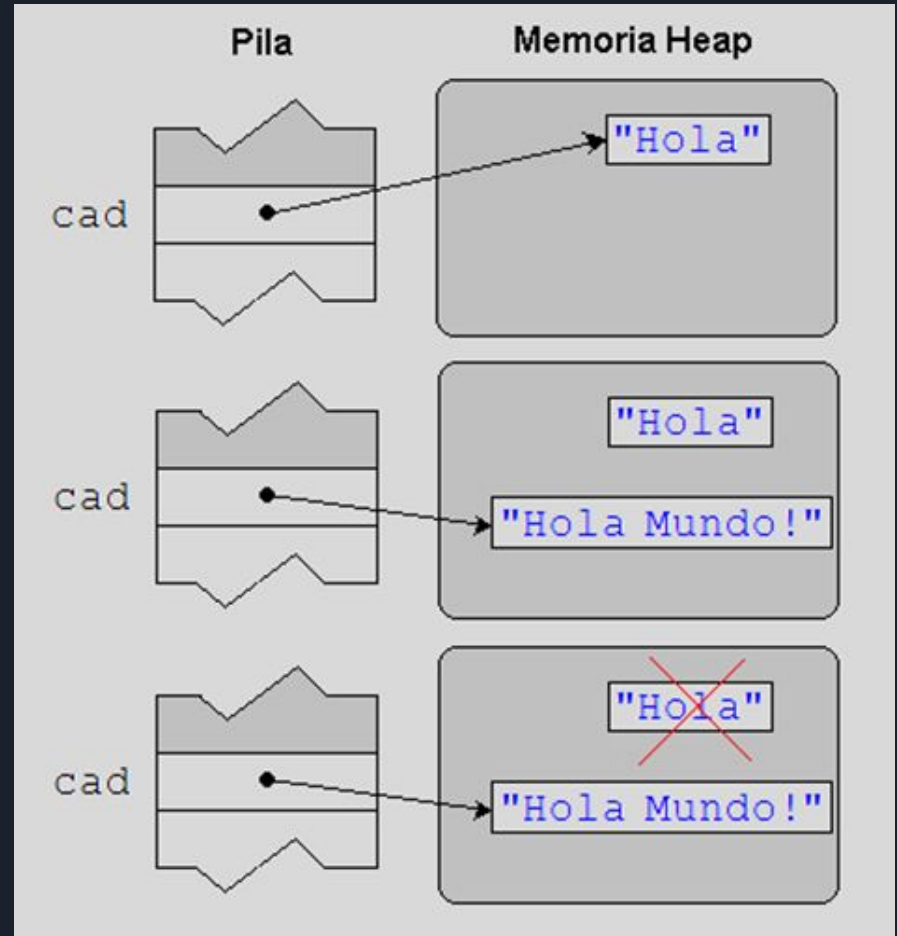
```
cad = cad + " Mundo!";
```

Correcto: se está creando un nuevo string

Sistema de Tipos - La clase String

```
string cad = "Hola";
```

```
cad = cad + " Mundo!";
```

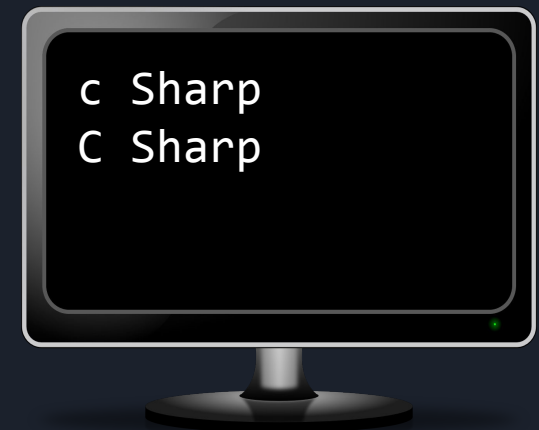


Sistema de Tipos - La clase `StringBuilder`

- `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`

```
using System;
using System.Text;
class Program
{
    static void Main(String[] args)
    {
        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

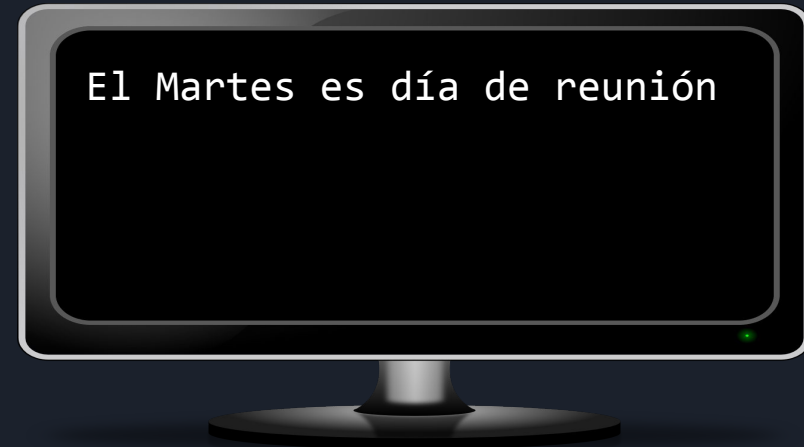
```
Tamaño t;
t = Tamaño.grande;
t = (Tamaño)0; //ahora t vale Tamaño.chico
```

Tipos enumerativos

```
using System;

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

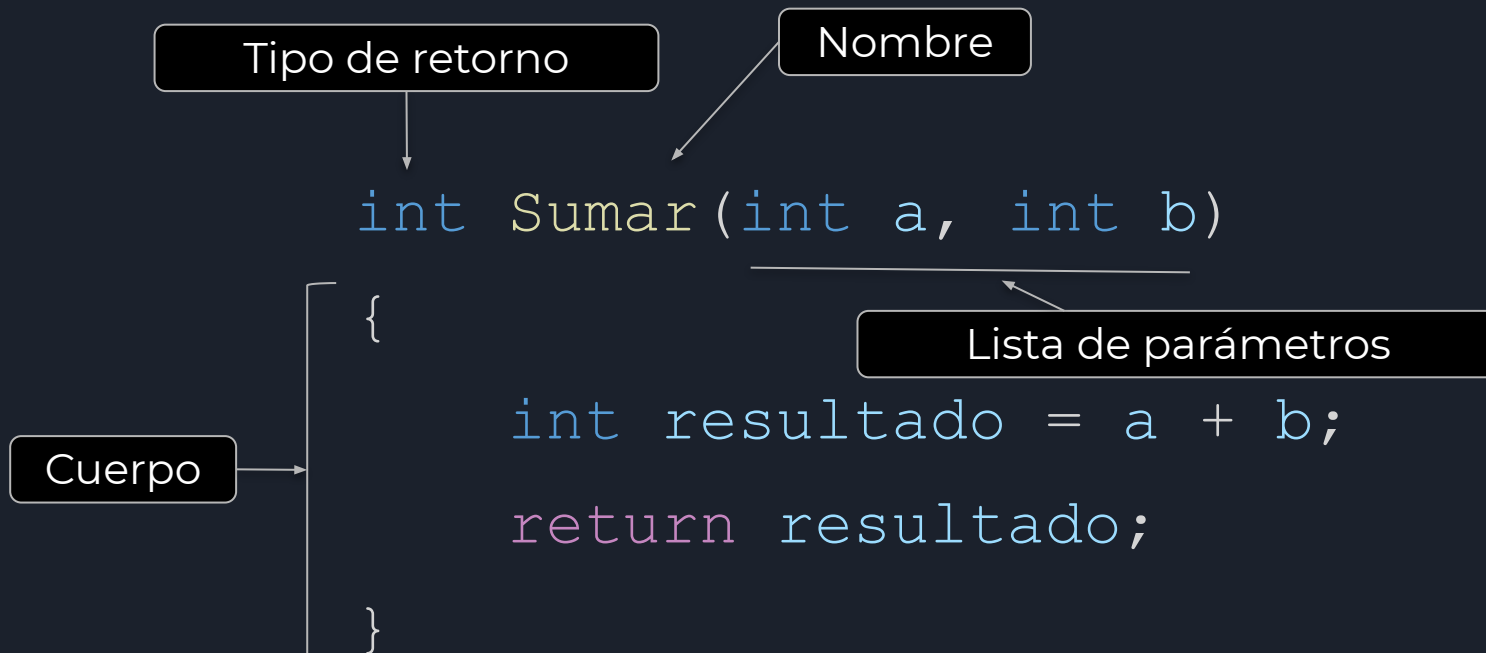
class Program
{
    public static void Main(string[] args)
    {
        DiaDeSemana diaDeReunion = DiaDeSemana.Martes;
        for (DiaDeSemana d = DiaDeSemana.Lunes; d <= DiaDeSemana.Viernes; d++)
        {
            if (d == diaDeReunion)
            {
                Console.WriteLine("El " + d + " es día de reunión");
            }
        }
    }
}
```



Métodos

Métodos

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



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

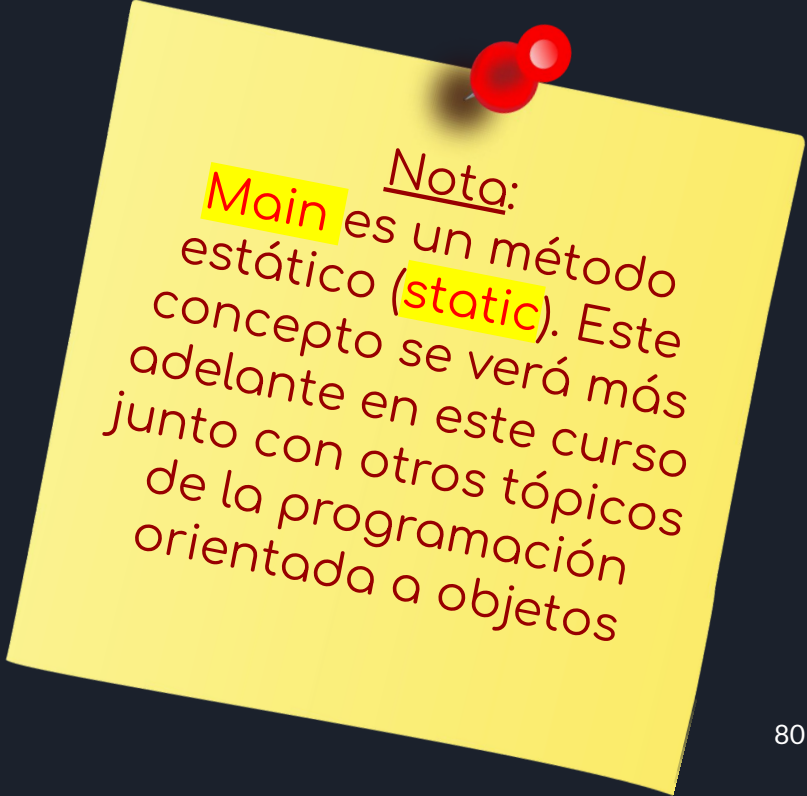
Formas del método Main

```
static void Main(string[] args) {...}
```

```
static int Main(string[] args) {...}
```

```
static void Main() {...}
```

```
static int Main() {...}
```




Nota:
Main es un método estático (**static**). Este concepto se verá más adelante en este curso junto con otros tópicos de la programación orientada a objetos

Pasaje de parámetros por la línea de comandos

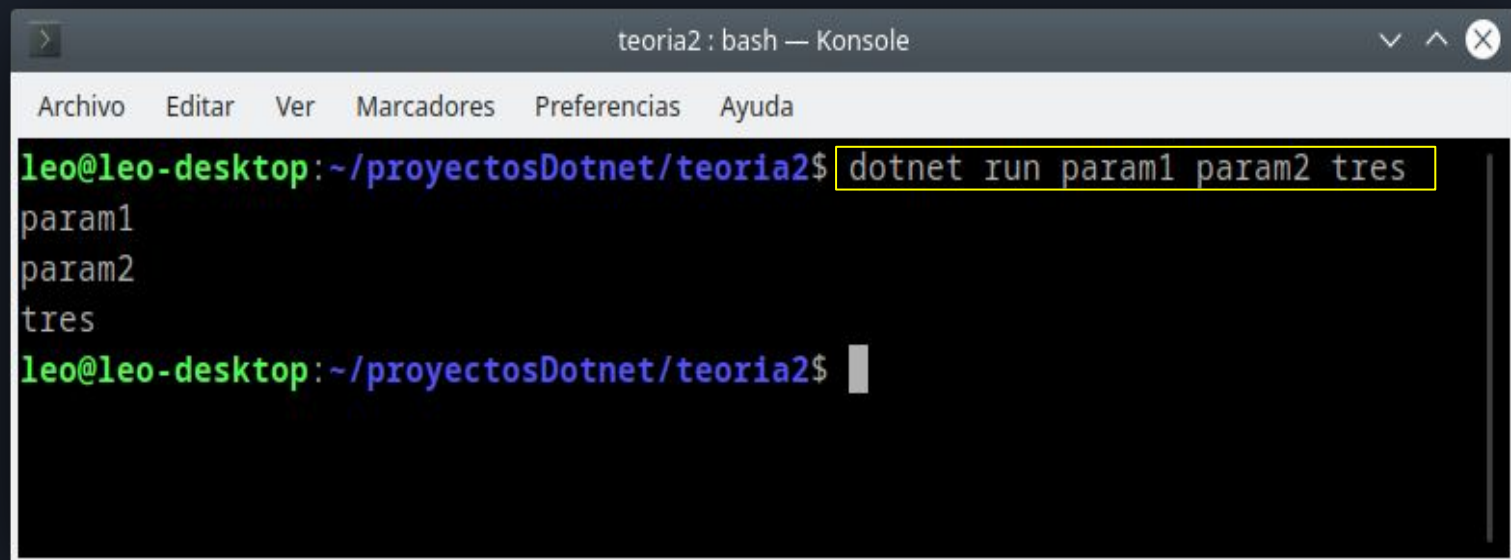
```
using System;
class Program
{
    static void Main(String[] args)
    {
        foreach (string st in args)
        {
            Console.WriteLine(st);
        }
    }
}
```

Los parámetros pasados por la línea de comandos se reciben en el vector `args`



Pasaje de parámetros por la línea de comandos

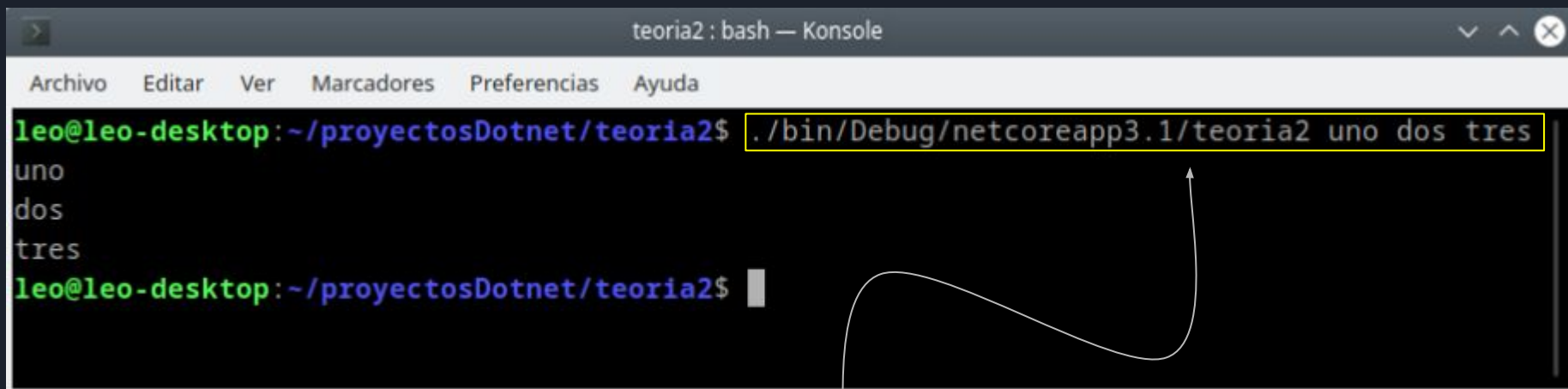
Es posible **compilar** y **ejecutar** la aplicación **pasando** los **parámetros** necesarios todo junto desde una **terminal** del sistema operativo



```
teoria2 : bash — Konsole
Archivo  Editar  Ver  Marcadores  Preferencias  Ayuda
leo@leo-desktop:~/proyectosDotnet/teoria2$ dotnet run param1 param2 tres
param1
param2
tres
leo@leo-desktop:~/proyectosDotnet/teoria2$
```

Pasaje de parámetros por la línea de comandos

Una vez compilado, puede invocarse directamente el ejecutable desde la carpeta en la que se generó



The screenshot shows a terminal window titled "teoria2 : bash — Konsole". The terminal displays the following commands and output:

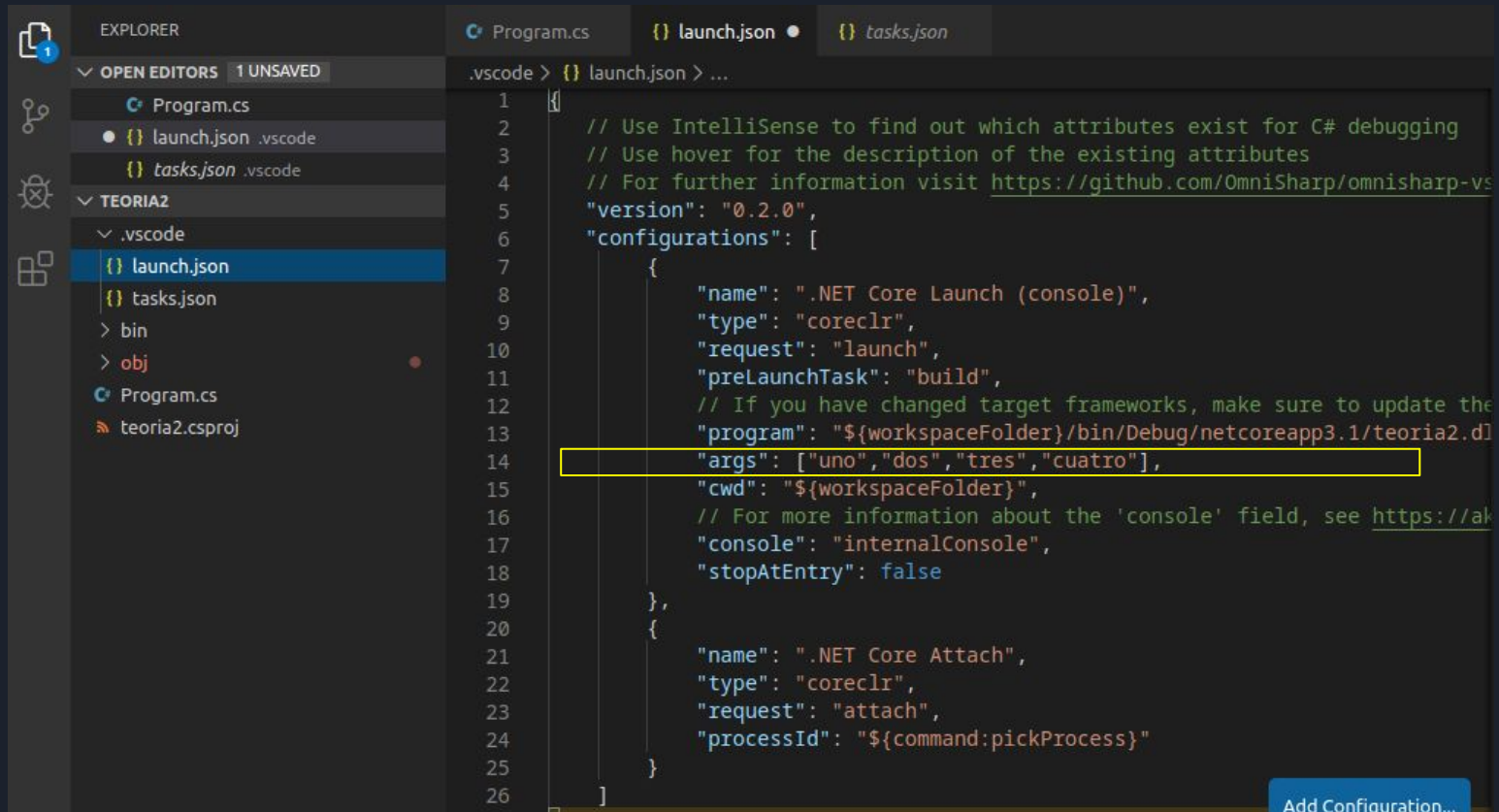
```
leo@leo-desktop:~/proyectosDotnet/teoria2$ ./bin/Debug/netcoreapp3.1/teoria2 uno dos tres
uno
dos
tres
leo@leo-desktop:~/proyectosDotnet/teoria2$
```

The command `./bin/Debug/netcoreapp3.1/teoria2 uno dos tres` is highlighted with a yellow box. A white arrow points from the text below to this command.

Cambia según la versión de dotnet que estemos utilizando.
En .Net 5.0 sería: `./bin/Debug/net5.0/teoria2`

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`



```
.vscode > {} launch.json > ...
1  // Use IntelliSense to find out which attributes exist for C# debugging
2  // Use hover for the description of the existing attributes
3  // For further information visit https://github.com/OmniSharp/omnisharp-vs
4  "version": "0.2.0",
5  "configurations": [
6      {
7          "name": ".NET Core Launch (console)",
8          "type": "coreclr",
9          "request": "launch",
10         "preLaunchTask": "build",
11         // If you have changed target frameworks, make sure to update the
12         "program": "${workspaceFolder}/bin/Debug/netcoreapp3.1/teoria2.dll",
13         "args": ["uno", "dos", "tres", "cuatro"],
14         "cwd": "${workspaceFolder}",
15         // For more information about the 'console' field, see https://aka.ms/VSCode-debug-console
16         "console": "internalConsole",
17         "stopAtEntry": false
18     },
19     {
20         "name": ".NET Core Attach",
21         "type": "coreclr",
22         "request": "attach",
23         "processId": "${command:pickProcess}"
24     }
25 ]
```

Método - parámetros

Parámetros de entrada (por valor): Recibe una copia del valor pasado como parámetro

```
class Program
{
    static void Main(String[] args)
    {
        int entero = 10;
        Imprimir(entero);
        Console.WriteLine(entero);
    }

    static void Imprimir(int n)
    {
        n = -5;
        Console.WriteLine(n);
    }
}
```

Parámetro de
entrada



Método - parámetros

Parámetros de entrada (por valor): Recibe una copia del valor pasado como parámetro

```
class Program
{
    static void Main(String[] args)
    {
        int entero = 10;
        Imprimir(entero);
        Console.WriteLine(entero);
    }

    static void Imprimir(int n)
    {
        n = -5;
        Console.WriteLine(n);
    }
}
```



Nota:
Desde un método estático (**static**) como es el caso de **Main**, puede invocarse de forma directa a otro método estático de la misma clase.

Método - parámetros

Parámetros 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

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

Parámetro de
salida

Método - parámetros

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

Los parámetros **out** se deben asignar dentro del cuerpo del método antes de cualquier lectura.

Método - parámetros

Parámetros de salida (out):

```
static void Main(String[] args)
{
    int r;
    Sumar(10, 20, out r);
    Console.WriteLine(r);
}
```

- En la invocación también se debe utilizar **out**
- La variable **r** no está inicializada pero es válido

Parámetro de salida

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



Método - parámetros

Parámetros por referencia (`ref`):

- Similar a los parámetros de salida, pero no es posible invocar 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

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

Parámetro
referencia

Válido. Se garantiza que
el parámetro estará
inicializado

Método - parámetros

Parámetros por referencia (ref):

```
static void Main(String[] args)
{
    int r = 0;
    Sumar(10, 20, ref r);
    Console.WriteLine(r);
}
```

- La variable **r** debe estar inicializada
- En la invocación también se debe utilizar **ref**

por referencia

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



30

Método - parámetros

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

```
public static void Main(string[] args)
{
    int n = 10;
    Imprimir(in n);
}
```

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

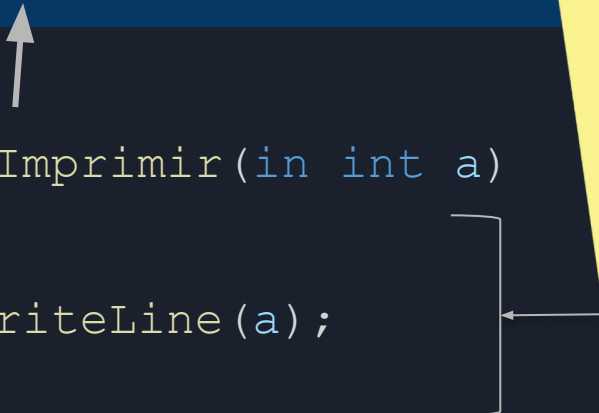
El valor de **a** no puede modificarse, es de sólo lectura

Método - parámetros

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

```
public static void Main(string[] args)
{
    int n = 10;
    Imprimir(in n);
}

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



Nota:
El modificador **in** podría omitirse en la invocación
Siempre que el método no esté sobrecargado (este concepto se verá más adelante en este curso)


Método - parámetros

Parámetros de entrada (in):

```
public static void Main(string[] args)
{
    int n = 10;
    Imprimir(33);
    Imprimir(in n);
    Imprimir(n);
}

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

Se admite pasar un literal omitiendo `in`.
El compilador crea una variable oculta
para poder pasar la referencia



33
10
10

Método - parámetros

Parámetros de entrada (in):

```
public static void Main()  
{  
    int i;  
    Impresora i1;  
    Impresora i2;  
    Impresora i3;  
}  
static void Main()  
{  
    Console.WriteLine("Inicio");  
}
```

Nota:
Usando el modificador **in** se consigue pasar un parámetro por referencia con la intención de evitar la copia pero asegurando la no modificación del valor

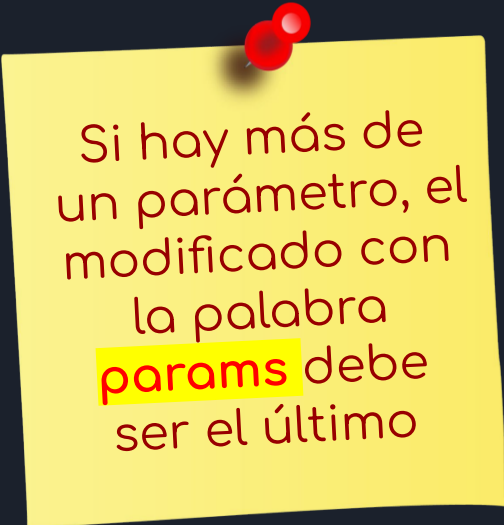
Método - parámetros

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

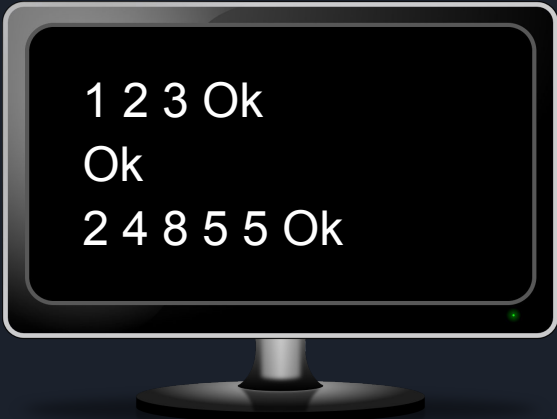
Método - parámetros

Uso de la palabra clave params :

```
public static void Main(string[] args)
{
    int[] vector = { 1, 2, 3 };
    Imprimir(vector);
    Imprimir();
    Imprimir(2, 4, 8, 5, 5);
}

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

Se puede pasar un vector o una lista de elementos que puede ser vacía



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;
```

Fin

