



# .NET

## Teoría 3

# Más sobre tipos

## Arreglos de 2 dimensiones

- Matriz

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

1	2	3	4
5	6	7	8
9	10	11	12

- Acceso

```
matriz[2, 2] = matriz[1, 1] + matriz[1, 2];
```

1	2	3	4
5	6	7	8
9	10	13	12

## Arreglos de 2 dimensiones

```
int[,] mat = new int[3, 4];  
for (int i = 0; i <= 11; i++)  
{  
    mat[i / 4, i % 4] = i;  
}
```



## Arreglos de 2 dimensiones

```
int[,] mat = new int[3, 4];  
for (int i = 0; i <= 11; i++)  
{  
    mat[i / 4, i % 4] = i;  
}
```

	0	1	2	3
0	0	1	2	3
1	4	5	6	7
2	8	9	10	11

## Arreglo de arreglos

Los elementos del arreglo son también un arreglo

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

0	0	
0	0	0

Es equivalente a:

```
int[][] tabla = { new int[2], new int[3] };
```

Acceso

```
tabla[1][2] = 8;
```

0	0	
0	0	8

## Arreglo de arreglos

```
int[][] tablaEscalonada = new int[5][];  
for (int i = 0; i < 5; i++)  
{  
    tablaEscalonada[i] = new int[i + 1];  
}
```

0				
0	0			
0	0	0		
0	0	0	0	
0	0	0	0	0

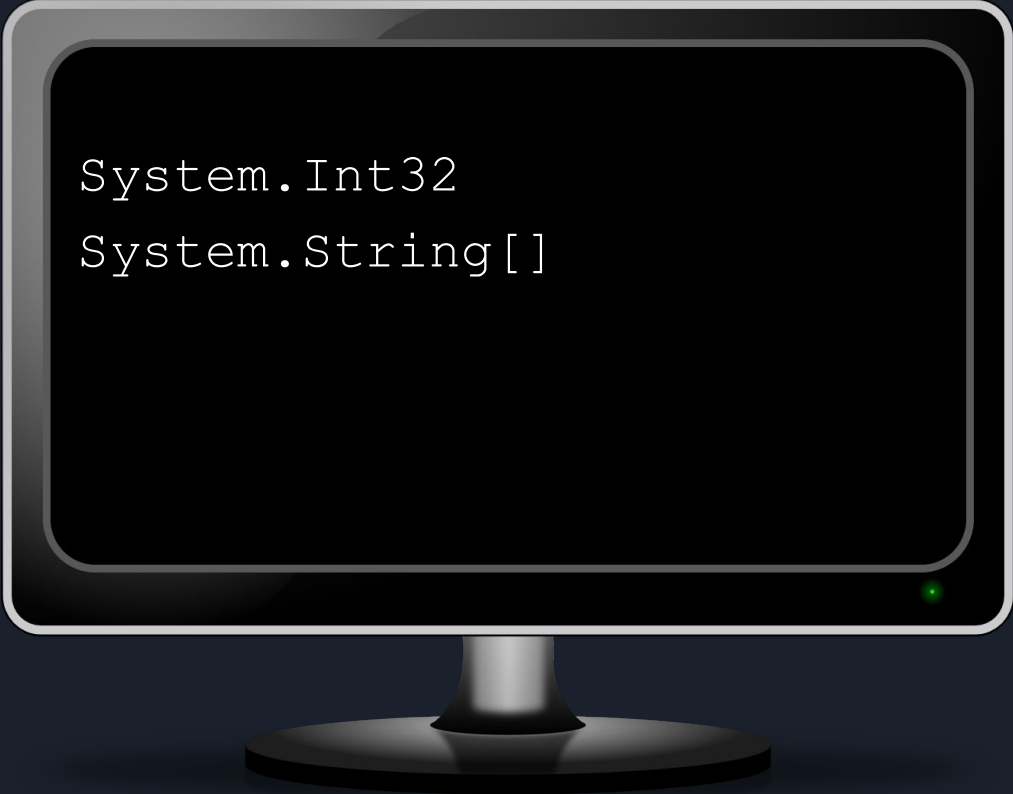
### Inferencia de tipos - palabra clave var

La palabra clave `var` en la `declaración` de una `variable local con inicialización` ( $\neq$  `null`) indica que el tipo de la misma es inferido por el compilador en función de la inicialización. Ejemplo:

```
var i = 15;  
var vector = new string[100];  
Console.WriteLine(i.GetType());  
Console.WriteLine(vector.GetType());
```



```
var i = 15;  
var vector = new string[100];  
Console.WriteLine(i.GetType());  
Console.WriteLine(vector.GetType());
```



System.Int32  
System.String[]

## Inferencia de tipos - palabra clave var

Una vez inferido el tipo de una variable por el compilador queda fijo e inmutable (no es un tipo dinámico)

```
var i = 15;
```

```
i = 140;
```

```
i = 13.2;
```

El tipo inferido de `i` es `int`

Error de compilación, no se puede convertir implícitamente el tipo `double` en `int`

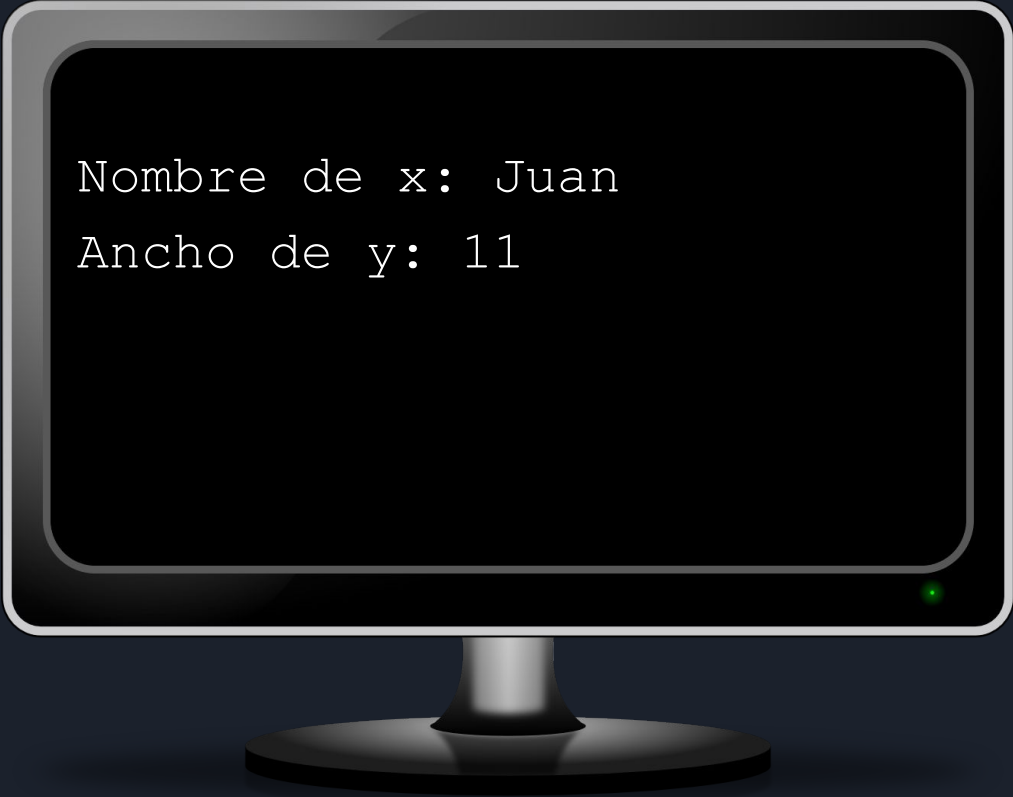
## Inferencia de tipos - tipos anónimos

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);
```

```
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);
```

**x** e **y** son variables de tipos anónimos distintos.



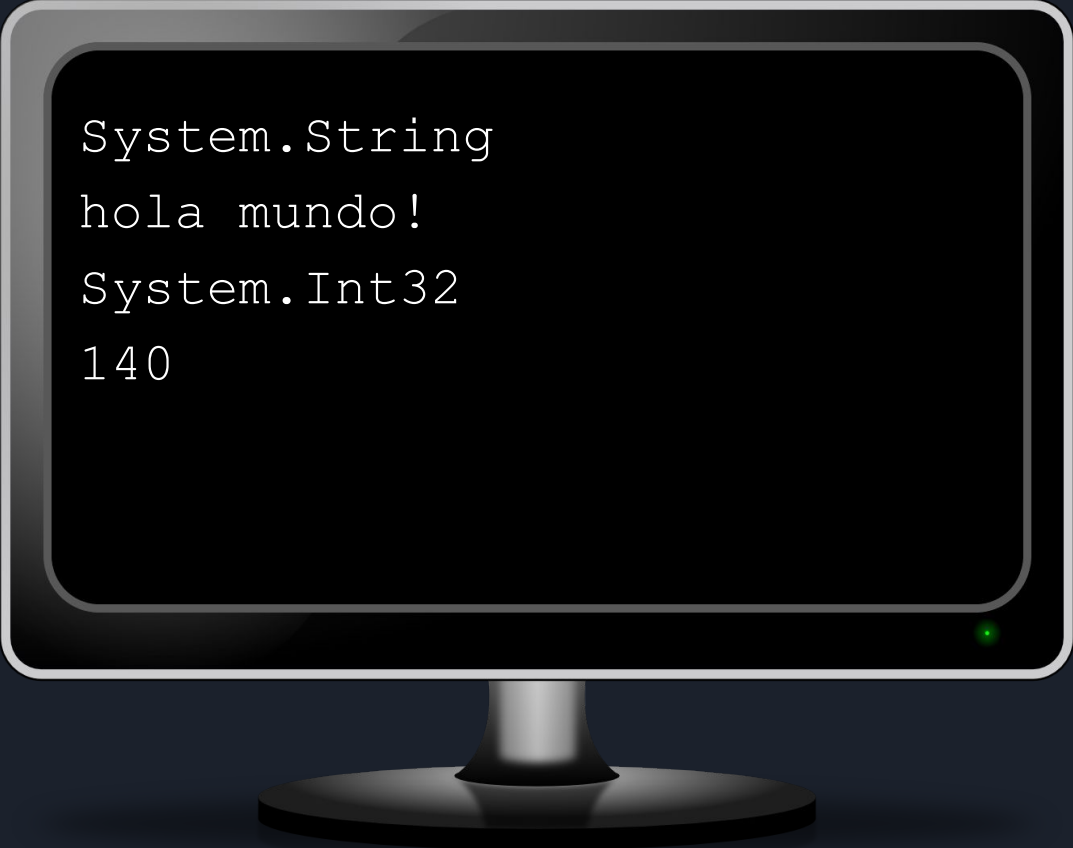
Nombre de x: Juan  
Ancho de y: 11

## Tipo dynamic

Una variable declarada de tipo `dynamic` admite la asignación de elementos de distintos tipos durante la ejecución

```
dynamic dy = "hola mundo!";  
Console.WriteLine(dy.GetType());  
Console.WriteLine(dy);  
dy = 140;  
Console.WriteLine(dy.GetType());  
Console.WriteLine(dy);
```

```
dynamic dy = "hola mundo!";  
Console.WriteLine(dy.GetType());  
Console.WriteLine(dy);  
dy = 140;  
Console.WriteLine(dy.GetType());  
Console.WriteLine(dy);
```



System.String  
hola mundo!  
System.Int32  
140

## Tipo dynamic

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 (excepciones).

Ejemplo:

```
dynamic dy = "hola mundo!";  
Console.WriteLine(dy.Length);  
dy = 140;  
Console.WriteLine(dy.Length);
```

Imprimir 11 en la consola

Error en tiempo de ejecución  
`int` no contiene una definición para `Length`

## Tipo dynamic

Debido a la falta de verificación de tipos en las expresiones donde hay elementos de tipo **dynamic** involucrados, tampoco son necesarias las conversiones explícitas. Ejemplo:

```
dynamic d1 = "hola mundo";  
dynamic d2 = 3;  
string st = d1;  
int i = d2 * 2;
```

No se necesita hacer casting Se realiza una conversión implícita Sigue existiendo riesgo de error en tiempo de ejecución



# Strings de formato compuesto y strings interpolados



## Haciendo pruebas



1. Abrir una terminal del sistema operativo
2. Cambiar a la carpeta `proyectosDotnet`
3. Crear la aplicación de consola `Teoria3`
4. Abrir `Visual Studio Code` sobre este proyecto



### Probar y contestar



¿Con qué valor queda asignada la variable st ?

```
string marca = "Ford";  
int modelo = 2000;  
string st = string.Format("Es un {0} año {1}", marca, modelo);
```

## Strings de formato compuesto y strings interpolados

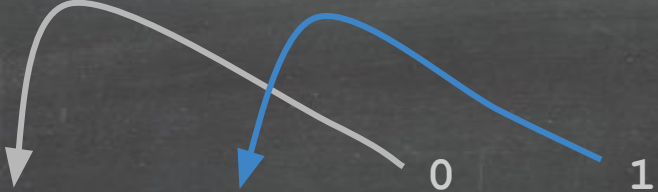
```
string marca = "Ford";  
int modelo = 2000;  
string st = string.Format("Es un {0} año {1}", marca, modelo);  
Console.WriteLine(st);
```



Es un Ford año 2000

## Cadenas de formato compuesto

```
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

vector de objetos  
¿recuerdan el  
modificador  
**params**?



## Cadenas de formato compuesto

A Los marcadores dentro del string de formato compuesto se los llama **elementos de formato**

"Es un {0} año {1}"

Elemento de  
formato

Elemento de  
formato





# Cadenas interpoladas Modificar y ejecutar



```
string marca = "Ford";
```

```
int modelo = 2000;
```

```
string st = $"Es un {marca} año {modelo}";
```

```
Console.WriteLine(st);
```

Anteponer  
signo \$

Interpolación de string  
(a partir de la versión C# 6.0)

## Strings de formato compuesto y strings interpolados

```
string marca = "Ford";  
int modelo = 2000;  
string st = $"Es un {marca} año {modelo}";  
Console.WriteLine(st);
```



Es un Ford año 2000



## Cadenas interpoladas

Las cadenas interpoladas también utilizan elementos de formato.

Son más legible y cómodas de utilizar que los strings de formato compuesto.

Las cadenas interpoladas llevan antepuesto el **símbolo \$**



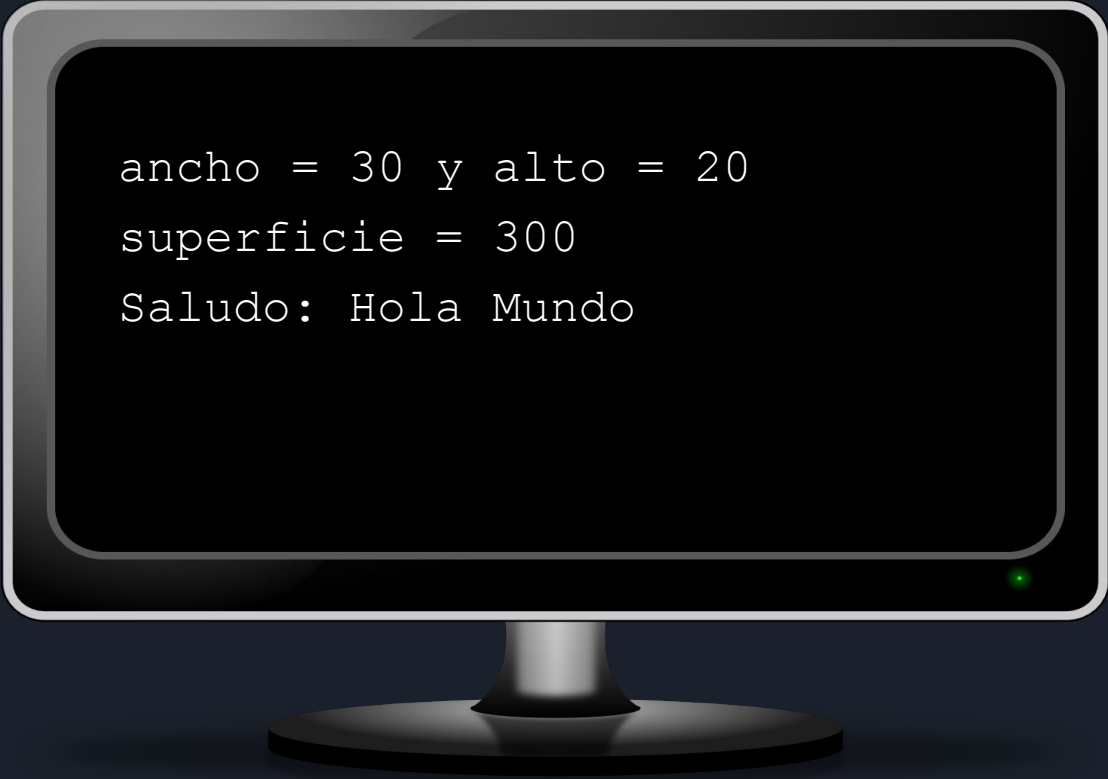
## Cadenas interpoladas

Los elementos de formato en las cadenas interpoladas también admiten expresiones

```
int ancho = 30;  
Console.WriteLine($"ancho = {ancho} y alto = {20}");  
Console.WriteLine($"superficie = {ancho * 20 / 2}");  
Console.WriteLine($"Saludo: {"Hola"+" Mundo"}");
```

## Strings de formato compuesto y strings interpolados

```
int ancho = 30;  
Console.WriteLine($"ancho = {ancho} y alto = {20}");  
Console.WriteLine($"superficie = {ancho * 20 / 2}");  
Console.WriteLine($"Saludo: {"Hola"+" Mundo"}");
```



ancho = 30 y alto = 20  
superficie = 300  
Saludo: Hola Mundo



# Cadenas interpoladas Modificar y ejecutar



```
string marca = "Ford";
```

```
int modelo = 2000;
```

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

```
Console.WriteLine($"Es un {"Nissan",7} año {2020}");
```

## Strings de formato compuesto y strings interpolados

```
string marca = "Ford";  
int modelo = 2000;  
Console.WriteLine($"Es un {marca,7} año {modelo}");  
Console.WriteLine($"Es un {"Nissan", 7} año {2020}");
```

Alineación derecha  
Completa con blancos a  
izquierda



```
Es un   Ford año 2000  
Es un  Nissan año 2020
```

7 caracteres de ancho

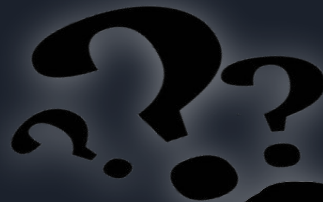


# Cadenas interpoladas Modificar y ejecutar



```
string marca = "Ford";  
int modelo = 2000;  
Console.WriteLine($"Es un {marca,-7} año {modelo}");  
Console.WriteLine($"Es un {"Nissan",-7} año {2020}");
```


Diagram illustrating string interpolation with format specifiers. The first line shows a variable `marca` with value `"Ford"` and a format specifier `,-7` (indicated by a downward arrow). The second line shows a string literal `"Nissan"` with a format specifier `,-7` (indicated by an upward arrow).



¿Y si se usan  
valores  
negativos?

## Strings de formato compuesto y strings interpolados

```
string marca = "Ford";  
int modelo = 2000;  
Console.WriteLine($"Es un {marca,-7} año {modelo}");  
Console.WriteLine($"Es un {"Nissan",-7} año {2020}");
```



Es un Ford año 2000  
Es un Nissan año 2020

7 caracteres de ancho  
alineación izquierda



# Cadenas interpoladas Modificar y ejecutar



```
double r = 2.417;  
Console.WriteLine($"Valor = {r:0.0}");  
Console.WriteLine($"Valor = {r:0.00}");
```



## Strings de formato compuesto y strings interpolados

```
double r = 2.417;  
Console.WriteLine($"Valor = {r:0.0}");  
Console.WriteLine($"Valor = {r:0.00}");
```

Máscaras de  
formato

```
Valor = 2,4  
Valor = 2,42
```

Redondea (no trunca)

## Sintaxis de elemento de formato

Cadena interpolada:

```
{expresión[,alignment][:format]}
```

Cadena de formato compuesto:

```
{indice[,alignment][:format]}
```



## Ejemplo máscara de formatos para DateTime

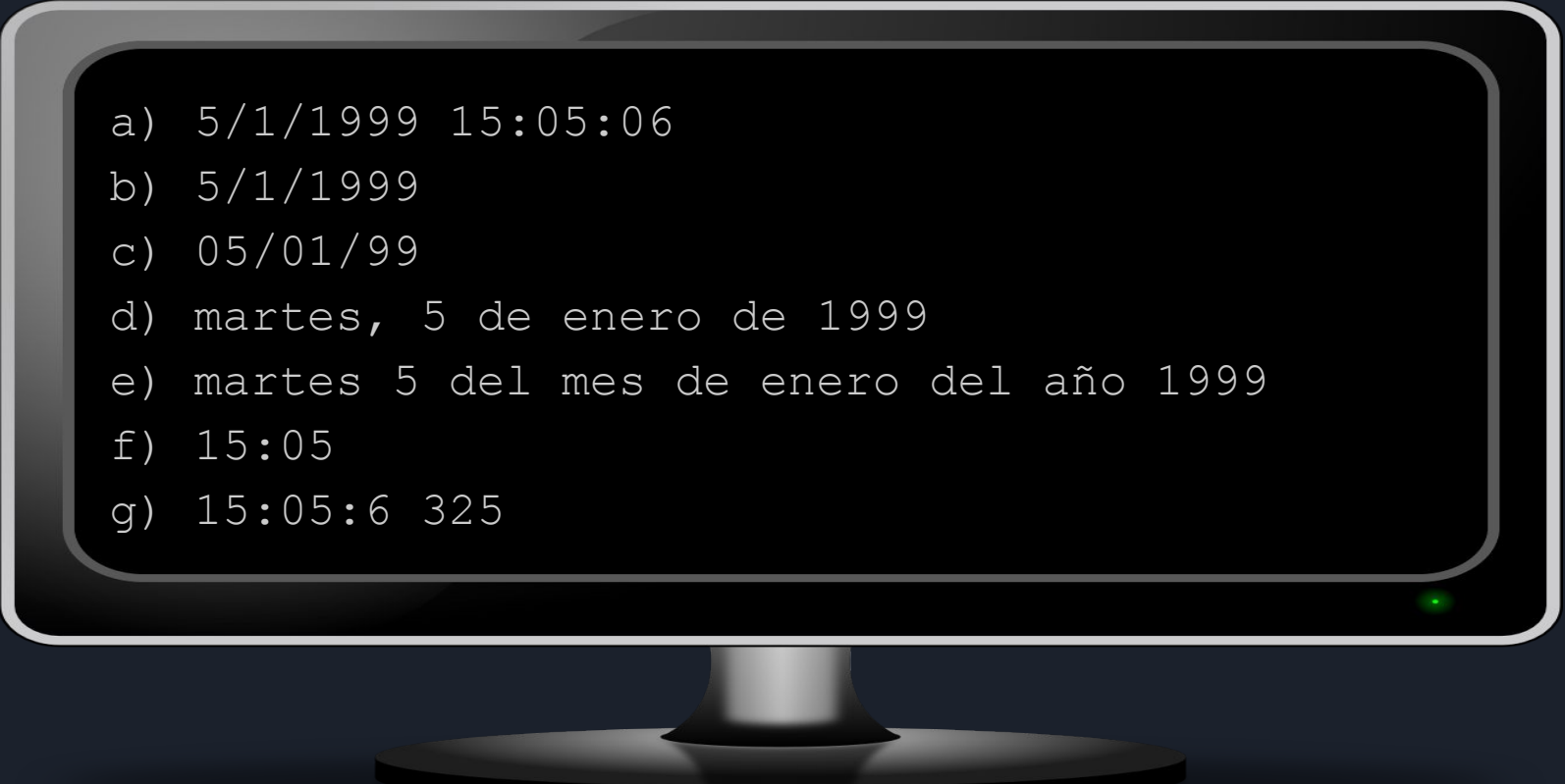
(año,mes,dia,hora,minutos,segundos,milisegundos)

```
DateTime fecha = new DateTime(1999,1,5,15,5,6,325);  
Console.WriteLine("a) {0}", fecha);  
Console.WriteLine("b) {0:d}", fecha);  
Console.WriteLine("c) {0:dd/MM/yy}", fecha);  
Console.WriteLine("d) {0:D}", fecha);  
Console.WriteLine($"e) {fecha:dddd d 'del mes de' MMMM 'del año' yyyy}");  
Console.WriteLine($"f) {fecha:t}");  
Console.WriteLine($"g) {fecha:HH:mm:s fff}");
```

`Console.WriteLine`  
también soporta strings  
de formato compuesto

## Strings de formato compuesto y strings interpolados

```
DateTime fecha = new DateTime(1999,1,5,15,5,6,325);  
Console.WriteLine("a) {0}", fecha);  
Console.WriteLine("b) {0:d}", fecha);  
Console.WriteLine("c) {0:dd/MM/yy}", fecha);  
Console.WriteLine("d) {0:D}", fecha);  
Console.WriteLine($"e) {fecha:dddd d 'del mes de' MMMM 'del año' yyyy}");  
Console.WriteLine($"f) {fecha:t}");  
Console.WriteLine($"g) {fecha:HH:mm:s fff}");
```



a) 5/1/1999 15:05:06  
b) 5/1/1999  
c) 05/01/99  
d) martes, 5 de enero de 1999  
e) martes 5 del mes de enero del año 1999  
f) 15:05  
g) 15:05:6 325

## Nota sobre formatos numéricos

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

Por ejemplo luego de hacer

```
double r = 2.417;
```

```
string st = r.ToString("0.00");
```

`st` queda asignado con el string `"2,42"`

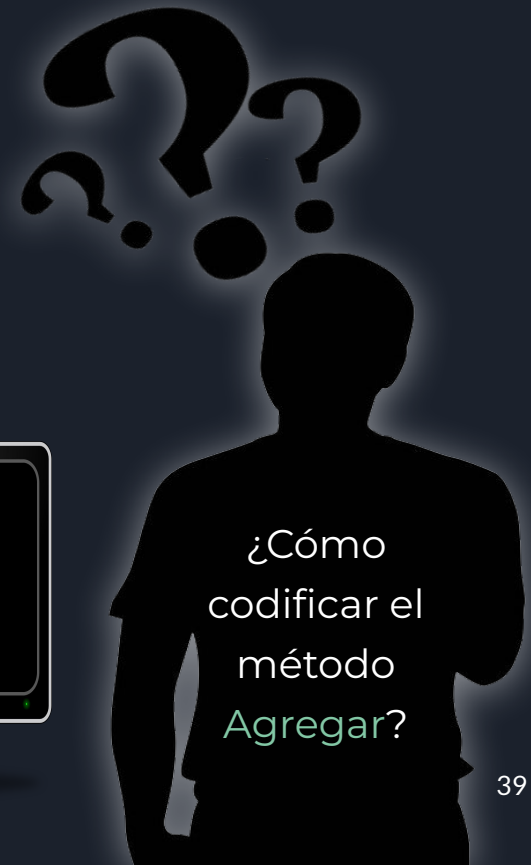


# Colecciones

## Para pensar

Los vectores en C# no pueden redimensionarse dinámicamente. Una vez instanciados queda fija su longitud hasta su destrucción. Sin embargo se podría hacer lo siguiente:

```
int[] vector = new int[0];  
for (int i = 1; i <= 5; i++)  
{  
    Agregar(ref vector, i);  
}  
foreach (int elem in vector)  
{  
    Console.WriteLine(elem);  
}
```



¿Cómo  
codificar el  
método  
Agregar?



## Posible solución

```
static void Agregar(ref int[] vector, int valor)
{
    int[] nuevo = new int[vector.Length + 1];
    for (int i = 0; i < vector.Length; i++)
    {
        nuevo[i] = vector[i];
    }
    nuevo[vector.Length] = valor;
    vector = nuevo;
}
```



## Posible solución

```
static void Agregar(ref int[] vector, int valor)
{
    int[] nuevo = new int[vector.Length + 1];
    for (int i = 0; i < vector.Length; i++)
    {
        nuevo[i] = vector[i];
    }
    nuevo[vector.Length] = valor;
    vector = nuevo;
}
```

En lugar de realizar la copia de esta manera, es más eficiente hacer:

**`Array.Copy(vector, nuevo, vector.Length);`**



## Dificultades con los arreglos

- Algunas funcionalidades no pueden resolverse con arreglos de manera conveniente. Por ejemplo:
  - 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

## Colecciones al rescate

- Las colecciones, al igual que los arreglos, gestionan un conjunto de elementos pero lo hacen de una manera especial.
- Se pueden considerar arreglos especializados.
- Existen distintos tipos de colecciones especializadas en distintas tareas.





## Colecciones genéricas

Entre las clases más significativas figuran:

- `List<T>`. Similar a un vector de elementos de tipo `T` pero con ciertas facilidades como por ejemplo ajustar su tamaño dinámicamente.
- `Stack<T>`. Pila de elementos de tipo `T`
- `Queue<T>`. Cola de elementos de tipo `T`
- `Dictionary<TKey,TValue>` Colección de pares (clave, valor). Recuperar un valor usando su clave es muy rápido, porque se implementa como una tabla hash.



## Ejercitando con List<int>



```
List<int> lista = new List<int>() { 10, 20, 30, 40 };  
for (int i = 0; i < lista.Count; i++)  
{  
    Console.WriteLine(lista[i]);  
}
```

```
List<int> lista = new List<int>() { 10, 20, 30, 40 };  
for (int i = 0; i < lista.Count; i++)  
{  
    Console.WriteLine(lista[i]);  
}
```



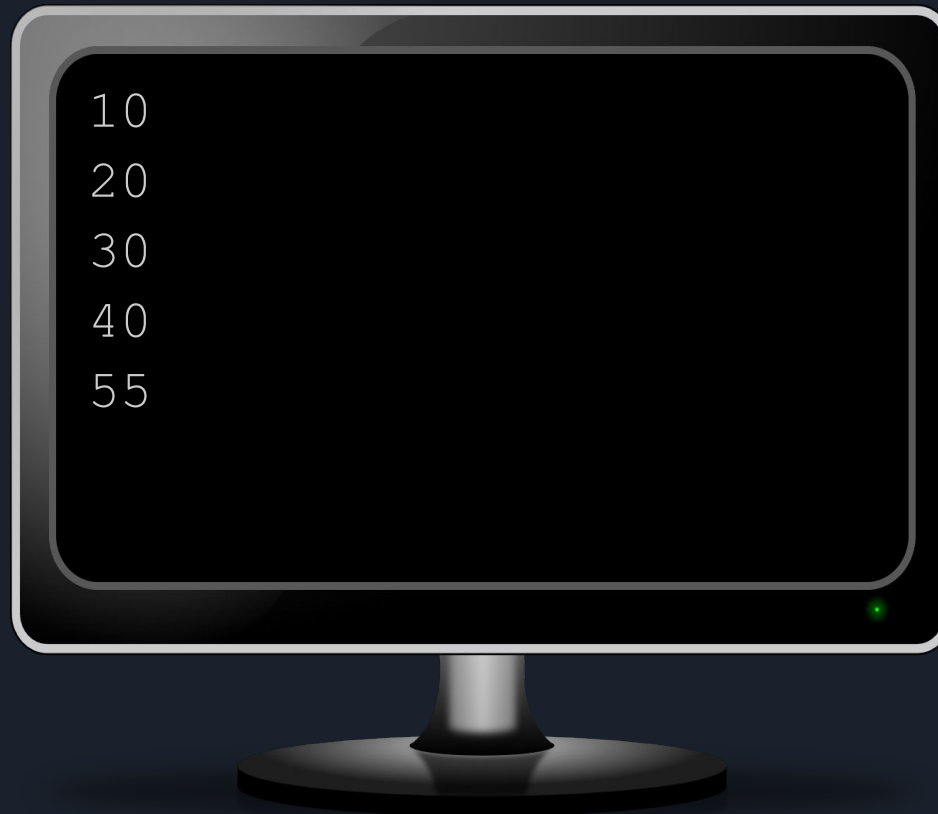


## Ejercitando con List<int>



```
List<int> lista = new List<int>() { 10, 20, 30, 40 };  
lista.Add(55);  
for (int i = 0; i < lista.Count; i++)  
{  
    Console.WriteLine(lista[i]);  
}
```

```
List<int> lista = new List<int>() { 10, 20, 30, 40 };  
lista.Add(55);  
for (int i = 0; i < lista.Count; i++)  
{  
    Console.WriteLine(lista[i]);  
}
```







## Ejercitando con List<int>



```
List<int> lista = new List<int>() { 10, 20, 30, 40 };  
lista.Remove(30);  
for (int i = 0; i < lista.Count; i++)  
{  
    Console.WriteLine(lista[i]);  
}
```

```
List<int> lista = new List<int>() { 10, 20, 30, 40 };  
lista.Remove(30);  
for (int i = 0; i < lista.Count; i++)  
{  
    Console.WriteLine(lista[i]);  
}
```





## Ejercitando con List<int>



```
List<int> lista = new List<int>() { 10, 20, 30, 40 };  
lista.RemoveAt(1);  
for (int i = 0; i < lista.Count; i++)  
{  
    Console.WriteLine(lista[i]);  
}
```

```
List<int> lista = new List<int>() { 10, 20, 30, 40 };  
lista.RemoveAt(1);  
for (int i = 0; i < lista.Count; i++)  
{  
    Console.WriteLine(lista[i]);  
}
```





## Ejercitando con List<int>



```
List<int> lista = new List<int>() { 10, 20, 30, 40 };  
lista.Insert(2,22);  
for (int i = 0; i < lista.Count; i++)  
{  
    Console.WriteLine(lista[i]);  
}
```

```
List<int> lista = new List<int>() { 10, 20, 30, 40 };  
lista.Insert(2,22);  
for (int i = 0; i < lista.Count; i++)  
{  
    Console.WriteLine(lista[i]);  
}
```





## Ejercitando con List<int>



```
List<int> lista = new List<int>() { 10, 20, 30, 40 };  
int[] vector = new int[] { 31, 32, 33 };  
lista.InsertRange(3,vector);  
for (int i = 0; i < lista.Count; i++)  
{  
    Console.WriteLine(lista[i]);  
}
```

```
List<int> lista = new List<int>() { 10, 20, 30, 40 };  
int[] vector = new int[] { 31, 32, 33 };  
lista.InsertRange(3,vector);  
for (int i = 0; i < lista.Count; i++)  
{  
    Console.WriteLine(lista[i]);  
}
```







## Stack<T>

- Es una pila de elementos de tipo `T`. Cuenta con métodos especializados:
- `Push(T elemento)`: Coloca el elemento indicado en el tope de la pila.
- `T Pop()`: Devuelve el elemento del tope de la pila y lo remueve.
- `T Peek()`: Devuelve el elemento del tope de la pila pero sin removerlo.

## Stack<T> - Ejemplo de uso

```
Stack<string> pila = new Stack<string>();  
pila.Push("Rojo");  
pila.Push("Verde");  
pila.Push("Azul");  
while (pila.Count > 0)  
{  
    Console.WriteLine(pila.Pop());  
}
```





## Queue<T>

- Es una cola de elementos de tipo **T**. Cuenta con métodos especializados:
- **Enqueue(T elemento)**: Coloca el objeto indicado al final de la cola.
- **T Dequeue()**: Devuelve el primer objeto de la cola y lo saca de ella.
- **T Peek()**: Devuelve el primer objeto de la cola pero no lo saca de ella.

## Queue<T> - Ejemplo de uso

```
Queue<char> cola = new Queue<char>();  
cola.Enqueue('A');  
cola.Enqueue('B');  
cola.Enqueue('C');  
while (cola.Count > 0)  
{  
    Console.WriteLine (cola.Dequeue());  
}
```



# Manejo de Excepciones



## Codificar y ejecutar



```
double[]? vector = new double[10];  
Procesar(vector, 1, 1);
```

```
void Procesar(double[]? v, int i, int c)  
{  
    c = c + 10;  
    v[i] = 1000 / c;  
    Console.WriteLine(v[i]);  
}
```



Código en el archivo  
03\_RecursosParaLaTeoria.txt



Agregar la instrucción resaltada y volver a ejecutar. ¿Qué es lo que ocurre?

```
double[]? vector = new double[10];
```

```
Procesar(vector, 1, 1);
```

→ 

```
Procesar(null, 1, 1);
```

```
void Procesar(double[]? v, int i, int c)
```

```
{
```

```
    c = c + 10;
```

```
    v[i] = 1000 / c;
```

```
    Console.WriteLine(v[i]);
```

```
}
```



# Se produce un error en tiempo de ejecución (Excepción)

C# Program.cs

```
1 double[]? vector = new double[10];  
2 Procesar(vector, 1, 1);  
3 Procesar(null, 1, 1);  
4  
5  
6 void Procesar(double[]? v, int i, int c)  
7 {  
8     c = c + 10;  
9     v[i] = 1000 / c;
```

**Exception has occurred: CLR/System.NullReferenceException** ✕

Excepción no controlada del tipo 'System.NullReferenceException' en nada2.dll: 'Object reference not set to an instance of an object.'

en Program.<<Main>>g\_\_Procesar|0\_0(Double[] v, Int32 i, Int32 c) en  
C:\Users\lccorbalan\proyectos\nada2\Program.cs: línea 9

en Program.<Main>\$(String[] args) en C:\Users\lccorbalan\proyectos\nada2\Program.cs: línea 3

```
10 Console.WriteLine(v[i]);  
11 }
```





### Codificar y ejecutar



Modificar el método **Procesar** utilizando una o más sentencias condicionales (**if**) para evitar que se produzca algún error en tiempo de ejecución debido a valores arbitrarios de los parámetros recibidos. En tal caso escribir en la consola "No procesado".

```
void Procesar(double[]? v, int i, int c)
{
    c = c + 10;
    v[i] = 1000 / c;
    Console.WriteLine(v[i]);
}
```

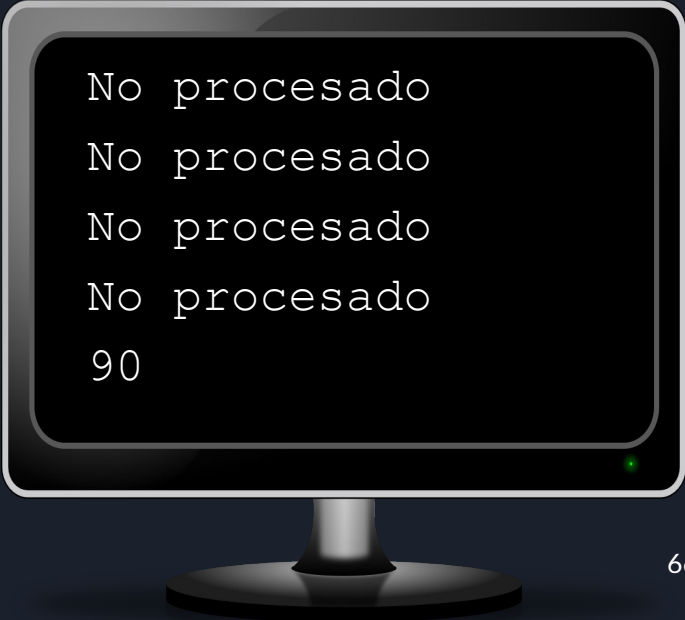


## Codificar y ejecutar



Verificar con estas invocaciones

```
double[]? vector = new double[10];  
Procesar(null, 1, 1);  
Procesar(vector, 10, 1);  
Procesar(vector, -1, 1);  
Procesar(vector, 1, -10);  
Procesar(vector, 1, 1);
```



```
No procesado  
No procesado  
No procesado  
No procesado  
90
```

Código en el archivo  
03\_RecursosParaLaTeoria.txt



## Codificar y ejecutar



Utilizar la instrucción `try ... catch`

```
void Procesar(double[]? v, int i, int c)
{
    try
    {
        c = c + 10;
        v[i] = 1000 / c;
        Console.WriteLine(v[i]);
    }
    catch
    {
        Console.WriteLine("No procesado");
    }
}
```



### Modificar y ejecutar



```
void Procesar(double[]? v, int i, int c)
{
    try
    {
        c = c + 10;
        v[i] = 1000 / c;
        Console.WriteLine(v[i]);
    }
    catch (Exception e)
    {
        Console.WriteLine(e.Message);
    }
}
```

```
void Procesar(double[]? v, int i, int c)
{
    try
    {
        c = c + 10;
        v[i] = 1000 / c;
        Console.WriteLine(v[i]);
    }
    catch (Exception e)
    {
        Console.WriteLine(e.Message);
    }
}
```

Object reference not set to an instance of an object.

Index was outside the bounds of the array.

Index was outside the bounds of the array.

Attempted to divide by zero.

90



# 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 comunes

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

... y muchas más

### Atención con el overflow !



Por defecto el chequeo de overflow **está deshabilitado**. El siguiente código no genera excepción alguna:


```
byte b = 255;  
b++;  
Console.WriteLine(b);
```





Se puede habilitar el chequeo de overflow desde el archivo de proyecto (extensión `.csproj`) en este caso `Teoria3.csproj`

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>net7.0</TargetFramework>
    <ImplicitUsings>enable</ImplicitUsings>
    <Nullable>enable</Nullable>
    <CheckForOverflowUnderflow>true</CheckForOverflowUnderflow>
  </PropertyGroup>
</Project>
```



Otra forma de habilitar el chequeo de **overflow** es directamente en el código **csharp**, utilizando un bloque **checked**.

```
byte b = 255;  
checked  
{  
    b++;  
}  
Console.WriteLine(b);
```

produce  
overflow



# Excepciones try/catch/finally

Boque try: Aquí dentro se controla la ocurrencia de excepciones

Cláusulas catch: Esta sección contiene manejadores para el caso de producirse excepciones en el bloque try

Bloque finally: Contiene código que se ejecuta siempre, se haya producido o no alguna excepción

```
try  
{  
    statements  
}
```

```
catch( ... )  
{  
    statements  
}  
catch( ... )  
{  
    statements  
}  
catch ...
```

```
finally  
{  
    statements  
}
```

} Esta sección es requerida

} Al menos una de estas secciones debe estar presente

# Excepciones

## Bloque catch

```
catch  
{  
    Statements  
}
```

### Cláusula catch general

- No lleva parámetro
- "Hace Matching" con cualquier tipo de excepción

```
catch( ExceptionType )  
{  
    Statements  
}
```

### Cláusula catch específica

- Toma como parámetro el nombre de una excepción
- "Hace Matching" con cualquier excepción de ese tipo


```
catch( ExceptionType ExceptionVariable )  
{  
    Statements  
}
```

### Cláusula catch específica con objeto

- Incluye un identificador luego del nombre de la excepción
- El identificador actúa como una variable local dentro del bloque catch



## Funcionamiento bloques catch

```
try
{
    object o = 3;
    int i = (int)o;
    Console.WriteLine("Sin error");
}
catch (InvalidCastException)
{
    Console.WriteLine("Error con el cast");
}
catch (DivideByZeroException)
{
    Console.WriteLine("división por cero");
}
catch
{
    Console.WriteLine("otra excepción");
}
Console.WriteLine("Continúa ejecución");
```



# Funcionamiento bloques catch

```
try
{
    object o = "hola";
    int i = (int)o;
    Console.WriteLine("Sin error");
}
catch (InvalidCastException)
{
    Console.WriteLine("Error con el cast");
}
catch (DivideByZeroException)
{
    Console.WriteLine("división por cero");
}
catch
{
    Console.WriteLine("otra excepción");
}
Console.WriteLine("Continúa ejecución");
```




Error con el cast  
Continúa ejecución



## Funcionamiento bloques catch

```
try
{
    int j = 0;
    int i = 1 / j;
    Console.WriteLine("Sin error");
}
catch (InvalidCastException)
{
    Console.WriteLine("Error con el cast");
}
catch (DivideByZeroException)
{
    Console.WriteLine("división por cero");
}
catch
{
    Console.WriteLine("otra excepción");
}
Console.WriteLine("Continúa ejecución");
```



división por cero  
Continúa ejecución

## Funcionamiento bloques catch

IndexOutOfRangeException

```
try
{
    int[] v = new int[] { 1, 2, 3 };
    int i = v[3];
    Console.WriteLine("Sin error");
}
catch (InvalidCastException)
{
    Console.WriteLine("Error con el cast");
}
catch (DivideByZeroException)
{
    Console.WriteLine("división por cero");
}
catch
{
    Console.WriteLine("otra excepción");
}

Console.WriteLine("Continúa ejecución");
```

otra excepción  
Continúa ejecución

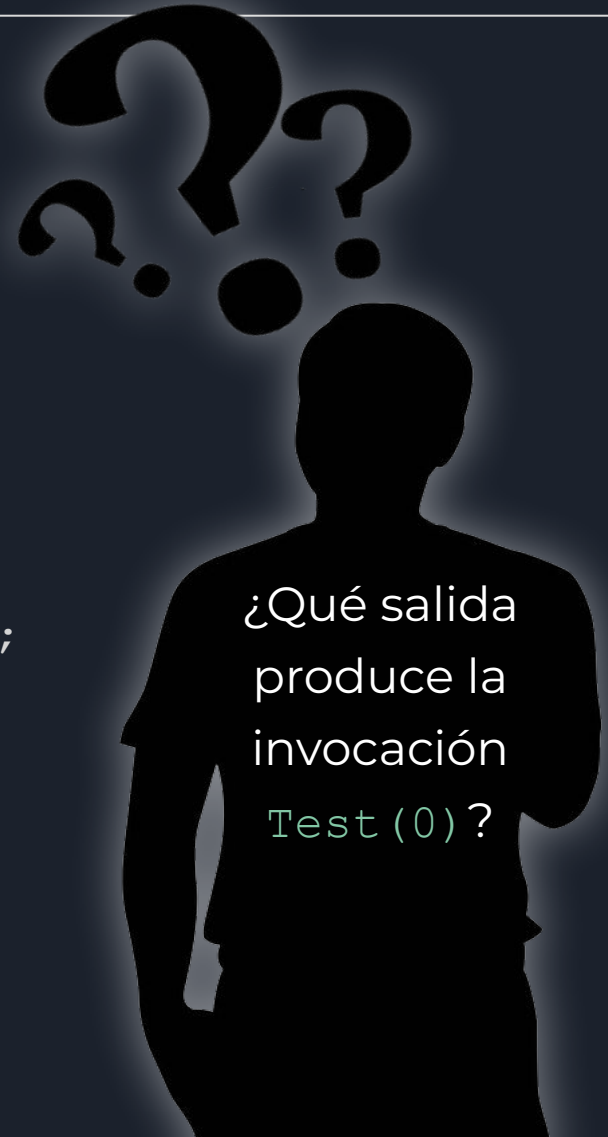


### Excepciones. Bloque finally

- 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 de los bloques `catch`


## Excepciones. Bloque finally

```
void Test(int x)
{
    try
    {
        x = 1 / x;
        return;
    }
    catch
    {
        Console.WriteLine("Excepción");
    }
    finally
    {
        Console.WriteLine("Finally");
    }
    Console.WriteLine("Continúa");
}
```



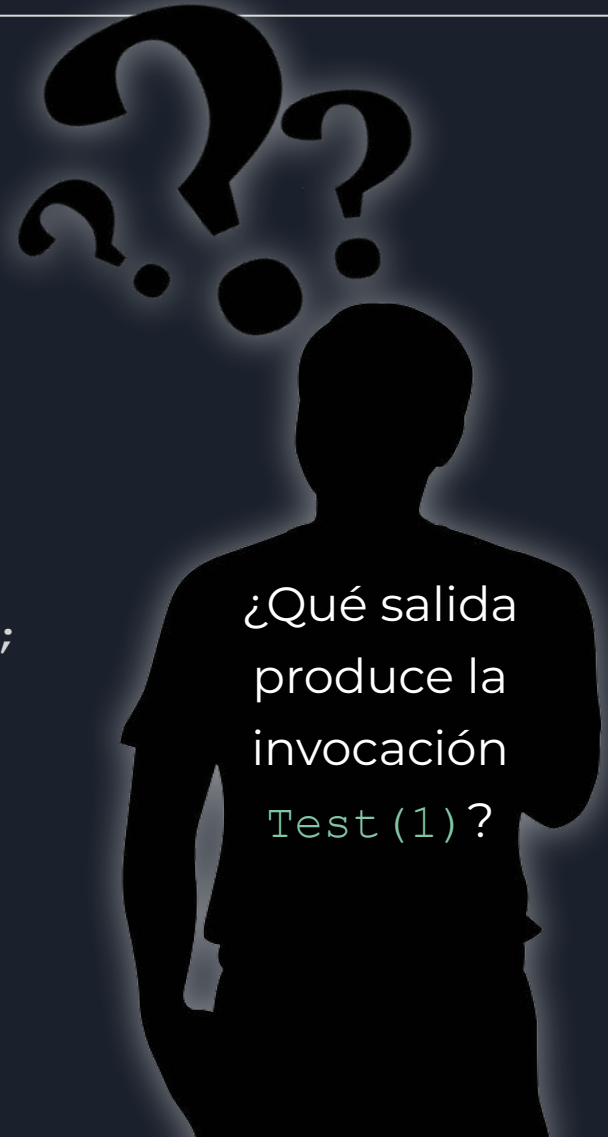
## Excepciones. Bloque finally

```
void Test(int x)
{
    try
    {
        x = 1 / x;
        return;
    }
    catch
    {
        Console.WriteLine("Excepción");
    }
    finally
    {
        Console.WriteLine("Finally");
    }
    Console.WriteLine("Continúa");
}
```



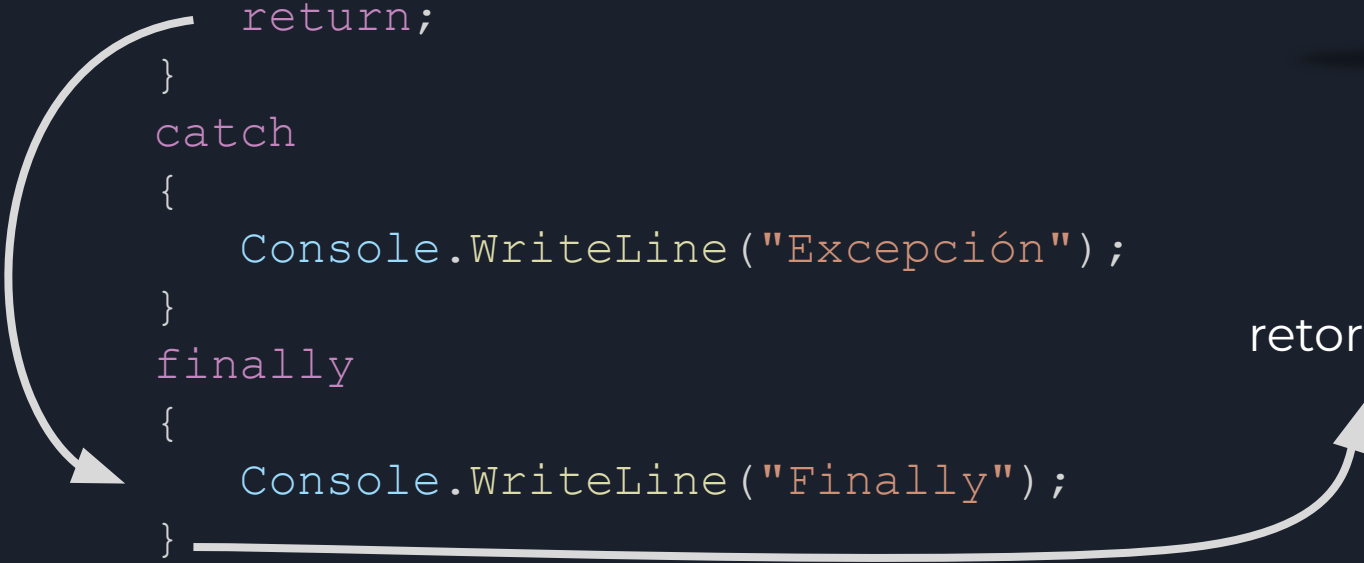
## Excepciones. Bloque `finally`

```
void Test(int x)
{
    try
    {
        x = 1 / x;
        return;
    }
    catch
    {
        Console.WriteLine("Excepción");
    }
    finally
    {
        Console.WriteLine("Finally");
    }
    Console.WriteLine("Continúa");
}
```



## Excepciones. Bloque **finally**


```
static void Test(int x)
{
    try
    {
        x = 1 / x;
        return;
    }
    catch
    {
        Console.WriteLine("Excepción");
    }
    finally
    {
        Console.WriteLine("Finally");
    }
    Console.WriteLine("Continúa");
}
```



# 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`

# Propagación de excepciones



```
void Metodo1()  
{  
    try  
    {  
        Metodo2();  
    }  
    catch  
    {  
        Console.WriteLine("Metodo2 generó excepción");  
    }  
    Console.WriteLine("fin Metodo1");  
}  
  
void Metodo2()  
{  
    object o = "hola";  
    int i = (int)o;  
    Console.WriteLine("fin Metodo2");  
}
```

# Propagación de excepciones

```
void Metodo1()  
{  
    try  
    {  
        Metodo2();  
    }  
    catch  
    {  
        Console.WriteLine("Metodo2 generó excepción");  
    }  
    Console.WriteLine("fin Metodo1");  
}  
  
void Metodo2()  
{  
    object o = "hola";  
    int i = (int)o;  
    Console.WriteLine("fin Metodo2");  
}
```

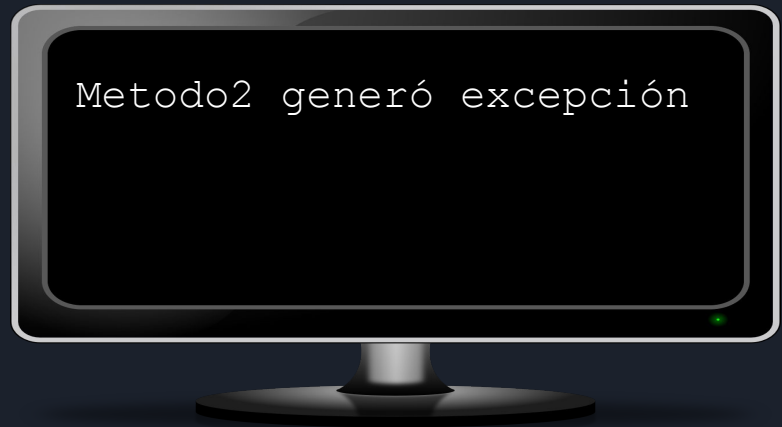

The diagram illustrates the flow of an exception. A curved arrow originates from the `try` block of `Metodo2()` and points to the `catch` block of `Metodo1()`, indicating that `Metodo1()` handles the exception. A second, larger curved arrow originates from the `catch` block of `Metodo1()` and points to the text "propaga excepción al invocador", indicating that the exception is propagated out of `Metodo1()` to its caller.



## Propagación de excepciones

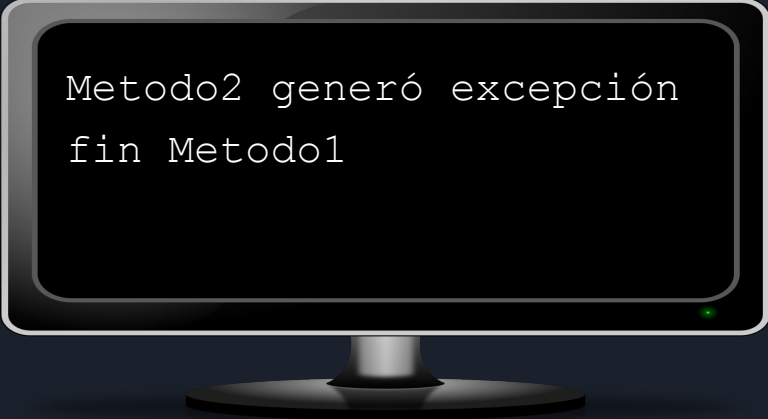
```
void Metodo1()  
{  
    try  
    {  
        Metodo2();  
    }  
    catch  
    {  
        Console.WriteLine("Metodo2 generó excepción");  
    }  
    Console.WriteLine("fin Metodo1");  
}  
  
void Metodo2()  
{  
    object o = "hola";  
    int i = (int)o;  
    Console.WriteLine("fin Metodo2");  
}
```

excepción



## Propagación de excepciones

```
void Metodo1()  
{  
    try  
    {  
        Metodo2();  
    }  
    catch  
    {  
        Console.WriteLine("Metodo2 generó excepción");  
    }  
    Console.WriteLine("fin Metodo1");  
}
```



Metodo2 generó excepción  
fin Metodo1

```
void Metodo2()  
{  
    object o = "hola";  
    int i = (int)o;  
    Console.WriteLine("fin Metodo2");  
}
```

```
void Metodo1 ()
{
    try
    {
        Metodo2 ();
    }
    catch
    {
        Console.WriteLine("Metodo2 genero excepción");
    }
    Console.WriteLine("fin Metodo1");
}
```

```
void Metodo2 ()
{
    try
    {
        object o = "hola";
        int i = (int)o;
    }
    finally
    {
        Console.WriteLine("finally Metodo2");
    }
    Console.WriteLine("fin Metodo2");
}
```

Atención!  
*Finally* no  
maneja la  
excepción



```
finally Metodo2
Metodo2 generó excepción
fin Metodo1
```

# 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

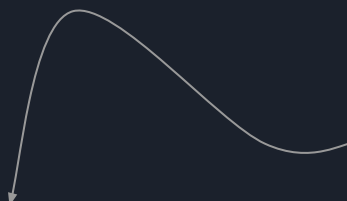
# Lanzar una excepción

```
try
{
    object o = 7;
    string? st = o as string;
    Imprimir(st);
}
catch (ArgumentNullException e)
{
    Console.WriteLine(e.Message);
}

void Imprimir(string? st)
{
    if (st == null)
    {
        throw new ArgumentNullException("st");
    }
    Console.WriteLine(st);
}
```




Si `st` es `null` se crea un objeto `ArgumentNullException` y se lanza esta excepción

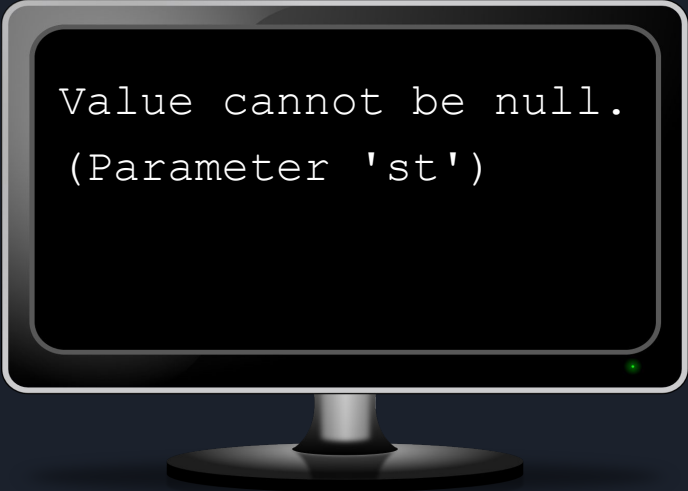


# Lanzar una excepción

```
try
{
    object o = 7;
    string? st = o as string;
    Imprimir(st);
}
catch (ArgumentNullException e)
{
    Console.WriteLine(e.Message);
}
```



```
void Imprimir(string? st)
{
    if (st == null)
    {
        throw new ArgumentNullException("st");
    }
    Console.WriteLine(st);
}
```



Value cannot be null.  
(Parameter 'st')

### Lanzar una excepción


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");
```

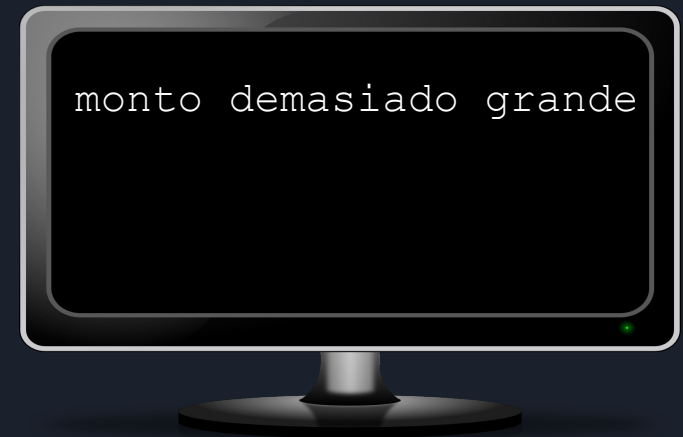
En la siguiente diapositiva se implementa un método extraer que lanza una excepción si el monto recibido como parámetro supera el valor de 1000

## Lanzar una excepción

```
try
{
    Extraer(2000);
}
catch (Exception e)
{
    Console.WriteLine(e.Message);
}
```



```
void Extraer(int monto)
{
    if (monto > 1000)
    {
        throw new Exception("monto demasiado grande");
    }
    // acá se procede con la extracción
}
```






## Relanzar una excepción

```
try {
    Metodo1();
} catch {
    Console.WriteLine("catch en Main");
}
```

```
void Metodo1() {
    try {
        throw new Exception();
    } catch {
        Console.WriteLine("catch en Metodo1");
        throw;
    }
}
```

Relanza la excepción que se propaga al método que invocó a `Metodo1`



```
catch en Metodo1
catch en Main
```

Fin  
de la teoría 3