



# TIPOS ANÓNIMOS EN C#



## ASIGNACIÓN IMPLÍCITA DE TIPOS

- La palabra clave **var** indica al compilador que deduzca el tipo de la variable a partir de la expresión que se encuentra en el lado derecho.
- Una vez deducido y no le podremos asignar cosas de otro tipo.

```
var i = 5;           // i es compilada como entero
var s = "Hola";      // s como string
var a = new[] { 0, 1, 2 }; // a como int[]
var list = new List<int>(); //list será List<int>
```

### ¿Por Qué Usar Este Tipo Si Podemos Especificar Uno?

- Existirán unos objetos especiales sin definición de clase que tendrán un **tipo anónimo** y por tanto no lo podremos poner en tiempo de compilación.
- Una forma de obtener objetos con resultados intermedios sin estado o una **Snapshot** (Instantánea) de datos de un objeto.



## TIPOS ANÓNIMOS INMUTABLES - I

- Los tipos anónimos son una manera de encapsular un conjunto de propiedades de solo lectura en un único objeto sin tener que definir un tipo.
- El compilador genera el **nombre del tipo no disponible** en el nivel de código fuente
- No son válidos ningún otro tipo de miembros de clase, como métodos o eventos.
- La expresión que se usa para inicializar una propiedad no puede ser null, una función anónima o un tipo de puntero.

### Ejemplo

- En el ejemplo siguiente se muestra dos tipo anónimo que se inicializa con una propiedad Name el primero y dos propiedades Nombre y Edad el segundo.

```
var estudianteDesconocido = new { Name = "Rigoberto" };  
var estudianteDesconocido2 = new { Nombre = "Pedro", Edad = "12" };
```



## TIPOS ANÓNIMOS INMUTABLES – II

- Podemos ahorrarnos el indicar los nombres de las propiedades directamente si utilizamos variables para inicializar el tipo anónimo:

### Ejemplo

```
double x = 9.1;  
float y = 3.2;  
var point1 = new { x, y };  
var point2 = new { x, SuperY = y };
```

- La variable point1 tendrá una propiedad llamada **X** de tipo double y otra llamada **Y** del tipo float. Inclusive podemos combinar las formas de inicializar, en este caso point2 tiene como propiedades X y SuperY.



## TIPOS ANÓNIMOS INMUTABLES – III

- Se puede crear una matriz de elementos con tipo anónimo combinando una variable local con tipo implícito y una matriz con tipo implícito.

### Ejemplo

```
var fruitSize = new[]  
{  
    new { Name = "Apple", Diameter = 4 },  
    new { Name = "Grape", Diameter = 1 }  
};
```

- En este caso hemos creados dos objetos anónimos con el mismo nombre de propiedades y del mismo tipo. **Internamente el CLR combinará ambos tipos anónimos creados.**



## TIPOS ANÓNIMOS INMUTABLES – IV

- Una de las restricciones más notables de los tipos anónimos es que su uso está limitado a hacerse dentro del cuerpo de un método.
- No pueden, por tanto declararse como anónimos:
  - Campos privados.
  - Tipos devueltos por métodos.
  - Propiedades.
  - Parámetros formales de métodos.
  - Eventos.
- Puesto que heredan de la clase object pueden mostrarse con `string ToString()` y compararse con `bool Equals(object obj)` que considerará iguales aquellos métodos anónimos que:
  - Las **mismas propiedades, en nombre y número.**
  - El **mismo orden** de declaración de las propiedades.
  - Los **mismos valores** para esas propiedades.



## INSTANCIANDO OBJETOS TIPADOS CON LA MISMA SINTÁXIS

- Podemos usar una sintaxis análoga para instanciar objetos tipados mutables, sin haber definido ningún constructor.
- En el siguiente ejemplo definimos la clase Persona y un constructor por defecto. Pero a través de sus propiedades auto-implementadas definidas podremos personas en una llista.

```
enum Sexo { Mujer, Varón };

class Persona
{
    public string Nombre { get; set; }
    public Sexo Sexo { get; set; }
    public stringCodigoPaís { get; set; }
}

List<Persona> Personas = new List<Persona> {
    new Persona { Nombre="Diana", Sexo=Sexo.Mujer, CodigoPaís="ES" },
    new Persona { Nombre="Juana", Sexo=Sexo.Mujer, CodigoPaís="RU" },
    new Persona { Nombre="Darío", Sexo=Sexo.Varón, CodigoPaís="CU" },
    new Persona { Nombre="Jenny", Sexo=Sexo.Mujer, CodigoPaís="CU" },
};
```



# MÉTODOS EXTENSORES





## MÉTODOS EXTENSORES EN C#

### Definición

- Los métodos de extensión permiten "agregar" operaciones sobre los tipos existentes sin crear un nuevo tipo derivado ni modificar el original.
- Los métodos de extensión son una clase especial de método estático, pero se les llama como si fueran métodos de instancia en el tipo extendido.
- Debemos usarlos poco y siempre que no sea posible realizar la extensión a través del mecanismo de herencia.
- No tendré acceso a los miembros privados del tipo extendido.

### Sintaxis

```
namespace <Tipo>Extensions {  
    public static class <Tipo>Extension {  
        public static void IdMétodoExtensor(this <Tipo> o) {  
            // Operaciones sobre o.  
        }  
    }  
}
```



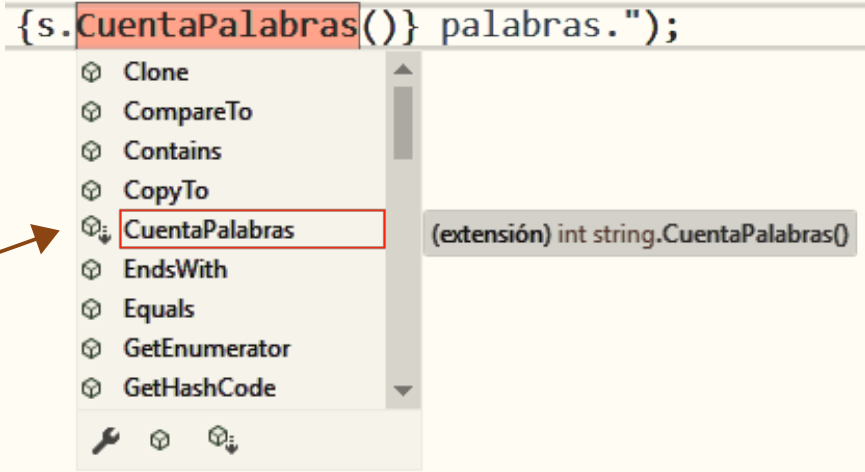
# MÉTODOS EXTENSORES EN C#

## Ejemplo

```
using System;
using StringExtensions;

namespace StringExtensions {
    public static class StringExtension {
        public static int CuentaPalabras(this string str)
        {
            return str.Split(
                new char[] { ' ', '.', '?' },
                StringSplitOptions.RemoveEmptyEntries).Length;
        }
    }
}

class Ejemplo {
    static void Main() {
        string s = "Hola caracola";
        Console.WriteLine($"{s}
        tiene {s.CuentaPalabras()}
        palabras.");
    }
}
```





# **RECORRIDO PEREZOSO DE SECUENCIAS**



## RECORRIDO PEREZOSO DE SECUENCIAS

### **IEnumerable<T>**

- Dispone de numerosos métodos de extensión pero para los ejemplos posteriores usaremos especialmente...
  - De instancia, el **ToList()** que convertirá un objeto enumerable en un List<T>
  - De clase, el **Range(int start, int count)** que genera una secuencia empezando en start, con count elementos.

```
List<int> sec = Enumerable.Range(2, 4).ToList();  
// Equivale a ...  
List<int> sec = new List<int> { 2, 3, 4, 5 };
```

### **List<T>**

- Recorrer un objeto lista con public void **ForEach(Action<T> action);**

```
List<int> sec = Enumerable.Range(0, 10).ToList();  
sec.ForEach(n => Console.WriteLine($"{n} "));  
// Mostrará por pantalla: 0 1 2 3 4 5 6 7 8 9
```



## RECORRIDO PEREZOSO DE DECUENCIAS

### La Palabra Reservada **Yield** - I

- Una posible traducción del inglés como verbo sería “**producir**”.
- La vamos a encontrar en otros lenguajes de scripting como: JavaScript, Php, Python, Scala o Ruby.
- Nos permite generar una secuencia enumerable sin implementar `IEnumerable<T>` ni por ende `IEnumerator<T>`
- Nos ayudará a implementar una especie de “**lazy loading**” en nuestro código introduciendo saltos entre un método y quien lo llamó para evitar así desperdiciar memoria en flujos de datos de tamaño medio grande.

### Sintaxis

```
yield return <expression>;  
yield break;
```



## RECORRIDO PEREZOSO DE DECUENCIAS

### La Palabra Reservada **yield** - I

#### Ejemplo

Supongamos el siguiente método **sin** yield:

```
// Devolverá una secuencia resultado de llenar
// una lista con los múltiplos de n entre ini y fin
static IEnumerable<int> ObtieneMultiplosDeN(int n, int ini, int fin) {
    List<int> multiplos = new List<int>();
    for (int i = ini; i < fin; i++) {
        if (i % n == 0) {
            Console.WriteLine($"Obtenido {i}");
            multiplos.Add(i);
        }
    }
    return multiplos;
}

// Vamos a obtener el 4º múltiplo de 2 entre 320 y 335
int cuartoMultobt = ObtieneMultiplosDeN(2, 320, 335).Skip(3).First();
Console.WriteLine($"El 4to multiplo es {cuartoMultobt}");
```



## RECORRIDO PEREZOSO DE DECUENCIAS

### La Palabra Reservada **Yield** - I

```
List<int> multimplos = new List<int>();  
for (int i = ini; i < fin; i++)  
{  
    if (i % n == 0)  
    {  
        Console.WriteLine($"Obtenido {i}");  
        multimplos.Add(i);  
    }  
}  
return multimplos;
```

2º realizamos **n**  
inserciones en  
la colección.

1º Llamamos  
al método.

ObtieneMultiplosDeN

3º un único  
return con  
toda la  
secuencia.

```
Obtenido 320  
Obtenido 322  
Obtenido 324  
Obtenido 326  
Obtenido 328  
Obtenido 330  
Obtenido 332  
Obtenido 334  
El 4to multiplo es 326
```



## RECORRIDO PEREZOSO DE DECUENCIAS

### La Palabra Reservada **yield** - I

Supongamos el siguiente método **con** yield:

```
// Devolverá una secuencia resultado de llenar
// una lista con los múltiplos de n entre ini y fin
static IEnumerable<int> ProduceMultiplosDeN(int n, int ini, int fin) {
    for (int i = ini; i < fin; i++) {
        if (i % n == 0) {
            Console.WriteLine($"Producido {i}");
            yield return i;
        }
    }
}

// Vamos a obtener el 4º múltiplo de 2 entre 320 y 335
int cuartoMultProd = ProduceMultiplosDeN(2, 320, 335).Skip(3).First();
Console.WriteLine($"El 4to multiplo es {cuartoMultProd}");
```





## RECORRIDO PEREZOSO DE DECUENCIAS

### La Palabra Reservada **Yield** - I

```
for (int i = ini; i < fin; i++)  
{  
    if (i % n == 0)  
    {  
        Console.WriteLine($"Obtenido {i}");  
        yield return i;  
    }  
}
```

1º Llamamos  
al método.

ProduceMultiplosDeN

2º produce cada elemento de la  
secuencia conforme se recorre.

3º salto de vuelta  
para producir un  
nuevo elemento.

```
Producido 320  
Producido 322  
Producido 324  
Producido 326  
El 4to multiplo es 326
```



## RECORRIDO PEREZOSO DE DECUENCIAS

### La Palabra Reservada **Yield** - I

- Si nos fijamos en los ejemplos...
  - En la implementación **sin yield**, debemos generar primero toda la secuencia para luego realizar la operación sobre la misma.
  - En la implementación **con yield**, realizamos la operación conforme generamos la secuencia.
- Esto último nos ahorrará bastante memoria e iteraciones si la secuencia tiene muchos elementos.



## RECORRIDO PEREZOSO DE DECUENCIAS

### La Palabra Reservada **Yield** – Múltiple Returns

```
public class Galaxy {
    public String Name { get; set; }
    public int DistanceLY { get; set; }
}

public class Galaxies {
    public IEnumerable<Galaxy> NextGalaxy {
        get {
            yield return new Galaxy { Name = "Tadpole", DistanceLY = 400 };
            yield return new Galaxy { Name = "Pinwheel", DistanceLY = 25 };
            yield return new Galaxy { Name = "Milky Way", DistanceLY = 0 };
            yield return new Galaxy { Name = "Andromeda", DistanceLY = 3 };
        }
    }
}

public static void Main()
{
    var theGalaxies = new Galaxies();
    foreach (Galaxy theGalaxy in theGalaxies.NextGalaxy) {
        Console.WriteLine($"{theGalaxy.Name} {theGalaxy.DistanceLY}");
    }
}
```