

Tema 11.3

Descargar estos apuntes en [pdf](#) o [html](#)

Índice

1. [Funcionalidades Interesantes del Lenguaje C#](#)
2. [Tipos Anónimos Inmutables](#)
 1. [Extendiendo al sintaxis a los tipos conocidos](#)
 2. [Métodos de Extensión](#)
 3. [Recorrido Perezoso De Secuencias](#)
 1. [Casos de uso de los generadores\)](#)
 2. [Sintaxis y ejemplo de uso de yield](#)

Funcionalidades Interesantes del Lenguaje C#

Tipos Anónimos Inmutables

Definición: 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**. Para ello, el compilador genera el nombre del tipo transparente para el programador y por tanto no disponible en el nivel de código fuente.

Restricciones: Solo podremos definir propiedades y por tanto, no son válidos ningún otro tipo de miembros de clase, como métodos o eventos. Además, la expresión que se usa para inicializar una propiedad no puede ser `null`.

En el ejemplo siguiente se muestra dos tipos 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" };
```

Podemos ahorrarnos el indicar los nombres de las propiedades si utilizamos **identificadores de variables** para inicializar el tipo anónimo ...

```
double X = 9.1;
float Y = 3.2;

// La variable point1 tendrá una propiedad llamada X de tipo double y
// otra llamada Y del tipo float.
var point1 = new { X, Y };

// Incluso podemos combinar las formas de inicializar,
// en este el siguiente caso point2 tiene como propiedades X y SuperY.
var point2 = new { X, SuperY = Y };
```

También se puede definir un array de elementos con tipo anónimo, combinando una variable local con tipo implícito y una matriz con tipo implícito. Por ejemplo ...

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

👉 **Importante:** En este caso hemos creado **dos objetos anónimos con el mismo nombre de propiedades** y por tanto del '*mismo tipo*'. Esto será detectado internamente por el Runtime que **combinará ambos tipos anónimos creados**, en uno solo.

Aunque aún no vamos a ver ejemplos de uso significativos. Más adelante, en el tema, los usaremos para obtener instantáneas de datos agrupados como resultado de consultas. Algo así como la obtención de un **snapshot** tras hacer un `select` en SQL. Por esa razón, 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 declararse como **tipos anónimos**:

- Campos privados.
- Propiedades.
- Eventos.
- Tipos devueltos por métodos.
- Parámetros formales de métodos.

Al compararse con `bool Equals(object obj)` **se considerará iguales** aquellos métodos anónimos que tengan:

- Las mismas propiedades, en nombre y número.
- El mismo orden de declaración de las propiedades.
- Los mismos valores para esas propiedades.

Por último, puesto que heredan de la clase `object` podrán mostrarse `string ToString()` por ejemplo...

```
var anonimo = new { Nombre = "María", Edad = 23 };  
// Mostrará "{ Nombre = \"María\", Edad = 23 }" sin necesidad de invalidar ToString().  
Console.WriteLine(anonimo);
```

Extendiendo al sintaxis a los tipos conocidos

Podemos usar una sintaxis análoga para instanciar objetos tipados, sin haber definido ningún constructor.

En el siguiente ejemplo definimos la clase `Persona` con solo el constructor por defecto.

```
enum Sexo { Mujer, Varón };
class Persona
{
    public string Nombre { get; set; }
    public Sexo Sexo { get; set; }
    public string CodigoPais { get; set; }
}
```

Pero a través de sus propiedades auto-implementadas definidas podremos definir una lista de personas por extensión, usando la sintaxis para definir tipos anónimos y que también podremos encontrar de forma similar en otros lenguajes de programación.

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

Fíjate que el tipo Persona es mutable que **todos los setters auto-implementados son publicos**. Pero como aprendimos, esto es peligroso porque perdemos la encapsulación de nuestras clases.

Para este tipo de inicializaciones, no nos valdría definir la propiedades de solo lectura **ej.**

`public string Nombre { get; }`. Esto nos permitía asignarle valor solo en el constructor, pero no usar la sintaxis anterior.


Por esta razón desde **C#9** además de `get` y `set`, tendremos la palabra reservada `init` para definir propiedades y que será equivalente a `set` pero solo nos permitirá asignarle valor una vez y por tanto sería otra forma de definirla como de solo lectura. Siendo más apropiado definir nuestras propiedades de solo lectura de esta forma `public string Nombre { get; init; }` a partir de C#9.

```
enum Sexo { Mujer, Varón };
class Persona
{
    public string Nombre { get; init; }
    public Sexo Sexo { get; init; }
    public string CodigoPais { get; init; }
}
```

De esta manera, ya no podríamos modificar el estado de los objetos de la lista y podríamos preservar su encapsulación aunque no podremos asegurarnos 100% de que el estado con el que se inicializan, es del todo correcto si las dejamos auto-implementadas.

Métodos de Extensión

Funcionalidad **interesantísima** de C# para extender la funcionalidad en clases selladas o de las que no disponemos el código porque es una librería de terceros, incluso para cumplir OCP (Open Closed Principle) de SOLID en nuestras propias implementaciones si usamos [arquitecturas de cebolla](#). De hecho, otros lenguajes modernos como **Kotlin** también los permiten. Pero dejando a un lado consideraciones complejas y de diseño, en este tema vamos definir simplemente el concepto y su sintaxis en C#.

 **Importante:** Desde Microsoft, [se recomienda no abusar de este tipos de métodos](#), y por tanto usarlos siempre que no sea posible realizar la extensión de dichas clases a través de un mecanismo de herencia o composición.

Características básicas de los métodos de extensión:

- Me permiten '*agregar*' operaciones sobre los tipos existentes, sin crear un nuevo tipo derivado y sin modificar el original.
- Se definen de forma especial a través de un método estático, pero se les llama como si fueran métodos de instancia en el tipo extendido.
- No tendré acceso a los miembros privados del tipo extendido.

Una propuesta de plantilla básica de sintaxis de definición de estos métodos podría ser...

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

Veámoslo a través de un ejemplo sencillo pero bastante '*esclarecedor*'...

Supongamos que queremos añadir métodos de utilidad sobre objetos cadena **string** que nos proporcionan las BCL. Sin embargo, nosotros no podemos modificar la implementación en la clase **string** para añadir nuevas operaciones.

Crearemos un fuente llamado **StringExtension.cs** que contendrá la clase estática **StringExtension** donde añadiremos todos los métodos de extensión sobre **string**.

En el siguiente ejemplo hemos añadido el método **Capitaliza** que pasa a mayúsculas la primera letra de cada palabra y el método **CuentaPalabras** que me retorna el número de palabras en una cadena.



```
1 namespace StringExtensions
{
3     public static class StringExtension
    {
5         public static string Capitaliza(this string s)
        {
            string sCapitalizada;
            if (!string.IsNullOrEmpty(s))
            {
                StringBuilder sb = new StringBuilder(s);
                sb[0] = char.ToUpper(sb[0]);
                for (int i = 1; i < s.Length; i++)
                    sb[i] = char.IsWhiteSpace(sb[i - 1]) ? char.ToUpper(sb[i]) : sb[i];
                sCapitalizada = sb.ToString();
            }
            else
                sCapitalizada = s;
            return sCapitalizada;
        }

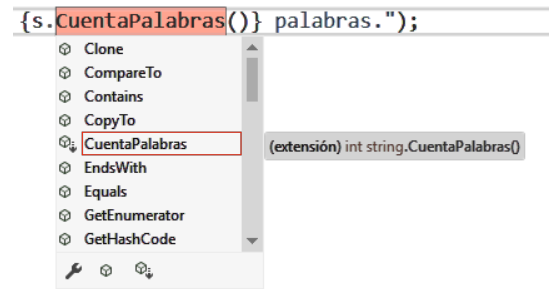
21        public static int CuentaPalabras(this string s) => s.Split(
                                new char[] { ' ', '.', '?' },
                                StringSplitOptions.RemoveEmptyEntries
                            ).Length;
    }
}
```

Ahora si quisiéramos utilizar estos métodos adicionales en un programa simplemente tendríamos que hacer un **using StringExtensions;** para que nos los ofrezca el '*IntelliSense*' al ver las la operaciones posibles sobre un objeto cadena.

```
1 using StringExtensions;

class Ejemplo
{
    static void Main()
    {
        string s = "hola caracola";
8        // Si no hacemos el using StringExtensions; los métodos Capitaliza
        // y CuentaPalabras no nos los ofrecerá.
10       Console.WriteLine($"{s.Capitaliza()} tiene { s.CuentaPalabras()} palabras.");
    }
}
```

💡 **Tip:** Dependiendo del IDE que estemos usando, el Intellisense normalmente nos ofrece un símbolo (ej. ) al lado del método público de la clase y si este es un método de extensión lo indicará con una flecha hacia abajo (ej. ) y/o la etiqueta **(Extensión)** precediendo a la descripción del método.



Un ejemplo de uso de métodos extensores que veremos más adelante y es muy utilizado en C# son los definidos en el espacio de nombres **System.Linq**. Estos extienden todas las operaciones definidas en **IEnumerable<T>** y por ende todas las operaciones que podremos realizar sobre nuestras colecciones.

Caso de estudio:

Prueba a copiar el siguiente programa ...

```
using System.Collections.Generic;

class CasoDeEstudio
{
    static void Main()
    {
        List<int> numeros = new List<int> { 9, 5, 7, 3, 1, 5 };
        numeros.
    }
}
```

Fíjate en las operaciones disponibles que tienes sobre el objeto **numeros**. Ahora incluiremos las definiciones en espacio de nombres **System.Linq**

```
using System.Collections.Generic;
using System.Linq;

...
```

Fíjate en las operaciones disponibles que tienes sobre el objeto **numeros**. Deberán aparecerte muchísimas más operaciones comunes a secuencias de datos. Precedidas todas por la etiqueta **(Extensión)**

Recorrido Perezoso De Secuencias

📌 **Nota:** Antes de hablar del recorrido perezoso de secuencias (`IEnumerable<T>`), vamos a comentar alguno de los métodos de extensión que proporciona System.Linq y que usaremos en algún ejemplo más adelante...

1. Método de instancia `secuencia.ToList()` que convertirá un objeto `IEnumerable<T>` en una lista `List<T>`
2. Método de clase, `Enumerable.Range(int start, int count)` que genera una secuencia de enteros empezando en `start` , con `count` elementos.
3. Método de instancia, `secuencia.Skip(int count)` que devolverá la secuencia resultante de haber saltado `count` posiciones en el objeto secuencia al que aplicamos la operación.
4. Método de instancia, `secuencia.First()` que devolverá el primer elemento del objeto secuencia al que aplicamos la operación.

```
1 List<int> secuencia = Enumerable.Range(2, 4).ToList();  
  // Equivale a ...  
  List<int> secuencia = new List<int> { 2, 3, 4, 5 };  
  
5 secuencia.Skip(2).First(); // Se evaluará al entero 4
```

Para hablar del recorrido '*perezoso*' de una secuencia, deberemos hablar de la palabra reservada `yield` . Es interesante indicar que una posible traducción del inglés de **to yield** como verbo sería '**generar o producir**' ya que más que de recorrido perezoso deberíamos hablar de '**generación perezosa de datos en una secuencia**'.

Se ha decidido incluir su explicación en este tema de funcionalidades interesantes de C# ya que, la palabra reservada `yield` la vamos a encontrar en otros lenguajes de programación tales como: **JavaScript, Php, Python, Scala o Ruby** con un significado y uso similar, aunque con variaciones en la sintaxis.

`yield` nos permite que un determinado método devuelva una secuencia `IEnumerable<T>` , sin que esta se base en una colección específica o en la definición de un iterador `IEnumerator<T>`

Casos de uso de los generadores

👉 **Importante:** Nos interesa usar secuencias para procesar datos, porque en la mayoría de lenguajes nos permitirán realizar consultas y agrupaciones complejas mediante programación funcional y declarativa. Más adelante en este tema haremos una introducción a estos procesos sobre secuencias.

Imaginemos un escenario, cada día más común, de **big data**, donde vamos a disponer de una gran cantidad de datos a procesar y donde no es tan importante el tiempo de proceso. Si vamos a procesar estos datos en forma de secuencia y los cargamos todos en una colección se nos pueden dar ciertos problemas en el proceso, como:

1. Nos quedamos sin memoria ya que hay demasiados datos y debemos realizar el proceso, cargando en varias secuencias con todo lo que ello conlleva de complejidad final.
2. Los datos pueden cargar en memoria pero tenemos que solicitarlos a un determinado servicio en Internet (endpoint). Sin embargo, son tantos datos que va a tardar mucho en mandármelos todos a la vez, además de que lo vamos a sobrecargar con nuestra petición
3. Derivado del anterior, no sabemos el tiempo que puede tardar el endpoint en generar cada dato y debemos procesar la secuencia de forma asíncrona. Esto es, el procesador estará atendiendo a otras tareas mientras se genera cada dato y en el momento que se genere un dato lo procesa en la secuencia.
4. Tenemos un stream a un fichero en disco con Terabytes (TB) de información a tratar y queremos aprovechar las funcionalidades de las secuencias para hacerlo.
5. Necesitemos hacer un búsqueda en una gran colección de objetos, de los que solo vamos a necesitar unos pocos hasta encontrar lo que buscábamos. Sin embargo, hemos tenido que cargar previamente todos en la colección para realizar la búsqueda.

En estos casos es mejor ir generando los elementos de la secuencia, conforme los necesitamos para su proceso y no todos a la vez como sucedería al cargarlos en una colección.

Sintaxis y ejemplo de uso de **yield**

En C# **yield** irá seguido de las cláusulas **return** o **break**.

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

Vamos a ver a través de un ejemplo sencillo, cómo sería un esquema básico de uso de **yield**. Para ello, supongamos el siguiente código simple donde **no usamos yield** ...

```

class Ejemplo
{
    // El siguiente método 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>();

        // Para ello, vamos añadiendo a una colección dichos números.
        for (int i = ini; i < fin; i++)
        {
            if (i % n == 0)
            {
                // Vamos generando un log del proceso.
                Console.WriteLine($"Obtenido {i}");
                multiplos.Add(i);
            }
        }

        // Hemos tenido que rellenar toda la colección de múltiplos
        // y la retornaremos en su forma de secuencia IEnumerable<T> (Sustitución)
        return multiplos;
    }

    static void Main()
    {
        // En el programa principal, vamos a obtener el 4º múltiplo
        // de 2 entre 320 y 335 pero ObtieneMultiplosDeN nos devuelve
        // ya toda la secuencia de múltiplos cargada en memoria.
        int cuartoMultObt = ObtieneMultiplosDeN(2, 320, 335).Skip(3).First();
        Console.WriteLine($"El 4to multiplo es {cuartoMultObt}");
    }
}

```

El flujo de ejecución sería....

1. **Línea 30** : Llamamos al métodos y la pasamos el control de ejecución.
2. **Líneas 10 a 18** : Realizamos la inserción de todos los múltiplos en la colección.
3. **Líneas 22** : Realizamos un único **return** con toda la secuencia.


Si ejecutamos el programa y comprobamos el log de salida, veremos que **hemos generado todos los múltiplos** en el rango dado. Para el rango del ejemplo son 8 pero **podría ser un número muy elevado y que además deberemos tener en un lista en memoria.**

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

Vamos a reinplementar el código de nuestro método y ahora **si usamos yield** .

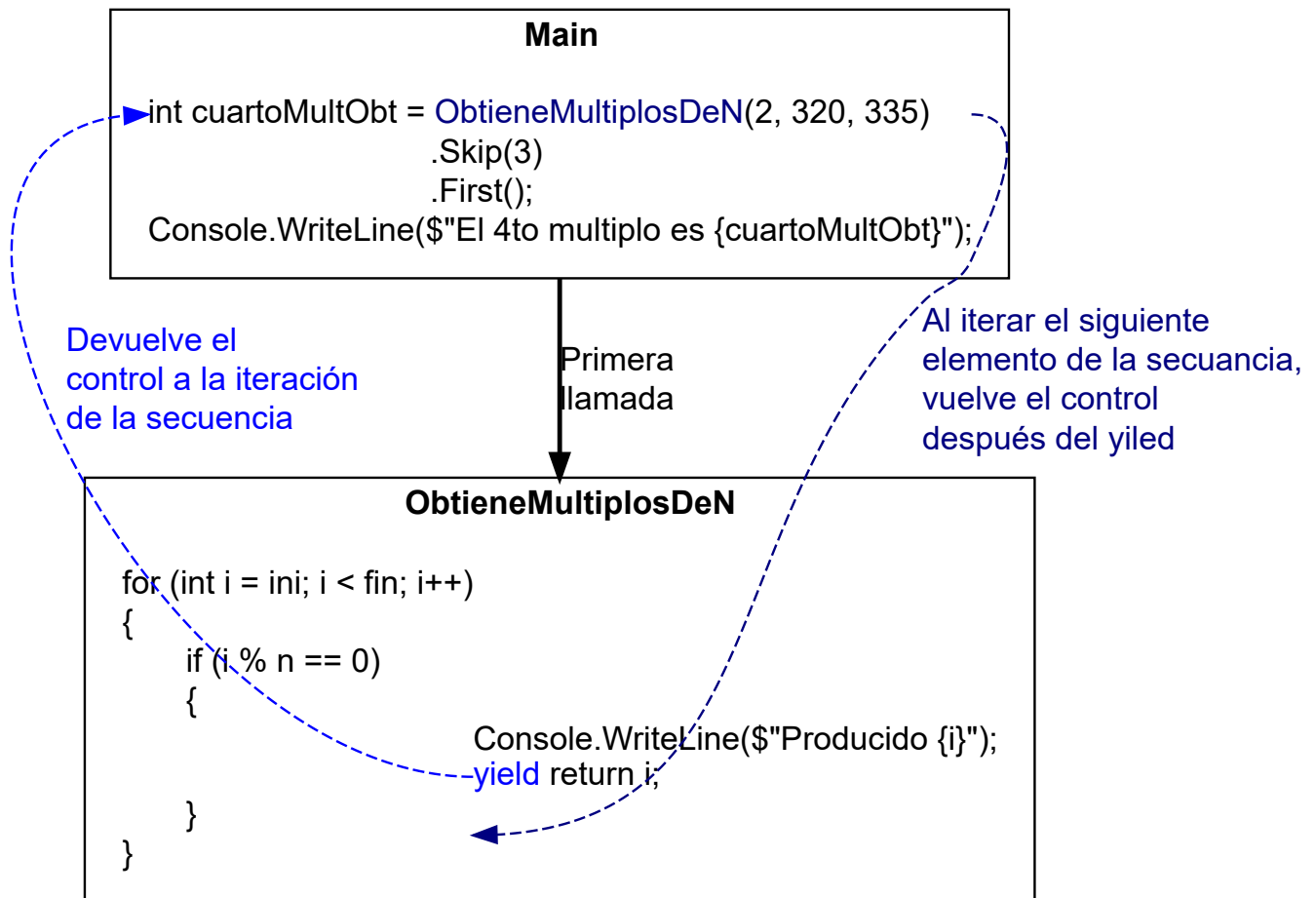
```
static IEnumerable<int> ObtieneMultiplosDeN(int n, int ini, int fin)
{
    for (int i = ini; i < fin; i++)
    {
        if (i % n == 0)
        {
            Console.WriteLine($"Producido {i}");
8           yield return i;
9        }
    }
}

static void Main()
{
15    int cuartoMultObt = ObtieneMultiplosDeN(2, 320, 335).Skip(3).First();
    Console.WriteLine($"El 4to multiplo es {cuartoMultObt}");
}
```

 **Importante:** Fíjate que al hacer un **yield return dato;** C# lo interpretará como que el método retorna una secuencia **IEnumerable<TipoDato>** . Por lo que podremos aplicar al valor retornado, todos los métodos de extensión definido para secuencias o incluso recorrer lo retornado con un **foreach** .

El flujo de ejecución frente al anterior sería....

1. **Línea 15** : Llamamos al métodos y la pasamos el control de ejecución.
2. **Líneas 8** : Obtenemos el siguiente valor de la secuencia y retornamos el control a la **línea 15** en ella si procesará el elemento de la secuencia y si necesitamos otro cuando llamemos al **it.MoveNext()** del iterador (**IEnumerator**) proporcionado por la secuencia devuelta, se volverá a pasar el control de ejecución a la **línea 9** para que el algoritmo me vuelva a generar otro elemento de la secuencia de llegar al **yield return i;** y de no ser así porque ha acabado el **for** entonces el **it.MoveNext()** devolverá **false** .



Si ejecutamos el programa y comprobamos el log de salida, veremos que **hemos generado solo los múltiplos hasta el 4** en el rango dado, en lugar de todos como antes. Esto será así independientemente del rango que le pasemos al método para obtener los múltiplos.

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

No hace falta que tengamos un bucle y un único `yield return dato;`. Podríamos, por ejemplo, *'hardcodear'* una secuencia con gran cantidad de datos predefinidos en nuestro programa, sin necesidad de cargarlos en ningún tipo de colección. Una posibilidad sería la siguiente ...

```

public class Empleado
{
    public string Nombre { get; init; }
    public int Edad { get; init; }
}

public static class Empleados
{
    public static IEnumerable<Empleado> DepartamentoDeVentas
    {
        get
        {
            // Después de cada yield el control volverá al main y
            // en el siguiente MoveNext() del foreach regresará para
            // generar el siguiente objeto de la secuencia.
            yield return new() { Nombre = "Xusa", Edad = 32 };
            yield return new() { Nombre = "Juanjo", Edad = 51 };
            yield return new() { Nombre = "Carmen", Edad = 27 };
            yield return new() { Nombre = "Pepe", Edad = 45 };
        }
    }
}

class Ejemplo
{
    public static void Main()
    {
        foreach (Empleado e in Empleados.DepartamentoDeVentas)
            Console.WriteLine($"{e.Nombre} {e.Edad}");
    }
}

```

Resumen:

yield en resumen, nos ayudará a implementar la generación incremental de una secuencia de datos, introduciendo saltos entre un método y quien lo llamó. De esta manera evitaremos consumir recursos de ejecución y memoria con **flujos de datos de gran tamaño**.