

# Tema 11.3

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

## Índice

### ▼ Funcionalidades Interesantes del Lenguaje C#

- Tipos Anónimos Inmutables
- Tipo Record
- Métodos de Extensión

### ▼ Recorrido Perezoso De Secuencias

- Casos de uso de los generadores
- Sintaxis y ejemplo de uso de yield

### ▼ Serialización de Objetos (Definiciones)

#### ▼ Persistencia

- Concepto de Anotación o Atributo

#### ▼ Serialización a JSON

- ¿Qué es JSON?
- JSON en CSharp
- Propiedades Opcionales

# 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.

**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.

```
var anonimo1 = new { Nombre = "María", Edad = 23 };
var anonimo2 = new { Nombre = "María", Edad = 23 };

// Mostrará True a pesar de que sean referencias a
// objetos diferentes en memoria.
Console.WriteLine(anonimo1.Equals(anonimo2));
```

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);
```

# Tipo Record

Más adelante, usaremos los tipos anónimos para obtener instantáneas de datos agrupados como resultado de consultas. Algo así como la obtención de un **snapshot** tras hacer un `select` en SQL.

A esta agrupación de datos resultado de una consulta sobre otros objetos se le denomina **DTO (Data Transfer Object)** y deben tener las **características de inmutabilidad y comparación de igualdad que tienen los tipos anónimos**.

No obstante, una de las **restricciones** más notables de los **tipos anónimos** es que **su uso está limitado** a hacerse dentro del **cuerpo de un método**. Esto significa, por ejemplo, que un método no podrá retornar un tipo anónimo que contenga el resultado de mi consulta.

Supongamos que definimos un método que crea un objeto anonimo con el **nombre** y el **código de país** de una persona.

✦ **Nota:** Aunque en el ejemplo no se hace por simplicidad, supondremos que los datos de **Nombre** y **CodigoPais** se han obtenido de la consulta sobre otros objetos.

```
// El código del cuerpo es válido pero como el tipo es anónimo
// no sabremos cómo indicárselo al método.
static ? PersonasPais()
{
    var PersonasPais = new { Nombre = "Diana", CodigoPais = "ES" };
    return PersonasPais;
}
```

1. Una **posible solución** es usar el tipo **dynamic** que equivaldría a la declaración de tipos que hacemos en los lenguajes de scripting como JavaScript donde el tipo no se conoce hasta el tiempo de ejecución.

```
static dynamic PersonasPais()
{
    var PersonasPais = new { Nombre = "Diana", CodigoPais = "ES" };
    return PersonasPais;
}
dynamic dato = PersonasPais();
```

💀 **Peligro:** Sin embargo, **no** deberíamos **usar nunca** este tipo ya que solo está pensado para usar en ciertos casos de **interoperabilidad** con otras librerías. Perdemos toda la seguridad de usar un lenguaje fuertemente tipado y pudiendo obtener errores en tiempo de ejecución fácilmente.

2. Otra posible solución es definir una clase específica que agrupe los datos y que tenga las características de inmutabilidad y comparación que tienen los tipos anónimos.

Tendríamos que definir algo así...

```
class PersonaPais
{
    public PersonaPais(string nombre, string codigoPais)
    {
        Nombre = nombre;
        CodigoPais = codigoPais;
    }

    public string Nombre { get; }
    public string CodigoPais { get; }

    public override bool Equals(object? obj) =>
        obj is PersonaPais pais &&
        Nombre == pais.Nombre &&
        CodigoPais == pais.CodigoPais;
    // Estoy obligado a implementarlo junto con el Equals.
    public override int GetHashCode() => GetHashCode.Combine(Nombre, CodigoPais);
}

static PersonaPais PersonasPais()
{
    PersonaPais PersonasPais = new PersonaPais("Diana", "ES");
    return PersonasPais;
}
```

Esta opción, nos obliga a definir una clase '*muy larga*' para unos datos que son el resultado de una consulta y así para otras combinaciones de datos que se nos ocurran. Además, perdemos la declaración la construcción que usábamos para los tipos anónimos

`new { Nombre = "Diana", CodigoPais = "ES" };` por un constructor tradicional

`new PersonaPais("Diana", "ES");`

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.

Usando la palabra reservada `init` podremos **ahorrarnos el constructor** y reescribir nuestro código anterior como ...

```
class PersonaPais
{
    public string? Nombre { get; init; }
    public string?CodigoPais { get; init; }

    public override bool Equals(object? obj) =>
        obj is PersonaPais pais &&
        Nombre == pais.Nombre &&
        CodigoPais == pais.CodigoPais;
    public override int GetHashCode() => GetHashCode.Combine(Nombre, CodigoPais);
}

static PersonaPais PersonasPais()
{
    // Esta sintaxis para inicializar propiedades,
    // la vamos a encontrar en otros lenguajes como Kotlin
    PersonaPais PersonasPais = new PersonaPais
    {
        Nombre = "Diana",
        CodigoPais = "ES"
    };
    return PersonasPais;
}
```

Podemos usar este tipo de definiciones de clases con `init` para retornar colecciones de objetos inmutables utilizando una sintaxis análoga a la de los objetos anónimos de la siguiente forma ...

```
static IEnumerable<PersonaPais> PersonasPorCodigoDePais()
{
    PersonaPais[] personasPorCodigoDePais =
    {
        new PersonaPais{ Nombre = "Diana", CodigoPais = "ES" },
        new PersonaPais{ Nombre = "Juana", CodigoPais = "RU" },
        new PersonaPais{ Nombre = "Dario", CodigoPais = "CU" },
        new PersonaPais{ Nombre = "Jenny", CodigoPais = "CU" }
    };
    return personasPorCodigoDePais;
}
```

3. Para evitar, este tipo de declaraciones tan largas. **C# 9** incluyó la palabra reservada **record** el cual me permite definir una tipo referencia para ser usado como **DTO** con las características de inmutabilidad y comparación que hemos indicado.

```
public record PersonaPais(string Nombre, stringCodigoPais);
public static PersonaPais PersonasPais()
{
    // Resultado de consulta en objeto anónimo.
    var dato = new { Nombre = "Diana", CodigoPais = "ES" };
    // Lo devolvemos tipado como DTO
    return new(dato.Nombre, dato.CodigoPais);
}
PersonaPais p = PersonasPais();
```

Podemos destacar las siguientes características a la hora de compararlos teniendo en cuenta que los operadores `==` y `!=` se redefinen automáticamente para comparar propiedad a propiedad.

```
PersonaPais p1 = PersonasPais();
PersonaPais p2 = PersonasPais();
Console.WriteLine(p1); // Mostrará PersonaPais { Nombre = Diana, CodigoPais = "ES" }
Console.WriteLine(p1.Equals(p2)); // True
Console.WriteLine(p1 == p2); // True
Console.WriteLine(ReferenceEquals(p1, p2)); // False referencias a diferentes objetos.
```

También es posible indicar la propiedad que estamos pasando al construir el registros con la sintaxis

**Propiedad: valor**

```
public static PersonaPais PersonasPais()
{
    var dato = new { Nombre = "Diana", CodigoPais = "ES" };
    return new (Nombre: dato.Nombre, CodigoPais: dato.CodigoPais);
}
```

Aunque son objetos inmutables, podemos crear fácilmente una nueva instancia modificando una propiedad con la **expresión with**

```
PersonaPais p1 = PersonasPais();
PersonaPais p2 = p1 with { CodigoPais = "RU" };
Console.WriteLine(p1); // Mostrará PersonaPais { Nombre = Diana, CodigoPais = ES }
Console.WriteLine(p2); // // Mostrará PersonaPais { Nombre = Diana, CodigoPais = RU }
Console.WriteLine(p1 == p2); // False
```

Al igual que sucedía con class, también podremos devolver **colecciones de registros inmutables**.

```
static IEnumerable<PersonaPais> PersonasPorCodigoDePais()
{
    PersonaPais[] personasPorCodigoDePais =
    {
        new ( Nombre: "Diana", CodigoPais: "ES" ),
        new ( Nombre: "Juana", CodigoPais: "RU" ),
        new ( Nombre: "Dario", CodigoPais: "CU" ),
        new ( Nombre: "Jenny", CodigoPais: "CU" )
    };
    return personasPorCodigoDePais;
}
```

✦ **Nota:** Si quieres saber más sobre **cómo usar registros**, puedes leer la documentación oficial en [este enlace](#).



# 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.

```
namespace StringExtensions
{
    public static class StringExtension
    {
        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;
        }

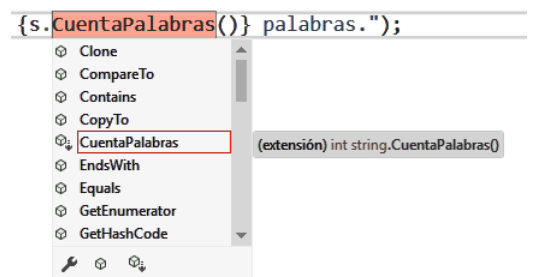
        public static int CuentaPalabras(this string s) => s.Split(
            new char[] { ' ', '.', '?' },
            StringSplitOptions.RemoveEmptyEntries
        ).Length;
    }
}
```

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.

```
using StringExtensions;

class Ejemplo
{
    static void Main()
    {
        string s = "hola caracola";
        // Si no hacemos el using StringExtensions; los métodos Capitaliza
        // y CuentaPalabras no nos los ofrecerá.
        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.

```
List<int> secuencia = Enumerable.Range(2, 4).ToList();
// Equivale a ...
List<int> secuencia = new List<int> { 2, 3, 4, 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 necesitemos para su proceso y no todos a la vez como sucedería al cargarlos en una colección.

📌 **Nota:** Puedes saber algo más sobre los generadores leyendo a [esta entrada en la Wikipedia](#)

## Sintaxis y ejemplo de uso de **yield**

En C# **yield** irá seguido de las clausulas **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** ...

```

1  class Ejemplo
2  {
3      // El siguiente método devolverá una secuencia resultado de llenar
4      // una lista con los múltiplos de n entre ini y fin.
5      static IEnumerable<int> ObtieneMultiplosDeN(int n, int ini, int fin)
6      {
7          List<int> multiplos = new List<int>();
8
9          // Para ello, vamos añadiendo a una colección dichos números.
10         for (int i = ini; i < fin; i++)
11         {
12             if (i % n == 0)
13             {
14                 // Vamos generando un log del proceso.
15                 Console.WriteLine($"Obtenido {i}");
16                 multiplos.Add(i);
17             }
18         }
19
20         // Hemos tenido que rellenar toda la colección de múltiplos
21         // y la retornaremos en su forma de secuencia IEnumerable<T> (Sustitución)
22         return multiplos;
23     }
24
25     static void Main()
26     {
27         // En el programa principal, vamos a obtener el 4º múltiplo
28         // de 2 entre 320 y 335 pero ObtieneMultiplosDeN nos devuelve
29         // ya toda la secuencia de múltiplos cargada en memoria.
30         int cuartoMultObt = ObtieneMultiplosDeN(2, 320, 335).Skip(3).First();
31         Console.WriteLine($"El 4to multiplo es {cuartoMultObt}");
32     }
33 }

```

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}");
            yield return i;
        }
    }
}

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

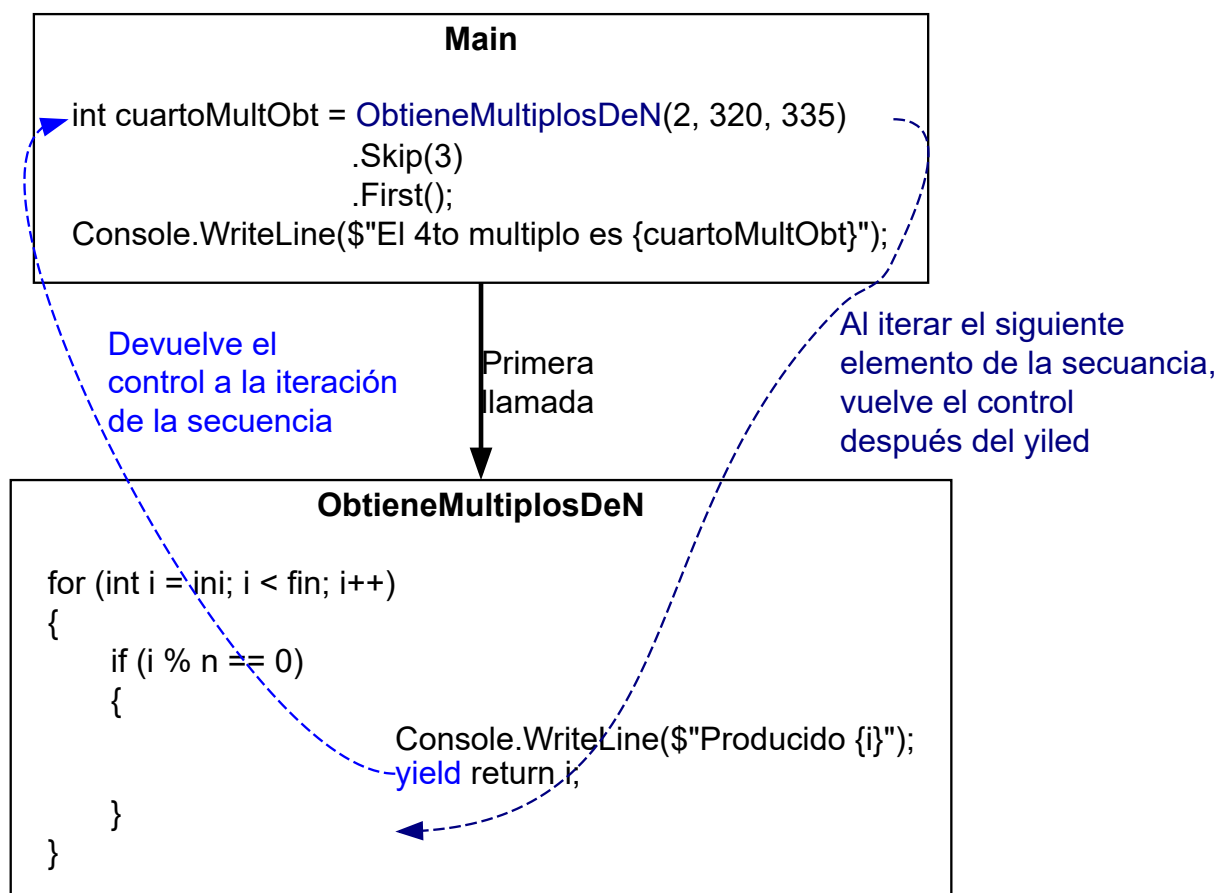
 **Importante:** Fíjate que al hacer un **yield return dato;** C# lo interpretará como que el método retorna una secuencia **IEnumerable<TipoDato>** . Por lo que podremos aplicar al valor retornado, todos los métodos de extensión definido para secuencias o incluso recorrer lo retornado con un **foreach**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

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 Empleado{ Nombre = "Xusa", Edad = 32 };
            yield return new Empleado{ Nombre = "Juanjo", Edad = 51 };
            yield return new Empleado{ Nombre = "Carmen", Edad = 27 };
            yield return new Empleado{ Nombre = "Simón", 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**.

## 🎓 Caso de estudio:

Serías capaz de reescribir el código del último ejemplo declarando `Empleado` y `Emplados` usando la palabra `record` .

📌 **Nota:** ten en cuenta que `record` no admite el modificador `static` .

```
public record Empleado(string Nombre, int Edad);

public record Empleados()
{
    public static IEnumerable<Empleado> DepartamentoDeVentas
    {
        get
        {
            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: "Simón", Edad: 45);
        }
    }
}

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

# Serialización de Objetos (Definiciones)

## Persistencia

Se define por persistencia en el mundo de la POO, como la **capacidad que tienen los objetos de sobrevivir al proceso padre que los creo**. Esto decir, que su ciclo de vida excede de la del programa que lo instanció.

La persistencia permite al programador pues almacenar, transferir y recuperar fácilmente el estado de los objetos.

### ¿Cómo podemos conseguir la persistencia?

La forma más común de conseguirlo es mediante la **serialización**.


La serialización es el proceso de convertir el estado de un objeto a un formato que se pueda **almacenar** o **transportar**. Normalmente el proceso producirá una **secuencia** de bytes o texto de marcado.

El complemento de la serialización es la **deserialización**, que convierte una secuencia de bytes o texto de marcado a un objeto. Ambos procesos pues permiten almacenar y transferir fácilmente datos.

### ¿A qué serializaremos una clase?

.NET ofrece dos tecnologías de serialización:

- La **serialización binaria** conserva la fidelidad de tipos, lo que resulta útil para conservar el estado de un objeto entre distintas llamadas a una aplicación. Puedes serializar un objeto en una secuencia de bytes que cómo hemos visto podemos pasar a disco, la memoria, a través de la red, etc. Ejemplos ...
  - Compartir un objeto entre distintas aplicaciones '*serializándolo*' en el Portapapeles.
  - Hacer **streaming** de vídeo serializando objetos con información de vídeo comprimido a través de la red.

 **Aviso:** En las últimas versiones de .NET se recomienda usar la serialización binaria sólo para casos muy específicos, como la comunicación entre aplicaciones .NET y ha sido marcada como **peligrosa** y **obsoleta**. Es por eso que en estos apuntes no se tratará la serialización binaria.

- La **serialización a lenguajes de marcado** como:
  - La **serialización a XML** sólo serializa las propiedades públicas y los campos, y no conserva la fidelidad de tipos. Esto resulta útil cuando desea proporcionar o consumir datos sin restringir la aplicación que utiliza los datos.

- La **serialización a alguna notación de objetos** estándar cada vez más comunes y cuya función sería la misma que el XML pero menos '*verbosas*' y más fáciles de leer y/o modificar para humanos. Los más comunes son:
  - **JSON**: Definir configuraciones o consumo de datos a través de microservicios web o bases de datos NoSQL.
  - **YAML**: Muy usado para definir archivos de configuración de sistemas.

La gran mayoría de librerías de serialización de objetos en .NET y en otros lenguajes utilizan el concepto de **anotaciones** o **atributos** para definir cómo se serializará un objeto. Es por eso, que antes de empezar a serializar un objeto, deberemos entender qué son las anotaciones o atributos.

## Concepto de Anotación o Atributo

👉 **Importante**: Las **anotaciones** son muy comunes en lenguajes como Java o Kotlin y básicamente son metadatos que se pueden añadir a clases, métodos, campos, etc. y **modificar el comportamiento justo del elemento al que antecede**. En C# se les denomina **atributos** aunque este término **puede ser confuso**, pues también es usado en la Programación Orientada a Objetos para hacer referencia a los que nosotros hemos llamado **campos y propiedades**.

Una **atributo** pues en .NET, es una etiqueta de la sintaxis `[nombre]` que podremos aplicar a un ensamblado, clase, constructor, enumeración, campo, interfaz, método, etc... y que **genera información en el ensamblado** en forma de metadatos heredando de la clase `Attribute`.

Por ejemplo, el atributo `[Obsolete]` es un atributo que se puede aplicar a un método o a una clase y que indica que el método o la clase están obsoletos y que no deberían usarse.

```
[Obsolete("Este método está obsoleto. Utilice en su lugar el método X")]  
public void MetodoObsoleto() // Código del método }
```

Otro ejemplo sería el atributo `[NotNull]` que se puede aplicar a un parámetro de un método y que indica que el parámetro no puede ser nulo incluso si el tipo de dato del parámetro es un tipo de dato que puede ser nulo.

```
public void Metodo([NotNull] string? parametro) { // Código del método }
```

En el ejemplo anterior, el atributo `[NotNull]` indica que el parámetro `parametro` no puede ser nulo.

## Serialización a JSON

La serialización a JSON es un proceso que convierte un objeto en una cadena JSON y viceversa. JSON es un formato de texto que es fácil de leer y escribir para los humanos y fácil de analizar y generar para las máquinas.

## ¿Qué es JSON?

Aunque este tema se tratará con más profundidad en el módulo de LM y en segundo curso. Vamos a realizar un resumen rápido sobre dicho formato para entenderlo por encima.

**JSON** (Javascript Object Notation) es un formato ligero de intercambio de datos entre clientes y servidores, basado en la sintaxis de Javascript para representar estructuras en forma organizada. Es un formato en texto plano independiente de todo lenguaje de programación, es más, soporta el intercambio de datos en gran variedad de lenguajes. El estándar de JSON se especifica en el [RFC 7159](#) y se ha estandarizado en la [ECMA-404](#). Así pues, puedes consultar la especificación del RFC para profundizar más en él.

### Tipos de datos en JSON

Similar a la estructuración de datos primitivos y complejos en los lenguajes de programación, JSON establece varios tipos de datos: **cadenas**, **números**, **booleanos**, **arrays** y **objetos**. El propósito es crear objetos que contengan varios atributos compuestos como pares clave valor. Donde la clave es un nombre que identifique el uso del valor que lo acompaña. Veamos un ejemplo:

El JSON de ejemplo está compuesto por los datos de un **estudiante**. Los objetos JSON contienen sus atributos entre llaves `{ }`, al igual que un bloque de código en Javascript, donde cada atributo debe ir separado por coma `,` para diferenciar cada par. La sintaxis de los pares debe contener dos puntos `:` para dividir la clave del valor. El nombre del par debe tratarse como cadena y añadirle **comillas dobles**.

```
{
  "id": 101,
  "nombre": "Carlos",
  "estaActivo": true,
  "notas": [2.3, 4.3, 5.0]
}
```

Si te fijas en nuestro ejemplo, este trae un ejemplo de cada tipo de dato:

- **id** es de tipo entero, ya que contiene un número que representa el código del estudiante.
- **nombre** es un string. Usa comillas dobles para definirlos.
- **estaActivo** es un tipo booleano que representa si el estudiante se encuentra en la institución educativa o no. Usa las palabras reservadas **true** y **false** para declarar el valor.
- **notas** es un arreglo de números reales. El conjunto de sus elementos debes incluirlos dentro de corchetes `[ ]` y separarlos por coma.

Básicamente esta sería la sintaxis, pero en los documentos se pueden dar casos más complejos como objetos anidados, arrays de objetos, etc. Por ejemplo si modificamos el ejemplo anterior para que el estudiante en lugar de una array de notas numérico tenga un array de objetos con las asignaturas y sus notas, el documento JSON quedaría así:

Fíjate que el documento tiene un **objeto raíz** que contiene el resto de objetos. En nuestro caso el objeto raíz es un estudiante. Pero también podríamos tener un array de estudiantes directamente en el documento JSON.

```
{
  "id": 101,
  "nombre": "Carlos",
  "estaActivo": true,
  "asignaturas": [
    {
      "nombre": "Matemáticas",
      "nota": 4.5
    },
    {
      "nombre": "Lengua",
      "nota": 5.0
    }
  ]
}
```

## JSON en CSharp

C# dentro de las BCL y dentro del espacio de nombres `System.Text.Json` define una serie de clases que nos permiten trabajar con JSON. EN concreto nos centraremos en la clase `JsonSerializer` que nos permitirá serializar y deserializar objetos a JSON y se define en el espacio de nombres `System.Text.Json.Serialization`.

Supongamos el siguiente **record** con datos de un estudiante:

```
record Estudiante(string Nombre, string Apellido, int Edad);
```

Si la serializamos tal cual a JSON, obtendremos un JSON como el del ejemplo usando como claves los nombres de las propiedades del record en PascalCase y el convenio en JSON es usar camelCase. Para cambiar el nombre de la clave asociada a una propiedad, podemos usar la anotación/atributo `[JsonPropertyName]` de la siguiente manera ...

```
{
  "Nombre": "Carlos",
  "Apellido": "García",
  "Edad": 18
}
```

```
record Estudiante(
  [property: JsonPropertyName("nombre")]
  string Nombre,
  [property: JsonPropertyName("apellido")]
  string Apellido,
  [property: JsonPropertyName("edad")]
  int Edad);
```

```
{
  "nombre": "Carlos",
  "apellido": "García",
  "edad": 18
}
```

Si quisiéramos serializar los datos de varios estudiantes en un documento JSON con un único objeto raíz, podríamos hacerlo a través de un **record** que contenga una colección de estudiantes como puede ser una clase completa con un nombre de clase, un tutor y una colección de estudiantes.

```
record Clase(  
    [property: JsonPropertyName("nombre")]  
    string Nombre,  
    [property: JsonPropertyName("tutor")]  
    string Tutor,  
    [property: JsonPropertyName("alumnos")]  
    IEnumerable<Estudiante> Estudiantes)  
{  
    public override string ToString() =>  
        $"{Nombre} ({Tutor})\n\n" +  
        string.Join("\n", Estudiantes.Select(a => $"{a}"));  
}
```

Para aplicar algunos de los conceptos que hemos visto en otros temas, como son el principio OCP (SOLID) y los métodos de extensión. Vamos a definir los métodos de serialización y deserialización en una clase estática que llamaremos **ClaseJson**.



```

static class ClaseJson
{
    public static Clase? Recupera(string path)
    {
        using FileStream s = new(path, FileMode.Open, FileAccess.Read);
        // Deserializamos el objeto JSON en el documento a un objeto de la clase o tipo Clase
        return JsonSerializer.Deserialize<Clase>(s);
    }

    // Método de extensión que serializa un objeto de la clase Clase a un documento JSON
    public static void Guarda(this Clase clase, string path)
    {
        using FileStream s = new(path, FileMode.Create, FileAccess.Write);
        JsonSerializerOptions options = new()
        {
            // Indentará la cadena con el JSON para hacerla más legible
            WriteIndented = true,
            // No permitirá comas al final de los arrays
            AllowTrailingCommas = false,
            // Codificará los caracteres Unicode básicos y los de la página de códigos ISO-8859-
            Encoder = JavaScriptEncoder.Create(
                UnicodeRanges.BasicLatin,
                UnicodeRanges.Latin1Supplement),
            // Ignorará una propiedad si su valor es null
            DefaultIgnoreCondition = JsonIgnoreCondition.WhenWritingNull
        };
        JsonSerializer.Serialize(s, clase, options);
    }
}

```

Ahora ya podríamos serializar y deserializar objetos de la clase `Clase` a JSON de la siguiente manera:

```

static class Program
{
    static void Main()
    {
        Clase clase = new(
            Nombre: "1º DAM",
            Tutor: "Juan",
            Estudiantes: [
                new("Pepa", "Pérez", 25),
                new("María", "Peláez", 22),
                new("Rosa", "López", 26)
            ]
        );

        clase.Guarda("1DAM.json");
        Clase? c = ClaseJson.Recupera("1DAM.json");
        Console.WriteLine(
            c?.ToString()
            ?? "No se ha podido recuperar la clase");
    }
}

```

```

{
    "nombre": "1º DAM",
    "tutor": "Juan",
    "alumnos": [
        {
            "nombre": "Pepa",
            "apellido": "Pérez",
            "edad": 25
        },
        {
            "nombre": "María",
            "apellido": "Peláez",
            "edad": 22
        },
        {
            "nombre": "Rosa",
            "apellido": "López",
            "edad": 26
        }
    ]
}

```

Podremos comprobar que al recuperar el objeto de la clase **Clase** de un documento JSON, se ha recuperado correctamente.

## Propiedades Opcionales

Puede darse el caso de que queramos que una propiedad de un objeto sea opcional. Para ello, fijate que en las opciones de serialización hemos añadido la propiedad `DefaultIgnoreCondition` con el valor `JsonIgnoreCondition.WhenWritingNull`. Esto hará que si una propiedad de un objeto es `null`, no se serialice en el documento JSON. Además, al deserializar el objeto, si esta propiedad no existe en el documento JSON, se inicializará a `null`.

Por ejemplo, supongamos que modificamos el record `Estudiante` para que tenga una propiedad que será un objeto de tipo `Direccion` que **sera opcional y por tanto anulable con valor por defecto a null**.

```
record Estudiante(  
    [property: JsonPropertyName("nombre")]  
    string Nombre,  
    [property: JsonPropertyName("apellido")]  
    string Apellido,  
    [property: JsonPropertyName("edad")]  
    int Edad,  
    [property: JsonPropertyName("direccion")]  
    Direccion? Direccion = null);
```

```
record Direccion(  
    [property: JsonPropertyName("calle")]  
    string Calle,  
    [property: JsonPropertyName("numero")]  
    int Numero,  
    [property: JsonPropertyName("ciudad")]  
    string Ciudad,  
    [property: JsonPropertyName("pais")]  
    string Pais);
```

Básicamente ya podríamos serializar y deserializar objetos de la clase `Estudiante` igual que antes. Teniendo en cuenta eso sí que la propiedad `Direccion` es opcional y por tanto podría ser `null` al leerse del documento JSON.

```
Clase clase = new(  
    Nombre: "1º DAM",  
    Tutor: "Juan",  
    Estudiantes: [  
        new("Rosa", "López", 26),  
        new("Juan", "Gómez", 24, new("Calle Falsa", 123, "Elche", "España"))  
    ]  
);
```

Se añadirán un objeto JSON anidado para la propiedad **direccion** en aquellos estudiantes que la tengan distinta de **null** y a la hora de deserializar el objeto, si la propiedad **direccion** no existe en el documento JSON, se inicializará a **null** y es por eso que sea anulable.

Por ejemplo si la edad fuera opcional, como es un campo numérico tendremos dos opciones, o bien lo inicializamos a **null** y el campo pasaría a ser de tipo **int?**.

```
[property: JsonPropertyName("edad")]  
int? Edad = null,
```

```
{  
  "nombre": "1º DAM",  
  "tutor": "Juan",  
  "alumnos": [  
    {  
      "nombre": "Rosa",  
      "apellido": "López",  
      "edad": 26  
    },  
    {  
      "nombre": "Juan",  
      "apellido": "Gómez",  
      "edad": 24,  
      "direccion": {  
        "calle": "Calle Falsa",  
        "numero": 123,  
        "ciudad": "Elche",  
        "pais": "España"  
      }  
    }  
  ]  
}
```