

# Unidad 20

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

## Índice

- [Índice](#)
- ▼ [Programación funcional](#)
  - [Introducción](#)
  - [Algunas definiciones interesantes](#)
- ▼ [Almacenando referencias a funciones en variables](#)
  - ▼ [Tipos Delegados en CSharp](#)
    - [Usando delegados en CSharp](#)
    - [!\[\]\(38441ceaa711016e0bf2ad46ad394ff4\_img.jpg\) Ampliación Opcional - Multidifusión de delegados](#)
  - ▼ [Delegados vs Interfaces en Csharp](#)
    - [¿Cuándo usar Delegados vs Interfaces en CSharp?](#)
  - ▼ [Delegados Parametrizados](#)
    - [Definiciones para procedimientos Action](#)
    - [Definiciones para funciones Func](#)
  - [Tipos Anónimos Inmutables](#)

# Programación funcional

## Introducción

La **programación funcional** es un **paradigma de programación declarativa** basado en el uso de **funciones matemáticas**, en contraste con la programación **imperativa**, que enfatiza los cambios de estado mediante la mutación de variables.

Existen lenguajes de programación funcionales puros como por ejemplo **Haskell** y **Clojure**. Sin embargo, el paradigma funcional ha ido cogiendo cada vez más peso en la programación moderna. Tanto es así, que lenguajes imperativos y orientados a objetos tradicionales como **C++**, **Java**, **C#** o **PHP** han incluido características de la programación funcional en su sintaxis o potenciando las que ya tenían lenguajes como **JavaScript**, **Python**. Además, los lenguajes multi-paradigma de nueva creación como **Scala**, **Go**, **F#** o **Kotlin** (este último en menor medida) se han diseñado para que tenga más peso la programación funcional que la orientada a objetos e imperativa.

Simplificando y a grandes rasgos podemos decir que **se basa en los siguientes pilares** matemáticos:

- [Cálculo Lambda](#).
- [Clausuras](#).
- [Teoría de categorías](#) y más en concreto el concepto de [functor](#) y su aplicación a las ciencias de la computación a través de:
  - [Mónadas](#).
  - [Covarianza y Contravarianza](#).
- [Recursión o Recursividad](#) (Formaría parte del Cálculo Lambda).

Cómo se puede observar, su bases son bastante amplias y llevaría bastante ahondar todos estos conceptos. Por esta razón, **este tema pretende ser meramente introductorio y por tanto en él solo vamos a abordar algunos conceptos a través del lenguaje C#**. Estos conceptos serán extrapolables a otros lenguajes ya que como hemos comentados todos utilizan, en mayor o medida, dicho paradigma.

## Algunas definiciones interesantes

Algunos de los **conceptos a destacar** en la programación funcional son:

- **Función**: En programación funcional, una **función** es un bloque de código que toma uno o más argumentos de entrada y produce un valor de salida. A diferencia de las funciones en la programación imperativa, las funciones en la programación funcional son '**ciudadanos de**

**primera clase**', lo que significa que pueden ser tratadas como cualquier otro valor, como números o cadenas. Esto implica que las funciones pueden ser:

- **Asignadas a variables.**
  - **Pasadas como argumentos a otras funciones.**
  - **Devueltas como valores desde otras funciones.**
  - Almacenadas en estructuras de datos.
  - Combinadas para crear nuevas funciones.
  - Anidadas dentro de otras funciones.
- **Funciones lambda:** Las funciones lambda son funciones anónimas que se pueden definir de manera concisa y se utilizan comúnmente para operaciones de orden superior. Se **tratan como expresiones** que pueden ser asignadas a variables o pasadas como argumentos. (Las trataremos más adelante).
  - **Funciones de orden superior (HOF):** Las funciones de orden superior son aquellas que pueden tomar otras funciones como argumentos o devolver funciones como resultados. Esto permite una mayor abstracción y reutilización del código.
  - **Funciones puras:** Las funciones puras son aquellas que siempre producen el mismo resultado para los mismos argumentos y no tienen efectos secundarios, es decir, no modifican ningún estado externo.
  - **Callbacks:** Nombre tradicional con el que se conoce a las funciones que se pasan como argumentos a otras funciones y **se ejecutan en un momento determinado, generalmente cuando ocurre un evento o se completa una tarea**. Por tanto, los callbacks **son un caso concreto de HOF**.
  - **Expresiones en lugar de declaraciones:** La programación funcional se basa en la evaluación de expresiones en lugar de la ejecución de declaraciones. Esto significa que el enfoque está en qué se quiere lograr en lugar de cómo lograrlo.

## Almacenando referencias a funciones en variables

El **mecanismo variará según el lenguaje** de programación. En C y C++ se usan **punteros a funciones**, en Java o Kotlin se usan **interfaces funcionales (SAM)**, en Python o JavaScript se usan **funciones anónimas (lambdas)** y en C# se usan **delegados**.

### Tipos Delegados en CSharp

Un delegado es un **tipo especial de clase cuyos objetos pueden almacenar referencias a uno o más métodos con la misma signatura** de tal manera que; a través del objeto, sea posible solicitar la ejecución en cadena de todos ellos. En otras palabras, podemos decir que es un objeto que almacena una o más referencias a un método para ejecutarlo posteriormente.

## Usando delegados en CSharp

Si un delegado en un objeto, deberá haber un tipo que lo defina. Este tipo tendrá un nombre (identificador) y me indicará la signatura de los métodos que referenciará el objeto delegado.

La sintaxis para definir el tipo será:

```
<modificadores> delegate <tipoRetorno> <TipoDelegado>(<parámetros formales>);
```

Donde:

- `<TipoDelegado>` será el **nombre del tipo** que me servirá para definir objetos delegado.
- `<tipoRetorno>` y `<parámetros formales>` se corresponderán, respectivamente, con el tipo del valor de retorno y la lista de parámetros formales que definirán la signatura de los métodos cuyas referencias contendrán los objetos de ese tipo delegado.

La sintaxis para instanciar objetos delegado del tipo definido será:

```
TipoDelegado oDelegado = IdMetodoQueCumpleLaSignaturaDelTipo;
```

que será un *syntactic sugar* del siguiente código...

```
TipoDelegado oDelegado = new TipoDelegado(IdMetodoQueCumpleLaSignaturaDelTipo);
```

La sintaxis para hacer una llamada al método o métodos que almacena un objeto delegado será:

```
tipoRetorno resultado = oDelegado(<parámetros reales>);
```

que será un *syntactic sugar* del siguiente código...

```
tipoRetorno resultado = oDelegado.Invoke(<parámetros reales>);
```

y que realmente estaremos haciendo una llamada al método...

```
tipoRetorno resultado = IdMetodoQueCumpleLaSignaturaDelTipo(<parámetros reales>);
```

**¿No te han quedado claro las diferentes sintaxis?** Veámoslo a través de un ejemplo concreto de uso comentado que puedes descargar de [este enlace](#).

Definamos un **tipo delegado** llamado `Operacion` que referenciará a métodos que reciban dos parámetros de tipo `double` y retornen un valor de tipo `double`. Fíjate, sobre todo, que lo que estamos definiendo es un **tipo** como si fuera una clase más.

```
public delegate double Operacion(double op1, double op2);
```

Definamos ahora dos métodos estáticos con cuerpo de expresión que cumplen la signatura definida en el tipo delegado anterior. Esto es, ambos métodos reciben dos parámetros de tipo `double` y retornan un valor de tipo `double`.

```
public static double Suma(double op1, double op2) => op1 + op2;
public static double Multiplica(double op1, double op2) => op1 * op2;
```

Método `OperaArrays` que recibe dos arrays de valores y un objeto delegado del tipo `Operacion`. El parámetro `operacion` indicará la 'estrategia' (Strategy) a seguir para operar con los valores de ambos arrays. El método devolverá un nuevo array con los resultados de aplicar la operación indicada a cada par de valores de los arrays de entrada.

```
public static double[] OperaArrays(  
    double[] ops1, double[] ops2,  
    ③ Operacion operacion)  
{  
    double[] resultados = new double[ops1.Length];  
    for (int i = 0; i < resultados.Length; ++i)  
        ⑦ resultados[i] = operacion(ops1[i], ops2[i]);  
    return resultados;  
}
```

Programa principal donde se definen dos arrays de valores y se usan ambos métodos definidos anteriormente para realizar sumas y multiplicaciones a través del método `OperaArrays`. Fíjate como en la **línea 6** estamos guardando la función suma en la variable `operacion` del tipo delegado `Operacion` y se la pasamos al método `OperaArrays`.

En la **línea 10** estamos pasando directamente la función multiplicación al método sin necesidad de definir una variable intermedia.

```
public static void Main()  
{  
    double[] ops1 = [ 5, 4, 3, 2, 1 ];  
    double[] ops2 = [ 1, 2, 3, 4, 5 ];  
  
    ⑥ Operacion operacion = Suma;  
    ⑦ double[] sumas = OperaArrays(ops1, ops2, operacion);  
    Console.WriteLine($"Sumas: {string.Join(" ", sumas)}");  
  
    ⑩ double[] multiplicaciones = OperaArrays(ops1, ops2, Multiplica);  
    Console.WriteLine($"Multiplicaciones: {string.Join(" ", multiplicaciones)}");  
}
```

## Ampliación Opcional - Multidifusión de delegados

Se producirá cuando un objeto delegado **llama a más de un método** cuando se invoca. Esta cualidad de los delegados nos será útil, más adelante, cuando veamos el concepto de **evento**.

- Para **encadenar** un método / delegado en la multidifusión usará el **operador +=**
- Para **retirar** un método / delegado de la multidifusión de llamadas usará el **operador -=**

### Aviso

Tiene sentido para **métodos que no retornan nada** (procedimientos), ya que si los delegados retornan algo como en el ejemplo anterior, se asignará el resultado **de la última llamada**.

```
class Ejemplo
{
    // Métodos a añadir al objeto delegado.
    public static void VerSuma(int op1, int op2) =>
        Console.WriteLine($"{op1} + {op2} = {op1 + op2}");
    public static void VerMultiplicacion(int op1, int op2) =>
        Console.WriteLine($"{op1} * {op2} = {op1 * op2}");
    public static void VerDivision(int op1, int op2) =>
        Console.WriteLine($"{op1} / {op2} = {op1 / op2}");

    // Definición del tipo delegado con la signatura de los métodos anteriores.
    public delegate void VerOperacion(int op1, int op2);

    public static void Main()
    {
        // La primera referencia al método a ejecutar la podemos asignar directamente.
        VerOperacion verOperaciones = VerSuma;
        // Las siguientes las añadimos con el operador +=
        verOperaciones += VerMultiplicacion;
        verOperaciones += VerDivision;

        for (int i = 1; i <= 10; ++i)
            // En esta invocación del objeto delegado se realizará una multidifusión
            // a los tres métodos que referencia, ejecutándose los tres.
            verOperaciones(i + 5, i);
    }
}
```

## Delegados vs Interfaces en Csharp

De lo visto en este tema, podemos deducir que hay **otra forma de aproximarnos al patrón Strategy** además de usando Interfaces como vimos en temas anteriores.

Vamos a tratar de aproximarnos a ambas a través de un **sencillo ejemplo** de uso ya definido en las BCL. Para ello supongamos la siguiente implementación de la clase **Persona** que hemos usado con anterioridad.

```
public record class Persona(string Nombre, int Edad);
```

Supongamos ahora el siguiente programa principal, donde instanciamos una lista de personas...

```
public class Principal
{
    public static void Main()
    {
        List<Persona> personas =
        [
            new ("Sonia", 35), new ("Antonio", 55), new ("Margarita", 32), new ("Manuel", 50)
        ];
    }
}
```

Si quisiéramos ordenar las personas por **Edad**, la clase **list** nos va a ofrecer el método **Sort**. Como nuestra clase **Persona** no implementa **IComparable<Persona>**, deberemos indicarle de algún modo al **Sort** la 'estrategia' de ordenación. Por esta razón **Sort** nos ofrecerá las siguientes sobrecargas ...

1. **public void Sort(IComparer<T>? comparer);**

y si buscáramos la definición del tipo **IComparer<T>** obtendríamos el siguiente **interfaz** parametrizado.

```
public interface IComparer<in T>
{
    int Compare(T? x, T? y);
}
```

2. **public void Sort(Comparison<T> comparison);**

y si buscáramos la definición del tipo **Comparison<T>** obtendríamos el siguiente **delegado** parametrizado.

```
public delegate int Comparison<in T>(T x, T y);
```

Recordemos que si quisiéramos usar el interfaz para ordenar por edad deberíamos definir una clase que implemente el interfaz. Por ejemplo ...

```
public record class Persona(string Nombre, int Edad)
{
    // Definición de la clase que implementa el interfaz
    // con la estrategia de comparación.

    public class ComparaEdad : IComparer<Persona>
    {
        int IComparer<Persona>.Compare(Persona? x, Persona? y) => (x, y) switch
        {
            (null, null) => 0,
            (null, _) => -1,
            (_, null) => 1,
            _ => x.Edad.CompareTo(y.Edad)
        };
    }
}
```

```
public static void Main()
{
    List<Persona> personas =
    [
        new ("Sonia", 35), new ("Antonio", 55),
        new ("Margarita", 32), new ("Manuel", 50)
    ];
    personas.Sort(new Persona.ComparaEdad());
    Console.WriteLine(string.Join(", ", personas));
}
```

Si ejecutamos el programa principal ahora veremos que las personas se han ordenado por edad. **Mostrando por consola:**

```
Persona { Nombre = Margarita, Edad = 32 }
Persona { Nombre = Sonia, Edad = 35 }
Persona { Nombre = Manuel, Edad = 50 }
Persona { Nombre = Antonio, Edad = 55 }
```

Sin embargo, si quisiéramos usar el delegado `Comparison<T>` tendríamos **una composición de código más sencilla...**

```
// Implementamos un método estático que cumpla la signatura del delegado
// e implemente la estrategia de ordenación.

public static int ComparaEdad(Persona p1, Persona p2) => p1.Edad.CompareTo(p2.Edad);

public static void Main()
{
    List<Persona> personas =
    [
        new ("Sonia", 35), new ("Antonio", 55), new ("Margarita", 32), new ("Manuel", 50)
    ];
    personas.Sort(ComparaEdad);
    Console.WriteLine(string.Join(", ", personas));
}
```

## ¿Cuándo usar Delegados vs Interfaces en CSharp?

### Usaremos delegados cuando:

- Se utilice un modelo de diseño de **eventos**.
- Se prefiere a la hora de encapsular un **método estático o de clase**.
- El autor de las llamadas no tiene ninguna necesidad de obtener acceso a otras propiedades, métodos o interfaces en el objeto que implementa el método.
- Se desea conseguir una **composición sencilla**.
- Una clase puede necesitar más de una implementación del método.

### Usaremos interfaces cuando:

- Haya un **grupo de métodos relacionados** a los que se pueda llamar.
- Una clase **sólo necesita una implementación del método**.
- La clase que utiliza la interfaz **deseará convertir esa interfaz en otra interfaz** o tipos de clase.

## Caso de estudio:

Supongamos el siguiente programa de ejemplo, donde dado un array de valores. Queremos calcular la media de las raíces cuadradas de los valores y también la media de elevar  $e$  a los valores.

Una posible propuesta sería la siguiente...

```
class Program
{
    public static double MediaRaiz(double[] valores)
    {
        double total = 0.0;
        for (int i = 0; i < valores.Length; i++) {
            total += Math.Sqrt(valores[i]);
        }
        return total / valores.Length;
    }

    public static double MediaExponencial(double[] valores)
    {
        double total = 0.0;
        for (int i = 0; i < valores.Length; i++) {
            total += Math.Exp(valores[i]);
        }
        return total / valores.Length;
    }

    public static void Main()
    {
        double[] valores = [ 1, 2, 3, 4 ];
        Console.WriteLine("Media raíces:" + MediaRaiz(valores));
        Console.WriteLine("Media exponentes:" + MediaExponencial(valores));
    }
}
```

Sin embargo en la propuesta anterior se repite el código para calcular la media y cómo nos sucedía en otros casos solo se repite la función aplicada al valor.

**Piensa cómo sería la solución usando interfaces.**

Si no se te ocurre puedes ver la solución en la siguiente página...

```

// Debemos definir el interfaz que implemente la función.
public interface IFuncion
{
    double Funcion(double valor);
}

// Definir tipos que implementen el interfaz con la función específica a aplicar.
public class MediaRaíz : IFuncion
{
    public double Funcion(double valor) => Math.Sqrt(valor);
}

public class MediaExponente : IFuncion
{
    public double Funcion(double valor) => Math.Exp(valor);
}

class Program
{
    // Media ahora recibe el objeto que implementa dicho interfaz.
    public static double Media(double[] puntos, IFuncion funcion)
    {
        double total = 0.0;
        for (int i = 0; i < puntos.Length; i++)
        {
            total += funcion.Funcion(puntos[i]);
        }
        return total / puntos.Length;
    }

    public static void Main()
    {
        double[] puntos = { 1, 2, 3, 4 };
        Console.WriteLine("Media raíces:" + Media(puntos, new MediaRaíz()));
        Console.WriteLine("Media exponentes:" + new MediaExponente());
    }
}

```

Cómo vemos en este caso es más apropiado usar delegados porque tenemos una única función. No hay extensión del interfaz y estamos generando mucho código de definición de tipos a cambio repetir el código de cálculo de la media.

### **Piensa cómo sería la solución usando delegados.**

Si no se te ocurre puedes ver la solución en la siguiente página...

Supongamos el siguiente programa de ejemplo, donde dado un array de valores.

```
class Program
{
    // Definimos el tipo delegado que más adelante incluso nos podremos ahorrar
    public delegate double Funcion(double valor);

    public static double Media(double[] puntos, Funcion funcion)
    {
        double total = 0.0D;
        for (int i = 0; i < puntos.Length; i++) {
            total += funcion(puntos[i]);
        }
        return total / puntos.Length;
    }

    public static void Main()
    {
        double[] puntos = [ 1, 2, 3, 4 ];
        Console.WriteLine("Media raíces:" + Media(puntos, Math.Sqrt));
        Console.WriteLine("Media exponentes:" + Media(puntos, Math.Exp));
    }
}
```

## Delegados Parametrizados

Podemos definir un delegado de forma parametrizada o que use genéricos. Esto me permitirá usar un **tipos delegado predefinidos para las signaturas de métodos más comunes que se me pueden dar.**

Por ejemplo, es muy común definir delegados en los que se evalúe un parámetro entrada a modo de '*predicado lógico*' que se evaluarán a cierto o falso.

Una posible definición sería...

```
public delegate bool Predicado<T>(T p);
```

y nos permitiría definir un tipo para cualquier firma de método que evalúe una entrada a modo de predicado. Por ejemplo...

```
// Esté método estático me permite evaluar si es cierto o no
// el predicado 'Valor es un número par'
public static bool EsPar(int valor) => valor % 2 == 0;

// Esté método estático me permite evaluar si es cierto o no
// el predicado 'Texto empieza por mayúscula'
public static bool EmpiezaPorMayuscula(string texto) => texto[0] == char.ToUpper(texto[0]);

static void Main()
{
    // En ambos casos puedo usar la misma definición de
    // delegado parametrizado para referenciar a los
    // métodos anteriores.
    Predicado<int> predicado1 = EsPar;
    Predicado<string> predicado2 = EmpiezaPorMayuscula;

    Console.WriteLine(predicado1(4));
    Console.WriteLine(predicado2("Hola"));
}
```

De hecho, ya existe una definición similar en el espacio de nombres **System** de C#...

```
public delegate bool Predicate<in T>(T obj);
```

y es usada en algunos métodos de las BCL **como por ejemplo** el método

`public List<T> FindAll(Predicate<T> match);` definido en `List<T>`. De tal manera que podríamos usar nuestro método `EsPar` para obtener todos los números pares en una lista de números de la siguiente manera ...

```
List<int> l = [ 1, 2, 3, 4, 5, 6, 7, 8, 9 ];
Console.WriteLine(string.Join(", ", l.FindAll(EsPar)));
```

Además de `delegate bool Predicate<in T>(T obj)` podemos destacar la siguientes definiciones en las BCL de delegados parametrizados...

## Definiciones para procedimientos Action

Bajo el tipo `Action` tendremos predefinidos delegados que admitirán métodos que no retornan nada (`void`) y que podrán tener de 0 a 16 parámetros.

```
// Si no parametrizamos es porque no habrá parámetros de entrada.
public delegate void Action()

...
public delegate void Action<in T1, in T2, ..., in T16>(T1 obj1, T2 obj2, ..., T16 obj16)
```

Un ejemplo de uso de este delegado sería el método `public void ForEach(Action<T> action);` definido en `List<T>`. De tal manera que me permitirá recorrer los elementos de la secuencia e ir aplicando a cada uno de ellos la acción especificada en forma de delegado...

```
public static void Muestra(int valor) => Console.WriteLine($"{valor:D2}");

static void Main()
{
    List<int> valores = [ 2, 6, 3, 8, 2 ];

    // Defino un delegado para métodos con un solo parámetro de entrada entero.
    Action<int> accion = Muestra;
    valores.ForEach(accion);
    // o directamente ... valores.ForEach(Muestra);
}
```

## Definiciones para funciones `Func`

Bajo el tipo `Func` tendremos predefinidos delegados que admitirán métodos que no retornan un tipo a modo de 'función' y que como los `Action` podrán tener de 0 a 16 parámetros.

```
// Fíjate que ahora la parametrización define al menos el tipo R de (Retorno) de la función.  
public delegate R Func<out R>()  
...  
// El tipo de retorno se definirá siempre al final en la parametrización.  
public delegate R Func<in T1, ..., in T16, out R>(T1 obj1, ..., T16 obj16)
```

Iremos encontrando este tipo de delegado más adelante, en ciertas definiciones de las BCL de `System.Linq`. Para poner nuestro ejemplo, vamos a fijarnos que el tipo delegado `Predicado<T>` que definimos en el nuestro primer ejemplo. Se puede definir también con `Func` de la siguiente manera `Func<T, bool>`. Aunque ambos delegados admitan métodos con la misma signatura, para C# serán tipos diferentes.

Veamos cómo reescribir el ejemplo de los predicados usando `Func` ...

```
public static bool EsPar(int valor) => valor % 2 == 0;  
public static bool EmpiezaPorMayuscula(string texto) => texto[0] == char.ToUpper(texto[0]);  
  
static void Main()  
{  
    // En ambos casos al ser el último tipo parametrizado un bool  
    // estaremos indicando que la signatura de los métodos debe retornar eso.  
    Func<int, bool> predicado1 = EsPar;  
    Func<string, bool> predicado2 = EmpiezaPorMayuscula;  
  
    Console.WriteLine(predicado1(4));  
    Console.WriteLine(predicado2("Hola"));  
}
```

Aunque solo se dará en casos aislados, a lo mejor queremos definir un delegado que refiera a un método con más de un parámetro de salida. En este caso, si nos acordamos de principio de curso, usaremos tuplas para definir el valor de retornos.

Recordemos el ejemplo que vimos para este caso...

```
static (double sen, double cos) Direccion(double anguloGr)  
{  
    double anguloRad = anguloGr * Math.PI / 180;  
    return (Math.Sin(anguloRad), Math.Cos(anguloRad));  
}
```

No obstante, como comentamos, a partir de C#7 no sería necesario. Si quisieramos usar `Func` para definir un tipo delegado que referenciase a los métodos anteriores podríamos hacer lo siguiente...

```
static void Main()
{
    Func<double, (double, double)> f = Direccion;
    (double seno, double coseno) = f(90);
    Console.WriteLine($"s={seno:F2}, c={coseno:F2}");
}
```

## 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);
```