

Tema 11.1

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

Índice

1. [Delegados en C#](#)
 1. [Definición](#)
 2. [Usos principales](#)
 3. [Usando delegados en C#](#)
 4. [Multidifusión de delegados](#)
 5. [Delegados vs Interfaces](#)
 1. [💡 ¿Cuándo usar Delegados vs Interfaces?](#)
6. [Delegados Parametrizados](#)
 1. [Definiciones para procedimientos](#)
 2. [Definiciones para funciones](#)

Delegados en C#

Definición

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.

Usos principales

- Ejecutar acciones asíncronas concurrentes tras un evento.
- Ejecutar el código de un método en otro hilo de forma paralela.
- Pasar una referencia a un método como parámetro.
- Simplificar la implementación del patrón Strategy.

Usando delegados en C#

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 sintáxis para definir el tipo será:

```
<modificadores> delegate <tipoRetorno> <TipoDelegado>(<parámetros formales>); (Ver línea 4 del ejemplo)
```

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 sintáxis para instanciar objetos delegado del tipo definido será:

```
TipoDelegado oDelegado = IdMetodoQueCumpleLaSignaturaDelTipo; (Ver línea 28 del ejemplo)
```

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

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

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

```
tipoRetorno resultado = oDelegado(<parámetros reales>); (Ver línea 18 del ejemplo)
```

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>);
```

Veámoslo a través de un ejemplo concreto de uso comentado...

```
class Principal
{
    // Definición del tipo delegado Operacion
    4 public delegate double Operacion(double op1, double op2);

    // Definición de métodos de clase con la misma signatura.
    static double Suma(double op1, double op2) => op1 + op2;
    static double Multiplica(double op1, double op2) => op1 * op2;

    // Recibe un objeto delegado del tipo Operación
    // Esto es, la 'estrategia' a seguir para operar con los arrays.
    static double[] OperaArrays(
        14 double[] ops1, double[] ops2,
        Operacion operacion)
    {
        double[] resultados = new double[ops1.Length];
        for (int i = 0; i < resultados.Length; ++i)
        18 resultados[i] = operacion(ops1[i], ops2[i]);
        return resultados;
    }

    public static void Main()
    {
        double[] ops1 = { 5, 4, 3, 2, 1 };
        double[] ops2 = { 1, 2, 3, 4, 5 };


        27 // Pasamos el nombre (id) la de función ya que Suma es de tipo Operacion
        Operacion op = Suma;
        29 double[] sumas = OperaArrays(ops1, ops2, Suma);
        Console.WriteLine($"Sumas: {string.Join(" ", sumas)}");

        // También podemos pasar el nombre de la función como parámetro real.
        33 double[] multiplicaciones = OperaArrays(ops1, ops2, Multiplica);
        Console.WriteLine($"Multiplicaciones: {string.Join(" ", multiplicaciones)}");
    }
}
```

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 -=**

 **Importante:** 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
{
    3 // 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 VerDividion(int op1, int op2) =>
        Console.WriteLine($"{op1} / {op2} = {op1 / op2}");

    // Definición del tipo delegado con la signarura de los métodos anteriores.
    12 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 +=
        19 verOperaciones += VerMultiplicacion;
        20 verOperaciones += VerDividion;

        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.
            25 verOperaciones(i + 5, i);
    }
}
```

Delegados vs Interfaces

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.

```
class Persona
{
    public string Nombre { get; }
    public int Edad { get; }
    public Persona(string nombre, int edad)
    {
        Nombre = nombre;
        Edad = edad;
    }
    public override string ToString() => $"{Nombre}\t\t{Edad}";
}
```

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

```
class Ejemplo
{
    public static void Main()
    {
        List<Persona> personas = new List<Persona>()
        {
            new Persona("Carmen", 35), new Persona("Antonio", 55),
            new Persona("Xusa", 32), new Persona("Juanjo", 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([AllowNull] T x, [AllowNull] 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 ...

```
class Persona
{
    // ... código omitido para abreviar.
}

5 // Definición de la clase que implementa el interfaz
6 // con la estrategia de comparación.
public class ComparaEdad : IComparer<Persona>
{
    public int Compare([AllowNull] Persona x, [AllowNull] Persona y) => x.Edad.Compar

}

class Ejemplo
{
    public static void Main()
    {
        // ... código omitido para abreviar.

        // Uso de la primera sobrecarga del Sort
20 personas.Sort(new Persona.ComparaEdad());
    }
}
```

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

```
class Ejemplo
{
    3 // Implementamos un método estático que cumpla la signatura del delegado
    // e implemente la estrategia de ordenación.
    5 public static int ComparaEdad(Persona p1, Persona p2) => p1.Edad.CompareTo(p2.Edad);

    public static void Main()
    {
        // ... código omitido para abreviar.

        // Uso de la primera sobrecarga del Sort
    12 personas.Sort(ComparaEdad);
    }
}
```

💡 ¿Cuándo usar Delegados vs Interfaces?

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...

Caso de estudio continuación ...

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

7 // 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.
20 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 };
33 Console.WriteLine("Media raíces:" + Media(puntos, new MediaRaíz()));
34 Console.WriteLine("Media exponentes:" + Media(puntos, 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 de 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...

Caso de estudio continuación ...

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 = 0.0;
        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 signatura de método que evalúe una entrada a modo de predicado. Por ejemplo...

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

// Este método estático me permite evaluar la si es cierto o no
// el predicado 'Texto empieza por mayúscula'
public static bool EmpiezaPorMayuscula(string texto)
=> texto != null && 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 = new List<int>() { 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

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 = new List<int> { 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

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>`.

 **Nota:** Aunque admitan métodos con la misma signatura, para C# serán tipos diferentes.

Sin embargo, podremos reescribir el código de nuestro primer ejemplo sin necesidad de definir el tipo `public delegate bool Predicado<T>(T p);` y usando los tipos ya definidos en las BCL.

```
public static bool EsPar(int valor) => valor % 2 == 0;
public static bool EmpiezaPorMayuscula(string texto)
=> texto != null && texto[0] == char.ToUpper(texto[0]);

static void Main()
{
    7 // 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;
    10 Func<string, bool> predicado2 = EmpiezaPorMayuscula;

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

Aunque solo se dará en casos aislados porque es fuente de baja cohesión, a lo mejor queremos definir un delegado que referencie 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 quisiéramos 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}");
}
```

Nota ampliación:

Aunque no lo comentamos a principio de curso, por no haber visto aún el concepto de tipo parametrizado. **(double sen, double cos)** es en realidad un tipo **Tuple<T1, T2, ..., Tn>** esto es, es un '*syntactic sugar*' de para la definición **Tuple<double, double>** por tanto sería equivalente escribir nuestra definición cómo

```
static Tuple<double, double> Direccion(double anguloGr)
{
    double anguloRad = anguloGr * Math.PI / 180;
    return new Tuple<double, double>(Math.Sin(anguloRad), Math.Cos(anguloRad));
}
```

Si quisiéramos que nuestro código funcionara también en las **versiones anteriores a C#7**, donde no están definida la sintaxis nueva de tuplas, deberíamos reescribir el código anterior de la siguiente manera...

```
static void Main()
{
    Func<double, Tuple<double, double>> f = Direccion;
    Tuple<double, double> direccion = f(90);
    Console.WriteLine($"s={direccion.Item1:F2}, c={direccion.Item2:F2}");
}
```

Si te fijas, es equivalente, pero es más simple e intuitiva la sintaxis a partir de C#7