

C#.Net

Propiedades implementadas automáticamente

- Con mucha frecuencia las propiedades están asociadas a campos privados de las clases

```
class Persona{  
    private string nombre;  
    public string Nombre {  
        get { return nombre; }  
        set { nombre = value; }  
    }  
}
```

Propiedades implementadas automáticamente

- Para facilitar la tarea del programador C#3.0 introdujo las propiedades auto-implementadas
- Con ellas es posible declarar la propiedad sin declarar el campo asociado
- El compilador crea un campo oculto que asocia a la propiedad auto-implementada
- Debido a que no es posible acceder al campo oculto, sólo se implementan automáticamente propiedades de lectura/escritura

Propiedades implementadas automáticamente

- La clase **Persona** puede re-escribirse de la siguiente manera:

```
class Persona{  
    public string Nombre{  
        get;  
        set;  
    }  
}
```

Propiedades implementadas automáticamente

- Un buen hábito es no codificar campos públicos, sino hacer públicas las propiedades y privados todos los campos
- Usando las propiedades auto-implementadas puede hacerse fácilmente simplemente agregando {get;set;} al final del campo

```
class Persona{  
    public string Nombre;  
    public int Edad;  
    public int Dni;  
}
```

Clase con tres campos públicos

```
class Persona{  
    public string Nombre{get;set;}  
    public int Edad{get;set;}  
    public int Dni{get;set;}  
}
```

Clase con tres propiedades públicas

Sintaxis de Inicialización de Objetos

- Para facilitar la inicialización de objetos, además de la utilización de constructores adecuados, C# provee una sintaxis específica para ello.
- Un inicializador consiste en una lista de elementos separada por comas encerradas entre llaves { } que sigue a la invocación del constructor
- Cada miembro de la lista mapea con un campo o propiedad pública del objeto al que le asigna un valor.

```
Persona p=new Persona ( ) {Nombre="Juan" , Edad=40 , Dni=28765421} ;
```



También se podría invocar cualquier otro constructor que se haya definido en la clase Persona

Delegados

- **Concepto:** Tipo especial de clase cuyos objetos almacenan referencias a uno o mas métodos de manera de poder ejecutar en cadena esos métodos.
- Permiten pasar métodos como parámetros a otros métodos
- Proporcionan un mecanismo para implementar eventos

Delegados

- Sintaxis definición de un delegado

<modif> **delegate** <tipoRet> <nombreDelegado> (<parám>);

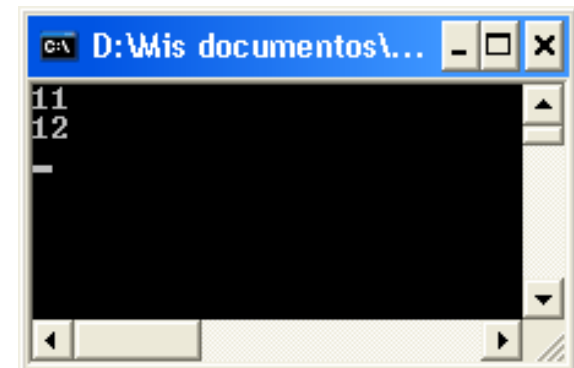
- Cuando se trabaja con una variable de un tipo delegado no es necesario instanciarla explícitamente simplemente se le asigna el nombre del método
- Si por el contrario si se instancia explícitamente utilizando el constructor se debe pasar como parámetro el nombre del método

Codifique y ejecute

```
using System;
delegate int FuncionEntera(int n);
class Program{
    static void Main(){
        FuncionEntera f = new FuncionEntera(sumaUno);
        Console.WriteLine(f(10));
        f=sumaDos;
        Console.WriteLine(f(10));
        Console.ReadKey(true);
    }
    static int sumaUno(int n){
        return n+1;
    }
    static int sumaDos(int n){
        return n+2;
    }
}
```

FuncionEntera es un tipo delegado que se corresponde con los métodos que reciben un parámetro de tipo **int** y que devuelven un valor de tipo **int**

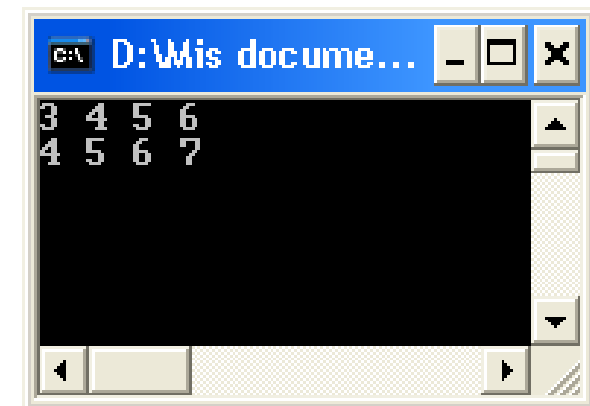
Para asignar **f**, en un caso se utiliza un constructor de **FuncionEntera** y en otro caso no. El resultado es el mismo



Codifique y ejecute

```
static void aplicar(int[] v, FuncionEntera f){  
    for(int i=0;i<v.Length;i++){  
        v[i]=f(v[i]);  
    }  
}  
  
static void Main(){  
    int[] v = new int[] {1,2,3,4};  
    aplicar(v,sumaDos);  
    foreach(int i in v) Console.Write(i+" ");  
    aplicar(v,sumaUno);  
    Console.WriteLine();  
    foreach(int i in v) Console.Write(i+" ");  
    Console.ReadKey(true);  
}
```

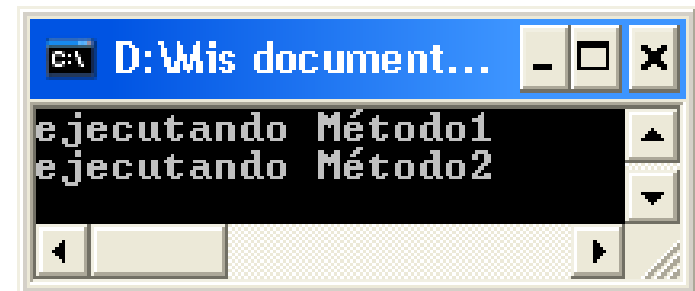
Agregue el método
aplicar y reescriba el
método Main



Codifique y ejecute

```
using System;
delegate void MetodoSinParametro();
class Program{
    static void Main(){
        MetodoSinParametro m;
        m=metodo1;
        m=m+metodo2; ←
        m();
        Console.ReadKey(true);
    }
    static void metodo1(){
        Console.WriteLine("ejecutando Método1");
    }
    static void metodo2(){
        Console.WriteLine("ejecutando Método2");
    }
}
```

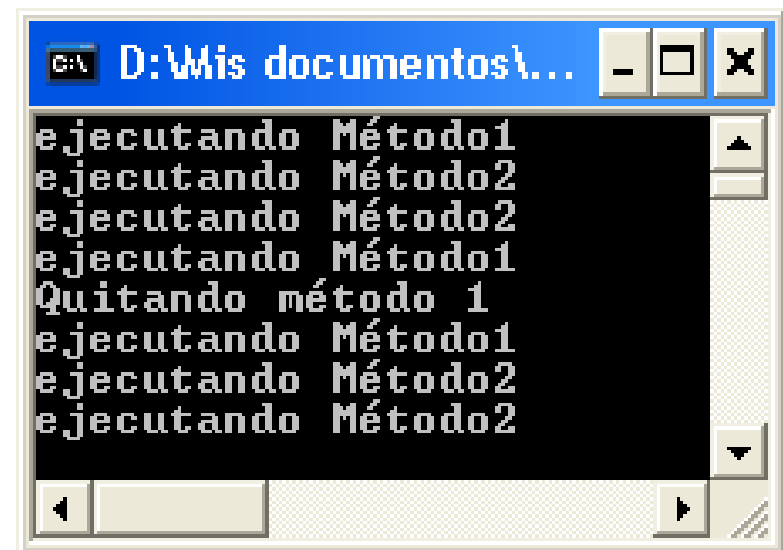
Una variable de tipo delegado puede contener una lista de métodos que serán invocados secuencialmente



Codifique y ejecute

```
static void Main(){  
    MetodoSinParametro m;  
    m=metodo1;  
    m+=metodo2;  
    m+=metodo2;  
    m+=metodo1;  
    m();  
    Console.WriteLine("Quitando método 1");  
    m-=metodo1;  
    m();  
    Console.ReadKey(true);  
}
```

Modifique el método
Main



```
ejecutando Método1  
ejecutando Método2  
ejecutando Método2  
ejecutando Método1  
Quitando método 1  
ejecutando Método1  
ejecutando Método2  
ejecutando Método2
```

Utilizar delegados a modo de eventos

```
using System;
```

```
delegate void TrabajoFinalizadoEventHandler();
```

```
class Trabajador
```

```
{
```

```
    public TrabajoFinalizadoEventHandler TrabajoFinalizado;
```

```
    public void Trabajar(){
```

```
        Console.WriteLine("trabajador trabajando");
```

```
        // hace algún trabajo
```

```
        if ( TrabajoFinalizado != null) {
```

```
            TrabajoFinalizado();
```

```
        }
```

```
    }
```

```
}
```

Por convención suele agregarse el subfijo EventHandler al nombre del tipo delegado que se utiliza para implementar eventos

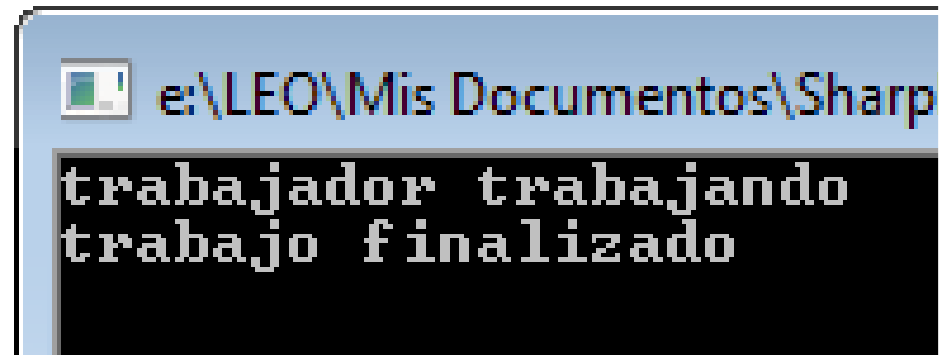
TrabajoFinalizado será un evento producido por un objeto Trabajador

Aquí se produce el evento invocando la lista de métodos encolados en el delegado. Si no se ha encolado ningún método la variable tiene el valor null, por eso es necesario verificarlo antes de intentar realizar la invocación.

Utilizar delegados a modo de eventos

```
class Program
{
    public static void Main()
    {
        Trabajador t = new Trabajador();
        t.TrabajoFinalizado = new TrabajoFinalizadoEventHandler(trabajoFinalizado);
        t.Trabajar();
        Console.ReadKey(true);
    }
    private static void trabajoFinalizado(){
        Console.WriteLine("trabajo finalizado");
    }
}
```

La clase Program se suscribe al evento **TrabajoFinalizado** del objeto **t**, asignando su propio método **trabajoFinalizado** para manejar dicho evento



e:\LEO\Mis Documentos\Sharp

```
trabajador trabajando
trabajo finalizado
```

Convenciones de nomenclatura

- Por convención, si es posible, los nombres que se usen para los eventos, los delegados y las clases de datos de evento deberían compartir una raíz común. Por ejemplo, si define un evento **CapacityExceeded** en una de sus clases, el delegado se debería denominar **CapacityExceededEventHandler** y los datos de evento se deberían denominar **CapacityExceededEventArgs**.
- Para los nombres de los eventos se utilizarán preferentemente verbos en **gerundio** (ejemplo *IniciandoTrabajo*) o **participio** (ejemplo *TrabajoFinalizado*) según se produzcan antes o después del hecho de significación.

Ejemplo

- Se implementará una clase **Ingresador** para introducir texto desde la consola repetidamente hasta que el usuario tipee el número 0 (cero)
- El ingresador provocará un evento **NumTipeado** cuando el texto introducido por el usuario se corresponda con un valor numérico válido. En caso contrario simplemente ignora la entrada y prosigue en el loop.
- La clase Program creará un objeto **Ingresador** y se suscribirá al evento **NumTipeado** de este objeto manejándolo con un método que recibirá un parámetro **NumTipeadoEventArgs** en cuya propiedad **Valor** se encontrará el número ingresado por el usuario

Ejemplo - Resolución

```
delegate void NumTipeadoEventHandler(NumTipeadoEvenArgs e);
```

```
class Program{
```

```
    static void Main(){
```

```
        Ingresador ing = new Ingresador(); ←
```

Se crea un objeto Ingresador

```
        ing.NumTipeado = ingNumTipeado; ←
```

Se suscribe el evento

```
        ing.Ingresar(); ←
```

Se invoca el método Ingresar

```
    }
```

```
    static void ingNumTipeado(NumTipeadoEvenArgs e){ ←
```

```
        Console.WriteLine("Se ha ingresado {0}", e.Valor);
```

```
    }
```

```
}
```

Método con el cual Program se suscribió al evento. Cada vez que se produzca el evento NumTipeado se ejecutará este método. Es decir que es el método con el cual Program maneja el evento

Ejemplo - Resolución

```
class NumTipeadoEventArgs : EventArgs {  
    public double Valor {get;set;}  
}
```

No es obligatorio, pero se recomienda que derive de EventArgs

```
class Ingresador {  
    public NumTipeadoEventHandler NumTipeado;  
    public void Ingresar(){  
        NumTipeadoEventArgs e = new NumTipeadoEventArgs(){Valor = -1};  
        while (e.Valor != 0){  
            try{  
                e.Valor = double.Parse(Console.ReadLine());  
                if (NumTipeado != null)  
                    NumTipeado(e);  
            }catch{ }  
        }  
    }  
}
```

Publica la variable NumTipeado para que puedan suscribirse al evento

Invoca todos los métodos encolados. Si no se ha encolado ningún método la variable tiene el valor null, por eso es necesario verificarlo antes de intentar realizar la invocación.

Ejemplo 2

- Se desea modificar la clase **Ingresador** para poder controlar externamente la finalización del loop de ingreso de datos.
- El que se suscriba al evento tendrá también la posibilidad de finalizar la entrada de datos estableciendo convenientemente una propiedad del objeto **NumTipoadoEventArgs** recibido como parámetro
- Por lo tanto **NumTipoadoEventArgs** tendrá dos propiedades, una para alojar el valor numérico ingresado (**Valor**) y la otra para ser establecida por el suscriptor indicando el fin a la entrada de datos (**FinDeIngreso**)

Ejemplo 2 - Resolución

```
class Program{
    static void Main(){
        Ingresador ing = new Ingresador();
        ing.NumTipeado = ingNumTipeado;
        ing.Ingresar();
    }
    static void ingNumTipeado(NumTipeadoEvenArgs e){
        Console.WriteLine("Se ha ingresado {0}",e.Valor);
        if (e.Valor == 0)
            e.FinDeIngreso = true;
    }
}
```

Ejemplo 2 - Resolución

```
class NumTipeadoEvenArgs:EventArgs {  
    public double Valor {get;set;}  
    public bool FinDeIngreso {get;set;}  
}  
  
class Ingresador {  
    public NumTipeadoEventHandler NumTipeado;  
    public void Ingresar(){  
        NumTipeadoEvenArgs e;  
        e=new NumTipeadoEvenArgs(){FinDeIngreso=false};  
        while ( !e.FinDeIngreso ){  
            try{  
                e.Valor = double.Parse(Console.ReadLine());  
                if (NumTipeado != null)  
                    NumTipeado(e);  
            }catch{ }  
        }  
    }  
}
```

Parámetro sender en los eventos

- Otra convención para los delegados asociados a eventos es la utilización de dos parámetros:
 - El primero, un objeto genérico (de la clase `object`) llamado `sender` utilizado para que el propio objeto que produce el evento se envíe a sí mismo
 - El segundo corresponde al argumento `EventArgs` o alguno derivado como ya hemos visto.
- Ejemplo:

```
delegate void AcontecimientoEventHandler(  
    object sender, EventArgs e);
```

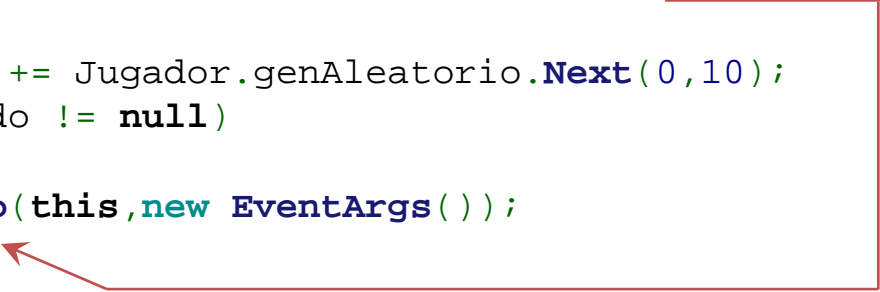
 - Aunque el argumento `EventArgs` no posee propiedades ni comportamiento alguno, podría invocarse con cualquier objeto de una clase derivada.

Ejemplo

```
delegate void MovidoEventHandler(object sender, EventArgs e);
```

```
class Jugador
{
    static Random genAleatorio=new Random();
    int posicion=0;
    public char Id{get;set;}
    public MovidoEventHandler Movido;
    public void Mover()
    {
        posicion += Jugador.genAleatorio.Next(0,10);
        if (Movido != null)
        {
            Movido(this,new EventArgs());
        }
    }
    public void Imprimirse()
    {
        Console.WriteLine("Jugador {0} => posición {1}",Id,posicion);
    }
}
```

El objeto Jugador se envía a sí mismo (**this**) anunciando que él es quien provocó el evento

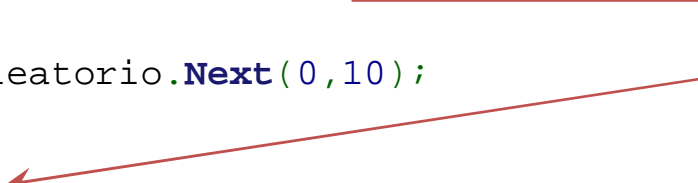


Ejemplo

```
delegate void MovidoEventHandler(object sender, EventArgs e);

class Jugador
{
    static Random genAleatorio=new Random();
    int posicion=0;
    public char Id{get;set;}
    public MovidoEventHandler Movido;
    public void Mover()
    {
        posicion += Jugador.genAleatorio.Next(0,10);
        if (Movido != null)
        {
            Movido(this,new EventArgs());
        }
    }
    public void Imprimirse()
    {
        Console.WriteLine("Jugador {0} => posición {1}",Id,posicion);
    }
}
```

Puesto que no es necesario proveer información adicional, podría enviarse simplemente null. Sin embargo se recomienda instanciar y enviar un objeto EventArgs



Ejemplo (Cont.)

```
using System;
class Programa
{
    static void Main()
    {
        Jugador j1=new Jugador() {Id='A'};
        Jugador j2=new Jugador() {Id='B'};
        j1.Movido=jugadorMovido;
        j2.Movido=jugadorMovido;
        for(int i=1;i<=10;i++)
        {
            j1.Mover();
            j2.Mover();
        }
        Console.ReadKey();
    }
    static void jugadorMovido(object sender, EventArgs e)
    {
        (sender as Jugador).Imprimirse();
    }
}
```

sender es de tipo object por lo tanto debe convertirse a Jugador para acceder a sus miembros.