

Hada T2: Programación Dirigida por Eventos

Un nuevo paradigma de programación.

Departamento de Lenguajes y Sistemas Informáticos Universidad de Alicante

Objetivos del tema

- Conocer el paradigma de la *Programación Dirigida por Eventos*.
- Aprender a programar bajo este nuevo paradigma.
- Saber y entender cómo se puede realizar en **C#**.
- Conocer, distinguir y saber emplear los conceptos de *señal* y *callback*.

Preliminares

- En términos de la estructura y la ejecución de una aplicación representa lo opuesto a lo que hemos hecho hasta ahora: *programación secuencial*.
- La manera en la que escribimos el código y la forma en la que se ejecuta éste está determinada por los sucesos (*eventos*) que ocurren como consecuencia de la interacción con el mundo exterior.
- Podemos afirmar que representa un nuevo *paradigma de programación*, en el que todo gira alrededor de los eventos o cambios significativos en el estado de un programa.

Programación secuencial vs. dirigida por eventos I

- En la *programación secuencial* le decimos al usuario lo que puede hacer a continuación, desde el principio al final del programa.
- El tipo de código que escribimos es como éste:

```
repetir
    presentar_menu ();
    opc = leer_opcion ();
    ...
    si (opc == 1) entonces accion1 ();
    si (opc == 2) entonces accion2 ();
    ...
hasta terminar
```

Programación secuencial vs. dirigida por eventos II

- En la *programación dirigida por eventos* indicamos:
 1. ¿Qué cosas -eventos- pueden ocurrir?
 2. Lo que hay que hacer cuando estos ocurran
- El tipo de código que escribimos es como éste:

```
son_eventos (ev1, ev2, ev3...);  
...  
cuando_ocurra ( ev1, accion1 );  
cuando_ocurra ( ev2, accion2 );  
repetir  
...  
hasta terminar
```

Programación secuencial vs. dirigida por eventos III

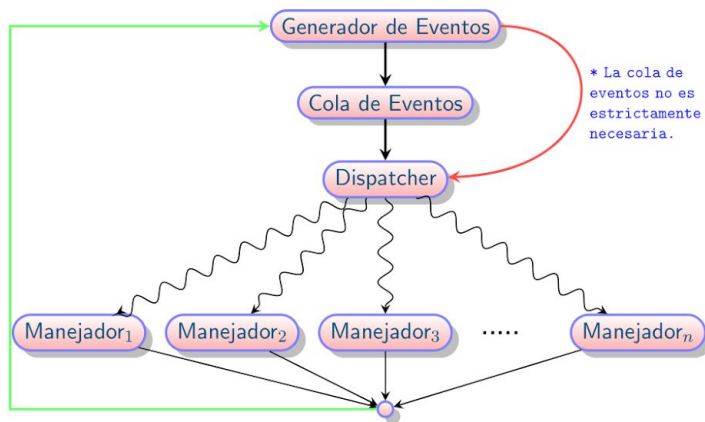
- A partir de este punto los eventos pueden ocurrir en cualquier momento y marcan la ejecución del programa.
- Aunque no lo parezca plantean un problema serio: el flujo de la ejecución del programa escapa al programador.
- El usuario (como fuente generadora de eventos) toma el control sobre la aplicación.
- Esto implica tener que llevar cuidado con el diseño de la aplicación teniendo en cuenta que el orden de ejecución del código no lo marca el programador y, además, puede ser distinto cada vez.

Esqueleto de una aplicación dirigida por eventos I

- Al principio de la misma llevamos a cabo una iniciación de todo el sistema de eventos.
- Se definen todos los eventos que pueden ocurrir.
- Se prepara el generador o generadores de estos eventos.
- Se indica qué código se ejecutará en respuesta a un evento producido *-ejecución diferida de código-*.
- Se espera a que se vayan produciendo los eventos.
- Una vez producidos son detectados por el “*dispatcher*” o planificador de eventos, el cual se encarga de invocar el código que previamente hemos dicho que debía ejecutarse.

Esqueleto de una aplicación dirigida por eventos II

- Todo esto se realiza de forma ininterrumpida hasta que finaliza la aplicación.
- A esta ejecución ininterrumpida es a lo que se conoce como el bucle de espera de eventos.
- Las aplicaciones con un interfaz gráfico de usuario siguen este esquema de programación que acabamos de comentar. Gráficamente sería algo así:



¿Necesitamos un lenguaje de programación especial? I

- No, La ejecución diferida de código tiene sus orígenes en el concepto de Callback.
- En Lenguaje **C** un *callback* no es más que un puntero a una función.
- En la propia biblioteca estándar de **C** hay varios ejemplos de ello.
 - Hagamos un pequeño ejercicio: en la biblioteca estándar de **C** (`#include <stdlib.h>`)

disponemos de la función “*qsort - ordena un vector*”. El prototipo de la misma es:

```
void qsort(void *base, size_t nmemb, size_t size,  
           int(*compar)(const void *, const void *));
```

- Identifica cada uno de sus parámetros. Trata de entender qué es el parámetro “*compar*”. **Propón un posible valor para ese parámetro.**

¿Necesitamos un lenguaje de programación especial? II

- Cuando tengas hecho todo este estudio trata de crear un programa ejecutable mínimo que te sirva para comprobar cómo funciona `qsort`.

```
int int_cmp (const void* pe1, const void* pe2) {  
    int e1 = *((const int*) pe1);  
    int e2 = *((const int*) pe2);  
    return e1-e2;  
}
```

```
int main () {  
    int v[4] = {4,0,-1,7};  
  
    for (int i = 0; i < 4; i++)  
        printf("v[%d]=%d\n", i, v[i]);
```

```
    qsort (v, 4, sizeof(int), int_cmp,  
    printf("\n");  
    for (int i = 0; i < 4; i++)  
        printf("v[%d]=%d\n", i, v[i]);  
    return 0;  
}
```



Ejecución diferida

¿Necesitamos un lenguaje de programación especial? III

- **C#** nos proporciona una construcción de más alto nivel que un puntero a una función.
- Es lo que denomina un “*delegado*”.
- Un “*delegado*” actúa como un puntero a una función pero añade comprobación de tipos en tiempo de compilación.

El principio de Hollywood I

- Las pautas de uso de la *programación dirigida por eventos* se pueden resumir con el llamado *principio de Hollywood*.
- Este es muy sencillo de entender: **No nos llame...ya le llamaremos**
- Se emplea sobre todo cuando se trabaja con *frameworks*.
- El flujo de trabajo se parece a esto:
 1. En el caso de trabajar con un *framework* implementamos un interfaz y en el caso más sencillo escribimos el código a ejecutar más adelante.
 2. Nos registramos...es decir, indicamos de algún modo cuál es el código a ejecutar posteriormente.
 3. Esperamos a que se llame -al código registrado previamente- cuando le corresponda, recuerda: *No nos llame...ya le llamaremos*.

El principio de Hollywood II

- El programador ya no *dicta* el flujo de control de la aplicación, sino que son los eventos producidos los que lo hacen.
- Puedes consultar más información sobre él [aquí](#).
- Si quieres ampliar más tus conocimientos sobre él debes saber que a este principio también se le conoce por otros nombres:
 1. Inversión de control: [IoC](#).
 2. Inyección de dependencias [DI](#).
- Puedes ampliar más información sobre estas técnicas de programación en asignaturas como Programación-3.

Delegados en C# I

- Un delegado es un *tipo de dato* que representa una referencia a un método con una serie de parámetros y un tipo de resultado.
- Podemos declarar variables de este tipo y asignarles un valor, el cual es un método cuya signatura se corresponde con la del delegado -se permite el uso de varianza en el tipo del resultado-.
- Posteriormente podemos invocar estos métodos a través del *delegado*.
- Se pueden usar delegados p.e. para pasar métodos como argumentos de otros métodos (repasa el ejemplo de *qsort* en **C** visto antes).
- A un delegado le podemos asignar métodos de clase y de instancia.

Delegados en C# II

- La sintaxis empleada para declarar un delegado es esta:

```
public delegate int PerformCalculation(int x, int y);
```

- Un extracto de ejemplo de uso:

```
delegate Person lesser (Person p1, Person p2);
class Person {
    ...
    public static Person nameLesser (Person p1, Person p2){...}
    public static Person ageLesser (Person p1, Person p2) {...}
    public static void showPrecedes (Person p1, Person p2,
                                     lesser cf) {

        Person p = cf(p1, p2);
        if (p != null) {
            Console.WriteLine ("{0} with age {1}.", p.name, p.age);
        }
    }
    // DATOS //
    public string name {get; set;}
    public int age {get; set;}
}
```

```
public static void Main () {
    Person juan = new Person ("Juan", 20);
    Person andres = new Person ("Andres", 30);
    Person.showPrecedes (juan, andres,
                        Person.ageLesser);

    Person.showPrecedes (juan, andres,
                        Person.nameLesser);
}
```

Funciones Lambda en C#

- C# permite crear funciones anónimas (**sin nombre**) también llamadas funciones lambda.
- Estas se definen en el punto donde se usan.
- Podemos hacerlo de varias maneras:

```
// Create a delegate.  
delegate void Tdel(int x);  
  
// Instantiate the delegate using an anonymous method.  
Tdel d = delegate(int k) { /* ... */ };
```

```
delegate int Tdel(int i);  
static void Main(string[] args) {  
    Tdel myDelegate = x => x * x;  
    int j = myDelegate(5); //j = 25  
}  
  
// Si tiene un num. de param. ≠ 1  
1. (x, y) => x == y  
2. (int x, string s) => s.Length > x  
3. () => SomeMethod()
```


Marco de *Prog. dirigida por eventos* en C#.Net

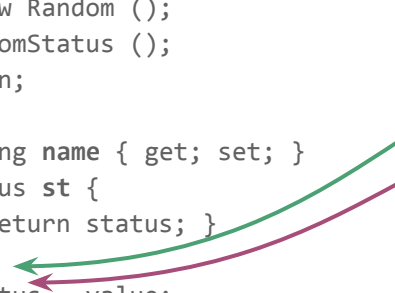
- El modelo empleado en C# para hacer uso de esta técnica es el siguiente:
 - Hace uso del concepto de *delegado* visto previamente.
 - Introduce un elemento nuevo, el *evento* (**event**).
 - Los eventos representan los cambios de estado de un objeto y a ellos les podemos “conectar” el código que queramos que se ejecute cuando se generen (*callback*).
- Cada vez que se produzca un evento para un objeto concreto de una clase, se ejecutarán todos los *callbacks* que le hayamos asociado, secuencialmente uno tras otro y no tiene porqué ser en el orden en que se conectaron.
- C# nos da soporte sintáctico para representar estas *ideas* en el código que escribamos, veámoslo con un ejemplo.

Ejemplo de eventos en C# I

- Vamos a crear una clase *abstracta* que representa un biestable, la llamaremos **FlipFlop**.
- Los objetos de esta clase (y clases derivadas) generarán el *evento* **statusChanged** cada vez que su estado cambie.
- De esta clase heredarán todas aquellas clases que representen realizaciones concretas de la misma (luz, puerta, etc...).

Ejemplo de eventos en C# II

```
public abstract class FlipFlop {  
    public enum Status { off, on }  
    public FlipFlop (string n = " --- ") {  
        rg = new Random ();  
        setRandomStatus ();  
        name = n;  
    }  
    public string name { get; set; }  
    public Status st {  
        get { return status; }  
        set {  
            status = value;  
            if (statusChanged != null)  
                statusChanged (  
                    this,  
                    new StatusChangeArgs (st)  
                );  
        }  
    }  
}
```



↓

```
public void toggle () {  
    if (st == Status.on)  
        st = Status.off; // Es una propiedad!  
    else  
        st = Status.on;  
}  
public void setRandomStatus () {  
    if (rg.Next (2) == 0)  
        st = Status.off; // ejecuta set  
    else  
        st = Status.on; // idem  
}  
public void showStatus () {  
    Console.WriteLine ("Status is {0}.", st);  
}
```

↓

Ejemplo de eventos en C# III

↓

```
public event EventHandler<StatusChangeArgs> statusChanged;  
private Status status;  
private Random rg;  
}  
} // Fin clase FlipFlop
```

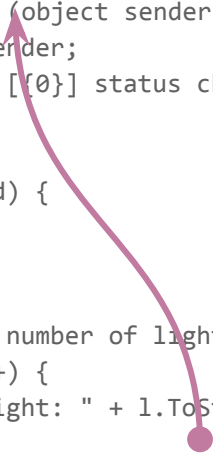
```
public class StatusChangeArgs : EventArgs {  
    private FlipFlop.Status s;  
    public StatusChangeArgs (FlipFlop.Status s) { this.s = s; }  
    public FlipFlop.Status status { get { return s; } }  
}
```

Ejemplo de eventos en C# IV

```
class Light : FlipFlop {  
    public Light (string n) : base (n) {  
        System.Console.WriteLine ("Light built.");  
    }  
}
```

Ejemplo de eventos en C# V

```
public class House {
    private Random rg;
    private List<Light> lal;
    ...
    private void itemStatusChanged (object sender, StatusChangeArgs e) {
        FlipFlop ff = (FlipFlop) sender;
        Console.WriteLine ("Object [{0}] status changed to: [{1}].", ff.name, e.status);
    }
    ...
    public House(int maxl, int maxd) {
        rg = new Random ();
        lal = new List<Light> ();
        dal = new List<Door> ();
        var nl = rg.Next(maxl); // number of lights
        for (int l = 0; l < nl; l++) {
            var lgt = new Light("light: " + l.ToString());
            lal.Add (lgt);
            lgt.statusChanged += itemStatusChanged; // Conexion del callback al evento!
            lgt.toggle ();                          // Forzamos a que se emita el evento!
        }
        ...
    }
}
```



Ejemplo de eventos en C# VI

Resumiendo:

- Para cada evento que pueda generar una clase debemos definir una variable en la clase de tipo **EventHandler<T>**:

```
public event EventHandler<...> statusChanged;
```

- El tipo genérico **T** se instanciará al de la clase **EventArgs** o derivada de ella:

```
public event EventHandler<StatusChangeArgs> statusChanged;
```

- A esta variable de tipo **evento** le conectaremos todos aquellos métodos que necesitemos, p.e.:

```
private void itemStatusChanged (object sender, StatusChangeArgs e)...
```

```
lgt.statusChanged += itemStatusChanged;
```

- Lo invocaremos así: `statusChanged (this, new StatusChangeArgs (st));`

