Tema 5 – Delegados y eventos

DELEGADOS

Un delegado es una referencia a un método, un delegado permite acceder a un método de forma casi anónima, ya que simplemente tiene la dirección de memoria de dicho método. Pero en .NET esto debe estar controlado, de forma que ese acceso no sea arbitrario. Un delegado es un tipo especial que permite definir la forma de acceder a un método.

Un delegado es un tipo especial de clase cuyos objetos pueden almacenar referencias a uno o más métodos de tal manera que a través del objeto sea posible solicitar la ejecución en cadena de todos ellos.

Declaración de delegados

La forma de controlar el acceso a un método es definiendo un delegado en el que se indica de qué tipo es ese método, si recibe parámetros, y de hacerlo cuántos y de qué tipo son. Se puede pensar que un delegado es en cierto modo similar a las interfaces ya que definen un contrato que hay que respetar.

Si se quiere acceder a un método que devuelve una cadena y que recibe un parámetro de tipo Cliente, habría que definir un delegado con esas características, y cuando posteriormente se quiera acceder a ese método, en lugar de hacerlo directamente, se usará un objeto del tipo definido por el delegado.

<modificadores> delegate <tipoRetorno> <nombreDelegado> (<parámetros>);

```
public delegate void MiDelegado(string mensaje);
```

<tipoRetorno> y <parámetros> se corresponderán, respectivamente, con el tipo del valor de retorno y la lista de parámetros de los métodos cuyos códigos puede almacenar en su interior los objetos de ese tipo delegado (objetos delegados). Lo que importa es que los tipos de los argumentos del método y el tipo devuelto coincidan con los del delegado. Esto hace que los delegados sean perfectos para una invocación "anónima" de métodos.

Ejemplo de método que devuelve una cadena y recibe un parámetro de tipo Cliente sin usar ningún puntero, simplemente se accede de forma directa. Es más, debido a que no se indica dónde está dicho método, se supone que se esta accediendo desde la propia clase en la que está definido.

```
class Cliente{
}

class Program{
	public static string MiFuncion(Cliente c){
		return "Hola";
	}

	public static void Main(){
		string s = MiFuncion( new Cliente() );
	}
}
```

Para asociar un método a un delegado e invocar con el delegado el método, primero hay que definir ese delegado.

```
// Definicion de un delegado para acceder a un
// metodo definido con la misma firma.
delegate string MiFuncionDelegado(Cliente c);
```

Se crea un método con la misma firma que el delegado en lo que se refiere a parámetros y tipos devueltos.

```
class Cliente{
	public static string MiFuncion(Cliente c){
		return "Hola soy el cliente";
	}
}
```

Para usar el delegado, se define una variable de ese tipo, y como los delegados en realidad son como clases, se puede usar el mismo código para crear cualquier tipo, con la diferencia de que en el constructor hay que indicar la método al que hace referencia el delegado.

Se puede usar los delegados de forma directa, es decir, sin necesidad de que crear una instancia de la clase del delegado.

```
class Program{
	public static void Main(){
		// dos formas equivalentes de asignar el puntero a una funcion.

		MiFuncionDelegado mfd = new MiFuncionDelegado(Cliente.MiFuncion);
		string s = mfd( new Cliente() );

		// uso de delegados de forma directa
		// sin necesidad de crear una instancia
		MiFuncionDelegado mfd2 = Cliente.MiFuncion;
		string s2 = mfd2( new Cliente() );
	}
}
```

En este caso hay que tener cuidado con los modificadores de acceso public del delegado y de la clase Cliente, El delegado apunta a un método que recibe una instancia de esta clase como argumento por lo que si el delegado usa public la clase también deberá usar public, o la clase tener un acceso menos restrictivo que el delegado.

```
public delegate string MiFuncionDelegado(Cliente c);

public class Cliente{
	public static string MiFuncion(Cliente c){
		return "Hola soy el cliente";
	}
}
```

Las clases y struct declarados directamente en un espacio de nombres pueden ser públicos o internos. Si no se especifica un modificador de acceso, internal es el valor predeterminado.

Usar un tipo delegado como parámetro de un método puede ayudar para separar diferentes implementaciones.

```
namespace back
{
	// Definicion de un delegado para acceder a un
	// metodo definido con la misma firma.
	delegate string MiFuncionDelegado(Cliente c);
	class Cliente{
		public string MiFuncion(Cliente c){
			return "Hola soy el cliente";
		}
	}
}

namespace front
{
	using back;
	class Program{
		// metodo que recibe un tipo delegado
		public static void usarMiFuncionDelegado(MiFuncionDelegado mfd){
			string s = mfd( new Cliente() );
			Console.WriteLine(s);
		}
		public static void Main(){
			Cliente c = new Cliente();
			usarMiFuncionDelegado(c.MiFuncion);
			// usarMiFuncionDelegado(new MiFuncionDelegado(c.MiFuncion));
		}
	}
}
```

Otro ejemplo

```
// declaracion de delegado
public delegate void MiDelegado(string mensaje);

class Program{
	// espera un delegado del tipo MiDelegado para
	// invocar el metodo que tiene encapsulado
	public void TestDelegado(MiDelegado delegado){
		// se pasa un argumento al metodo encapsulado
		// que sera invocado por el delegado
		delegado("hola delegado");
	}

	// metodo que encapsulara el delegado
	// tiene el mismo tipo de retorno y parametros que
	// el delegado MiDelegado
	public void MetodoDelegado(string mensaje){
		Console.WriteLine(mensaje);
	}

	static void Main(string[] args){
		Program app = new Program();
		// objeto delegado que encapsula el metodo MetodoDelegado
		MiDelegado delegado = new MiDelegado(app.MetodoDelegado);
		// se pasa como argumento el objeto delegado a TestDelegado
		app.TestDelegado(delegado);
	}
}
```

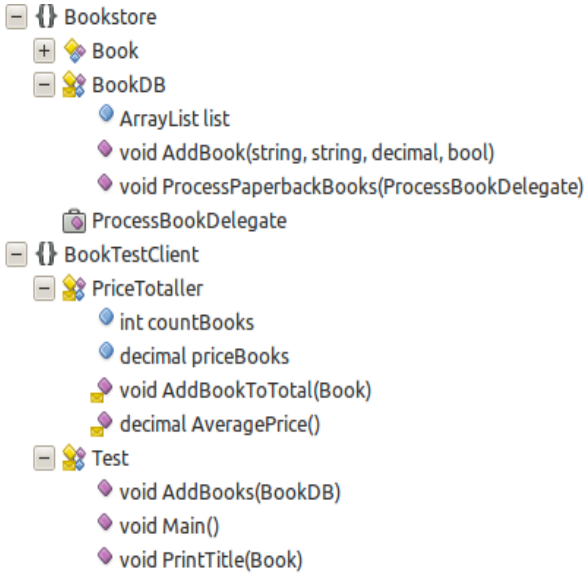
Si se prescinde del método TestDelegado:

```
// declaracion de delegado
public delegate void MiDelegado(string mensaje);

class Program{

	public void MetodoDelegado(string mensaje){
		Console.WriteLine(mensaje);
	}

	static void Main(string[] args){
		Program app = new Program();
		MiDelegado delegado = new MiDelegado(app.MetodoDelegado);
		delegado("Hola delegado");
	}
}
```

Ejemplo Bookstore

El siguiente ejemplo ilustra la declaración, creación de instancias y uso de un delegado.

En el ejemplo hay dos namespace (Bookstore y BookTestClient).



La clase **BookDB** encapsula la base de datos de los libros de una librería en donde se usa un tipo ArrayList para almacenar los libros de tipo Book. Y Expone un método ProcessPaperbackBooks, el cual busca todos los libros físicos(paperback) de la base de datos y llama a un delegado para cada uno.

El tipo delegate que se utiliza se denomina ProcessBookDelegate.

La clase Test utiliza esta clase para imprimir los títulos y el precio medio de los libros en formato físico. Para ello asocia el tipo delegate ProcessBookDelegate con el método estático PrintTitles() de la clase Test y con el método de instancia AddBookToTotal() de la clase PriceTotaller.

El uso de delegados promueve una buena separación de la funcionalidad entre la base de datos de la librería y el código del programa cliente. El código del cliente no tiene conocimiento de cómo están almacenados los libros ni de cómo se buscan los libros. El código de la librería no conoce qué procesamiento se realiza sobre los libros después de encontrarlos (precio medio, recuento de libros, imprimir títulos etc).

```
// ejemplo 02 bookstore
using System;

// A set of classes for handling a bookstore:
namespace Bookstore
{
	using System.Collections;

	// Describes a book in the book list:
	public struct Book{

		public string Title;	// Title of the book.
		public string Author;	// Author of the book.
		public decimal Price;	// Price of the book.
		public bool Paperback;	// Is it paperback? (libro fisico?)

		public Book(string title, string author, decimal price, bool paperBack){
			Title = title;
			Author = author;
			Price = price;
			Paperback = paperBack;
		}
	}

	// Declare a delegate type for processing a book:
	public delegate void ProcessBookDelegate(Book book);

	// Maintains a book database.
	class BookDB{
		// List of all books in the database:
		ArrayList list = new ArrayList();

		// Add a book to the database:
		public void AddBook(string title, string author, decimal price, bool
paperBack){
			list.Add(new Book(title, author, price, paperBack));
		}

		// Call a passed-in delegate on each paperback book to process it:
		public void ProcessPaperbackBooks(ProcessBookDelegate processBook){
			foreach (Book b in list){
				if (b.Paperback)
					// Calling the delegate:
					processBook(b);
			}
		}
	}
}
```

```
// Using the Bookstore classes:
namespace BookTestClient
{
	using Bookstore;

	// Class to total and average prices of books:
	class PriceTotaller{

		int countBooks = 0;
		decimal priceBooks = 0.0M;

		internal void AddBookToTotal(Book book){
			countBooks += 1;
			priceBooks += book.Price;
		}

		internal decimal AveragePrice(){
			return priceBooks / countBooks;
		}
	}

	// Class to test the book database:
	class Test{
		// Print the title of the book.
		static void PrintTitle(Book b){
			Console.WriteLine("   {0}", b.Title);
		}

		// Execution starts here.
		static void Main(){
			BookDB bookDB = new BookDB();

			// Initialize the database with some books:
			AddBooks(bookDB);

			// Print all the titles of paperbacks:
			Console.WriteLine("Paperback Book Titles:");
			// Create a new delegate object associated with the static
			// method Test.PrintTitle:
			// PrintTitle y ProcessBookDelegate
			// tienen el mismo tipo de retorno y parametros
			// el objeto delegado encapsula el metodo PrintTitle
			ProcessBookDelegate delegateObj1 = new
ProcessBookDelegate(PrintTitle);
			// a traves del delegado se ejecuta PrintTitle
			// ProcessPaperbackBooks va pasando como argumentos los libros que
			// son fisicos al delegado y al estar asociado con PrintTitle
			// invoca a este metodo para que se encargue en cada libro de
			// imprimir su titulo
			bookDB.ProcessPaperbackBooks(delegateObj1);

			// Get the average price of a paperback by using
			// a PriceTotaller object:
			PriceTotaller totaller = new PriceTotaller();
			// Create a new delegate object associated with the nonstatic
			// method AddBookToTotal on the object totaller:
			// totaller.AddBookToTotal y ProcessBookDelegate
			// tienen el mismo tipo de retorno y parametros
			// el objeto delegado encapsula el metodo totaller.AddBookToTotal
			ProcessBookDelegate delegateObj2 = new
ProcessBookDelegate(totaller.AddBookToTotal);
```

```
                    // a traves del delegado se ejecuta totaller.AddBookToTotal
                    bookDB.ProcessPaperbackBooks(delegateObj2);
                    Console.WriteLine("(precio medio libros) " +
                          "Average Paperback Book Price: ${0:#.##}",
                                totaller.AveragePrice());
              }

              // Initialize the book database with some test books:
              static void AddBooks(BookDB bookDB){
                    bookDB.AddBook("The C Programming Language",
                                "Dennis M. Ritchie", 19.95M, true);
                    bookDB.AddBook("The Unicode Standard 2.0",
                                "The Unicode Consortium", 39.95M, true);
                    bookDB.AddBook("The MS-DOS Encyclopedia",
                                "Ray Duncan", 129.95M, false);
                    bookDB.AddBook("Dogbert's Clues for the Clueless",
                                "Scott Adams", 12.00M, true);
              }
        }
}
```

- Declarar un delegado

```
public delegate void ProcessBookDelegate(Book book);
```

declara un nuevo tipo delegado. Cada tipo delegado describe el número y tipo de los argumentos, así como el tipo del valor devuelto de los métodos que puede encapsular. **Cuando se necesita un nuevo conjunto de tipos de argumentos o de valor devuelto, se debe declarar un nuevo tipo delegado.**

- Crear una instancia de un delegado

Una vez declarado un tipo delegado, debe crearse un objeto delegado y asociarlo con un determinado método. Al igual que los demás objetos, un objeto delegado se crea mediante una expresión **new**. Sin embargo, cuando se crea un delegado, **el argumento que se pasa a la expresión new es especial: se escribe como una llamada a un método, pero sin los argumentos**.

```
ProcessBookDelegate delegateObj1 = new ProcessBookDelegate(PrintTitle);
ProcessBookDelegate delegateObj2 = new ProcessBookDelegate(totaller.AddBookToTotal);
```

Una vez que se crea el delegado, el método con el que está asociado no cambia nunca (los objetos delegados son inmutables).

- Llamar a un delegado

Una vez creado un objeto delegado, éste se pasa normalmente a otro código que llamará al delegado.

```
bookDB.ProcessPaperbackBooks(delegateObj1);
bookDB.ProcessPaperbackBooks(delegateObj2);
```

La llamada a un objeto delegado se realiza mediante el nombre del objeto delegado, seguido por los argumentos entre paréntesis que se pasarán al delegado.

```
processBook(b);
```

Delegados compuestos

Una propiedad útil de los objetos delegados es que se pueden componer mediante el operador "+". Un delegado compuesto llama a los dos delegados de los que se compone. Solo pueden ser compuestos los delegados del mismo tipo.

El operador "-" se puede utilizar para quitar un delegado componente de un delegado compuesto.

```
// declaracion de delegado
delegate void MyDelegate(string s);

class Program{
	public static void Hello(string s){
		Console.WriteLine("  Hello, {0}!", s);
	}

	public static void Goodbye(string s){
		Console.WriteLine("  Goodbye, {0}!", s);
	}

	public static void Main(){
		MyDelegate a, b, c, d;

		// Create the delegate object a that references
		// the method Hello:
		a = new MyDelegate(Hello);
		// Create the delegate object b that references
		// the method Goodbye:
		b = new MyDelegate(Goodbye);
		// The two delegates, a and b, are composed to form c:
		c = a + b;
		// Remove a from the composed delegate, leaving d,
		// which calls only the method Goodbye:
		d = c - a;

		Console.WriteLine("Invoking delegate a:");
		a("A");
		Console.WriteLine("Invoking delegate b:");
		b("B");
		Console.WriteLine("Invoking delegate c:");
		c("C");
		Console.WriteLine("Invoking delegate d:");
		d("D");
	}
}
```

Métodos anónimos

Un método anónimo es aquel método que no tiene un nombre específico. Y la manera de invocarlos es usando delegados.

Mediante los métodos anónimos, se reduce la sobrecarga de codificación al crear instancias de delegados sin tener que crear un método independiente.

Un método anónimo es declarado con la palabra clave **delegate** seguido de una lista de parámetros. Las líneas de código para el método están dentro del par de llaves {}. Todo esto debe ser referenciado por un tipo delegate que tenga la misma firma que el método anónimo, es así que, la invocación de los métodos anónimos se realiza usando esta referencia.

```
// declaracion de delegado
public delegate void toString(Boolean b);

class Program{
	static void Main(string[] args){
		//referencia a delegado de tipo toString.
		toString ts;

		//declaracion de metodo anonimo usando delegado.
		ts = delegate(Boolean b) {
			Console.WriteLine("\ntoString: {0}", b == true ? "Verdadero" :
"Falso");
		};

		ts(true); //invocacion del metodo anonimo usando delegado.
	}
}
```

Ejemplo donde hay varios métodos anónimos:

```
// declaracion de delegado
public delegate void printIntView(int valor);

class Program{
	static void Main(string[] args){
		// delegados a partir de tipo definido printIntView
		printIntView entero, hexa, EH;

		// metodos anonimos
		entero=delegate(int nro) { Console.Write("{0} ", nro); };
		hexa = delegate(int nro) { Console.Write("0x{0:X} ", nro); };

		Console.Write("\nentero: "); entero(128);
		Console.Write("\nhexa: "); hexa(128);

		// delegado compuesto
		EH = entero + hexa;
		Console.Write("\nEH: ");
		EH(128);//invocacion
	}
}
```

El ejemplo siguiente muestra las dos maneras de crear instancias de un delegado:

- Asociar el delegado a un método anónimo.
- Asociar el delegado a un método con nombre (MetodoDelegado).

En cada uno de los casos, se muestra un mensaje cuando se invoca al delegado.

```
// declaracion de delegado
public delegate void MiDelegado(string mensaje);

class Program{
	// metodo que encapsulara el delegado
	// tiene el mismo tipo de retorno y parametros que
	// el delegado MiDelegado
	public static void MetodoDelegado(string mensaje){
		Console.WriteLine(mensaje);
	}

	static void Main(string[] args){
		// instanciacion del delegado usando
		// un metodo anonimo
		MiDelegado delegado = delegate(string mensaje){
			Console.WriteLine(mensaje);
		} ;

		delegado("el delegado usa un metodo anonimo");

		// objeto delegado que encapsula el metodo MetodoDelegado
		// delegado = new MiDelegado(Program.MetodoDelegado);
		delegado = new MiDelegado(Program.MetodoDelegado);

		delegado("el delegado usa el metodo asociado");
	}
}
```

Encapsular varios métodos en un objeto delegado

El objeto delegado puede recibir n métodos, para ello se usa el operador += para indicarle al objeto delegado que va a contener mas métodos y cuando se llame al objeto delegado llamará a todos los métodos contenidos

```
// definicion de delegado
public delegate void OperacionesBasicas(int a, int b);

class Program{

	static OperacionesBasicas operacion;

	static void Suma(int a, int b){
		Console.WriteLine("sumando {0} y {1} resultado es {2}", a, b, a + b);
	}

	static void Resta(int a, int b){
		Console.WriteLine("restando {0} y {1} resultado es {2}", a, b, a - b);
	}

	static void Multiplicacion(int a, int b){
		Console.WriteLine("multiplicar {0} y {1} resultado es {2}", a, b, a * b);
	}
```

```
        static void Division(int a, int b){
                Console.WriteLine("division {0} y {1} resultado es {2}", a, b, a / b);
        }

        static void Main(string[] args){
                operacion += new OperacionesBasicas(Suma);
                operacion += new OperacionesBasicas(Resta);
                operacion += new OperacionesBasicas(Multiplicacion);
                operacion += new OperacionesBasicas(Division);

                operacion(20,5);
         }
}
```

Otra forma de hacer lo mismo en donde se usa una propiedad

```
class Program{
        // definicion de delegado
        public delegate void OperacionesBasicas(int a, int b);

        // atributo privado que guarda objetos del tipo delegado
        private static OperacionesBasicas operacion;

        // propiedad que asigna objetos y devuelve objetos delegados
        public static OperacionesBasicas Operacion{
                get{return operacion;}
                set{  operacion = value;}
        }

        static void Main(string[] args){
                // instancia de Implementacion
                Implementacion a = new Implementacion();
                // valores que reciben los metodos de los objetos
                // delegados contenidos en operacion
                Program.Operacion(20,5);
         }
}

class Implementacion{
        public Implementacion(){
                Program.Operacion += new Program.OperacionesBasicas(Suma);
                Program.Operacion += new Program.OperacionesBasicas(Resta);
                Program.Operacion += new Program.OperacionesBasicas(Multiplicacion);
                Program.Operacion += new Program.OperacionesBasicas(Division);
        }

        private void Suma(int a, int b){
                Console.WriteLine("sumando {0} y {1} resultado es {2}", a, b, a + b);
        }
        private void Resta(int a, int b){
                Console.WriteLine("restando {0} y {1} resultado es {2}", a, b, a - b);
        }
        private void Multiplicacion(int a, int b){
                Console.WriteLine("multiplicar {0} y {1} resultado es {2}", a, b, a * b);
        }
        private void Division(int a, int b){
                Console.WriteLine("division {0} y {1} resultado es {2}", a, b, a / b);
        }
}
```

Covarianza y contravarianza en los delegados

La covarianza permite que un método tenga un tipo de valor devuelto más derivado que lo que se define en el delegado.

```
// declaracion o definicion de delegado
delegate Persona PersonaCallback();
class Persona{
	// Omitidas las definiciones
	// de los atributos Nombre y Apellidos
}
class Colega : Persona{
	public static Colega NuevoColega(){
		return new Colega();
	}
}
class Program{
	static void Main(string[] args){
		PersonaCallback nColega;
		nColega = Colega.NuevoColega;
		Colega unColega = (Colega)nColega();
	}
}
```

En la asignación a la variable unColega hay que hacer una conversión (cast) ya que el valor devuelto por el delegado es del tipo Persona, independientemente del tipo que devuelva en realidad el método a que hace referencia ese delegado.

La contravarianza define que los tipos de datos que se pueden usar como parámetros al llamar a un delegado pueden ser del mismo tipo que está definido en el delegado o de cualquier tipo derivado. Si el compilador ve una relación de herencia entre el tipo usado y el definido en el delegado, lo permitirá.

```
// declaracion o definicion de delegado
delegate string MiFuncionDelegado(Cliente c);

class Cliente{
	public string Nombre;
	public Cliente(){}
	public Cliente(string nombre){
		this.Nombre = nombre;
	}
	public string MiFuncion(Cliente c) {
		return "Que tal " + c.Nombre;
	}
}

class ClienteOro : Cliente{
	public ClienteOro(string nombre){
		this.Nombre = nombre;
	}
}
```

```
class Program{
	static void Main(string[] args){
		ClienteOro co = new ClienteOro("Paco");
		// se asocia delegado con metodo MiFuncion
		// usando el tipo derivado ClienteOro
		MiFuncionDelegado mfd2 = co.MiFuncion;

		// el tipo derivado se pasa como argumento
		// al delegado en vez de un tipo base
		string sco = mfd2(co);
		Console.WriteLine(sco);
	}
}
```

EVENTOS

Un evento es un mensaje (o notificación) que lanza un objeto de una clase determinada cuando algo ha ocurrido. El ejemplo más clásico y fácil de entender es cuando el usuario pulsa con el ratón en un botón de un formulario.

Si interesa interceptar el evento hay que comunicárselo a la clase que define el botón. Como es de suponer, los eventos se pueden definir en cualquier clase, y no solo en clases que tengan algún tipo de interacción con el usuario, aunque lo más habitual es que precisamente se usen con clases que forman parte de la interfaz gráfica que se le presenta al usuario de una aplicación.

Definir eventos

La definición de un evento está estrechamente ligada con los delegados, **cada evento está relacionado con un delegado** ya que la forma de lanzar ese evento (o lo que es lo mismo, la forma de notificar que ese evento ha ocurrido) es llamando al delegado con los parámetros que se hayan definido.

Por ejemplo, si se quiere tener un evento en una clase para que notifique cada vez que se modifica el contenido de una propiedad, habrá que **definir un delegado que indique la "firma" que debe tener el método que vaya a recibir dicha notificación**.

Además, hay que **definir el evento dentro de una clase, el cual será una variable especial del tipo del delegado**. Lo de "variable especial" es porque en realidad se define como cualquier otra variable (solo que con un tipo delegado como tipo de datos), **pero añadiéndole la instrucción event**.

<private/public/protected> delegate <tipoRetorno> <nombreDelegado> (<parámetros>);

<private/public/protected> ***event*** *<nombreDelegado> <nombreEvento>;*

Según las recomendaciones de nomenclatura de .NET Framework, los nombres de los delegados deben terminar con EventHandler.

Ejemplo básico de uso de eventos:

Clase llamada Empleado, con un evento llamado datosModificados, el cual se producirá cada vez que se modifiquen las dos propiedades que tienen acceso de escritura. Este evento no tendrá ningún parámetro.

```
// delegado sin parametros,
// y al utilizarse para recibir eventos es de tipo void
public delegate void DatosCambiadosEventHandler();

class Empleado {

	// evento
	public event DatosCambiadosEventHandler datosCambiados;

	private string nombre;
	private string apellidos;
	private decimal salario;

	public Empleado(string nombre, string apellidos, decimal salario){
		this.nombre = nombre;
		this.apellidos = apellidos;
		this.salario = salario;
	}
	public string Nombre{
		get{  return nombre; }
		set{
			nombre = value;
			// comprobar si algun metodo sera notificado
			if (datosCambiados != null)
				datosCambiados(); // lanzar o producir evento
		}
	}
	public string Apellidos{
		get{  return apellidos; }
		set{
			apellidos = value;
			// comprobar si algun metodo sera notificado
			if (datosCambiados != null)
				datosCambiados(); // lanzar o producir evento
		}
	}
	public decimal Salario{
		get{  return salario; }
	}
	// reemplaza al metodo ToString de la clase object,
	// el cual se utilizara para mostrar los datos de la clase
	public override string ToString(){
		return Apellidos + ", " + Nombre;
	}
}

class Program{

	// metodo que servira para asociarlo con el evento
	static void emp_DatosCambiados(){
		Console.WriteLine("Datos cambiados en el Empleado");
	}
```

```
        static void Main(string[] args){
                Empleado emp = new Empleado("Paco", "Aragon", 950.50M);

                // Asociar el evento de la clase Empleado
                // con el metodo emp_DatosCambiados
                // evento += new delegado(metodo)
                emp.datosCambiados += new DatosCambiadosEventHandler(emp_DatosCambiados);

                Console.WriteLine(emp.ToString());
                emp.Nombre = "Jose";
                Console.WriteLine(emp.ToString());
        }
}
```

Explicación del ejemplo.

Definición del delegado:

```
public delegate void DatosCambiadosEventHandler();
```

Definición del evento dentro de la clase Empleado usando el tipo del delegado:

```
public event DatosCambiadosEventHandler datosCambiados;
```

Lanzar o producir el evento en donde se usa el nombre del evento, pero hay que comprobar si algún método será notificado, es decir, si se ha añadido a ese evento una dirección de memoria gestionada por el delegado. En caso de que no haya ningún método esperando ser notificado, se lanzará una excepción del tipo NullReferenceException. Para que no se produzca ese error hay que comprobar si el "evento" contiene un valor nulo, y en caso de que no lo tenga se lanza el evento:

```
// comprobar si algun metodo sera notificado
if (datosCambiados != null)
        datosCambiados(); // lanzar o producir evento
```

Antes de asociar el evento con un método, hay que declarar un método con la misma firma que ha definido el delegado:

```
static void emp_DatosCambiados(){
        Console.WriteLine("Datos cambiados en el Empleado");
}
```

Asociar evento de la clase Empleado con el método emp_DatosCambiados;

nombreEvento += new nombreDelegado(nombreMetodo);

```
emp.datosCambiados += new DatosCambiadosEventHandler(emp_DatosCambiados);
```

Asociar eventos con un método

La forma que tiene C# de asociar eventos con los métodos que lo interceptarán es usando el operador += para asignar al evento el delegado, el cual recibe un parámetro que es el método que recibirá la notificación de que dicho evento se ha producido.

nombreEvento += new nombreDelegado(nombreMetodo);

Desasociar eventos

Cuando no se quiera que un evento sea interceptado, se puede eliminar el delegado que previamente se ha asociado. Esto se hace de la misma forma que cuando se asocia un método a un evento, pero en lugar de utilizar el operador += se usa el operador -=.

nombreEvento -= new nombreDelegado(nombreMetodo);

```
emp.datosCambiados -= new DatosCambiadosEventHandler(emp_DatosCambiados);
```

Detectar cuándo se añade o se elimina un delegado de un evento

Para detectar cuándo se añade un delegado a un evento o cuándo se elimina, se puede utilizar las propiedades de eventos o eventos de propiedades.

Esta característica se puede usar cuando interese tener una relación de los métodos que interceptan los eventos o para solventar ciertos conflictos que se pueden presentar cuando una clase implementa más de una interfaz en las que se ha definido un evento que tiene el mismo nombre en las interfaces, y el delegado usado en cada uno de ellos sea diferente.

Para aplicar esta nueva forma de definir un evento en el ejemplo de la clase Empleado, hay que eliminar la declaración del evento y utilizar otra:

```
// La variable privada se usara
// para almacenar el valor interno del evento-propiedad
private DatosCambiadosEventHandler datosCambiados;

// El evento declarado al estilo de una propiedad.
public event DatosCambiadosEventHandler DatosCambiados{
	add{
		datosCambiados += value;
		Console.WriteLine("El metodo a notificar es: {0}", value.Method.Name);
		Console.WriteLine("Se añade un delegado al evento DatosCambiados");
	}
	remove{
		datosCambiados -= value;
		Console.WriteLine("Se elimina un delegado al evento DatosCambiados");
	}
}
```

Lo primero es declarar una variable privada del mismo tipo que el delegado, esa variable será la que se use para almacenar el valor interno del evento-propiedad.

Después se declara el evento-propiedad, que se hace de la misma forma que un evento, con la diferencia de que en lugar de terminarlo con un punto y coma, se utiliza un bloque de código. Dentro se definen otros dos bloques, pero en lugar de utilizar los típicos get y set de las propiedades normales, se usa un bloque para cuando se añada el delegado (add) y otro para cuando se elimine (remove).

Esos dos bloque reciben un parámetro implícito (value), el cual representa al delegado que se quiere añadir o quitar, por tanto se deben realizar esas mismas operaciones dentro de cada bloque, utilizando los operadores += y -=. También se puede añadir más código a cada bloque, incluso se puede obtener información del delegado usado para asociar el evento con un método.

Para lanzar este evento, en lugar de usar el declarado como propiedad, se usa la variable privada:

```
// comprobar si algun metodo sera notificado
if (datosCambiados != null)
	datosCambiados(); // lanzar o producir evento
```

Otro ejemplo de eventos:

```
public delegate void NombreCambiadoEventHandler(string nuevo, string anterior);

class Empleado {

	public event NombreCambiadoEventHandler nombreCambiado;
	private string nombre;

	public Empleado(string nombre){
		this.nombre = nombre;
	}

	public string Nombre{
		get { return nombre; }
		set{
			if(nombreCambiado != null ){
				nombreCambiado(value, nombre);
			}
			nombre = value;
		}
	}

	public override string ToString(){
		return nombre;
	}
}
class Program{

	static void m_NombreCambiado(string nuevo, string anterior){
		Console.WriteLine("Se ha cambiado el nombre:\n" +
			"Nuevo valor:" + nuevo + "\n" +
			"Valor anterior: " + anterior);
	}

	static void Main(string[] args){
		Empleado emp = new Empleado("Paco");
		emp.nombreCambiado += new NombreCambiadoEventHandler(m_NombreCambiado);

		Console.WriteLine(emp.ToString());
		emp.Nombre = "Jose";
	}
}
```

El delegado EventHandler

Generalmente, los delegados asociados a un evento son declarados con la siguiente firma:

```
public delegate void EventHandler(Object sender, EventArgs e);
public event EventHandler Changed;
```

No retornan nada (void), el primer parámetro es la fuente del evento y el segundo una instancia de EventArgs o de alguna subclase. EventArgs contiene los datos de evento.

Dado que los delegados con esta firma son muy frecuentes, se puede utilizar EventHandler, que lo proporciona el framework y no necesita ser declarado explícitamente.

El siguiente ejemplo muestra una clase, ListWithChangedEvent, similar a la clase estándar ArrayList, pero también llama a un evento Changed cuando el contenido de la lista cambia.

Por ejemplo, un procesador de textos podría mantener una lista de los documentos abiertos. Cuando la lista cambia, puede que sea necesario notificarlo a diferentes objetos del procesador de textos para poder actualizar la interfaz de usuario.

Mediante el uso de eventos, el código que mantiene la lista de documentos no necesita saber quién necesita notificación (una vez que se cambia la lista de documentos, se llama automáticamente al evento y se notifica correctamente a todos los objetos que lo necesitan). Mediante el uso de eventos, la **modularidad** del programa se incrementa.

```
namespace MyCollections
{
	using System.Collections;

	//public delegate void EventHandler(Object sender, EventArgs e);

	// similar a la clase estandar ArrayList,
	// llama al evento Changed cuando el contenido de
	// la lista cambia y lo hace a traves del metodo OnChanged
	class ListWithChangedEvent: ArrayList {
		// Evento que se usa para notificar que
		// un elemento de la lista ha cambiado
		public event EventHandler Changed;

		// Invoca el evento Changed; lo llama cuando cambia la lista
		protected virtual void OnChanged(EventArgs e) {
			// se comprueba si algun metodo sera notificado
			if (Changed != null)
				Changed(this, e);
		}

		// Reemplazo de algunos metodos que pueden cambiar
		// la lista invoca evento en cada metodo

		// Añade elemento al final de la lista
		// devuelve valor del indice
		public override int Add(object value) {
			// base se utiliza para acceder a los
			// metodos de la clase base desde una derivada
			int i = base.Add(value);
			// Empty representa un evento sin datos de eventos
			OnChanged(EventArgs.Empty);
			return i;
		}

		// Quita todos los elementos de la lista
		public override void Clear() {
			base.Clear();
			OnChanged(EventArgs.Empty);
		}
```

```
		// Obtiene o establece el elemento que se
		// encuentra en el indice especificado.
		// en este caso solo se establece con set
		public override object this[int index] {
			set {
				base[index] = value;
				OnChanged(EventArgs.Empty);
			}
		}
	}
}

namespace TestEvents
{
	using MyCollections;

	// Clase para controlar el evento
	class EventListener {
		// campo privado que guarda instancia
		// de la clase ListWithChangedEvent
		private ListWithChangedEvent _list;

		// constructor
		public EventListener(ListWithChangedEvent list) {
			_list = list;
			// Asocia evento con ListChanged
			_list.Changed += new EventHandler(ListChanged);
		}

		// Metodo que es invocado por el evento
		// cuando hay un cambio en la lista
		private void ListChanged(Object sender, EventArgs e) {
			Console.WriteLine("Esto se llama cuando el evento se activa.");
		}

		public void Detach() {
			// Quita la asociacion del evento
			// con ListChanged y elimina la lista
			_list.Changed -= new EventHandler(ListChanged);
			_list = null;
		}
	}

	class Test {
		public static void Main() {
			// Create a new list.
			ListWithChangedEvent list = new ListWithChangedEvent();

			// Instancia de clase que controla los eventos de cambios en la lista
			EventListener listener = new EventListener(list);

			// Add and remove items from the list.
			list.Add("item 1"); // lanza evento
			list.Clear(); // lanza evento
			listener.Detach(); // Quita la asociacion del evento
			list.Add("item 1"); // no lanza evento
			list.Clear(); // no lanza evento
		}
	}
}
```

Eventos y herencia

Según las recomendaciones de .NET Framework sobre los eventos es que estos sólo se pueden invocar desde dentro de la clase que los declaró, las clases derivadas no pueden invocar directamente eventos declarados dentro de la clase base.

Para lanzar un evento en una clase se debería de declarar un método protegido llamado OnNombreDelEvento, al que se llamará cuando se quiera lanzar el evento.

```
// Invoca el evento Changed; lo llama cuando cambia la lista
protected virtual void OnChanged(EventArgs e) {
	// se comprueba si algun metodo sera notificado
	if (Changed != null)
		Changed(this, e);
}
```

Para obtener más flexibilidad, el método que invoca el evento se suele declarar como virtual, lo cual permite a la clase derivada reemplazarlo(override). Esto permite a la clase derivada interceptar los eventos que la clase base está invocando, y, posiblemente, realizar su propio procesamiento sobre ellos. Si se sobrescribe dicho método para adaptarlo a las necesidades, ese método debería llamar al de la clase base para que los delegados que existan sigan notificando dicho evento.

```
class L2 : ListWithChangedEvent{
	protected override void OnChanged(EventArgs e) {
		Console.WriteLine("OnChanged Reemplazada");
		base.OnChanged(e);
	}
}
```

Eventos en interfaces

Otra diferencia entre eventos y campos(atributos) es que un evento se puede colocar en una interfaz mientras que un campo no. Cuando se implementa la interfaz, la clase que la implementa debe suministrar un evento correspondiente.

```
interface ITest{
	event EventHandler ChangeReg;
}

class Registro: ITest{
	// Evento implementado
	public event EventHandler ChangeReg;

	private string _email;
	private string _pass;

	public Registro(string email, string pass){
		_email = email;
		_pass = pass;
	}

	public string Email{
		set{
			_email = value;
			OnChangeReg(EventArgs.Empty);
		}
	}
```

```
        public string Pass{
                set{
                        _pass = value;
                        OnChangeReg(EventArgs.Empty);
                }
        }

        // Invoca el evento ChangeReg
        protected virtual void OnChangeReg(EventArgs e) {
                // se comprueba si algun metodo sera notificado
                if (ChangeReg != null)
                        ChangeReg(this, e);
        }

        public override string ToString(){
                return _email + " " + _pass;
        }

}

// Clase para controlar el evento
class EventListener {
        // campo privado que guarda instancia
        private Registro _reg;

        // constructor
        public EventListener(Registro reg) {
                _reg = reg;
                // Asocia evento
                _reg.ChangeReg += new EventHandler(ChangedReg);
        }

        // Metodo que es invocado por el evento
        private void ChangedReg(Object sender, EventArgs e) {
                Console.WriteLine("Datos cambiados de registro");
                Console.WriteLine("{0}", _reg.ToString());

        }
}

class Test {
        public static void Main() {
                // Create a new register
                Registro r1 = new Registro("paco@paco.net", "paco234");
                // Instancia de clase que controla los eventos
                EventListener listener = new EventListener(r1);

                r1.Email = "paco@paco.com";
                r1.Pass = "23487";
        }
}
```

El delegado genérico EventHandler

La firma estándar del delegado de un controlador de eventos no devuelve ningún valor, cuyo primer parámetro es de tipo Object y hace referencia a la instancia que provoca el evento, y cuyo segundo parámetro se deriva del tipo EventArgs y contiene los datos de eventos. Si el evento no genera datos de eventos, el segundo parámetro es tan sólo una instancia de EventArgs. De lo contrario, el segundo parámetro es un tipo personalizado derivado de EventArgs y que proporciona los campos o propiedades necesarios para contener los datos de eventos.

```
public delegate void EventHandler<TEventArgs> (
       Object sender,
       TEventArgs e
) where TEventArgs : EventArgs
```

EventHandler es un delegado predefinido que representa un método controlador de un evento, independientemente de si el evento genera datos de eventos o no. Si el evento no genera datos de eventos, se sustituye el parámetro de tipo genérico por EventArgs; en caso contrario, se utiliza un tipo de datos de eventos personalizado para sustituir el parámetro de tipo genérico.

Ejemplo:

```
// public delegate void EventHandler<TEventArgs> (Object sender, TEventArgs e);

// clase personalizada derivada de EventArgs
// ya que se desea que el evento genere datos
public class MyEventArgs : EventArgs {
       private string msg;
       // constructor
       public MyEventArgs( string messageData ) {
              msg = messageData;
       }
       public string Message {
              get { return msg; }
              set { msg = value; }
       }
}

public class HasEvent {
       // Evento con el tipo personalizado de delegado
       public event EventHandler<MyEventArgs> SampleEvent;

       // metodo que lanza el evento
       // al evento se le pasa un objeto del tipo MyEventArgs
       // junto con el string como argumento
       public void DemoEvent(string val) {
              if (SampleEvent != null)
                     SampleEvent(this, new MyEventArgs(val));
       }
}

public class Sample {
       public static void Main() {
              HasEvent he = new HasEvent();
              // evento asociado
              he.SampleEvent += new EventHandler<MyEventArgs>(SampleEventHandler);

              he.DemoEvent("Saludo 1");
              he.DemoEvent("Saludo 2");
       }
       // metodo que usa el evento
       private static void SampleEventHandler(object src, MyEventArgs mes) {
              Console.WriteLine(mes.Message);
       }
}
```

Parámetros por referencia en eventos

Para poder hacer que la clase principal (Program) pueda comunicarse con la clase que produce el evento, por ejemplo, para cancelar una acción o para pasar cierta información , se declara por referencia uno (o varios) de los parámetros. De esta forma se puede asignar un valor a ese parámetro y usarlo en la clase que produce el evento.

```
// delegado
public delegate void NombreCambiadoEventHandler(string nuevo, string anterior, ref bool
cancelar);

class Empleado {
	// evento
	public event NombreCambiadoEventHandler NombreCambiadoRef;

	private string nombre;

	public Empleado(string nombre){
		this.nombre = nombre;
	}

	public string Nombre{
		get{  return nombre; }
		set{
			// Lanzar el evento
			// indicando el nuevo valor y el anterior,
			// ademas de la variable para saber si cancela
			if( NombreCambiadoRef != null ){
				bool cancelado = false;
				NombreCambiadoRef(value, nombre, ref cancelado);
				// si el metodo asociado al evento cambia el valor de
				// cancelado a true al ser una referencia
				// se sale de la propiedad sin asignar el nuevo nombre
				if( cancelado ){
					return;
				}
			}
				nombre = value;
		}
	}

	// reemplaza al metodo ToString de la clase object,
	// el cual se utilizara para mostrar los datos de la clase
	public override string ToString(){
		return nombre;
	}
}

class Program{

	// metodo que servira para asociarlo con el evento
	static void cli_NombreCambiadoRef(string nuevo, string anterior, ref bool
cancelar) {
		// Cancelamos la accion si el nuevo valor no
		// cumple las condiciones que estimemos oportunas
		if( string.IsNullOrEmpty(nuevo) || nuevo.Length < 3 ) {
			Console.WriteLine( "cancelando.. debe tener mas de 2 caracteres...");
			cancelar = true; // cambia valor de variable cancelado
		}
	}
```

```
	static void Main(string[] args){
		Empleado emp = new Empleado("Paco");
		// Asociar el evento
		// evento += new delegado(metodo)
		emp.NombreCambiadoRef += new
NombreCambiadoEventHandler(cli_NombreCambiadoRef);

		Console.WriteLine(emp.ToString());
		emp.Nombre = "Jo";// se lanza evento
		Console.WriteLine(emp.ToString());
	}
}
```

La comprobación del número de caracteres se podría haber hecho en la propiedad, pero se trata de ver un ejemplo de cómo cancelar (o informar a la clase). Además, al hacerlo en la clase principal(Program), esto da la posibilidad de que cada aplicación que use esa clase decida la comprobación que se debe hacer.

Usando clases como parámetro de eventos

.NET permite pasar más datos usando un tipo personalizado como parámetro, es decir, una clase. Para ello hay la clase EventArgs. De hecho, todos los eventos de los controles del espacio de nombres Windows.Forms se basan (o se derivan) de esa clase; por tanto, se puede crear una clase basada en EventArgs para pasar (y recuperar) datos en ambos sentidos, y evitar la definición de parámetros por referencia.

```
public class DatosCambiadosEventArgs : EventArgs {
	//constructor
	public DatosCambiadosEventArgs(string nuevo, string anterior) {
		m_Nuevo = nuevo;
		m_Anterior = anterior;
	}

	// Propiedad de solo lectura cuando se accede
	// desde fuera del propio ensamblado
	private string m_Nuevo;
	public string Nuevo {
		get { return m_Nuevo; }
		internal set { m_Nuevo = value; }
	}

	// Propiedad de solo lectura cuando se accede
	// desde fuera del propio ensamblado
	private string m_Anterior;
	public string Anterior {
		get { return m_Anterior; }
		internal set { m_Anterior = value; }
	}

	private bool m_Cancelar;
	public bool Cancelar {
		get { return m_Cancelar; }
		set { m_Cancelar = value; }

	}
}
// delegado
public delegate void DatosCambiadosEventHandler(DatosCambiadosEventArgs e);
```

```
class Empleado {
	// evento
	public event DatosCambiadosEventHandler NombreCambiado;

	private string nombre;

	// constructor
	public Empleado(string nombre){
		this.nombre = nombre;
	}

	public string Nombre{
		get{  return nombre; }
		set{
		if( NombreCambiado != null ){
			// Usamos el metodo de apoyo para lanzar el evento
			if( OnNombreCambiado(value, nombre) ) {
				// Lo que haya que hacer para avisar
				// que se ha cancelado la asignacion
				Console.WriteLine( "Cancelado el nuevo nombre: {0}.", value);
					return;
				}
			}
			nombre = value;
		}
		}

	// Si vamos a usar un metodo de apoyo para lanzar
	// el evento y vamos a tener en cuenta la propiedad
	// Cancelar el metodo deberia devolver el valor de cancelacion
	protected bool OnNombreCambiado(string nuevo, string anterior) {
		DatosCambiadosEventArgs e = new DatosCambiadosEventArgs(nuevo, anterior);
		e.Cancelar = false;
		NombreCambiado(e);
		return e.Cancelar;
	}

	// reemplaza al metodo ToString de la clase object,
	// el cual se utilizara para mostrar los datos de la clase
	public override string ToString(){
		return nombre;
	}
}
class Program{
	// metodo que servira para asociarlo con el evento
	static void cli_NombreCambiado(DatosCambiadosEventArgs e) {
		if( string.IsNullOrEmpty(e.Nuevo) || e.Nuevo.Length < 3 ) {
			Console.WriteLine("cancelando..debe tener mas de 2 caracteres.");
			e.Cancelar = true;
		}
	}
	static void Main(string[] args){
		Empleado emp = new Empleado("Paco");
		// Asociar el evento
		// evento += new delegado(metodo)
		emp.NombreCambiado += new DatosCambiadosEventHandler(cli_NombreCambiado);

		Console.WriteLine(emp.ToString());
		emp.Nombre = "Jo";// se lanza evento
		Console.WriteLine(emp.ToString());
	}
}
```

Como en otro ejemplo ya visto, otra de las recomendaciones que atañen a los eventos definidos en una clase (o tipo), es la de crear un método (habitualmente definido como protegido) que sea el que en realidad se encargue de producir el evento. De esa forma, para lanzar un evento, en lugar de hacerlo directamente se usará ese método, al que se le pasarán los parámetros correspondientes y será el encargado de producir el evento.

La nomenclatura recomendada para este tipo de métodos es que tenga el mismo nombre que el evento, pero empezando con **On**. Esos métodos de apoyo también suelen ser de tipo void (no devuelven un valor), pero en el ejemplo devuelve un valor de tipo bool, de forma que indica si se cancela o no la asignación de la propiedad.

ACTIVIDADES

1.- ¿Cómo se declararía un delegado que no retorne nada y con un parámetro de tipo entero?

2.- Asocia un método que imprima un entero por consola al delegado de la anterior pregunta e invoca con el delegado ese método: (recuerda que hay dos formas de asociar un método a un delegado)

3.- Si se usa un método para invocar un delegado que encapsule a otro método:

```
class Program{
	public static void MiFuncion(int num){
		Console.WriteLine(num);
	}
	public static void usarDelegado(MiDelegado md, int num){
		md(num);
	}
	public static void Main(){
		// implementar
	}
}
```

¿Qué sentencia dentro de Main sería necesaria para que se imprimiese por consola el numero 10?

a) `usarDelegado(MiFuncion, 10);`
b) `usarDelegado(new MiDelegado(MiFuncion), 10);`
c) Las dos son correctas

4.- Convierte el siguiente método en un método anónimo usando el delegado de la actividad 1, invoca ese método anónimo para obtener por consola el numero 10:

```
public static void MiFuncion(int num){
	Console.WriteLine(num);
}
```

5.- Implementa el método Main para llamar a los métodos Pregunta1, Pregunta2 y Pregunta3 usando solo una llamada al objeto delegado md. Usa operador += para indicar al objeto delegado que va a contener mas métodos.

```
public delegate void MiDelegado();
	class Program{
		static MiDelegado md;
		static void Pregunta1(){
			Console.WriteLine("Pregunta 1");
		}
		static void Pregunta2(){
			Console.WriteLine("Pregunta 2");
		}
		static void Pregunta3(){
			Console.WriteLine("Pregunta 3");
		}
		public static void Main(){
		}
}
```

6.- Declara delegado llamado “CuentaEventHandler” que no devuelva nada y con tres parámetros; uno de tipo decimal llamado saldo, uno de tipo decimal llamado cantidad y otro de tipo string llamado operacion.

7.- Define evento llamado infoOperacion del tipo delegado de la actividad anterior.

8.- Crea programa que conste de dos clases (Cuenta y Program).

La clase Cuenta guarda en campos privados el numero de cuenta y el saldo de cuenta. A estos campos se accede para su consulta dos propiedades publicas. Aparte hay dos métodos:

- Ingresar de una cantidad que además tiene un parámetro de tipo string para poder saber el concepto (descripción)
- Reintegro de una cantidad que también tiene un parámetro de tipo string para poder saber el concepto (descripción)

Estos dos métodos aparte de modificar el saldo de la cuenta lanzan un evento. Este evento imprimirá por consola la siguiente información cada vez que que se haga una operación:

- Concepto
- Cantidad ingresada o restada
- Saldo Actual de la cuenta

Usa el evento de la actividad 7.

En la clase Program hay dos métodos; uno para asociarlo con el evento que tendrá la misma firma que el delegado de la actividad 6 y que mostrará la información antes mencionada cada vez que se lance el evento. Y el método Main en donde se hará uso de la clase Cuenta y se asociará el evento al método. En Main realiza un ingreso y un reintegro para que se lance el evento.