Introducción a C#

1. Una visión general de C#.
2. Variables, constantes y operadores.
3. Sentencias de selección e iteración.
4. Programación modular.
5. Clases y objetos.
6. Herencia.
7. Delegados y eventos.

Una Visión General de C#

- Creado por *Microsoft* en 2000 como parte integrante de su plataforma .NET.
- Simple, distribuido, moderno y de propósito general.
- Proporciona soporte para comprobación estricta de tipos, compobación de límites en arrays, detección de intentos de utilizar variables no inicializadas y recolector de basura automático.
- Lenguaje *orientado a objetos*. Cualquier aplicación se apoya en un número de *clases preexistentes*, diseñadas por el usuario o del propio lenguaje (C# API).
- Adecuado para escribir aplicaciones convencionales sobre S.O. sofisticados y pequeños programas con funciones dedicadas para sistemas empotrados.
- El *lenguaje intermedio común* (CIL) procedente de la compilación de un fichero fuente (.cs) es traducido por un motor de ejecución (*Common Language Runtime*) que posee un compilador *Just In Time* (JIT).

Un programa sencillo

```
// Un programa sencillo.

using System;

class Primero {
  public static void Main() {

    int num;

    num = 1;

    Console.WriteLine("\nMaravilla, Colega!");
    Console.WriteLine("Mi numero es el " + num +
                      " por ser el primero.\n\n");
  }
}
```

Variables, Constantes y Operadores

- Un programa necesita trabajar con *datos*.

- Algunos están *preseleccionados* antes de la ejecución del programa y mantienen sus valores inalterados durante la misma. Se denominan **constantes**. Otros pueden recibir nuevas *asignaciones* de valor durante la ejecución del programa. Se trata de **variables**.

- Declaración de variables: Se emplea la palabra clave del tipo seguida de los identificadores de las variables separados por comas. La sentencia acaba en punto y coma. Las variables se pueden *inicializar* (asignarles un valor inicial) en la propia sentencia de declaración.

- El valor que contiene una constante no puede modificarse tras su asignación (si se intenta se produce un error). Para declarar una constante hay que añadir al principio la palabra reservada `const`.

```
using System; // Pasa docenas a huevos

class Huevos {

  public static void Main() {

    const int doce = 12;
    int huevos;
    int docenas = 4;

    huevos = doce * docenas;
    Console.WriteLine("Hay " + huevos + " huevos en " + docenas +
                      " docenas.");
  }
}
```

Tipos básicos.

Tipos enteros.

- Tipos de 8 bits: `sbyte` [-128, 127] y `byte` [0, 255].

- Tipos de 16 bits: `short` [-32768, 32767] y `ushort` [0, 65535].
- Tipos de 32 bits: `int` [-2147483648, 2147483647] y `uint` [0, 4294967295].
- Tipo de 64 bits: [-2^{63}, $2^{63}-1$] y `ulong` [0, $2^{64}-1$].

Tipos reales.

- Tipo `float` (32 bits): [$-3.4\ 10^{38}$, $3.4\ 10^{38}$].
- Tipo `double` (64 bits): [$-1.7\ 10^{308}$, $1.7\ 10^{308}$].

- Para representar una constante en punto flotante se escribe una serie de cifras con signo incluyendo un punto decimal, a continuación la letra `e` o `E` seguida de un exponente con signo, que indica la potencia de diez a usar.

```
double num1 = -1.56E+8, // -156000000
       num2 = 2.87e-3,  // .00287
       num3 = 35e2;     // 3500
```

Tipo carácter (`char`).

- Se utiliza para almacenar caracteres del conjunto de Unicode (16 bits), tal como letras, dígitos, caracteres especiales y signos de puntuación.

- Para manejar caracteres muchos ordenadores emplean el codigo ASCII, mediante el cual asocian números enteros a caracteres. UNICODE contiene el conjunto de caracteres ASCII.

- Cuando se encierra un único carácter entre comillas simples, C# lo identifica como una *constante de tipo carácter* y almacena su representación interna en el ordenador (en formato binario) según el código de entrada/salida UNICODE.

- Cuando se usa un código UNICODE, se debe tener en cuenta la diferencia entre un número y un carácter numérico representativo de una cifra.

```
using System;

class cifra {
```

```
  public static void Main() {
    char cifra = (char) 52; // char cifra = '4';
    Console.WriteLine("Cifra: " + cifra);
  }
}
```

- Existen ciertos *caracteres no imprimibles*, pues representan acciones a ejecutar por la máquina, tales como el salto de línea, la tabulación, el retroceso, etc. Para representarlos podemos usar las *secuencias de escape*.

Carácter	**Acción**
`\n`	nueva línea
`\t`	tabulador
`\b`	retroceso
`\r`	retorno de carro
`\f`	salto de página
`\\`	barra atrás
`\'`	comilla simple
`\"`	comillas

Tipo lógico (`bool`).

- Representa dos valores lógicos posibles: verdadero (`true`) y falso (`false`).

Operadores y precedencias.

- **Operador de asignación:** =. El operando de la izquierda debe ser una variable y el de la derecha puede ser cualquier expresión.

- **Operadores aritméticos:** `+`, `-`, `*`, `/`, `%`, `()`. El operador de división actúa de manera diferente en tipos reales y enteros. El operador menos unario se usa para cambiar el signo de una expresión.

- Operadores de *actualización de variables*: `+=`, `-=`, `*=`, `/=`, `%=`.

- El operador *incremento* `++` aumenta el valor de su operando en una unidad y el *decremento* `--` lo disminuye en una unidad.

- **Operadores relacionales.** Se emplean para realizar comparaciones, y son: <, <=, ==, !=, >, >=.

- **Operadores lógicos.** Se utilizan para combinar dos o más expresiones de relación y son los siguientes: && (AND), || (OR) y ! (NOT).

Sentencias de Selección e Iteración

Expresiones y sentencias.

- **Expresión:** Combinación de operandos mediante operadores. La expresión más simple es un operando aislado (una constante o una variable).

- La asignación *también posee un valor*, que es el que adquiere la variable a la izquierda del signo igual. Así se pueden realizar *asignaciones múltiples*.

- **Un programa es un conjunto de sentencias.** En Java cada sentencia debe finalizar con un *punto y coma*.

- Una **sentencia compuesta** (*bloque*) es un conjunto de sentencias encerrado entre llaves.

La sentencia `if`.

- Permite elegir la acción o acciones a realizar en función del valor de verdad de una expresión.

```
if (<expresión>)
   <sentencia>
else   // opcional
   <sentencia>
```

Ejemplo:

```
if (y > 0)
   x = y;
else
   x = -y;
```

- Pueden escribirse *sentencias anidadas*, donde cada sentencia `else` es, a su vez, otra sentencia `if-else`.

- Si existen sentencias `if-else` anidadas, cada sentencia `else` corresponde siempre a la sentencia `if` más próxima.

El operador condicional.

- Es una *forma abreviada* de expresar la sentencia `if-else`. Si la primera expresión es cierta la expresión condicional toma el valor de la segunda expresión, y si es falsa el de la tercera.

```
<expr_1>? <expr_2> : <expr_3>
```

Ejemplo: `x = (y > 0)? y: -y;`

Elección múltiple: La sentencia **`switch`**.

- Si un programa debe elegir una opción entre varias, resulta más legible si se usa una sentencia `switch` en vez de varias sentencias `if-else` anidadas.

```
switch (<expresión_entera>) {
       case const_1: sentencia/s
                     break; // obligatorio
       case const_2: sentencia/s
                     break; // obligatorio

       case const_n: sentencia/s
                     break; // obligatorio
       default: sentencia/s
                     break; // obligatorio
       }
```

Ejemplo:

```
String msj;

switch (nota) { // la variable es de tipo int
```

```
        case 0: msj = "Ni idea."; break;
        case 1: case 2: case 3: msj = "Casi ni idea."; break;
        case 4: msj = "Casi apruebas."; break;
        case 5: msj = "Por los pelos. A esforzarse."; break;
        case 6: msj = "Bien aprobado."; break;
        case 7: case 8: msj = "Notable."; break;
        case 9: case 10: msj = "Sobresaliente."; break;
        default: msj = "Nota incorrecta."; break;
        }

Console.WriteLine(msj);
```

El bucle **`while`**.

- Pertenece a la categoría de *bucles condicionales*. La sentencia (simple o compuesta) se ejecuta de cero a n veces según el valor de verdad de la expresión (*condición de entrada*).

```
while (<expresión>)
      <sentencia>
```

Ejemplo:

```
// programa sencillo de encriptación

using System;

class Encripta {
  public static void Main() {

    const int K = 10; // clave
    int i = 0;
    string str;

    if (args.Length != 0) {
       str = args[0];
       while (i < str.Length) {
             // encriptación
             Console.Write((char) (str[i] + K));
             i++;
             }
       }
  }
}
```

El bucle **do-while**.

- La sentencia (simple o compuesta) se ejecuta de una a n veces según el valor de verdad de la expresión (*condición de salida*), la cual se comprueba tras cada iteración.

```
do <sentencia>
while (<expresión>);
```

Ejemplo:

```
using System; // Programa para adivinar un número

class Adivina {
  public static void Main() {

    const int NUM = 5;    // número a adivinar
    int elegido;

    do {
         Console.Write("\nDame un entero: ");
         elegido = Int32.Parse(Console.ReadLine());
       } while (elegido != NUM);
    Console.WriteLine("\nAcertaste.");
  }
}
```

El bucle **`for`**.

- Permite aumentar la legibilidad de los bucles en algunos casos.

```
for (<inic.>; <test>; <act.>)
    <sentencia>
```

- El formato anterior puede reescribirse del siguiente modo:

```
<inicialización>;
while (<test>) {
      <sentencia>
      <actualización>;
}
```

Ejemplos:

```
for (n = 10; n > 0; n--)
    Console.WriteLine(n);
```

```
for (ch = 'a'; ch <= 'z'; ch++)
    Console.WriteLine("Letra: " + ch);

n = 1;
for (; n <= 10; n++)
    Console.WriteLine(n);

for (; ; )
    Console.WriteLine("Eternidad.");
```

- Ejercicio: Escribir un programa que sume los primeros m términos de la sucesión $A_n = 1/2^n$.

Las sentencias **`break;`** y **`continue;`**.

- La sentencia `break;` puede usarse con los tres bucles anteriores. Cuando el programa llega a esta sentencia dentro de un bucle, su flujo lo abandona y pasa a ejecutar la siguiente sentencia del programa.

Ejemplo:

```
i = Int32.Parse(Console.ReadLine());
while (i <= 10) {
      if (i < 5) break;
      Console.WriteLine(i);
      i = Int32.Parse(Console.ReadLine());
      }
```

- La sentencia `continue;` dentro de un bucle evita el resto de la iteración y desvía el flujo del programa de nuevo hacia la expresión de test. Puede usarse en los tres bucles anteriores, pero **no** en una sentencia `switch`.

Ejemplo:

```
do {
   i = Int32.Parse(Console.ReadLine());
   if (i < 5) continue;
   Console.WriteLine(i);
   } while (i <= 100);
```

El bucle **`foreach`**.

- Permite iterar sobre los items de colecciones sin utilizar índices. La variable tomará todos los posibles valores de la expresión enumerable. El tipo de dicha expresión debe implementar la interfaz `Ienumerable`, de modo que ésta puede ser un array o una colección (una de las clases C# específicas para guardar datos).

```
foreach (<declación variable> in <expresión enumerable>)
        <sentencia>
```

Ejemplo:

```
using System;

public class Paracada {
  public static void Main() {
    string[] itemSet = {"Alpha", "Bravo", "Charlie"};
    foreach (string item in itemSet)
            Console.WriteLine(item);
  }
}
```

Programación Modular

Definición de método:

Unidad de código diseñada para llevar a cabo una tarea determinada (generalización de la noción de operador).

- En orientación a objetos un método se asocia con una clase particular.
- El tipo de un método C# coincide con el valor devuelto por éste.
- En C# todos los métodos deben devolver un valor excepto aquéllos cuyo tipo es `void`.
- Un método puede considerarse como una **caja negra**, definido exclusivamente por la información suministrada *(parámetros de entrada)* y el producto recibido *(resultado de salida).*

  Ejemplo: `Console.WriteLine()`

- Parámetros formales: identificadores de objetos o de datos de tipo primitivo que se colocan en la definición del método, entre los paréntesis, separados por comas e indicando su tipo.

```
using System;

class Maximo {

  public static void Main() {

    int x = 4, y = 10;
    int z;

    z = max(x, y); // parámetros reales: x e y
    Console.WriteLine("Mayor de " + x + " y " + y + ": " + z + ".");
  }

  public static int max(int a, int b) {

    int x; // variable local a max()

    x = (a > b)? a: b;
    return x;
  }
}
```

- Paso de parámetros:

  - Por *valor*: Los valores de los parámetros reales se copian en los parámetros formales. En java los datos de tipo primitivo se pasan por valor.

  - Por *referencia* o por *variable*: Los valores de los parámetros reales (cuando son variables) se modifican dentro de la función y permanecen alterados al abandonarla. En C# los objetos se pasan por referencia.

- Parámetros reales o efectivos: valores particulares que se asignan a los parámetros formales durante la llamada a un método concreto.

  - dato de tipo primitivo (constante o variable).
  - expresión (válida en C#).
  - objeto.

- Devolución de valores:

```
return [expresión];
```

El tipo de un método se antepone a su identificador.

Ejemplo:

```
public static int abs(int x) {
  if (x > 0) return x;
  else return -x;

  Console.WriteLine("Esta sentencia NO se ejecuta.");
}
```

- El tipo void (vacío) se usa cuando un método carece de valor de retorno.

Ejemplo:

```
public static void Main() {
  Console.WriteLine("No hay valor de retorno.");
  return; // sentencia opcional
}
```

```
using System;

class Intercam {

  public static void Main() {

    int x = 5, y = 8;

    Console.WriteLine("X: " + x + ", Y: " + y + ".");
    intercambia(x, y);
    Console.WriteLine("X: " + x + ", Y: " + y + ".");
  }

  public static void intercambia(int a, int b) {

    int temp;

    temp = a;
    a = b;
    b = temp;
  }
}
```

```
using System;

class Intercam {

  public static void Main() {

    int x = 5, y = 8;

    Console.WriteLine("X: " + x + ", Y: " + y + ".");
    intercambia(ref x, ref y);
    Console.WriteLine("X: " + x + ", Y: " + y + ".");
  }

  public static void intercambia(ref int a, ref int b) {

    int temp;

    temp = a;
    a = b;
    b = temp;
  }
}
```

- Alcance de una variable:

  - Local a un método: Su ámbito se reduce al método donde ha sido declarada y fuera de él la variable es desconocida. Pueden existir dos variables locales declaradas en dos métodos distintos pero con el mismo identificador .

  - Global a una clase: Se define dentro de una clase, pero fuera de todos sus métodos y es conocida por **todos** los métodos de la clase. No se recomienda usar una variable global debido a los *efectos colaterales*, excepto cuando ésta se utilice en muchos métodos de la clase donde está declarada.

Ejemplo:

```
using system;

class global {

  static int a = 4; /* variable global y estática porque no se crea
                       ningún objeto */

  public static void Main() {
```

```
    int a = 1; // local a Main()
    Console.WriteLine("Main(): " + a + ".");
    metodo();
  }

  public static void metodo() {
    Console.WriteLine("Método(): " + a + ".");
  }
}
```

- Sobrecarga de métodos: C# permite utilizar el mismo identificador de método para múltiples métodos.

  - *Firma del método*: Es el nombre del método junto con el tipo, orden y número de sus parámetros. El tipo de retorno del método no es parte de la firma.

  - Ejemplo de utilización: Uso de varios métodos constructores sobrecargados para inicializar de modo diferente un objeto cuando éste se crea.

```
int metodo_1(int var_1);
int metodo_1(double var_1);
int metodo_1(int var_1, double var_2);
int metodo_1(double var_2, int var_1);
```

Clases y objetos

- Clase: Implementación de un tipo abstracto de datos en el paradigma de la Programación Orientada a Objetos. Los atributos y los métodos de una clase son los miembros de la misma.

- Objeto: Instancia o ejemplar de una clase. Cada objeto es único porque posee su propio espacio de datos (estado).

- Propiedades de una clase: *Encapsulación* y *Abstracción*:

  - Encapsulación: Un objeto encapsula *datos* y *métodos*. El objeto se usa como una *caja negra* (filosofía cliente-servidor).

    Interfaz pública de una clase: conjunto de métodos que sirven para que los objetos de una clase proporcionen sus servicios.

    Métodos de servicio: definen los servicios que el objeto proporciona y pueden ser invocados por un cliente.

Métodos de soporte: métodos adicionales en un objeto que no definen un servicio utilizable por el cliente pero auxilian a otros métodos en sus funciones.

El estado de un objeto (sus datos) sólo debe ser modificado mediante los métodos de dicho objeto.

- Abstracción: Permite ocultar los detalles de un concepto en el momento oportuno para permitir enfocar la atención en la dirección adecuada.

  Para trabajar con un objeto tan sólo debemos conocer la interfaz pública de su clase.

  La encapsulación es un mecanismo para llevar a la práctica la abstracción.

- Ejemplo:

```
using System;

class Ejempila {

  public static void Main() {

    Pila mipila = new Pila(10);
    int i;
```

```
    for (i = 1; i <= 10; i++)
        mipila.apila(i);

    while (!mipila.pila_vacia()) {
         i = mipila.desapila();
         Console.WriteLine("Dato: " + i + ".");
         }
  } // Fin método "Main()"
}

class Pila {

  private int pila, tam;
  private int [] dato;

  public Pila(int tam) {
    pila = -1;
    this.tam = tam;
    dato = new int [tam];
  } // Fin método constructor "Pila()"

  public bool pila_vacia() {
    return pila == -1;
  } // Fin método "pila_vacia()"
```

```
  public void apila(int dato) {
    if (pila < tam) {
       pila++;
       this.dato[pila] = dato;
       }
    else
       Console.WriteLine("Error: Pila llena.");
  } // Fin método "apila()"

  public int desapila() {

    int dato = 0;

    if (!pila_vacia()) {
       dato = this.dato[pila];
       pila--;
       }
    else
       Console.WriteLine("Error: Pila vacía.");

    return dato;
  } // Fin método "desapila()"
}
```

- Ejercicio: Modificar el programa anterior para que apile números consecutivos a partir del uno inclusive hasta que la pila de diez elementos se haya llenado (para ello diseñar el método `pila_llena`).

- Relaciones entre clases: *generalización*, *asociación* y *dependencia*.

  - Generalización (*herencia*): Se da entre una clase padre o *superclase* y una clase hija o *subclase*. Esta última hereda los datos y métodos del padre, pudiendo añadir los suyos propios.

  - Asociación: Una clase se compone estructuralmente de otras clases: Se utiliza un objeto de una de las clases como atributo (*dato*) de la clase compuesta. La *Agregación* es un caso particular donde existe un todo compuesto por partes (un objeto que contiene otros objetos se denomina *objeto agregado*).

  - Dependencia: Es una relación de utilización, donde un cambio en el estado de un objeto (*independiente*) afecta al estado de otro (*dependiente*), pero no al revés. Aparece en la práctica cuando una clase se relaciona con otra a través de los mensajes que le envía: `mi_monitor.dibujar(mi_circulo)`.

- Definición de clases y creación de objetos.

```
class Nombre_clase {
  <declaraciones>
  <constructores>
  <métodos>
}
```

Ejemplo:

```
using System;

class Cuenta {
  private int num;
  private double saldo;

  public Cuenta(int num, double ini) {
    this.num = num;
    saldo = ini;
  } // Fin método constructor "Cuenta()"

  public void ingreso(int cant) {
```

```
    saldo += cant;
  } // Fin método "ingreso()"
  public void reintegro(int cant) {
    saldo -= cant;
  } // Fin método "reintegro()"

  public double Saldo {
    get { return saldo; }
  } // Fin propiedad "Saldo"
} // Fin clase "Cuenta"

class Banco {
  public static void Main() {
    double s;
    Cuenta c = new Cuenta(325, 6000);
    s = c.Saldo;
    Console.WriteLine("Saldo:" + s + " Eur.");
    c.ingreso(3000);
    s = c.Saldo;
    Console.WriteLine("Saldo:" + s + " Eur.");
  }
}
```

Categorías de tipos

- Todos los tipos pertenecen a dos categorías: tipos por valor y tipos por referencia.

  - *Tipos por valor*: Una variable de este tipo consta de una zona de memoria donde se almacena su valor. Cuando se copia en una variable el valor de otra, se realiza una copia del valor de la variable original en la zona de memoria de la segunda variable. Así pues, una modificación del valor de la primera variable no afectará al valor de la segunda y viceversa.

    ```
    int a, b;
    a = 4;
    b = a;
    ```

  - *Tipos por referencia*: Una variable de este tipo almacena una referencia a la zona de la memoria libre donde se encuentran los datos, en lugar de almacenar estos directamente. Por tanto, para acceder a los datos, el soporte en tiempo de ejecución leerá la dirección de memoria guardada en la variable y, a continuación, accederá a la posición de memoria donde están los datos.

```
Cuenta c1 = new Cuenta(1, 10);
Cuenta c2 = new Cuenta(2, 20);

c1 = c2;
// ambas variables apuntan ahora al mismo objeto
```

Al hacer la asignación, `c1 = c2;` no se ha sustituido el contenido del objeto `c1` con el del objeto `c2`, sino que ahora la variable `c1` ahora apunta al objeto `c2` (`c1` y `c2` hacen referencia al mismo conjunto de datos). Por tanto, a partir de ahora la modificación de un dato a través de la variable `c1` afectará del mismo modo al dato referenciado por la variable `c2`.

El objeto `c1` sigue ocupando memoria libre, pero no tiene ninguna referencia que lo apunte. C# dispone de un *recolector automático de basura* (*garbage collector*), que elimina los objetos que ya no son accesibles e interviene cuando la memoria libre de la aplicación disminuye de forma significativa. Durante la eliminación de un objeto, el recolector de basura invoca a su *método destructor*, si éste ha sido programado. Un destructor se utiliza para liberar los posibles recursos utilizados por la clase.

- Modificador `static`.

  - *Variables de clase*: A diferencia de las variables de instancia (existentes para cada objeto de la clase), son compartidas por todos los objetos de una clase. Únicamente son accesibles a través del nombre de la clase.

  - *Constantes estáticas*: Sus valores son compartidos por todos los objetos de la clase.

  Ejemplo:

```
Using System;

class Principal {
  public static void Main() {
    StatCont cont1 = new StatCont();
    Console.WriteLine(StatCont.num);
    StatCont cont2 = new StatCont();
    Console.WriteLine(StatCont.num);
    StatCont cont3 = new StatCont();
    Console.WriteLine(StatCont.num);
  }
} // fin clase "Principal"
```

```
class StatCont {
  public static long num;

  public StatCont() { // constructor
    num++; // incremento var. estática
  }
} // fin clase "StatCont"
```

- *Métodos de clase*: Están asociados a la clase y no a sus objetos. Se invocan a través de la propia clase. Sólo pueden utilizar variables estáticas o locales al método, ya que al no operar en el contexto de un objeto particular, no pueden usar variables de instancia.

- Ejemplo: métodos de la clase `Math` (`System`).

```
double Abs(double a);
double Pow(double a, double b);
double Sqrt(double a);
couble Sin(double a);
double Cos(double a);
double Exp(double a);
double Log(double a);
double Log10(double a);
```

Definiciones

- La especificación de *Infraestructura de Lenguaje Común* (CLI) incluye el *Sistema de Ejecución Virtual* (VES) también denominado *motor de ejecución* (*Runtime*). El soporte en tiempo de ejecución compila en modo *just-in-time* los programas en C# traducidos ya al *Lenguaje Intermedio Común* (CIL). El código ejecutado bajo el control del soporte en tiempo de ejecución se denomina *código gestionado*.

- *Base Class Library* (BCL): Biblioteca de clases común a todos los lenguajes .NET que todos los desarrolladores pueden utilizar. Los nombres BCL de los tipos numéricos básicos incluyen atributos constantes para obtener sus valores máximo (`MAX_VALUE`) y mínimo (`MIN_VALUE`) y métodos conversores (*parsers*) de cadenas (`string`) a tipos numéricos.

- *Ensamblado* (*assembly*): Agrupación de tipos, recursos y funcionalidades diseñados para operar conjuntamente. Se almacenan en ficheros “.exe” o “.dll” y son generados a partir de una compilación.

La clase cadena (`String`)

- En C# las cadenas de caracteres alfanúmericos pertenecen al tipo básico `string` cuyo nombre BCL es la clase `System.String`. El índice del primer carácter es el cero.

```
string cadena = "Hola" + " amigo.";
cadena += " ¿Cómo estás?";
```

Método	Significado
`CompareTo(`<cadena>`)`	Compara dos cadenas
`Length()`	Longitud cadena
`IndexOf(`<carácter>`)`	Posición primera ocurrencia
`LastIndexOf(`<carácter>`)`	Posición última ocurrencia
`Replace(`<c1>`, `<c2>`)`	Sustituye carácter c1 por c2
`Substring(`<n>, <l>`)`	Subcadena en posición *n* y longitud *l*
`ToUpper()`	Cadena en mayúsculas
`ToLower()`	Cadena en minúsculas
`Trim()`	Elimina todos los espacios
`valueOf(`<n1>`)`	Convierte *n1* a cadena

- Las cadenas se pueden concatenar utilizando el operador +. También puede usarse el operador +=.

- Ejercicio: Codificar un programa que use los métodos anteriores para manipular cadenas de caracteres.

La clase array

- Definición de array:

  - Estructura de Datos.
  - Serie de elementos.
  - Todos del mismo tipo.

- Acceso a un elemento:

  - Nombre del array.
  - Posición del elemento dentro de éste (uso de **índices**).

- Tipos de arrays:

  - Unidimensionales (vectores).
  - Bidimensionales (tablas).
  - Tridimensionales (cubos).
  - Multidimensionales (en general).

- Los arrays deben tener la longitud mínima necesaria. Nunca deben declararse arrays excesivamente grandes.

- En C# los arrays son objetos y sus elementos pueden ser de cualquier tipo, incluso objetos. El primer elemento siempre tiene como índice el cero.

- Declaración de un array:

```
Tipo [] Nombre = new Nombre[Tam];

// array de 365 reales "dobles"
double [] dato = new double [365];
```

```
// tabla de 10x50 enteros
int [, ] tabla = new int [10, 50];

// tres arrays unidimensionales
int [] m1, m2, m3;
m1 = new int [10];
m2 = new int [20];
m3 = new int [30];
```

- Inicialización de un array:

```
/* el compilador cuenta el número de elementos del array */
char [] abc = {'A', 'B', 'C'};

// array unidimensional de enteros
int [] nums = {1, 2, 3, 4, 5};

// matriz triangular inferior
int [, ] dato = {{1, 0, 0}, {2, 3, 0}, {4, 5, 6}};
```

- Nunca se debe referenciar una posición cuyo índice esté fuera del intervalo del array.

- El operador `[]` de C# realiza un control de límites automático: Un índice fuera de rango genera una excepción (`IndexOutOfRangeException`) que es posible capturar para actuar en consecuencia.

- Si las dimensiones de un array se especifican en tiempo de compilación (*dimensionamiento estático*) se reserva la memoria necesaria para dicho array, aunque no se usen todos sus elementos.

- En C#, como en muchos lenguajes modernos, se puede reservar la memoria necesaria para un array en tiempo de ejecución (*dimensionamiento dinámico*). Así es posible asignar las dimensiones en función de variables, cuyos valores son leídos durante la ejecución del programa: C# soporta *arrays dinámicos*.

- Los arrays son de longitud fija. Una vez creados (aunque sean dinámicos), no se puede modificar su tamaño.

- En C# se puede conocer en cualquier momento el número de elementos que tiene un array utilizando la propiedad `Length` de la propia clase.

  Ejemplo:

```
using System;

class Vector {
  public static void Main() {

    int tam, i;

    Console.Write("Tamaño matriz: ");
    tam = Int32.Parse(Console.ReadLine());
    int [] datos = new int [tam];

    // inicializando matriz
    for (i = 0; i < datos.Length; i++) datos[i] = i + 1;
    // escribiendo matriz
    for (i = 0; i < datos.Length; i++)
        Console.Write(datos[i] + " ");
  }
}
```

- Ejercicio:

```
using System;

class Matriz {
  public static void Main() {

    int tam1, tam2, i, j;

    Console.Write("Numero de filas: ");
    tam1 = Int32.Parse(Console.ReadLine());
    ...

    int [, ] datos = new int [tam1, tam2];

    // inicializando matriz
    ...
            datos[i, j] = 10 * i + (j + 1);

    // escribiendo matriz
   ...
  }
}
```

- En el método `Main()` puede declararse una matriz de tipo cadena (`string`). Los elementos del array `args` capturan las cadenas de caracteres introducidos en la línea de ejecución.

```
public static void main(String [] args) {
  <bloque de sentencias>
}
```

Ejemplo:

```
using System;

class Argumentos {
  public static void Main(String [] args) {
    for (int i = 0; i < args.Length; i++)
      Console.WriteLine("args[{0}]: {1}.", i,
                        Int32.Parse(args[i]));
  }
}
```

```
// Uso de matriz triangular

using System;

class Triangular {
  public static void Main() {

  int [][] dato = new int [4][];
  int i, j, suma;

  dato[0] = new int [] {1};
  dato[1] = new int [] {2, 3};
  dato[2] = new int [] {4, 5, 6};
  dato[3] = new int [] {7, 8, 9, 10};

  // Escribiendo la matriz
  for (i = 0; i < dato.Length; i++) {
      for (j = 0; j < dato[i].Length; j++)
          Console.Write(dato[i][j] + " ");
      Console.WriteLine();
      }
```

```
  Console.WriteLine();

  // Sumando filas
  for (i = 0; i < dato.Length; i++) {
      for (j = suma = 0; j < dato[i].Length; j++)
          suma += dato[i][j];
      Console.WriteLine("Fila {0}: Suma: {1}.", i+1, suma);
      }
  }
}
```

- En C# los arrays son objetos (tipos por referencia). Por tanto, sus valores pueden ser modificados por los métodos a los cuales los arrays se transfieren como parámetros, y dichos valores permanecen modificados al abandonar estos métodos.

- Ejercicio: Escribir un programa donde el método principal invoque a otro auxiliar, el cual modifique un array unidimensional declarado en el principal.

Herencia

- *Mecanismo de abstracción que permite crear una clase nueva a partir de otra ya existente*. La nueva clase contiene automáticamente todos los atributos y métodos de la clase original.

- El programador puede añadir nuevos atributos y métodos a la nueva clase o modificar los heredados para definir de modo apropiado la nueva clase.

- Las clases creadas mediante la herencia son menos complejas que si tuviesen que incluir los métodos y atributos que ya heredan de otras. *La herencia hace posible la reutilización del software*.

- La derivación de una clase nueva (*clase derivada* o *subclase*) de otra (*clase base* o *superclase*) por herencia da lugar a la relación “es-un” entre las dos.

```
class clase_derivada : clase_base {
  <contenido de la clase>
}
```

- Los atributos y métodos heredados retienen en la subclase las características de visibilidad originales.

- La herencia es unidireccional: de la clase padre a la clase hija. *C# sólo soporta la herencia simple*.

- Los modificadores de visibilidad controlan los atributos y métodos que se heredan: Los miembros con visibilidad `public` se heredan y los que tienen visibilidad `private` no.

- El modificador `protected` establece un nivel intermedio de protección: Permite que los miembros de una superclase así etiquetados sean accesibles para las subclases.

- La referencia `base` permite a una clase derivada reutilizar el código de un método de la clase base o de su constructor.

- Si el constructor de la clase base está sobrecargado, la referencia `base` puede ser escrita utilizando cualquier forma definida en la clase base. Se ejecutará el constructor que tenga la misma firma.

- En una jerarquía de varios niveles, la referencia `base` se refiere siempre a la clase base inmediatamente superior a la clase derivada que la utiliza.

Modificador	**Clases e Interfaces**	**Métodos y variables**
public	Visibles desde cualquier lugar	Accesibles desde cualquier lugar
private	Clases anidadas visibles sólo desde sus respectivas clases contenedoras	Accesibles sólo desde la propia clase
protected	Clases anidadas visibles desde sus clases contenedoras y desde cualquier clase derivada	Accesibles desde la clase base y desde cualquier clase derivada
internal	Visibles sólo dentro del ensamblado	Accesibles sólo dentro del ensamblado
protected internal	Clases anidadas visibles desde cualquier clase derivada y dentro del ensamblado	Accesibles desde la clase base, desde cualquier clase derivada y dentro del ensamblado

```
using System;

class Libro {
  // campos de instancia (instance fields)
  protected string titulo, autor;
  protected int [] fecha = new int [3];

  // método constructor
  public Libro(String titulo, String autor, int dia, int mes, int anio){
    this.titulo = titulo; this.autor = autor;
    fecha[0] = dia; fecha[1] = mes; fecha[2] = anio;
  }

  // métodos de servicio
  public string Titulo() {
    return titulo;
  }
  public string Autor() {
    return autor;
  }
  public int [] Fecha() {
    return fecha;
  }
} // Fin clase base "Libro"
```

```
class Tesis: Libro {
  protected String departamento;

  public Tesis(String titulo, String autor, String departamento, int dia,
               int mes, int anio): base(titulo, autor, dia, mes, anio) {
    this.departamento = departamento; // "this" se refiere al objeto
  }

  public string Departamento() { return departamento; }
} // Fin clase derivada "Tesis"

class Programa {
  public static void Main() {
    Tesis mi_tesis = new Tesis("Sistemas Alternativos",
                               "Juan Guerra Paz",
                               "Ciencia Alternativa", 6, 2, 2002);
    int [] fecha = mi_tesis.Fecha();

    Console.WriteLine("Título: " + mi_tesis.Titulo());
    Console.WriteLine("Autor: " + mi_tesis.Autor());
    Console.WriteLine("Departamento: " + mi_tesis.Departamento());
    Console.WriteLine("Fecha: " + fecha[0] + "-" + fecha[1] + "-" +
                      fecha[2]);
  }
} // Fin clase "Programa"
```

- Cuando una variable se hereda y se declara en la clase derivada una nueva variable con el mismo identificador y el modificador `new`, la variable de la clase base ha sido *ocultada* en la clase derivada.

- La variable de la clase base aún existe, y se puede acceder a ella indicando expresamente que se hace referencia a la clase base usando la referencia `base`.

```
using System;

class Padre {
  protected int dato = 10;
} // Fin clase "Padre"

class Hija : Padre {
  new private int dato;

  public Hija(int dato) {
    this.dato = dato;
  }

  public void valor_hija() {
    Console.WriteLine("H: " + dato);
  }
```

```
  public void valor_padre() {
    Console.WriteLine("P: " + base.dato);
  }
} // Fin clase "Hija"

class Programa {
  public static void Main() {
    Hija mi_hija = new Hija(5);
    mi_hija.valor_hija();
    mi_hija.valor_padre();
  }
} // Fin clase "Programa"
```

- Cuando se escribe un método con la misma firma que un método que se hereda, el método accedido a través de las instancias de la clase derivada es el definido en ésta. El método de la clase base debe declararse usando el modificador `virtual` para permitir su sobrescritura en la clase derivada.

- La *sobrescritura de métodos* (*method overriding*) permite el uso de una única firma (identificador y lista de parámetros) para distintas clases relacionadas por herencia. El método que sobrescribe debe usar la palabra reservada `override`.

```
using System;

class X {
  protected int m = 10;

  virtual public void imprimir() { Console.WriteLine("X: " + m); }
} // Fin clase "X"

class Y: X {
  protected int n = 25;

  override public void imprimir() { Console.WriteLine("Y: " + n); }
} // Fin clase "Y"

class Programa {
  public static void Main() {
    X x = new X();
    Y y = new Y();
    x.imprimir();
    y.imprimir();
  }
} // Fin clase "Sobrescribe"
```

- Se debe distinguir entre *sobrescritura* y *sobrecarga* (mismo identificador pero distinta firma).

- En C# todas las clases, predeterminadas o creadas por el usuario, heredan la clase `Object`. Ésta constituye la *clase raíz* de toda la jerarquía de clases de C# y posee métodos interesantes que heredan todos sus desdendientes.

```
// Definiciones equivalentes

class nombre_clase {                    class nombre_clase : Object {
  <contenido de la clase>                 <contenido de la clase>
}                                       }
```

- El método `ToString()` devuelve una cadena que contiene una representación del objeto que lo llama. Cualquier clase puede sobrescribirlo para proporcionar una representación `string` más apropiada de una clase particular.

- Si no se sobrescribe y se llama al método, se usará el de la clase `Object`. Cuando se trata de clases definidas por el usuario, invocando a `ToString()` puede saberse a qué clase pertenece un objeto concreto.

```
using System;

class Alumno {
  private string nombre;
  private int num;

  public Alumno(string nombre, int num) {
    this.nombre = nombre;
    this.num = num;
  } // método constructor

  public override string ToString() {
    return "Datos: " + nombre + ", " + num;
  }
}

class Programa {
  public static void Main() {
    Alumno alum1 = new Alumno("Juan", 123);
    Console.WriteLine(alum1);
    Alumno alum2 = new Alumno("Ana", 456);
    Console.WriteLine(alum2);
  }
}
```

```
using System;

class Cuenta {
  protected int num;
  protected double saldo;

  public Cuenta(int num, double saldo) {
    this.num = num; this.saldo = saldo;
  }

  public virtual void Ingreso(double cant) {
    Console.WriteLine("Ingreso en: " + num);
    Console.WriteLine("Cantidad: " + cant + " Eur");
    saldo += cant;
    Console.WriteLine("Saldo: " + saldo + " Eur");
  }

  public virtual void Reintegro(double cant) {
    Console.WriteLine("Reintegro de: " + num);
    Console.WriteLine("Cantidad solicitada: " + cant + " Eur");
    if (saldo < cant) Console.WriteLine("Saldo insuficiente");
    else { saldo -= cant;
           Console.WriteLine("Saldo: " + saldo + " Eur");
         }
  }
```

```
  public double Saldo { get { return saldo; } }
} // Fin clase "Cuenta"

class Cuenta_ahorro: Cuenta {

  protected double interes;

  public Cuenta_ahorro(int num, double saldo, double interes):
         base(num, saldo) { this.interes = interes; }

  public void Intereses() {
    Console.WriteLine("Actualizando intereses en " + num);
    saldo += saldo * (interes / 100);
    Console.WriteLine("Saldo: " + saldo + " Eur");
  }
} // Fin clase "Cuenta_ahorro"

class Cuenta_2000: Cuenta_ahorro {

  private double penaliza;

  public Cuenta_2000(int num, double saldo, double interes,
                     double penaliza): base(num, saldo, interes) {
    this.penaliza = penaliza; }
```

```
  public override void Reintegro(double cant) {
    Console.WriteLine("\nReintegro con penalizacion de: " + num);
    Console.WriteLine("Cantidad solicitada: " + cant + " Eur");

    if (saldo < cant)
       Console.WriteLine("Saldo insuficiente");
    else {
          saldo -= cant * (1 + (penaliza / 100));
          Console.WriteLine("Saldo: " + saldo + " Eur");
        }
  }
} // Fin clase "Cuenta_2000"

class Cuenta_corriente: Cuenta {
  private Cuenta_ahorro ahorro;

  public Cuenta_corriente(int num, double saldo, Cuenta_ahorro ahorro):
         base(num, saldo) {
    this.ahorro = ahorro;
  }

  public override void Reintegro(double cant) {
    Console.WriteLine("\nReintegro de: " + num);
    Console.WriteLine("Cantidad solicitada: " + cant + " Eur");
```

```
    if (saldo < cant) {
       if (saldo + ahorro.Saldo < cant)
          Console.WriteLine("Saldo insuficiente");
       else {
          Console.WriteLine("Retirando fondos de la cuenta asociada");
          ahorro.Reintegro(cant - saldo);
          saldo = 0;
          }
       }
    else saldo -= cant;
    Console.WriteLine("Saldo cuenta corriente: " + saldo + " Eur");
  }
} // Fin clase "Cuenta_corriente"

class Programa {
  public static void Main() {
    Cuenta_ahorro libreta = new Cuenta_ahorro(1, 1000, 4);
    Cuenta_2000 libreton = new Cuenta_2000(2, 2000, 8, 5);
    Cuenta_corriente cuenta = new Cuenta_corriente(3, 500, libreta);
    libreta.Reintegro(100);
    libreton.Reintegro(200);
    libreton.Intereses();
    cuenta.Reintegro(1000);
  }
} // Fin clase "Programa"
```

Polimorfismo

- El *polimorfismo* es la capacidad que posee una referencia para permitir la asignación de objetos de distintas clases relacionadas por herencia. Esta capacidad es útil si estos objetos tienen métodos sobrescritos, que son los que se van a invocar a través de la *referencia polimórfica*.

- En la clase base se define un método, que es el que se va a usar polimórficamente. Dicho método se redefine en las clases derivadas de la clase base, de modo que el método original se sobrescribe sucesivamente.

- Cuando se invoca a un método polimórfico, el motor de ejecución determina la clase real del objeto para invocar la definición apropiada del mismo: Es la clase del objeto referido la que indica qué método utilizar, y no la clase con que se declara inicialmente la referencia.

- El polimorfismo es un concepto importante en la programación orientada a objetos al que están asociados otros, como la herencia, la sobrecarga de métodos y propiedades, y la sobrescritura de los mismos.

```
using System;

class Padre {
  public virtual void Miclase() {
    Console.WriteLine("Padre.");
  }
}

class Hijo: Padre {
  public override void Miclase() {
    Console.WriteLine("Hijo.");
  }
}

class Nieto: Hijo {
  public override void Miclase() {
    Console.WriteLine("Nieto.");
  }
}
```

```
class Polimorfismo {
  public static void Main() {
    Padre padre = new Padre();
    Hijo hijo = new Hijo();
    Nieto nieto = new Nieto();

    Padre referencia = padre; // referencia polimórfica
    referencia.Miclase();

    referencia = hijo;
    referencia.Miclase();

    referencia = nieto;
    referencia.Miclase();
  }
}
```

Clases abstractas

- Puede ser conveniente disponer de clases en las que todas las propiedades y los métodos estén declarados, aunque no desarrollados. Estas clases se denominan *clases diferidas* o *abstractas*. Su utilidad consiste en especificar la abstracción representada por la clase.

```
abstract class nombre_clase {
  // código de la clase
}
```

- Los métodos especificados (firma y tipo de retorno) pero no definidos se llaman *métodos abstractos*. Una clase abstracta debe contener al menos una propiedad o un método abstracto.

- Un método abstracto no puede declararse como `sealed` (se debe poder sobrescribir) ni `static`, aunque es declarado implícitamente como `virtual` para que pueda sobrescribirse. Tampoco se pueden declarar constructores abstractos.

- En la práctica, las clases abstractas se utilizan dentro de una jerarquía de clases en la cual *la clase abstracta define sólo parte de su implementación, y difiere el resto de la misma a las clases derivadas por medio de la sobrescritura de propiedades y métodos abstractos*.

- No hay restricciones acerca de dónde se puede definir una clase abstracta en la jerarquía de clases. Sin embargo, lo lógico es que aparezca en niveles altos, sino en el primero, de la jerarquía de clases.

- Una clase abstracta puede contener datos, y combinar propiedades y métodos abstractos con no abstractos.

- Cualquier clase que contenga propiedades o métodos abstractos debe declarse como abstracta. Una clase abstracta puede no contener propiedades y métodos abstractos propios, ya que podría derivar de una clase base abstracta de la cual heredase sus propiedades y métodos abstractos.

- Las clases descendientes de una clase base abstracta no tienen que sobrescribir todas sus propiedades y métodos abstractos. No obstante, si una clase derivada no

sobrescribe todos las propiedades y los métodos abstractos de la clase base, debe declararse también como abstracta.

- Las clases abstractas pueden usarse para declarar referencias a objetos. Una referencia de una clase abstracta puede referir a objetos de clases derivadas mediante referencias polimórficas.

- Un modo de asegurar la sobrescritura de un método heredado con la definición apropiada para la clase derivada, y así evitar usar por error el método heredado de la clase base consiste en utilizar una clase base abstracta.

- Mediante el uso correcto de las clases abstractas se asegura que el futuro personal de mantenimiento de la aplicación entienda la estructura de clases y su propósito.

```
using System;

abstract class Automovil {
  abstract public void Mensaje();
}
```

```
class Deportivo: Automovil {
  public override void Mensaje() {
    Console.WriteLine("Veloz como el rayo.");
  }
}

class Turismo: Automovil {
  public override void Mensaje() {
    Console.WriteLine("Para el uso diario.");
  }
}

class Familiar: Automovil {
  public override void Mensaje() {
    Console.WriteLine("Gran capacidad.");
  }
}
```

```
class Auto {
  public static void Main() {
    // Array de referencias polimórficas
    Automovil [] auto = new Automovil[3];

    auto[0] = new Deportivo();
    auto[1] = new Turismo();
    auto[2] = new Familiar();
    imprmsj(auto);
  }

  public static void imprmsj(Automovil []auto){
    for (int i = 0; i < 3; i++) auto[i].Mensaje();
  }
}
```

- Conclusión: *La Programación orientada a objetos implica el uso de la herencia*. Sin herencia se tiene únicamente *programación basada en objetos*.

Delegados

- Un delegado (*delegate*) encapsula métodos como objetos, y de este modo permite realizar una invocación indirecta a un método en tiempo de ejecución. Es una abstracción de alto nivel del tipo de dato equivalente a un puntero a una función en C, o a un puntero a un método en C++.

```
using System;

delegate void Procedimiento(string arg);

class DemoDelegado {

  private static void Metodo_1(string arg) {
    Console.WriteLine("Metodo 1: {0}.", arg);
  }
  private static void Metodo_2(string arg) {
    Console.WriteLine("Metodo 2: {0}.", arg);
  }
  private void Metodo_3(string arg) {
    Console.WriteLine("Metodo 3: {0}.", arg);
  }
```

```
  public static void Main() {

    Procedimiento misProcs = null;
    DemoDelegado demo = new DemoDelegado();

    misProcs += new Procedimiento(DemoDelegado.Metodo_1);
    misProcs += new Procedimiento(DemoDelegado.Metodo_2);
    misProcs += new Procedimiento(demo.Metodo_3);
    misProcs("Prueba"); // llamada indirecta
  }
}
```

- La declaración de un delegado es una abstracción que no se traduce en un código ejecutable que realice algún tipo de tarea. En el ejemplo anterior, los dos primeros métodos son estáticos, mientras que el tercero no lo es. Por tanto, este último método debe ir precedido por la correspondiente referencia a un objeto, el cual es creado en el programa.

- Un objeto de un tipo delegado puede tener como valor *null* (no se le ha asignado ningún método). En tal caso, si dicho objeto se intenta utilizar se producirá una excepción.

- Los métodos a asociar a un delegado deben tener el mismo número y tipo de parámetros, y el mismo valor de retorno que posee el tipo delegado. Dichos métodos no pueden ser abstractos pero sí ser virtuales y, por tanto, sobrescritos.

- El operador += permite añadir más de un método a un objeto delegado (*multicast delegate*). Por tanto, éste puede almacenar mútiples valores simultáneamente. Asímismo, el operador −= elimina el método que figura a su derecha.

- El uso de un objeto delegado con múltiples métodos asociados devolverá el valor de retorno o el parámetro de salida correspondiente a la última llamada, si éstos forman parte de la declaración del tipo delegado.

- La utilización de delegados permite aumentar la flexibilidad de los programas mediante la transferencia a un método de un objeto delegado como parámetro.

```
using System;

class Programa {

  private delegate bool Comparador(int primero, int segundo);
```

```
private static bool Mayor(int primero, int segundo) {
  return primero > segundo;
}

private static bool AlfaMayor(int primero, int segundo) {

  int resultado;

  resultado = (primero.ToString().CompareTo(segundo.ToString()));
  return resultado > 0;
}

private static void Burbuja(int [] items, Comparador comp) {

  int i, j, temp;

  for (i = items.Length - 1; i >= 0; i--)
      for (j = 1; j <= i; j++)
          if (comp(items[j - 1], items[j])) {
             temp = items[j - 1];
             items[j - 1] = items[j];
             items[j] = temp;
             }
}
```

```
  public static void Main() {

    int i;
    int[] items = new int[5];
    string op;
    Comparador comp;

    Console.Write("Introducir \'1\' para comparar enteros u " +
                  "otro valor para comparar caracteres: ");
    op = Console.ReadLine();

    if (op != null && op == "1") comp = new Comparador(Mayor);
    else comp = new Comparador(AlfaMayor);

    for (i = 0; i < items.Length; i++) {
        Console.Write("Introducir un entero: ");
        items[i] = int.Parse(Console.ReadLine());
        }

    Burbuja(items, comp);

    for (i = 0; i < items.Length; i++) Console.WriteLine(items[i]);
  }
}
```

Eventos

- Un *evento* es una condición detectable capaz de enviar una notificación.

- Una *notificación* es una señal generada por un evento que es enviada a un destinatario, el cual se determina en tiempo de ejecución.

- En el lenguaje C#, un evento es un tipo especial de delegado que se utiliza en la *programación basada en eventos*.

- Un evento declarado como público permite a otras clases usar los operadores += y −= sobre el mismo. Por tanto, un método de otra clase puede suscribir un evento añadiendo métodos a su delegado.

- Un evento puede dispararse (de modo que se invoquen los métodos asociados al mismo) únicamente dentro de la clase que lo declara, independientemente de su modificador de visibilidad. Por esta razón, los eventos proporcionan un mayor nivel de encapsulamiento con respecto a los delegados.

```
using System;

public class MyEventTarget {
  public void MyMethod() { Console.WriteLine("Target method."); }
}

public class MyEventSource {
  public delegate void MyEventHandler(); // declare a delegate
  public event MyEventHandler OnMyEvent; // declare an event field

  public void FireMyEvent() {
    if (OnMyEvent != null) OnMyEvent(); // fire the event
  }
}

public class Program {
  public static void Main() {
    MyEventSource myeventsource = new MyEventSource();
    MyEventTarget myeventtarget = new MyEventTarget();
    // subscription to the event by adding a method
    myeventsource.OnMyEvent +=
      new MyEventSource.MyEventHandler(myeventtarget.MyMethod);
    myeventsource.FireMyEvent();
  }
}
```

```
using System;

public class MyTargetObject {

  private string id;

  public MyTargetObject(string id) { this.id = id; } // constructor
  public string ID { get { return id; } } // property
}

public class MySourceEvent {

  // declare a delegate
  public delegate void MyEventHandler(MyTargetObject obj, bool status);
  public event MyEventHandler MyEvent; // declare an event field

  MyTargetObject obj = new MyTargetObject("Class: My Object");

  public void FireMyEvent(bool status) { // notification launcher method
    if (MyEvent != null) MyEvent(obj, status);
  }
}
```

```
public class Program {
  public static void Main() {
    MySourceEvent obj = new MySourceEvent();
    bool eventFired = false;
    int i, num;

    // subscription to event by adding handler method to event delegate
    obj.MyEvent += OnMyEvent;
    Console.Write("Enter a number from 0 to 10 for a true status: ");
    num = Int16.Parse(Console.ReadLine());

    for (i = 0; i <= 10; i++) {
        if (i == num) { // first event condition
           Console.WriteLine("First event condition detected.");
           eventFired = true;
           // fire event and send notification (event handler method)
           obj.FireMyEvent(eventFired);
           }
        }

    if (!eventFired) { // second event condition.
       Console.WriteLine("Second event condition detected.");
       obj.FireMyEvent(eventFired);
       }
  }
```

```
  // event handler method
  private static void OnMyEvent(MyTargetObject obj, bool status) {
    Console.WriteLine("Target object ID: {0}. Method status: {1}.",
                      obj.ID, status);
    Console.WriteLine("Event handler method \"OnMyEvent()\" " +
                      "successfully called.");
  }
}
```

Bibliografía

- J. Rumbaugh. *Object-Oriented Modeling and Design*. Prentice Hall, 1991.
- B. Meyer. *Object-oriented Software Construction (2nd ed.)*. Prentice Hall, 1997.
- *C# Language Specification (4th ed.)*. ECMA International, 2006.
- *C# Language Specification. Version 4.0*. Microsoft Corp., 1999-2010.
- M. Michaelis. *Essential C# 4.0 (3rd ed.)*. Pearson Educación, 2010.
- F.J. Ceballos Sierra. *Microsoft C#. Curso de Programación. Segunda Edición*. RA-MA Editorial, 2011.
- T. Faison. *Event-Based Programming: Taking Events to the Limit*. Apress, 2006.