C#

Otros aspectos del lenguaje

Índice

Programación funcional

Paradigma de programación **declarativo**, no imperativo
Se dice cómo es el problema a resolver, no los pasos a seguir para resolverlo.

Lenguajes funcionales: **Haskell**, **Miranda**, **Scala**, **Clojure**

Muchos lenguajes populares han incorporado características funcionales.

Características principales

Transparencia referencial: si llamamos a una función varias veces pasándole los mismos parámetros, debe producir el mismo resultado.

Inmutabilidad: los datos deben ser inmutables para evitar efectos colaterales.

Composición de funciones: la salida de una función se puede tomar como entrada de la siguiente.

Funciones de primer orden: funciones que se pasan como parámetro a otras funciones.

Código imperativo vs código declarativo

Código imperativo

```
List<Persona> adultos = new List<Persona>();
for(int i = 0; i < personas.Count; i++)
{
    if (personas[i].Edad >= 18)
    {
        adultos.Add(personas[i]);
    }
}
```

Código declarativo

```
List<Persona> adultos =
    personas.Where(p => p.Edad >= 18)
            .ToList();
```

Expresiones Lambda

Expresiones breves que simplifican la implementación de ciertas funciones.

Normalmente se aplican a implementar interfaces, aunque en algunos lenguajes tienen más usos.

Se les suele llamar también **arrow functions** por la flecha característica en su código.

Estructura

1- Parámetros de la interfaz, entre paréntesis (o sin paréntesis si sólo hay uno).

2- Símbolo de la flecha =>.

3- Cuerpo de la función. Si sólo es un return se pueden omitir las llaves y el return. Si no, se pone todo el código entre llaves.

```
(p1, p2) => p1.Nombre.CompareTo(p2.Nombre));
```

LINQ

Language-INtegrated Query.

Librería para manejar colecciones de datos en C#.

Sintaxis similar a **SQL**.

Necesitamos incluir **System.Linq**.

El resultado devuelto se puede asignar a un **IEnumerable** (interfaz implementada por **List**).

Sintaxis básica

```
IEnumerable<Persona> adultos =
    from persona in personas
    where persona.Edad >= 18
    select persona;

foreach(Persona p in adultos)
{
    Console.WriteLine(p);
}
```

FROM

Primera línea de **LINQ**. Determina la variable que recorre la colección.

```
IEnumerable<Persona> adultos =
    // Recorremos la lista "personas" con la variable "persona"
    from persona in personas
    ...
```

SELECT

Última línea de **LINQ**. Determina lo que se devuelve como resultado de cada elemento recorrido.

```
IEnumerable<Persona> adultos =
    from persona in personas
    ...
    select persona;
    // Devolvemos todo el objeto Persona
```

WHERE

Determina la condición (o condiciones) a cumplir para los objetos que pasen el filtro.

```
IEnumerable<Persona> adultos =
    from persona in personas
    // Personas adultas
    where persona.Edad >= 18
    select persona;
```

ORDERBY

Determina la ordenación del resultado por uno de sus campos.

```
IEnumerable<Persona> adultos =
    from persona in personas
    where persona.Edad >= 18
    orderby persona.Edad descending
    select persona;
```

IEnumerable

Where(condición)

filtra elementos de la colección según la condición indicada.

Select(dato)

permite seleccionar un atributo de los objetos de la colección.

OrderBy(comparador) y OrderByDescending(comparador)

ordenan la colección por el atributo indicado.

ToList()

obtiene una lista resultado de acumular una o varias de las operaciones anteriores.

Otros: **Average**, **Max**, **Min**, **Any**...

```
List<Persona> adultasOrdenadas =
    personas.Where(p => p.Edad >= 18)
            .OrderByDescending(p => p.Edad)
            .ToList();
```

```
double mediaEdad =
    personas.Where(p => p.Edad >= 18)
            .Average(p => p.Edad);
```

```
List<string> nombresAdultos =
    personas.Where(p => p.Edad >= 18)
            .Select(p => p.Nombre)
            .ToList();

string nombresUnidos =
    String.Join(", ", nombresAdultos);
```

Operaciones con diccionarios

El funcionamiento es similar, pero en la consulta hay que distinguir si nos referimos a la parte de la **clave** (**Key**) o del **valor** (**Value**).

```
Dictionary<string, Persona> personas = new
Dictionary<string, Persona>();
personas.Add("11A", new Persona("11A", "Nacho", 44));
personas.Add("22B", new Persona("22B", "May", 43));
personas.Add("33C", new Persona("33C", "Mario", 9));
personas.Add("44D", new Persona("44D", "Laura", 8));
personas.Add("55C", new Persona("55E", "Juan", 70));

IEnumerable<Persona> adultos =
    from persona in personas
    where persona.Value.Edad >= 18
    select persona.Value;
```

```
List<Persona> adultos =
    personas.Where(p => p.Value.Edad >= 18)
        .Select(p => p.Value)
        .ToList();
```

Gestión de fechas y horas

Uso de fechas(DATETIME)

Constructor para crear fechas (año, mes y día, o también hora minuto y segundo),

```
// 20 de Marzo de 2022
DateTime fecha1 = new DateTime(2022, 3, 20);
// 20 de Marzo de 2022 a las 22:30:15
DateTime fecha2 = new DateTime(2022, 3, 20, 22, 30, 15);
```

Propiedades de las fechas

Day, **Month**, **Year**, **Hour**, **Minute**, **Second** para acceder a partes de la fecha.

```
DateTime ahora = DateTime.Now;
int mes = ahora.Month;
int segundo = ahora.Second;
```

Sumar partes de fecha

Métodos como **AddDays** y similares para sumar partes a la fecha.

```
DateTime fecha = new DateTime(2022, 3, 20, 22, 30, 15);

// 23 de Marzo de 2022 a las 22:30:15
DateTime fecha2 = fecha.AddDays(3);
// 20 de Marzo de 2022 a las 23:30:15
DateTime unaHoraDespues = fecha.AddHours(1);
```

Períodos de tiempo

Usamos el método **Subtract** para restar fechas y construir un **TimeSpan** con el que ver la diferencia en días o unidades inferiores (horas, minutos…).

```
DateTime ahora = DateTime.Today;
DateTime fechaPasada = new DateTime(2000, 4, 14);
TimeSpan diferencia = ahora.Subtract(fechaPasada);
Console.WriteLine("Han pasado {0} días",
diferencia.Days);
```

Leer fechas de teclado

Hay varias formas.

podemos usar **DateTime.ParseExact.**

-Texto a procesar

-Formato de fecha esperado

-Cultura en que viene la fecha (o **null** para no indicar ninguna)

```
DateTime fecha = DateTime.ParseExact(Console.ReadLine(), "dd/MM/yyyy", null);
```

Mostrar fechas por pantalla

Usamos **ToString** indicando el formato de salida.

```
// The example displays the following output:

//    d Format Specifier     de-DE Culture                 31.10.2008
//    d Format Specifier     en-US Culture                 10/31/2008
//    d Format Specifier     es-ES Culture                 31/10/2008
//    d Format Specifier     fr-FR Culture                 31/10/2008
//
//    D Format Specifier     de-DE Culture          Freitag, 31. Oktober 2008
//    D Format Specifier     en-US Culture          Friday, October 31, 2008
//    D Format Specifier     es-ES Culture          viernes, 31 de octubre de 2008
//    D Format Specifier     fr-FR Culture          vendredi 31 octobre 2008
//
//    f Format Specifier     de-DE Culture       Freitag, 31. Oktober 2008 17:04
//    f Format Specifier     en-US Culture       Friday, October 31, 2008 5:04 PM
//    f Format Specifier     es-ES Culture    viernes, 31 de octubre de 2008 17:04
//    f Format Specifier     fr-FR Culture       vendredi 31 octobre 2008 17:04
//
//    F Format Specifier     de-DE Culture      Freitag, 31. Oktober 2008 17:04:32
//    F Format Specifier     en-US Culture      Friday, October 31, 2008 5:04:32 PM
//    F Format Specifier     es-ES Culture  viernes, 31 de octubre de 2008 17:04:32
//    F Format Specifier     fr-FR Culture      vendredi 31 octobre 2008 17:04:32
//
//    g Format Specifier     de-DE Culture               31.10.2008 17:04
//    g Format Specifier     en-US Culture               10/31/2008 5:04 PM
//    g Format Specifier     es-ES Culture               31/10/2008 17:04
//    g Format Specifier     fr-FR Culture               31/10/2008 17:04
//
//    G Format Specifier     de-DE Culture               31.10.2008 17:04:32
//    G Format Specifier     en-US Culture               10/31/2008 5:04:32 PM
//    G Format Specifier     es-ES Culture               31/10/2008 17:04:32
//    G Format Specifier     fr-FR Culture               31/10/2008 17:04:32
//
//    m Format Specifier     de-DE Culture                    31. Oktober
//    m Format Specifier     en-US Culture                    October 31
//    m Format Specifier     es-ES Culture                    31 de octubre
//    m Format Specifier     fr-FR Culture                    31 octobre
//
//    o Format Specifier     de-DE Culture          2008-10-31T17:04:32.0000000
//    o Format Specifier     en-US Culture          2008-10-31T17:04:32.0000000
//    o Format Specifier     es-ES Culture          2008-10-31T17:04:32.0000000
//    o Format Specifier     fr-FR Culture          2008-10-31T17:04:32.0000000
//
```

```
//    r Format Specifier     de-DE Culture         Fri, 31 Oct 2008 17:04:32 GMT
//    r Format Specifier     en-US Culture         Fri, 31 Oct 2008 17:04:32 GMT
//    r Format Specifier     es-ES Culture         Fri, 31 Oct 2008 17:04:32 GMT
//    r Format Specifier     fr-FR Culture         Fri, 31 Oct 2008 17:04:32 GMT
//
//    s Format Specifier     de-DE Culture              2008-10-31T17:04:32
//    s Format Specifier     en-US Culture              2008-10-31T17:04:32
//    s Format Specifier     es-ES Culture              2008-10-31T17:04:32
//    s Format Specifier     fr-FR Culture              2008-10-31T17:04:32
//
//    t Format Specifier     de-DE Culture                   17:04
//    t Format Specifier     en-US Culture                   5:04 PM
//    t Format Specifier     es-ES Culture                   17:04
//    t Format Specifier     fr-FR Culture                   17:04
//
//    T Format Specifier     de-DE Culture                   17:04:32
//    T Format Specifier     en-US Culture                   5:04:32 PM
//    T Format Specifier     es-ES Culture                   17:04:32
//    T Format Specifier     fr-FR Culture                   17:04:32
//
//    u Format Specifier     de-DE Culture              2008-10-31 17:04:32Z
//    u Format Specifier     en-US Culture              2008-10-31 17:04:32Z
//    u Format Specifier     es-ES Culture              2008-10-31 17:04:32Z
//    u Format Specifier     fr-FR Culture              2008-10-31 17:04:32Z
//
//    U Format Specifier     de-DE Culture      Freitag, 31. Oktober 2008 09:04:32
//    U Format Specifier     en-US Culture      Friday, October 31, 2008 9:04:32 AM
//    U Format Specifier     es-ES Culture      viernes, 31 de octubre de 2008 9:04:32
//    U Format Specifier     fr-FR Culture      vendredi 31 octobre 2008 09:04:32
//
//    Y Format Specifier     de-DE Culture              Oktober 2008
//    Y Format Specifier     en-US Culture              October 2008
//    Y Format Specifier     es-ES Culture              octubre de 2008
//    Y Format Specifier     fr-FR Culture              octobre 2008
```

```
DateTime ahora = DateTime.Now;
Console.WriteLine(ahora.ToString("d")); // dd/MM/yyyy
```

Expresiones regulares

Permiten detectar si un texto cumple un determinado patrón.

Usaremos la clase **Regex** del espacio **System.Text.RegularExpressions**

-Construimos la expresión a comprobar.

-Usamos el método **IsMatch** de la expresión para ver si encaja en un texto determinado.

```
using System.Text.RegularExpressions;

...

Regex expresion = new Regex("^[0-9]+$");
if (expresion.IsMatch(Console.ReadLine()))
{
    Console.WriteLine("Es un número entero");
}
```

Símbolos básicos

Símbolo	Significado
x	Carácter 'x'
[abc]	Caracteres 'a', 'b' o 'c'
[^abc]	Cualquier cosa menos los caracteres 'a', 'b' o 'c'
[a-zA-Z]	Rango de 'a' a 'z' o de 'A' a 'Z'
[a-z]	Rango de 'a' a 'z'
.	Cualquier carácter
^	Inicio de expresión (no puede haber nada antes)
$	Fin de expresión (no puede haber nada después)
\d	Dígito de 0 a 9
\D	Cualquier cosa menos un dígito
\s	Espaciado (espacio, tabulación, nueva línea...)
\S	Cualquier cosa menos espaciado
\w	Alfanumérico (letra, número o subrayado)
\W	Cualquier cosa menos alfanumérico

Cardinalidad

Símbolo	Significado
x?	x aparece 0 o 1 vez
x+	x aparece 1 o más veces
x*	x aparece 0 o más veces
x{n}	x aparece n veces
x{n,}	x aparece al menos n veces
x{n, m}	x aparece entre n y m veces (inclusive)

Ejemplo

```
// Texto compuesto de 4 dígitos
Regex exp1 = new Regex("^\\d{4}$");

// DNI compuesto por 8 dígitos y letra mayúscula
Regex exp2 = new Regex("^\\d{8}[A-Z]$");
```

Acceso a bases de datos relacionales

Uso de bases de datos

Conectaremos desde aplicaciones **C#** a bases de datos **SQLite**.

De forma similar se puede conectar a otros **SGBD** más potentes, como **MySQL** u **Oracle**.

SQLITE

SGBD de pequeño tamaño.

Tiene la ventaja de que se puede distribuir conjuntamente con el proyecto, sin necesitar ningún software externo.

Se puede instalar haciendo clic derecho en Dependencias e instalando como un paquete **NuGet**.

Debemos buscar **System.Data.SQLite**.

Creación de la base de datos

Usamos un objeto **SQLiteConnection** para crear el archivo de la base de datos.

Una vez establecida la conexión, usamos el método **Open** para abrirla.

Nos guardamos ese objeto como un atributo para usarlo más adelante en el programa y hacer operaciones contra la base de datos.

```
class PruebaSQLite
{
   private SQLiteConnection conexion;

   public void CrearConexion()
   {
      if (!File.Exists("basedatos.sqlite"))
      {
         conexion = new SQLiteConnection
         ("DataSource=basedatos.sqlite;Version=3;New=True;Compress=True;");
         conexion.Open();
      }
      else
      {
         conexion = new SQLiteConnection
            ("Data Source=basedatos.sqlite;Version=3;New=False;Compress=True;");
            conexion.Open();
      }
   }
}
```

Creación de las tablas

A diferencia de otros **SGBD**, las tablas deberemos crearlas a mano con instrucciones **SQL**.

Ejecutaremos estas instrucciones sólo la primera vez (cuando creamos la base de datos).

```
class PruebaSQLite
{
   private SQLiteConnection conexion;

   public void CrearConexion()
   {
      if (!File.Exists("basedatos.sqlite"))
      {
         conexion = new SQLiteConnection
("DataSource=basedatos.sqlite;Version=3;New=True;Com
press=True;");
         conexion.Open();
         CrearTablas();
      }
      else
      {
         // Este código no cambia
         ...
      }
   }
```

```
    public void CrearTablas()
    {
        string creacion = "CREATE TABLE personas " +
"(id INTEGER PRIMARY KEY AUTOINCREMENT, " +
"nombre VARCHAR(200) NOT  NULL, " +  "edad
INTEGER NOT NULL);";

        SQLiteCommand cmd = new
SQLiteCommand(creacion, conexion);
        cmd.ExecuteNonQuery();
    }
}
```

Operaciones sobre la base de datos

Seguimos un patrón similar a la creación de tablas, pero con operaciones **INSERT**, **UPDATE**, **DELETE** o **SELECT**.

En el caso de **INSERT**, **UPDATE** o **DELETE**, ejecutamos una **non query**, y obtendremos como resultado el número de filas afectadas por la operación.

En el caso de **SELECT**, obtendremos un **SQLiteDataReader**, y con su método **Read** iremos de registro en registro, consultando sus campos.

Inserción

```
string insercion = "INSERT INTO personas (nombre, edad) VALUES ('Nacho', 44)";
SQLiteCommand cmd = new SQLiteCommand(insercion, conexion);
int cantidad = cmd.ExecuteNonQuery();
if (cantidad < 1)
    Console.WriteLine("No se ha podido insertar");
```

Búsqueda

```
string consulta = "SELECT * FROM personas WHERE edad > 20";
SQLiteCommand cmd = new SQLiteCommand(consulta, conexion);
SQLiteDataReader resultados = cmd.ExecuteReader();
while (resultados.Read())
{
    // Campo 0 = id, campo 1 = nombre, campo 2 = edad
    string nombre = Convert.ToString(resultados[1]);
    int edad = Convert.ToInt32(resultados[2]);
    Console.WriteLine("{0}: {1} años", nombre, edad);
}
```

Cerrar conexión

Usamos el método **Close** de la conexión.

Podemos cerrar al final del programa, o tras cada operación (mejor esto en programas largos o con muchos accesos).

```
conexion.Close();
```

Operaciones parametrizadas

Cuando queremos pasar datos variables a una operación **SQL**, tenemos dos opciones:

Concatenar esos datos en medio de la instrucción.

El código puede resultar algo confuso en ocasiones

Dejar definidas unas marcas y luego reemplazarlas por los datos reales.

Sentencias conocidas como **prepared statements.**

Especificamos luego el valor de cada dato con **AddWithValue**.

```
string insercion = "INSERT INTO videojuegos (titulo, precio)" +
" VALUES (@titulo, @precio)";
SQLiteCommand cmd = new SQLiteCommand(insercion, conexion);
cmd.Parameters.AddWithValue("@titulo", titulo);
cmd.Parameters.AddWithValue("@precio", precio);
cmd.Prepare();
int cantidad = cmd.ExecuteNonQuery();
if (cantidad < 1)
    Console.WriteLine("No se ha podido insertar");
```

Bases de datos y objetos

En ocasiones nos puede interesar gestionar los registros de la base de datos como colecciones de objetos, en lugar de como simples **arrays** de datos sueltos.

Deberemos crear la(s) clase(s) que encapsule(n) la información de cada tabla.

A esa clase le podemos pasar la conexión a la base de datos por si queremos hacer alguna operación de actualización.

```
class Persona
{
    private int id;
    private string nombre;
    private int edad;

    public int Id
    {
        get { return id; }
    }

    public string Nombre
    {
        get { return nombre; }
        set { nombre = value; }
    }

    public int Edad
    {
        get { return edad; }
        set { edad = value; }
    }

    public Persona(string nombre, int edad)
    {
        this.nombre = nombre;
        this.edad = edad;
    }

    public Persona(int id, string nombre, int edad)
    {
        this.id = id;
        this.nombre = nombre;
        this.edad = edad;
    }

    public override string ToString()
    {
        return id + ": " + nombre + ", " + edad + " años.";
    }
```

```
    // Métodos adicionales para gestión del objeto en la base de datos

    public bool Insertar(SQLiteConnection conexion)
    {
        string insercion = "INSERT INTO personas (nombre, edad)" +
            " VALUES (@nombre, @edad)";
        SQLiteCommand cmd = new SQLiteCommand(insercion, conexion);
        cmd.Parameters.AddWithValue("@nombre", nombre);
        cmd.Parameters.AddWithValue("@edad", edad);
        cmd.Prepare();
        int cantidad = cmd.ExecuteNonQuery();
        return cantidad == 1;
    }

    public bool Borrar(SQLiteConnection conexion)
    {
        string borrado = "DELETE FROM personas WHERE id = @id";
        SQLiteCommand cmd = new SQLiteCommand(borrado, conexion);
        cmd.Parameters.AddWithValue("@id", id);
        cmd.Prepare();
        int cantidad = cmd.ExecuteNonQuery();
        return cantidad == 1;
    }

    public bool Actualizar(SQLiteConnection conexion)
    {
        string actualizacion = "UPDATE personas SET nombre = @nombre, " +
            "edad = @edad WHERE id = @id";
        SQLiteCommand cmd = new SQLiteCommand(actualizacion,
conexion);
        cmd.Parameters.AddWithValue("@nombre", nombre);
        cmd.Parameters.AddWithValue("@edad", edad);
        cmd.Parameters.AddWithValue("@id", id);
        cmd.Prepare();
        int cantidad = cmd.ExecuteNonQuery();
        return cantidad == 1;
    }
```

```
    static List<Persona> Listar(SQLiteConnection conexion)
    {
        List<Persona> resultado = new List<Persona>();
        string consulta = "SELECT * FROM personas";
        SQLiteCommand cmd = new SQLiteCommand(consulta, conexion);
        SQLiteDataReader resultados = cmd.ExecuteReader();
        while (resultados.Read())
        {
            int id = Convert.ToInt32(resultados[0]);
            string nombre = Convert.ToString(resultados[1]);
            int edad = Convert.ToInt32(resultados[2]);
            resultado.Add(new Persona(id, nombre, edad));
        }
        return resultado;
    }
}

class Main
{
    static void Main()
    {
        SQLiteConnection conexion = ... // Obtenemos la conexion

        List<Persona> personas = Persona.Listar(conexion);

        Persona nueva = new Persona("John Doe", 40);
        nueva.Insertar(conexion);

        personas = Persona.Listar();
        personas[1].Nombre = "Otro nombre";
        personas[1].Actualizar(conexion);
        personas[0].Borrar(conexion);
        personas.RemoveAt(0);

        foreach (Persona p in personas)
            Console.WriteLine(p);

        conexion.Close();
    }
}
```

Otros aspectos del lenguaje

Obtener información del sistema

Dentro del propio espacio **System.Diagnostics**, la clase Environment permite obtener información del sistema:

Environment.CurrentDirectory obtiene la ruta del directorio actual.

Environment.UserName el nombre del usuario actual.

Environment.OSVersion la versión del sistema operativo.

Invocar procesos externos

Usamos el método **Process.Start** de la clase **Process**

Espacio **System.Diagnostics**.

Opcionalmente, podemos llamar al método **WaitForExit** tras invocarlo, para que el programa espere a que se cierre.

Si ponemos sólo el nombre del ejecutable, el programa debe estar en el **PATH** del sistema operativo.

Alternativa: poner la ruta completa hasta el ejecutable.

```
using System.Diagnostics;

...

Process proc = Process.Start("notepad.exe");
proc.WaitForExit();
Console.WriteLine("Notepad finalizado");
```

Parámetros de entrada

i un proceso necesita parámetros de entrada, se los podemos pasar todos juntos como segundo parámetro del método **Start.**

```
Process proc = Process.Start("copy", "fichero1.txt fichero2.txt");
```

Resultado de la ejecución

Podemos esperar a que un proceso termine y recoger su estado con la propiedad **ExitCode**.

Si es 0 todo ha ido correctamente.

```
Process proc = Process.Start("copy", "fichero1.txt fichero2.txt");
proc.WaitForExit();
if (proc.ExitCode != 0)
    Console.WriteLine("Error ejecutando proceso");
```

Otras opciones

Podemos controlar el estado de la ventana del proceso, y minimizarla o incluso ocultarla, usando **ProcessStartInfo**

Mismos parámetros que **Process.Start** (si queremos pasar argumentos, usamos un segundo parámetro).

```
ProcessStartInfo pInfo = new ProcessStartInfo("notepad.exe");
pInfo.WindowStyle = ProcessWindowStyle.Minimized;
Process proc = Process.Start(pInfo);
```