

Tema Colecciones en C#.LINQ

Contenido

1	Introducción	2
1.1	Clases del espacio de nombres System.Collections.Generic	2
1.2	Iteradores (C#)	4
1.3	Colecciones	10
1.3.1	List<T> Clase	10
1.3.2	SortedList	15
1.3.3	Dictionary	18
1.3.4	Ejercicion	19
1.3.5	La Colección Queue<T>	19
1.3.6	Ejercicio. Notas Cornell	21
1.3.7	Stack<T>	22
1.3.8	Ejercicio. Notas Cornell	23
2	Creando datos aleatorios para Colecciones	24
2.1	Clases para generar colecciones aleatorias	28
2.1.1	Ejercicio	28
2.1.2	Ejercicios	32
3	LINQ	32
3.1	El interfaz IQueryable	34
3.2	Partes de una consulta LINQ	35
3.3	Consultas básicas LINQ	38
3.3.1	LINQ con sintaxis de query	38
3.3.2	Esta sintaxis es muy parecida a SQL, los conocedores de SQL la manejan fácilmente y parece más legible para ellos.	38
3.3.3	Ejercicios	39
3.3.4	Ejercicios	41
3.3.5	LINQ con sintaxis de método y expresiones lambda.	42
3.3.6	Ejercicios	46
3.4	LINQ y XML	47
3.5	Ejercicio	52
3.6	Consultas LINQ en xml	53
4	Entity Framework	60
4.1	Que se un ORM (Objet Relational Model)	60
4.1.1	Mapeadores	61
4.2	Paquetes NuGet	64
4.2.1	El flujo de paquetes entre creadores, hosts y consumidores	65
4.2.2	Compatibilidad de destino de paquetes	66
4.2.3	¿Qué más hace NuGet?	67
4.2.4	Herramientas para usar NuGet.	68
4.3	Instalación de Entity Framework en los proyectos Visual Studio. Primer proyecto	69
4.3.1	Modelo de clases	70
4.3.2	Modelo Entity. Code First. Creación de la base de datos con la API Fluent. ..	75
4.3.3	Ejercicios	88
4.3.4	Modelos mas avanzados. Anotaciones.	88
4.3.5	Ejercicio Notas Cornell.	91

4.3.6	Ejercicio	92
4.3.7	Añadiendo clases al modelo.	94
4.3.8	Varias clases relacionadas en nuestro modelo. Entity Framework.....	99
5	Proyecto Final de la Unidad.	111

1 Introducción

Para muchas aplicaciones, puede que desee crear y administrar grupos de objetos relacionados. Existen dos formas de agrupar objetos: mediante la creación de matrices de objetos y con la creación de colecciones de objetos.

Las matrices son muy útiles para crear y trabajar con un número fijo de objetos fuertemente tipados. Para obtener información sobre las matrices, vea *Matrices*.

Las colecciones proporcionan una manera más flexible de trabajar con grupos de objetos. A diferencia de las matrices, el grupo de objetos con el que trabaja puede aumentar y reducirse de manera dinámica a medida que cambian las necesidades de la aplicación. Para algunas colecciones, puede asignar una clave a cualquier objeto que incluya en la colección para, de este modo, recuperar rápidamente el objeto con la clave.

Una colección es una clase, por lo que debe declarar una instancia de la clase para poder agregar elementos a dicha colección.

Si la colección contiene elementos de un solo tipo de datos, puede usar una de las clases del espacio de nombres *System.Collections.Generic*. Una colección genérica cumple la seguridad de tipos para que ningún otro tipo de datos se pueda agregar a ella. Cuando recupera un elemento de una colección genérica, no tiene que determinar su tipo de datos ni convertirlo.

1.1 Clases del espacio de nombres *System.Collections.Generic*

Puede crear una colección genérica mediante una de las clases del espacio de nombres *System.Collections.Generic*. Una colección genérica es útil cuando todos los elementos de la colección tienen el mismo tipo. Una colección genérica exige el establecimiento de fuertes tipos al permitir agregar solo los tipos de datos deseados.

En la tabla siguiente se enumeran algunas de las clases usadas con frecuencia del espacio de nombres *System.Collections.Generic*:

CLASES *SYSTEM.COLLECTIONS.GENERIC*

Dictionary<TKey,TValue>	Representa una colección de pares de clave y valor que se organizan según la clave.
List<T>	Representa una lista de objetos a los que puede tener acceso el índice. Proporciona métodos para buscar, ordenar y modificar listas.
Queue<T>	Representa una colección de objetos de primeras entradas, primeras salidas (FIFO).
SortedList<TKey,TValue>	Representa una colección de pares clave-valor que se ordenan por claves según la implementación de IComparer<T> asociada.
Stack<T>	Representa una colección de objetos de últimas entradas, primeras salidas (LIFO).

Las siguientes colecciones pertenecen a otro espacio de nombres System.Collections no almacenan los elementos como objetos de tipo específico, sino como objetos del tipo Object.
YA NO SE USAN.

Clase	Descripción
ArrayList	Representa una matriz cuyo tamaño aumenta dinámicamente cuando es necesario.
Hashtable	Representa una colección de pares de clave y valor que se organizan por código hash de la clave.
Queue	Representa una colección de objetos de primeras entradas, primeras salidas (FIFO).
Stack	Representa una colección de objetos de últimas entradas, primeras salidas (LIFO).

1.2 Iteradores (C#)

Un iterador puede usarse para recorrer colecciones como listas y matrices.

Un método iterador o un descriptor de acceso `get` realiza una iteración personalizada en una colección. Un iterador usa la instrucción `yield return` para devolver cada elemento de uno en uno. Cuando se alcanza una instrucción `yield return`, se recuerda la ubicación actual en el código. La ejecución se reinicia desde esa ubicación la próxima vez que se llama a la función del iterador.

Para consumir un método iterador desde código de cliente, se usa una instrucción `foreach` o una consulta de LINQ.

Intefaz `IEnumerable`

Usamos el interfaz `IEnumerable` de Espacio de nombres: `System.Collections`

Expone un enumerador, que admite una iteración simple en una colección no genérica.

```
public interface IEnumerable
```

Devuelve un enumerador que recorre en iteración una colección.

```
public System.Collections.IEnumerator GetEnumerator ();
```

Interfaz `IEnumerable<T>` Espacio de nombres: `System.Collections.Generic`

```
public interface IEnumerable<out T> : System.Collections.IEnumerable
```

Método a implementar

```
public System.Collections.Generic.IEnumerator<out T> GetEnumerator ();
```

En el ejemplo siguiente, la primera iteración del bucle `foreach` hace que continúe la ejecución del método de iterador `IteradorNumeros` hasta que se alcance la primera instrucción

yield return. Esta iteración devuelve un valor de 3, y la ubicación actual del método de iterador se conserva. En la siguiente iteración del bucle, la ejecución del método iterador continúa desde donde se dejó, deteniéndose de nuevo al alcanzar una instrucción yield return. Esta iteración devuelve un valor de 5, y la ubicación actual del método de iterador se vuelve a conservar. El bucle se completa al alcanzar el final del método iterador. Recordar que yield devuelve el control al programa que llamo al método y se queda en el punto tras ejecutar la instrucción.

Vamos a analizar el ejemplo, donde se verá más claramente. En el método IteradorNumeros que devuelve un IEnumerable se devuelven uno a uno los tres números al foreach posterior.

```
public static System.Collections.IEnumerable IteradorNumeros ()
{
    yield return 3;
    yield return 5;
    yield return 8;
}

foreach (int num in IteradorNumeros())
{
    Console.WriteLine(" [{0}] ", num);
}
```

Se puede hacer también con un bucle for. Como veis en secuencia pares construimos un iterador sobre los números pares haciendo yield return en cada par.

```
SecuenciaPares(int primero, int ultimo)
{
    // Yield even numbers in the range.
    for (int numero = primero; numero <= ultimo; numero++)
    {
        if (numero % 2 == 0)
        {
            yield return numero;
        }
    }
}

foreach (int num in SecuenciaPares(0,10))
{
    Console.WriteLine(" [{0}] ", num);
}
```

O con una clase aparte. La siguiente clase contiene un array con los días de la semana que

es recorrido con yield return y devolviendo día a día. La clase implementa el interfaz IEnumerable que obliga a implementar el método GetEnumerator que devuelve un enumerator.

```
public class ClaseConIterador : IEnumerable
{
    private string[] dias = { "Lunes", "Martes", "Miercoles", "Jueves", "Viernes",
    "Sabado", "Domingo" };

    public IEnumerator GetEnumerator()
    {
        for (int i = 0; i < dias.Length; i++)
        {
            yield return dias[i];
        }
    }
}
```

Lo usamos en el programa principal en un foreach igualmente. Declaramos una variable de tipo ClaseConIterador() y recorremos el objeto clase con un foreach

```
ClaseConIterador clase = new ClaseConIterador();
```

```
foreach (string dia in clase)
{
    Console.WriteLine(" {0}| ", dia);
}
```

Ejecución ejemplo:

iteradorNumeros

[3] [5] [8]

SecuenciaPares

[0] [2] [4] [6] [8] [10]

Clase con iterador

Lunes| Martes| Miercoles| Jueves| Viernes| Sabado| Domingo|

Ejemplo Completo

ClaseConIterador.cs

```
using System;
using System.Collections.Generic;
using System.Text;
using System.Collections;
```

```
namespace Iteradores
```

```

{

    public class ClaseConIterador : IEnumerable
    {
        private string[] dias = { "Lunes", "Martes", "Miercoles", "Jueves", "Viernes",
        "Sabado", "Domingo" };

        public IEnumerator GetEnumerator()
        {
            for (int i = 0; i < dias.Length; i++)
            {
                yield return dias[i];
            }
        }
    }
}

```

Program.cs

```

using System;
using System.Collections;

namespace Iteradores
{
    class Program
    {
        public static System.Collections.IEnumerable IteradorNumeros()
        {
            yield return 3;
            yield return 5;
            yield return 8;
        }

        public static System.Collections.Generic.IEnumerable<int>
        SecuenciaPares(int primero, int ultimo)
        {
            // Yield even numbers in the range.
            for (int numero = primero; numero <= ultimo; numero++)
            {
                if (numero % 2 == 0)
                {
                    yield return numero;
                }
            }
        }

        static void Main(string[] args)
        {
            Console.WriteLine("\n iteradorNumeros");

            foreach (int num in IteradorNumeros())
            {
                Console.Write(" [{0}] ", num);
            }
            Console.WriteLine("\n SecuenciaPares");
            foreach (int num in SecuenciaPares(0,10))

```

```

    {
        Console.WriteLine(" [{0}] ", num);
    }

    Console.WriteLine("\n Clase con iterador");
    ClaseConIterador clase = new ClaseConIterador();

    foreach (string dia in clase)
    {
        Console.WriteLine(" {0}| ", dia);
    }
}
}
}

```

El interfaz Enumerator

Espacio de nombres:
System.Collections

Admite una iteración simple a través de una colección no genérica.

```
public interface IEnumerator
```

Método a implementar

```
public bool MoveNext ();
```

Propiedades a implementar el get

```
public T Current { get; }
```

Es true si el enumerador avanzó con éxito hasta el siguiente elemento; es false si el enumerador alcanzó el final de la colección.

IEnumerator<T> Interfaz

Espacio de nombres:

[System.Collections.Generic](#)

Admite una iteración simple en una colección genérica.

```
public interface IEnumerator<out T> : IDisposable, System.Collections.IEnumerator
```

En el siguiente ejemplo podéis ver una implementación del interfaz. Mantenemos un índice en index con valor inicial cero.

```
private int index = 0;
```


La propiedad Current devuelve el día en el índice actual en el array.

El Método moveNext() devuelve false si no hemos llegado al final del array.

El método reset() pone el index a cero.

```
using System;
using System.Collections;
using System.Collections.Generic;
using System.Text;

namespace EnumeratorEnumerable
{
    class DiasDeLaSemanaEnumerator : IEnumerator<String>
    {
        private string[] dias = { "Lunes", "Martes", "Miercoles", "Jueves", "Viernes",
        "Sabado", "Domingo" };

        private int index = 0;

        public string Current => dias[index];

        object IEnumerator.Current => dias[index];

        public void Dispose()
        {
            throw new NotImplementedException();
        }

        public bool MoveNext()
        {
            index++;
            if (index < dias.Length)
            {
                return true;
            }
            else
            {
                return false;
            }
        }

        public void Reset()
        {
            index = 0;
        }
    }
}
```

En el programa principal recorreremos el enumerator con un do while llamando a MoveNext. Usamos la propiedad Current para recoger el valor.

```
using System;

namespace EnumeratorEnumerable
{
    class Program
```

```

{
    static void Main(string[] args)
    {
        DiasDeLaSemanaEnumerator enumer = new DiasDeLaSemanaEnumerator();
        Console.WriteLine("Dias de la semana");
        do
        {
            Console.Write(" {0} | ", enumer.Current);

        } while (enumer.MoveNext());

        enumer.Reset();
    }
}

```

1.3 Colecciones

Como ya hemos visto las colecciones para muchas aplicaciones, puede que desee crear y administrar grupos de objetos relacionados. Existen dos formas de agrupar objetos: mediante la creación de matrices de objetos y con la creación de colecciones de objetos. Vamos a ahora a ver colección a colección.

1.3.1 List<T> Clase

Representa una lista de objetos fuertemente tipados a la que se puede obtener acceso por índice. Proporciona métodos para buscar, ordenar y manipular listas. Implementa IEnumerable con lo que se puede recorrer con un foreach, además como veremos más adelante es requisito para que se pueda realizar consultas LINQ sobre objetos, que sus clases implementen IEnumerable.

Espacio de nombres: System.Collections.Generic

Interfaces implementados

```

public class List<T> : System.Collections.Generic.ICollection<T>,
System.Collections.Generic.IEnumerable<T>, System.Collections.Generic.IList<T>,
System.Collections.Generic.IReadOnlyCollection<T>,
System.Collections.Generic.IReadOnlyList<T>, System.Collections.IList

```

Propiedades:

- Capacity: capacidad máxima de la colección.
- Count: numero de elementos de la colección

- `Item[index]`: para acceder a los elementos de la colección

Métodos:

Add : Añade un elemento
 AddRange: aumenta el rango
 AsReadOnly: la marca como solo lectura
 BinarySearch: permite búsqueda binaria.
 Clear: deja sin elementos
 Contains: comprueba si contiene
 ConvertAll: convierte todos los elementos a otro tipo
 CopyTo: copia la colección
 Exists: comprueba si un elemento existe
 Find: busca el primero
 FindAll: busca todos
 FindIndex
 FindLast
 FindLastIndex
 ForEach
 GetEnumerator: devuelve el enumerator
 GetRange: crea una copia de una sublista con el rango indicado
 IndexOf: devuelve el índice del elemento
 Insert: inserta en la posición indicada
 InsertRange
 LastIndexOf
 Remove
 RemoveAll
 RemoveAt
 RemoveRange
 Reverse: le da la vuelta a la colección
 Sort: ordena la colección conforme al CompareTo del Tipo
 ToArray: transforma array
 TrimExcess: cambia el rango a los elementos sobrantes
 TrueForAll

Vamos a ver unos cuantos ejemplos:

Creamos una lista con un `new` y añadimos un rango 10, tendrá como máximo 10 elementos

```
List<int> numerosPrimos = new List<int>(10);
```

Añadimos cuatro elementos con `Add`

```
numerosPrimos.Add(1);
numerosPrimos.Add(3);
numerosPrimos.Add(5);
numerosPrimos.Add(7);
Console.WriteLine("Lista de primos recorrida con un bucle");
```

Recorreremos la lista con un bucle `foreach`, ya que implementa `IEnumerable` lo podemos hacer.

```
foreach (int num in numerosPrimos)
{
```

```

        Console.WriteLine("|{0}|", num);
    }

    Console.WriteLine("\n Lista de ciudades recorrida con el método foreach");

```

Podemos crear una lista sin un rango predeterminado.

```

var ciudades = new List<string>();

```

Añadir elementos con Add

```

    ciudades.Add("New York");
    ciudades.Add("London");
    ciudades.Add("Mumbai");

```

Insertar en el índice indicado

```

    ciudades.Insert(3, "Chicago");

```

Añadir nulos

```

    ciudades.Add(null); // Podemos añadir nulos a las listas

```

Recorrer la lista con el método ForEach que recibe un delegado de tipo Action

```

    ciudades.ForEach((s) => Console.WriteLine(s));

```

Recoger una sublista que empiece en el índice indicado y con los elementos indicados

```

    Console.WriteLine("sub Lista de ciudades desde el indice 1 tres elementos");
    List<string> sublista = ciudades.GetRange(1, 3);

    sublista.ForEach((s) => Console.WriteLine(s));

```

Borra en el índice indicado en este caso 3.

```

    Console.WriteLine("Quitamos el elemento de indice 3 con removeAt");
    ciudades.RemoveAt(3);

    ciudades.ForEach((s) => Console.WriteLine(s));

    Console.WriteLine("borramos el null");

```

Quitar un elemento por su valor

```

    ciudades.Remove(null);

    ciudades.ForEach((s) => Console.WriteLine(s));

```

Borrar la lista entera

```

    ciudades.Clear();

    Console.WriteLine("limpiamos la lista con clear e intentamos recorrer pero esta vacia");

    ciudades.ForEach((s) => Console.WriteLine(s));

```

Crear una lista con el inicializador llaves.

```
var grandesCiudades = new List<string>()
{
    "New York",
    "London",
    "Mumbai",
    "Chicago"
};
```

Buscar usando el método find que recibe un delegado Predicate

```
String nombreCiudad = grandesCiudades.Find(s=> s.StartsWith("L"));
Console.WriteLine("Primera ciudad que empieza por L {0}", nombreCiudad);
```

Buscar una lista de ciudades que contengan o dentro de la lista grandesCiudades

```
Console.WriteLine("Buscamos la ciudad que contienen o en la lista grandesCiudades");

List<String> ciudadesQueContieneno = grandesCiudades.FindAll(s =>
s.Contains("o"));
ciudadesQueContieneno.ForEach((s) => Console.WriteLine(s));
```

Ejecución completa

Lista de primos recorrida con un bucle

|1|3|5|7

Lista de ciudades recorrida con el método foreach

New York

London

Mumbai

Chicago

sub Lista de ciudades desde el indice 1 tres elementos

London

Mumbai

Chicago

Quitamos el elemento de indice 3 con removeAt

New York

London

Mumbai

borramos el null

New York

London

Mumbai

limpiamos la lista con clear e intentamos recorrer pero esta vacia

Primera ciudad que empieza por L London

Buscamos las ciudades que contienen o en la lista grandes ciudades

New York

London
Chicago

Ejemplo completo:

```
using System;
using System.Collections.Generic;

namespace Listas
{
    class Program
    {
        static void Main(string[] args)
        {
            List<int> numerosPrimos = new List<int>(10);
            numerosPrimos.Add(1);
            numerosPrimos.Add(3);
            numerosPrimos.Add(5);
            numerosPrimos.Add(7);
            Console.WriteLine("Lista de primos recorrida con un bucle");
            foreach (int num in numerosPrimos)
            {
                Console.WriteLine($"{num}");
            }

            Console.WriteLine("Lista de ciudades recorrida con el método foreach");

            var ciudades = new List<string>();
            ciudades.Add("New York");
            ciudades.Add("London");
            ciudades.Add("Mumbai");
            ciudades.Insert(3, "Chicago");
            ciudades.Add(null); // Podemos añadir nulos a las listas
            ciudades.ForEach((s) => Console.WriteLine(s));

            Console.WriteLine("sub Lista de ciudades desde el indice 1 tres elementos");
            List<string> sublista = ciudades.GetRange(1, 3);

            sublista.ForEach((s) => Console.WriteLine(s));

            Console.WriteLine("Quitamos el elemento de indice 3 con removeAt");
            ciudades.RemoveAt(3);

            ciudades.ForEach((s) => Console.WriteLine(s));

            Console.WriteLine("borramos el null");

            ciudades.Remove(null);

            ciudades.ForEach((s) => Console.WriteLine(s));

            ciudades.Clear();

            Console.WriteLine("limpiamos la lista con clear e intentamos recorrer pero esta vacia");

            ciudades.ForEach((s) => Console.WriteLine(s));
        }
    }
}
```

```

var grandesCiudades = new List<string>()
{
    "New York",
    "London",
    "Mumbai",
    "Chicago"
};

String nombreCiudad = grandesCiudades.Find(s=> s.StartsWith("L"));

Console.WriteLine("Primera ciudad que empieza por L {0}", nombreCiudad);

Console.WriteLine("Buscamos la ciudad que contienen o en la lista
grandesCiudades");
List<String> ciudadesQueContieneno = grandesCiudades.FindAll(s =>
s.Contains("o"));
ciudadesQueContieneno.ForEach((s) => Console.WriteLine(s));

    }
}
}

```

1.3.2 SortedList

Representa una colección de pares clave-valor que se ordenan por claves según la implementación de `IComparer<T>` asociada. Cada par se guarda en un objeto de tipo `KeyValuePair<T,U>`.

Propiedades

Capacity

Comparer: Obtiene el `IComparer<T>` para la lista ordenada.

Count

Item[]: nos permite acceder al par clave valor por valor de clave.

Keys: devuelve una colección con las claves, tipo `ICollection<T>`.

Values: devuelve una colección con los valores `ICollection<U>`.

Métodos

Add: añade un par clave valor

Clear: deja vacía la colección

ContainsKey: comprueba si la clave está en la colección

ContainsValue: comprueba si el valor está en la colección

GetEnumerator

IndexOfKey: devuelve el índice para la clave pasada como parámetro

IndexOfValue: devuelve el valor para la clave pasada como parámetro

Remove

RemoveAt

TrimExcess

TryGetValue: Obtiene el valor asociado a la clave especificada.

Declaramos un SortedList con el tipo para la clave y para el valor. Podríamos darle rango para fijar el tamaño.

```
SortedList<string, string> abrirCon =  
    new SortedList<string, string>();
```

Añadimos elementos con Add. C# los ordena por clave. Lo veremos tras el recorrido.

```
abrirCon.Add("txt", "notepad.exe");  
abrirCon.Add("bmp", "paint.exe");  
abrirCon.Add("dib", "paint.exe");  
abrirCon.Add("rtf", "wordpad.exe");  
  
Console.WriteLine("Coleccion completa con clave valor");  
foreach (KeyValuePair<string, string> element in abrirCon)  
{  
    Console.WriteLine("Elemento: {0}", element);  
}
```

Los elementos esta ordenados, lo podemos ver tras la ejecución del recorrido.

Elemento: [bmp, paint.exe]

Elemento: [dib, paint.exe]

Elemento: [rtf, wordpad.exe]

Elemento: [txt, notepad.exe]

Podemos buscar una clave con el método TryGetValue. El método devolverá true si la clave está presente. En el parámetro de salida marcado con el modificador out coloca el valor asociado a esa clave.

```
Console.WriteLine("\nBusqueda de valor a partir de clave");  
string s;  
if (abrirCon.TryGetValue("txt", out s))  
{  
    Console.WriteLine("Encontrada la clave txt con valor {0}", s);  
}
```


Podemos recorrer el conjunto de claves con el atributo Keys que es de tipo colección e implementa el **interfaz IList**. Como IEnumerable puedes recorrerlo con un foreach. Podríamos hacer lo mismo con los valores con la propiedad values.

```
Console.WriteLine("Recorriendo la coleccion de valores con el atributo Keys");
IList<string> keys = abrirCon.Keys;
foreach( string valor in keys)
{
    Console.Write("| {0}", valor);
}
```

Finalmente podemos acceder a un valor por su clave, con la propiedad item.

```
Console.WriteLine("\nAccediendo a un valor por clave {0} ", abrirCon["rtf"]);
```

Ejecución completa:

Colección completa con clave valor

Elemento: [bmp, paint.exe]

Elemento: [dib, paint.exe]

Elemento: [rtf, wordpad.exe]

Elemento: [txt, notepad.exe]

Búsqueda de valor a partir de clave

Encontrada la clave txt con valor notepad.exe

Recorriendo la coleccion de valores con el atributo Keys

| bmp| dib| rtf| txt

Accediendo a un valor por clave wordpad.exe

Ejemplo completo

```
using System;
using System.Collections.Generic;

namespace SortedList
{
    class Program
    {
        static void Main(string[] args)
        {
            SortedList<string, string> abrirCon =
                new SortedList<string, string>();

            abrirCon.Add("txt", "notepad.exe");
            abrirCon.Add("bmp", "paint.exe");
            abrirCon.Add("dib", "paint.exe");
            abrirCon.Add("rtf", "wordpad.exe");

            Console.WriteLine("Coleccion completa con clave valor");
            foreach (KeyValuePair<string, string> element in abrirCon)
            {
```

```

        Console.WriteLine("Elemento: {0}", element);
    }

    Console.WriteLine("\nBusqueda de valor a partir de clave");
    string s;
    if (abrirCon.TryGetValue("txt", out s))
    {
        Console.WriteLine("Encontrada la clave txt con valor {0}", s);
    }
    Console.WriteLine("Recorriendo la coleccion de valores con el atributo Keys");
    IList<string> keys = abrirCon.Keys;
    foreach (string valor in keys)
    {
        Console.WriteLine("| {0}", valor);
    }

    Console.WriteLine("\nAccediendo a un valor por clave {0} ", abrirCon["rtf"]);
}
}
}

```

1.3.3 Dictionary

Representa una colección de pares claves y valores de tipo clase KeyValuePair. **No esta ordenada como el SortedList.**

Interfaces que implementa:

```

public class Dictionary<TKey,TValue> :
    System.Collections.Generic.ICollection<System.Collections.Generic.KeyValuePair<TKey,TValue>>,
    System.Collections.Generic.IDictionary<TKey,TValue>,
    System.Collections.Generic.IEnumerable<System.Collections.Generic.KeyValuePair<TKey,TValue>>,
    System.Collections.Generic.IReadOnlyCollection<System.Collections.Generic.KeyValuePair<TKey,TValue>>,
    System.Collections.Generic.IReadOnlyDictionary<TKey,TValue>,
    System.Collections.IDictionary, System.Runtime.Serialization.IDeserializationCallback,
    System.Runtime.Serialization.ISerializable

```

Propiedades:

- Comparer
- Count
- Item[]
- Keys

- Values

Métodos:

- Add
- Clear
- ContainsKey
- ContainsValue
- EnsureCapacity: Garantiza que el diccionario puede contener hasta un número especificado de entradas sin más expansión de su almacenamiento de respaldo.
- GetEnumerator
- GetObjectData
- OnDeserialization: implementa la interfaz ISerializable y genera el evento de deserialización cuando esta ha finalizado.
- Remove
- TrimExcess: Establece la capacidad de este diccionario para contener hasta un número especificado de entradas sin más expansión de su almacenamiento de respaldo.
- TryAdd: intenta añadir un par clave valor, comprobando si esta o no. Devuelve true si se ha agregado correctamente. False si no lo ha hecho.
- TryGetValue

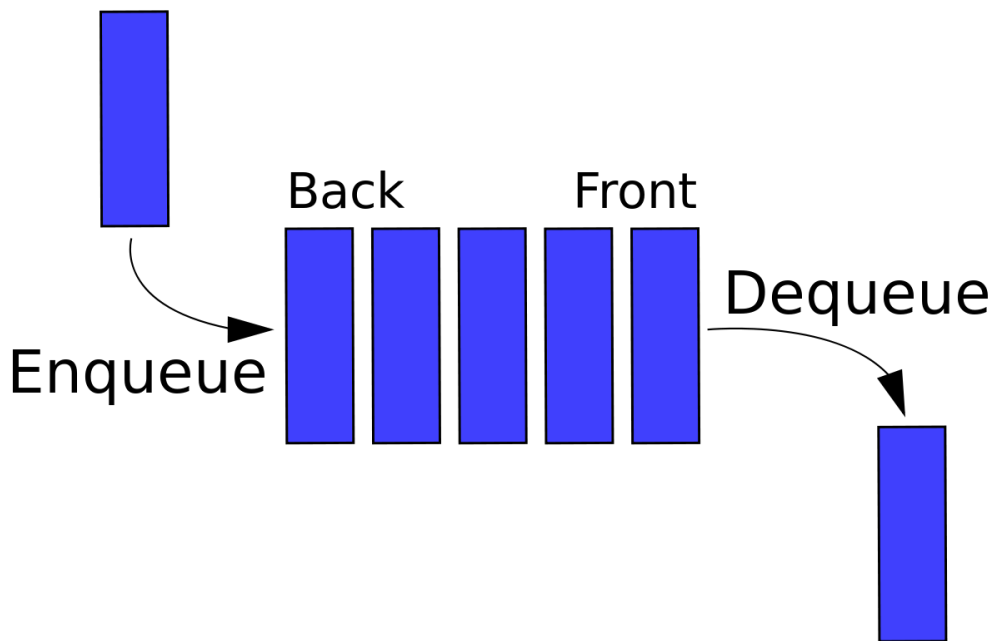
Funciona prácticamente igual que el SortedList, podríamos intercambiar el ejemplo anterior con dictionary y cambiaría básicamente que no hay orden.

1.3.4 Ejercicio

Transformar el ejemplo de SortedList a Dictionary.

1.3.5 La Colección Queue<T>

Representa una colección de objetos de tipo primero en entrar, primero en salir. Cuando realizamos un Dequeue sacamos el elemento de la cola y por tanto se borra. El tipo T especifica el tipo de elementos en la cola.



Herencia

Object -> Queue<T>

Implementacion de interfaces

IEnumerable<T> IReadOnlyCollection<T> ICollection IEnumerable

Propiedades

Count -> contiene el número de elementos en la cola.

Métodos

Clear: limpia la cola

Contains: comprueba si contiene un elemento

CopyTo

Dequeue: Saca un elemento del principio de la cola

Enqueue: Inserta un elemento al final de la cola.

GetEnumerator

Peek: Devuelve un objeto al principio de Queue<T> sin eliminarlo.

ToArray

TrimExcess

TryDequeue: parecido a Dequeue pero devuelve false si no se consigue

TryPeek: parecido a Peek pero devuelve false si no se consigue.

1.3.6 Ejercicio. Notas Cornell

Para el siguiente código en el que usamos colas comentar que hace cada línea tras estudiar y ejecutar el código.

```
using System;
using System.Collections.Generic;

namespace Queue
{
    class Program
    {
        static void Main(string[] args)
        {
            Queue<string> numeros = new Queue<string>();
            numeros.Enqueue("uno");
            numeros.Enqueue("dos");
            numeros.Enqueue("tres");
            numeros.Enqueue("cuatro");
            numeros.Enqueue("cinco");

            foreach (string numero in numeros)
            {
                Console.WriteLine(numero);
            }

            Console.WriteLine("\nSacando de la cola '{0}'", numeros.Dequeue());
            Console.WriteLine("Peek el siguiente item: {0}",
                numeros.Peek());
            Console.WriteLine("Sacando '{0}'", numeros.Dequeue());

            Queue<string> queueCopy = new Queue<string>(numeros.ToArray());

            Console.WriteLine("\nCopia cola transformando a array y usando constructor");
            foreach (string number in queueCopy)
            {
                Console.WriteLine(number);
            }

            Console.WriteLine("\nCopiando con el método CopyTo a array");
            string[] array2 = new string[numeros.Count * 2];
            numeros.CopyTo(array2, numeros.Count);

            // Create a second queue, using the constructor that accepts an
            // IEnumerable(Of T).
            Queue<string> queueCopy2 = new Queue<string>(array2);

            Console.WriteLine("\nContenidos de la segunda copia con nulos y duplicados:");
            foreach (string number in queueCopy2)
            {
                Console.WriteLine(number);
            }

            Console.WriteLine("\nqueueCopy.Contains(\"four\") = {0}",
                queueCopy.Contains("four"));

            Console.WriteLine("\nqueueCopy.Clear()");
            queueCopy.Clear();
        }
    }
}
```

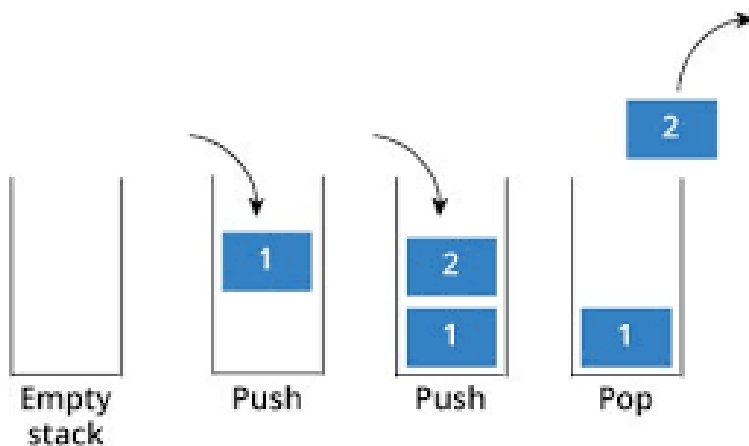
```

    Console.WriteLine("\nqueueCopy.Count = {0}", queueCopy.Count);
}
}
}

```

1.3.7 Stack<T>

Representa una colección LIFO (el último en entrar es el primero en salir) de tamaño variable de instancias del mismo tipo especificado. En este caso el último elemento introducido se pone “encima” de los anteriores (push), en lo que se llama cabecera de la pila. Cuando extraigamos un elemento de la pila, será el primero en salir (pop).



T : Especifica el tipo de elementos de la pila.

Herencia

Object -> Stack<T>

Implementacion de interfaces

IEnumerable<T> IReadOnlyCollection<T> ICollection IEnumerable

Propiedades

Count: devuelve o contiene el numero de elementos de la pila.

Métodos

Clear

Contains

CopyTo

GetEnumerator

Peek: lee el element en la cabecera de la pila
Pop: saca el elemento de la cabecera de la pila
Push: introduce un elemento en la cabecera de la pila.
ToArray
TrimExcess
TryPeek
TryPop

1.3.8 Ejercicio. Notas Cornell

Comentar cada paso que realizamos en el siguiente código después de ejecutarlo

```
using System;
using System.Collections.Generic;

namespace Stack
{
    class Program
    {
        static void Main(string[] args)
        {
            Stack<string> numeros = new Stack<string>();
            numeros.Push("uno");
            numeros.Push("dos");
            numeros.Push("tres");
            numeros.Push("cuatro");
            numeros.Push("cinco");

            foreach (string numero in numeros)
            {
                Console.WriteLine(numero);
            }

            Console.WriteLine("\nSacando '{0}'", numeros.Pop());
            Console.WriteLine("Haciendo peek del elemento superior {0}",
                numeros.Peek());
            Console.WriteLine("Sacando '{0}'", numeros.Pop());

            Stack<string> stack2 = new Stack<string>(numeros.ToArray());

            Console.WriteLine("\nContenidos de la copia:");
            foreach (string number in stack2)
            {
                Console.WriteLine(number);
            }

            string[] array2 = new string[numeros.Count * 2];
            numeros.CopyTo(array2, numeros.Count);

            Stack<string> stack3 = new Stack<string>(array2);

            Console.WriteLine("\nContenidos de la secuencia copia con duplicados y nulos");
            foreach (string number in stack3)
```

```

        {
            Console.WriteLine(number);
        }

        Console.WriteLine("\nstack2.Contains(\"four\") = {0}",
            stack2.Contains("four"));

        Console.WriteLine("\nstack2.Clear()");
        stack2.Clear();
        Console.WriteLine("\nstack2.Count = {0}", stack2.Count);
    }
}

```

2 Creando datos aleatorios para Colecciones

Vamos a proporcionar el modelo anterior de empleados añadiendo el atributo dirección a empleado, y generando campos aleatorios usando random, Arrays y colecciones con el fin de generar colecciones de empleados aleatorios. Nos servirá de base para realizar algún ejercicio de colecciones y enlazar con consultas LINQ.

El interfaz

La clase Persona

```

using System;
using System.Collections.Generic;
using System.Text;

namespace ModeloConGeneracionAleatoria
{
    public class Persona
    {

        private String elApellido;
        private String elNombre;
        private DateTime laFechaNac;
        private int laEdad;

        public Persona()
        {

        }

        public Persona(String Apellido, String Nombre, DateTime FechaNac)
        {

            this.Apellido = Apellido;
            this.Nombre = Nombre;
            this.FechaNac= FechaNac;
        }
    }
}

```



```

        laEdad = this.Edad;
    }

    public String Apellido
    {
        get { return elApellido; }
        set { elApellido = value.ToUpper(); }
    }
    public String Nombre
    {
        get { return elNombre; }
        set { elNombre = value.ToLower(); }
    }
    public DateTime FechaNac
    {
        get { return laFechaNac; }
        set
        {
            if (value.Year >= 1900)
            {
                laFechaNac = value;
            }
        }
    }

    public int Edad
    {
        get { return DateTime.Now.Year - FechaNac.Year; }
    }

    public override string ToString()
    {
        return String.Format("Apellidos: {0}, Nombre: {1}, FechaNac: {2:d}, Edad: {3} ",
this.Apellido, this.Nombre, this.FechaNac, this.Edad);
    }
}
}

```

La clase Empleado

```

using System;
using System.Collections.Generic;
using System.Text;

namespace ModeloConGeneracionAleatoria
{
    public abstract class Empleado : Persona
    {
        private double salario;
    }
}

```

```

        private string direccion;

        public Empleado()
        {

        }

        public Empleado(String Apellido, String Nombre, DateTime FechaNac, string direccion,
double elSalario) : base(Apellido, Nombre, FechaNac) {
            this.Direccion = direccion;
            this.Salario = elSalario;
        }

        public abstract double calculoImpuestos();

        public abstract double salarioNeto();

        public string Direccion { get { return direccion; } set { direccion=value; } }

        public double Salario { get { return salario; } set { salario = value; } }


        public override string ToString()
        {
            return String.Format("{0} Direccion: {1} Salario: {2}", base.ToString(),
this.Direccion, this.Salario);
        }
    }
}

```

El interfaz IBonus

```

using System;
using System.Collections.Generic;
using System.Text;

namespace PrimeraClase
{
    public interface IBonus
    {

        public double calculoBonus();
    }
}

```

La clase Administrativo

```

using System;
using System.Collections.Generic;
using System.Text;

namespace PrimeraClase

```

```

{
    class Administrativo : Empleado
    {
        private const double PORCENTAJEIMPUESTOS = 0.10;
        public override double calculoImpuestos()
        {
            return PORCENTAJEIMPUESTOS*this.Salario;
        }

        public override double salarioNeto()
        {
            return this.Salario -this.calculoImpuestos();
        }
    }
}

```

La clase Ejecutivo

```

using System;
using System.Collections.Generic;
using System.Text;

namespace ModeloConGeneracionAleatoria
{
    class Ejecutivo : Empleado, IBonus
    {
        private const double PORCENTAJEIMPUESTOS = 0.30;

        private const double BONUS = 400;

        public Ejecutivo()
        {
        }

        public Ejecutivo(String Apellido, String Nombre, DateTime FechaNac, string Direccion,
double elSalario) : base(Apellido, Nombre, FechaNac, Direccion, elSalario)
        {
        }

        public double calculoBonus()
        {
            return BONUS;
        }

        public override double calculoImpuestos()
        {
            return (this.Salario + BONUS) * PORCENTAJEIMPUESTOS;
        }
    }
}

```

```

    public override double salarioNeto()
    {
        return Salario + calculoBonus() - calculoImpuestos();
    }
}

```

2.1 Clases para generar colecciones aleatorias

La clase GenerarCamposAleatorios esta dentro del Espacio de nombres del resto de clases, pero añade un subespacio de nombres Debug , ModeloConGeneracionAleatoria.Debug

Tenemos 3 arrays para combinar nombres, apellidos y direcciones en los empleados

```

    public static string[] nombres = {"ANTONIO", "MANUEL", "JOSE", "FRANCISCO", "DAVID",
    "JUAN", "JOSE ANTONIO", "JAVIER", "DANIEL", "JOSE LUIS",
    "FRANCISCO JAVIER", "CARLOS", "JESUS", "ALEJANDRO", "FRANCISCA", "LUCIA", "MARIA
    ISABEL", "MARIA JOSE", "ANTONIA", "DOLORES", "SARA",
    "PAULA", "ELENA", "MARIA LUISA", "RAQUEL"};
    public static string[] apellidos = {"García", "González", "Rodríguez", "Fernández",
    "López", "Martínez", "Sánchez", "Pérez", "Gómez",
    "Martín", "Jiménez", "Ruiz", "Hernández", "Díaz", "Moreno", "Muñoz", "Álvarez", "Romero",
    "Alonso", "Gutiérrez", "Navarro",
    "Torres", "Domínguez", "Vázquez", "Ramos", "Gil", "Ramírez", "Serrano", "Blanco", "Molina",
    "Morales", "Suarez", "Ortega",
    "Delgado", "Castro", "Ortiz", "Rubio", "Marín", "Sanz", "Núñez", "Iglesias", "Medina",
    "Garrido", "Cortes", "Castillo", "Santos"};
    public static string[] direcciones = {"Alfonso López de Haro", "Calle de Alvarfáñez de
    Minaya", "Calle de Arcipreste de Hita",
    "Calle de Arrabal del Agua", "Plaza de Beladiez", "Plaza de Caídos en la Guerra civil", "Calle
    Minaya", "Calle Hermanos Galiano"};

```

Usamos dos delegados para generar aleatorios en un rango, uno para enteros, otro para doubles.

```

    private static Func<int, int, int> randomFromTo = (ini, fin) => rand.Next() % (fin-
    ini) +ini;

    private static Func<int, int, double> randomFromToDouble = (ini, fin) =>
    rand.NextDouble() * (fin - ini) + ini;

```

2.1.1 Ejercicio

Comentar los métodos marcados en azul y comentar que hacen cada uno:

Ejemplo completo

```
using System;
using System.Collections.Generic;
using System.Text;

namespace ModeloConGeneracionAleatoria.Debug
{
    class GenerarCamposAleatorios
    {
        public static string[] nombres = {"ANTONIO", "MANUEL", "JOSE", "FRANCISCO", "DAVID",
            "JUAN", "JOSE ANTONIO", "JAVIER", "DANIEL", "JOSE LUIS",
            "FRANCISCO JAVIER", "CARLOS", "JESUS", "ALEJANDRO", "FRANCISCA", "LUCIA", "MARIA
            ISABEL", "MARIA JOSE", "ANTONIA", "DOLORES", "SARA",
            "PAULA", "ELENA", "MARIA LUISA", "RAQUEL"};
        public static string[] apellidos = {"García", "González", "Rodríguez", "Fernández",
            "López", "Martínez", "Sánchez", "Pérez", "Gómez",
            "Martín", "Jiménez", "Ruiz", "Hernández", "Díaz", "Moreno", "Muñoz", "Álvarez", "Romero",
            "Alonso", "Gutiérrez", "Navarro",
            "Torres", "Domínguez", "Vázquez", "Ramos", "Gil", "Ramírez", "Serrano", "Blanco", "Molina",
            "Morales", "Suarez", "Ortega",
            "Delgado", "Castro", "Ortiz", "Rubio", "Marín", "Sanz", "Núñez", "Iglesias", "Medina",
            "Garrido", "Cortes", "Castillo", "Santos"};
        public static string[] direcciones = {"Alfonso López de Haro", "Calle de Alvarfáñez de
            Minaya", "Calle de Arcipreste de Hita",
            "Calle de Arrabal del Agua", "Plaza de Beladiez", "Plaza de Caídos en la Guerra civil", "Calle
            Minaya", "Calle Hermanos Galiano"};

        private static Random rand = new Random();

        private static Func<int, int, int> randomFromTo = (ini, fin) => rand.Next() % (fin -
            ini) + ini;

        private static Func<int, int, double> randomFromToDouble = (ini, fin) =>
            rand.NextDouble() * (fin - ini) + ini;

        public static string nombreAleatorio()
        {
            return nombres[randomFromTo(0, nombres.Length - 1)];
        }

        public static string apellidosAleatorio()
        {
            return apellidos[randomFromTo(0, apellidos.Length - 1)] + " " +
                apellidos[randomFromTo(0, apellidos.Length - 1)];
        }

        public static DateTime fechaNacimientoAleatoria()
        {
            return new DateTime(randomFromTo(1960, 2001), randomFromTo(1, 12), randomFromTo(1,
                28));
        }
    }
}
```

```

        public static string direccionAleatoria()
        {
            return direcciones[randomFromTo(0, direcciones.Length - 1)];
        }

        public static double SalarioAleatorio()
        {
            return randomFromToDouble(600, 4000);
        }
    }
}

```

La clase GeneraColeccionesAleatorias

Tenemos el método `public static List<Empleado> ListaEmpleados(int numEmpleados)` que genera una lista de Empleados de tipo o Administrativo o Ejecutivo, de tamaño numEmpleados, el parámetro pasado.

```

using System;
using System.Collections.Generic;
using System.Text;

namespace ModeloConGeneracionAleatoria.Debug
{
    class GeneraColeccionesAleatorias
    {
        public static List<Empleado> ListaEmpleados(int numEmpleados)
        {
            List<Empleado> listaEmpleados = new List<Empleado>();

            Random r = new Random();

            for (int i= 0; i < numEmpleados; i++ )
            {
                Empleado emp=null;

                switch (r.Next()%2)
                {
                    case 0:
                        emp = new Administrativo(GenerarCamposAleatorios.apellidosAleatorio(),
                                                GenerarCamposAleatorios.nombreAleatorio(),
                                                GenerarCamposAleatorios.fechaNacimientoAleatoria(),
                                                GenerarCamposAleatorios.direccionAleatoria(),
                                                GenerarCamposAleatorios.SalarioAleatorio());

                        break;

                    case 1:

```

```

        emp = new Ejecutivo(GenerarCamposAleatorios.apellidosAleatorio(),
                             GenerarCamposAleatorios.nombreAleatorio(),
                             GenerarCamposAleatorios.fechaNacimientoAleatoria(),
                             GenerarCamposAleatorios.direccionAleatoria(),
                             GenerarCamposAleatorios.SalarioAleatorio());

        break;
    }

    listaEmpleados.Add(emp);
}

return listaEmpleados;
}
}
}

```

El programa principal genera una lista de 200 empleados aleatorios y la visualiza la lista.

```

using System;
using System.Collections.Generic;
using ModeloConGeneracionAleatoria.Debug;

namespace ModeloConGeneracionAleatoria
{
    class Program
    {
        static void Main(string[] args)
        {
            List<Empleado> listaEmp = GeneraColeccionesAleatorias.ListaEmpleados(200);

            listaEmp.ForEach((emp) => Console.WriteLine(emp.ToString()));
        }
    }
}

```

Ejemplo de ejecución:

Apellidos: MARTÍNEZ MARÍN, Nombre: jose antonio, FechaNac: 01/10/1962, Edad: 59
 Direccion: Plaza de Caídos en la Guerra civil Salario: 862,9111398304399
 Apellidos: DOMÍNGUEZ DELGADO, Nombre: dolores, FechaNac: 16/03/1975, Edad: 46
 Direccion: Plaza de Beladiez Salario: 1388,5897048695897
 Apellidos: ORTEGA SERRANO, Nombre: antonia, FechaNac: 27/02/1967, Edad: 54 Direccion:
 Calle de Arcipreste de Hita Salario: 1292,06384219791
 Apellidos: CASTILLO GARCÍA, Nombre: francisco, FechaNac: 24/09/2000, Edad: 21 Direccion:
 Calle Minaya Salario: 3666,419420981044

2.1.2 Ejercicios

1. Crear una nueva clase `Tecnico`, que herede de `Empleado` e implemente `IBonus`. Su bonus será de 300 euros. Modifica `GeneraColeccionesAleatorio` para que se generen técnicos en la lista de `Empleados`
2. Añade a `Empleado` el campo `id`, un entero de nueve dígitos:
 - a. Añade getters y setters
 - b. Modifica los constructores de la clase y subclases para que reciban el `Id` del `Empleado`
 - c. En `GeneraCamposAleatorios` añade un método estático `GeneralDAleatorio` para que genere un entero de 9 dígitos aleatorio.
 - d. Modifica `GeneraColeccionesAleatorio` para incluir el `ID` en la generación de empleados.
3. Añade un método estático en `GeneraColeccionesAleatorio` para que devuelva un `Dictionary` de `Empleados`, cuya clave sea el `Id`, y el valor el `Empleado` entero:
`public static Dictionary<long,Empleado> diccionarioEmpleados(int numEmpleados)`
4. Añade un método estático en `GeneraColeccionesAleatorio` para que devuelva un `SortedList` de `Empleados`, cuya clave sea el `Id`, y el valor el `Empleado` entero:
`public static SortedList<long,Empleado> diccionarioEmpleados(int numEmpleados)`
5. Añade un método estático en `GeneraColeccionesAleatorio` para que devuelva un `Queue` de `Empleados`, cuya clave sea el `Id`, y el valor el `Empleado` entero:
`public static Queue<Empleado> colaEmpleados(int numEmpleados)`

3 LINQ

Language-Integrated Query (LINQ) es el nombre de un conjunto de tecnologías basadas en la integración de capacidades de consulta directamente en el lenguaje `C#`. Tradicionalmente, las consultas con datos se expresaban como cadenas simples sin comprobación de tipos en tiempo de compilación ni compatibilidad con `IntelliSense`. Además, tendrá que aprender un lenguaje de consulta diferente para cada tipo de origen de datos: bases de datos SQL, documentos XML, varios servicios web y así sucesivamente. Con LINQ, una consulta es una construcción de lenguaje de primera clase, como clases, métodos y eventos. Escribe consultas en colecciones de objetos fuertemente tipadas con palabras clave del lenguaje y operadores familiares. La familia de tecnologías de LINQ proporciona una experiencia de

consulta coherente para objetos (LINQ to Objects), bases de datos relacionales (LINQ to SQL) y XML (LINQ to XML).

Para un desarrollador que escribe consultas, la parte más visible de "lenguaje integrado" de LINQ es la expresión de consulta. Las expresiones de consulta se escriben con una sintaxis de consulta declarativa. Con la sintaxis de consulta, puede realizar operaciones de filtrado, ordenación y agrupamiento en orígenes de datos con el mínimo código. Utilice los mismos patrones de expresión de consulta básica para consultar y transformar datos de bases de datos SQL, conjuntos de datos de ADO .NET, secuencias y documentos XML y colecciones. NET.

Puede escribir consultas LINQ en C# para bases de datos de SQL Server, documentos XML, conjuntos de datos ADO.NET y cualquier colección de objetos que admita IEnumerable o la interfaz genérica IEnumerable<T>. La compatibilidad con LINQ también se proporciona por terceros para muchos servicios web y otras implementaciones de base de datos.

En el ejemplo siguiente se muestra la operación de consulta completa. La operación completa incluye crear un origen de datos, definir la expresión de consulta y ejecutar la consulta en una instrucción foreach.

A destacar que podemos realizar la consulta LINQ de manera tradicional. Como veis el lenguaje es muy parecido a SQL, con lo que el aprendizaje no es complicado. Fijaos como las sentencias llevan orden invertido.

```
IEnumerable<int> notasQuery =  
    from nota in notas  
    where nota > 80  
    orderby nota descending  
    select nota  
    ;
```

Pero también está integrada en métodos en el interfaz IEnumerable y IQueryable que implementa el anterior, usando delegados, en este caso expresiones lambda por comodidad. Ambas consultas van a producir los mismos resultados. Intentaremos aprender este sistema de manejo de datos en C# de aquí en adelante.

```
IEnumerable<int> notaQueryConLambda = notas.Where(nota => nota >  
80).OrderByDescending(a=> a);
```

Ejemplo completo

```
using System;  
using System.Collections.Generic;  
using System.Linq;  
using System.Xml.Linq;  
  
namespace IntroduccionALinq  
{
```

```

class Program
{

    static void Main()
    {

        // Specify the data source.
        int[] notas = new int[] { 97, 92, 81, 60 };

        Console.WriteLine("Y las notas con LINQ de consulta son:");

        // Define the query expression.
        IEnumerable<int> notasQuery =
            from nota in notas
            where nota > 80
            orderby nota descending
            select nota
            ;

        // Execute the query.
        foreach (int i in notasQuery)
        {
            Console.Write(i + " |");
        }

        IEnumerable<int> notaQueryConLambda = notas.Where(nota => nota >
80).OrderByDescending(a=> a);

        Console.WriteLine("Y las notas con LINQ de métodos son:");
        notaQueryConLambda.ToList().ForEach(nota => Console.Write(" {0} |", nota));

        Console.WriteLine("Trabajando con XML:");

        XElement Factura = XElement.Load(
@"C:\Users\carlo\OneDrive\Brianda20202021\interfaces\.NET\Tema1\Proyectos\Tema2\IntroduccionALinq\IntroduccionALinq\empleados.xml");

        IEnumerable<XElement> queryFactura = Factura.Elements("empleado").Where(emp =>
emp.Element("nombre").Value.StartsWith("A"));

        foreach (XElement elem in queryFactura)
        {

            Console.WriteLine(elem);

        }

    }
}

```

3.1 *El interfaz IQueryable*

La IQueryable<T> interfaz está pensada para que la implementan los proveedores de

consultas.

Esta interfaz hereda la `IEnumerable<T>` interfaz para que, si representa una consulta, se puedan enumerar los resultados de esa consulta. La enumeración fuerza el árbol de expresión asociado a un `IQueryable<T>` objeto que se va a ejecutar. Las consultas que no devuelven resultados enumerables se ejecutan cuando `Execute<TResult>(Expression)` se llama al método.

La `IQueryable<T>` interfaz permite que las consultas sean polimórficas. Es decir, dado que una consulta en un `IQueryable` origen de datos se representa como un árbol de expresión, se puede ejecutar con distintos tipos de orígenes de datos.

Propiedades

PROPIEDADES

ElementType	Obtiene el tipo de los elementos que se devuelven cuando se ejecuta el árbol de expresión asociado a esta instancia de <code>IQueryable</code> . (Heredado de <code>IQueryable</code>)
Expression	Obtiene el árbol de expresión que está asociado a la instancia de <code>IQueryable</code> . (Heredado de <code>IQueryable</code>)
Provider	Obtiene el proveedor de consultas que está asociado a este origen de datos. (Heredado de <code>IQueryable</code>)

Métodos

GetEnumerator()

Devuelve un enumerador que recorre en iteración una colección. (Heredado de `IEnumerable`)

Métodos extendidos

Proporciona una gran cantidad de métodos extendidos para realizar consultas que veremos en los ejemplos, pues son demasiados. Como ejemplos podemos enumerar `Where`, `Select`, `OrderBy`, `Any`, `All`, `Join`, `GroupJoin`, `ToList`, `ToDictionary`, `ToSortedList` y muchos más. El enlace complete a todos los métodos es el siguiente:

<https://docs.microsoft.com/es-es/dotnet/api/system.linq.iqueryable-1?view=net-5.0>

3.2 Partes de una consulta LINQ

Todas las operaciones de consulta LINQ constan de tres acciones distintas:

- Obtener el origen de datos.
- Crear la consulta.
- Ejecutar la consulta.

En el siguiente ejemplo se muestra cómo se expresan las tres partes de una operación de consulta en código fuente. En el ejemplo se usa una matriz de enteros como origen de datos para su comodidad, aunque se aplican los mismos conceptos a otros orígenes de datos. En el resto de este tema se hará referencia a este ejemplo.

Orígenes de datos

- XML

```
XElement Factura = XElement.Load(
@"C:\Users\carlo\OneDrive\Brianda20202021\interfaces\.NET\Tema1\Proyectos\Tema2\IntroduccionALinq\IntroduccionALinq\empleados.xml");

IEnumerable<XElement> queryFactura = Factura.Elements("empleado").Where(emp =>
emp.Element("nombre").Value.StartsWith("A"));

foreach (XElement elem in queryFactura)
{
    Console.WriteLine(elem);
}
```

XElement implementa IQueryable<T>, se deriva de IEnumerable<T>.y se puede consultar un XML con LINQ

- Bases de datos como SQLServer

```
Northwnd db = new Northwnd(@"c:\northwnd.mdf");

// Query for customers in London.
IQueryable<Customer> custQuery =
    from cust in db.Customers
    where cust.City == "London"
    select cust;
```

La consulta

La consulta especifica la información que se debe recuperar de los orígenes de datos. Opcionalmente, una consulta también especifica cómo se debe ordenar, agrupar y conformar esa información antes de que se devuelva. Las consultas se almacenan en una variable de consulta y se inician con una expresión de consulta. Para facilitar la escritura de

consultas, C# ha incorporado una nueva sintaxis de consulta.

La consulta del ejemplo anterior devuelve todos los números pares de la matriz de enteros. La expresión de consulta contiene tres cláusulas: `from`, `where` y `select` (si está familiarizado con SQL, habrá observado que el orden de las cláusulas se invierte respecto al orden de SQL). La cláusula `from` especifica el origen de datos, la cláusula `where` aplica el filtro y la cláusula `select` especifica el tipo de los elementos devueltos. Estas y otras cláusulas de consulta se tratan con detalle en la sección Language Integrated Query (LINQ). Por ahora, lo importante es que en LINQ la variable de consulta no efectúa ninguna acción y no devuelve ningún dato. Lo único que hace es almacenar la información necesaria para generar los resultados cuando se ejecuta la consulta en algún momento posterior.

```
IEnumerable<int> notasQuery =  
    from nota in notas  
    where nota > 80  
    orderby nota descending  
    select nota  
    ;
```

Ó

```
IQueryable<Customer> custQuery =  
    from cust in db.Customers  
    where cust.City == "London"  
    select cust;
```

Ejecución aplazada

Como se ha indicado anteriormente, la variable de consulta solo almacena los comandos de consulta. La ejecución real de la consulta se aplaza hasta que se procese una iteración en la variable de consulta en una instrucción `foreach`. Este concepto se conoce como ejecución aplazada o Lazy Evaluation y se muestra en el ejemplo siguiente:

```
IEnumerable<int> notaQueryConLambda = notas.Where(nota => nota >  
80).OrderByDescending(a=> a);  
  
Console.WriteLine("\nY las notas con LINQ de métodos son:");  
notaQueryConLambda.ToList().ForEach(nota => Console.Write(" {0} |", nota));  
}
```

Ejecución forzada

Podemos forzar la ejecución transformando a colección a `Array`, llamado a `Métodos` como `ToArray()` o `ToList()`.

```
notaQueryConLambda.ToList().ForEach(nota => Console.Write(" {0} |",  
nota)).ToList();
```

Variables de LINQ

Las variables de consulta LINQ tienen el tipo `IEnumerable<T>` o un tipo derivado como `IQueryable<T>`. Cuando vea una variable de consulta que tiene el tipo `IEnumerable<Customer>`, significa que, al ejecutarse, la consulta generará una secuencia de cero o más objetos `Customer`.

3.3 Consultas básicas LINQ

3.3.1 LINQ con sintaxis de query.

3.3.2 Esta sintaxis es muy parecida a SQL, los conocedores de SQL la manejan fácilmente y parece más legible para ellos.

From y select

En una consulta LINQ, el primer paso es especificar el origen de datos. En C#, como en la mayoría de los lenguajes de programación, se debe declarar una variable antes de poder usarla. En una consulta LINQ, la cláusula `from` aparece en primer lugar para introducir el origen de datos (`listaEmp`) que es una lista de 1000 empleados y la variable de rango (`emp`). Esta es la consulta más básica que podemos realizar. Vamos a añadirle más cosas para hacerla un poquito más compleja

La variable de rango hace referencia a cada una de las entidades que contiene la fuente de datos. En este caso son objetos de tipo `Empleado`. Podemos acceder a las propiedades y métodos de dicho objeto dentro de la consulta

```
List<Empleado> listaEmp = GeneraColeccionesAleatorias.ListaEmpleados(1000);

IEnumerable<Empleado> consulta = from emp in listaEmp
                                select emp;

foreach( var empleado in consulta)
{
    Console.WriteLine(empleado);
}
```

Filtrado con Where

Usamos la sentencia `Where` para filtrar en nuestra consulta. El `Where` admite operadores

lógicos como || (or), && (and), y ! (not)

```
IEnumerable<Empleado> consulta = from emp in listaEmp
                                  where emp.Edad == 50 || emp.Edad== 48
                                  select emp;
```

3.3.3 Ejercicios

1. Modificar la consulta anterior para que obtenga todos los empleados cuya edad es distinto de 50.
2. Modificar la consulta para que obtenga todos los empleados distintos de 50 y cuyo nombre no empiece por "A".

Ordenando con order by

Con la orden order by podemos ordenar de manera ascendente (ascending) o descendiente (descending). Ordenamos por el nombre en la siguiente consulta.

```
IEnumerable<Empleado> consulta = from emp in listaEmp
                                  where emp.Edad == 50 || emp.Edad== 48
                                  orderby emp.Nombre ascending
                                  select emp;
```

Podemos ordenar por más de un campo como en el siguiente ejemplo. Los campos separados por comas.

```
IEnumerable<Empleado> consulta = from emp in listaEmp
                                  where emp.Edad == 50 || emp.Edad== 48
                                  orderby emp.Apellido, emp.Nombre descending
                                  select emp;
```

Ordenando con group by

Podemos agrupar por campos con la orden group by como en una sentencia sql. Cuando agrupamos por order by nos devolverá un IEnumerable que contiene objetos de tipo IGrouping, la clave será el valor del campo por el que agrupamos, el objeto, el empleado. Por cada clave diferente tendremos una colección de Empleados. En este caso, son dos claves 50 y 48, tendremos dos colecciones, una para los de Edad 50, otro para los de edad 48.

```
IEnumerable<IGrouping<int, Empleado>> consultaAgrupada = from empl in listaEmp
                                                         where empl.Edad == 50 || empl.Edad == 48
                                                         orderby empl.Apellido, empl.Nombre descending
                                                         group empl by empl.Edad;
```

Tenemos que realizar para recorrer la consulta un foreach externo para recorrer todos los

miembros del `IEnumerable<IGrouping<int,Empleado>>`. Luego para cada empleado de la colección asociada a la clave debemos recorrer todos los empleados de cada `IGrouping<int,Empleado>`.

```
foreach (IGrouping<int, Empleado> empleadoGroup in consultaAgrupada)
{
    foreach (Empleado empleado in empleadoGroup) {
        Console.WriteLine(empleado);
    }
}
```

Mejoramos la consulta anterior mandando la agrupación a otra variable de rango `emplGroup`. De esta manera podremos hacer una selección sobre esa variable.

```
IEnumerable<IGrouping<int, Empleado>> consultaAgrupada = from empl in listaEmp
where empl.Edad == 50 || empl.Edad == 48
orderby empl.Apellido, empl.Nombre descending
group empl by empl.Edad into emplGroup
select emplGroup;
```

Tenéis información completa de `group by` en el siguiente enlace:

<https://docs.microsoft.com/es-es/dotnet/api/system.linq.igrouping-2?view=net-5.0>

Selección o proyección.

Estamos trabajando con objetos todo el rato. Podemos realizar la selección de un campo del objeto empleado `select emp.Nombre`, que devolverá un `IEnumerable` de tipo `string`.

```
IEnumerable<string> consulta2 = from emp in listaEmp
where emp.Edad == 50 || emp.Edad == 48
orderby emp.Apellido, emp.Nombre descending
select emp.Nombre;

foreach (var obj in consulta2)
{
    Console.WriteLine(obj);
}
```


Si queremos seleccionar más de una propiedad del objeto Empleado debemos construir un objeto anónimo nuevo con un new. Al ser anónimos los tipos que definimos, debe ser el compilador quien los resuelva no podemos tipar las variables resultado de nuestra consulta. Usamos la **opción var del compilador** cuando no sabemos que tipo va a devolver una operación.

Seleccionamos tres campos en este caso para crear un objeto nuevo

```
select new { emp.Nombre, emp.Apellido, emp.Edad } ;
```

```
var consulta2 = from emp in listaEmp
                where emp.Edad == 50 || emp.Edad == 48
                orderby emp.Apellido, emp.Nombre descending

                select new { emp.Nombre, emp.Apellido, emp.Edad } ;

foreach (var obj in consulta2)
{
    Console.WriteLine(obj);
}
```

El resultado de esta ejecución se vería como objetos anónimos.

```
{ Nombre = manuel, Apellido = ALONSO ALONSO, Edad = 48 }
{ Nombre = elena, Apellido = ÁLVAREZ RODRÍGUEZ, Edad = 48 }
{ Nombre = alejandro, Apellido = BLANCO LÓPEZ, Edad = 48 }
```

3.3.4 Ejercicios

1. Modificar la consulta del ejercicio anterior para que además de obtener todos los empleados cuya edad es distinto de 50, ordene por Apellidos
2. Modificar la consulta para que obtenga todos los empleados distintos de 50 y cuyo nombre no empiece por "A". Ordenar por apellidos y agrupar por Nombre.

Haciendo uniones con la orden join

Con la orden join podemos seleccionar dos fuentes de datos y hacer un join como si de una query SQL se tratará. En este caso seleccionamos los valores de la consulta y de consulta2 que creaba un objeto anónimo, y las unimos por el campo Edad

Cogemos los empleados de consulta 1

```
from emp in consulta
```

los unimos con los objetos anónimos de consulta 2

```
join empanon in consulta2
```

y los cruzamos por edad

```
on emp.Edad equals empanon.Edad
```

Finalmente seleccionamos dos campos

```
select new { emp.Nombre, emp.Edad };
```

```
var consultaJoin = from emp in consulta
                    join empanon in consulta2 on emp.Edad equals empanon.Edad
                    select new { emp.Nombre, emp.Edad };
```

```
foreach (var obj in consultaJoin)
{
    Console.WriteLine(obj);
}
```

3.3.5 LINQ con sintaxis de método y expresiones lambda.

Podemos realizar las anteriores consultas LINQ aprovechando la programación funcional de C# y usando LINQ como llamadas a métodos o funciones. Todo lo que podemos hacer con LINQ tradicional lo podemos hacer con LINQ con métodos como vamos a demostrar en el siguiente ejemplo. Realmente lo que hace el compilador de C# es traducir las consultas con sintaxis de Query a métodos internamente en compilación. Si las escribimos directamente con métodos ahorramos un paso al compilador, y programamos de una manera más pura en C#. La desventaja es que para quien sabe SQL las consultas con sintaxis Query son un poco más legibles. En cualquier caso en muchas consultas con sintaxis de Query hay que llamar a métodos.

Método Where

El where como método requiere como parámetro un tipo delegado o función. Por comodidad y siendo más funcionales usamos expresiones lambda. Para el Where el delegado que recibe es un tipo Predicate, con lo que la expresión lambda o método debe devolver un booleano

```
emp => emp.Edad == 50 || emp.Edad == 48
```

```
IEnumerable<Empleado> consulta = listaEmp.Where<Empleado>(emp => emp.Edad == 50 ||
emp.Edad == 48)
```

Equivalente al ejemplo anterior `where emp.Edad == 50 || emp.Edad== 48`

Método OrderBy y ThenBy

Para ordenar tenemos el método OrderBy. Va a recibir como parámetro un delegado que devuelva la clave o campo por el que se ordena. En el order by de LINQ podemos indicar varios campos, aquí sólo podemos indicar uno.

```
.OrderByDescending(emp=>emp.Apellido)
```

Para suplir esta carencia tenemos el método `ThenByDescending(emp=>emp.Nombre)` en el que podemos indicar un siguiente campo para ordenar tras el primero. Debemos tener en cuenta que el campo que pasamos en la clave debe tener implementado IComparable, el método CompareTo para poder comparar. Los tipos de C# lo tienen, si hemos construido un tipo propio debemos hacer que implemente IComparable.

```
IEnumerable<Empleado> consulta = listaEmp.Where<Empleado>(emp => emp.Edad == 50 ||
emp.Edad == 48)
.OrderByDescending(emp=>emp.Apellido).ThenByDescending(emp=>emp.Nombre);
```

Equivalente a la expresión LINQ `orderby emp.Apellido, emp.Nombre descending`

GroupBy

Este método también requerirá un delegado en que se indique el campo o clave por el que se va a ordenar.

```
IEnumerable<IGrouping<int, Empleado>> consultaAgrupada =
listaEmp.Where<Empleado>(emp => emp.Edad == 50 || emp.Edad == 48)
.OrderByDescending(emp =>
emp.Apellido).ThenByDescending(emp => emp.Nombre)
.GroupBy(emp=> emp.Edad);
```

Equivalente a la expression LINQ `group empl by empl.Edad;`

El metodo Select()

Va a recibir de nuevo un delegado que devuelve el campo que vamos a seleccionar. Si

queremos seleccionar varios campos tendremos que construir un objeto anónimo nuevo. Como veis hemos preferido primero hacer la proyección select y luego agrupar con el Groupby.

```
.Select(emp=> new { emp.Nombre, emp.Apellido, emp.Edad } ).GroupBy(emp => emp.Edad);
```

El método Join()

Para combinar dos consultas u orígenes de datos usaremos Join. En el siguiente ejemplo combinamos dos consultas, consulta y consulta 3. Los parámetros de Join son

```
consulta.Join(consulta3, emp=> emp.Edad, emp3=>emp3.Edad , (emp, emp3)=> new { emp.Nombre, emp.Apellido, emp.Direccion })
```

Origen de datos a la que combinar la origen de datos inicial.

consulta3

Delegado indicando el campo o clave por el que hacemos el Join en el origen de datos inicial

emp=> emp.Edad

Delegado indicando el campo o por el que hacemos el Join en el segundo origen de datos

emp3=>emp3.Edad

Delegado indicando que seleccionamos tras la unión. Tiene dos parámetros la expresión lambda, uno por cada origen de datos.

```
(emp, emp3)=> new { emp.Nombre, emp.Apellido, emp.Direccion }
```

```
IEnumerable<dynamic> consulta3 = listaEmp.Where<Empleado>(emp => emp.Edad == 50 || emp.Edad == 48)
    .OrderByDescending(emp => emp.Apellido).ThenByDescending(emp => emp.Nombre)
    .Select(emp => new { emp.Nombre, emp.Apellido, emp.Edad });
```

```
var consultaJoin = consulta.Join(consulta3, emp=> emp.Edad, emp3=>emp3.Edad , (emp, emp3)=> new { emp.Nombre, emp.Apellido, emp.Direccion });
```

```
foreach (var obj in consultaJoin)
{
    Console.WriteLine(obj);
}
```

Ejemplo completo LINQ con expresiones lambda.

```
using LinqConsultasLambda.Debug;
using System;
using System.Collections.Generic;
using System.Linq;

namespace LinqConsultasLambda
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Hello World!");

            List<Empleado> listaEmp = GeneraColeccionesAleatorias.ListaEmpleados(1000);

            IEnumerable<Empleado> consulta = listaEmp.Where<Empleado>(emp => emp.Edad == 50 ||
emp.Edad == 48)
.OrderByDescending(emp=>emp.Apellido).ThenByDescending(emp=>emp.Nombre);

            foreach (var empleado in consulta)
            {
                Console.WriteLine(empleado);
            }

            IEnumerable<IGrouping<int, Empleado>> consultaAgrupada =
listaEmp.Where<Empleado>(emp => emp.Edad == 50 || emp.Edad == 48)
.OrderByDescending(emp =>
emp.Apellido).ThenByDescending(emp => emp.Nombre)
.GroupBy(emp=> emp.Edad);

            foreach (IGrouping<int, Empleado> empleadoGroup in consultaAgrupada)
            {
                Console.WriteLine("Vueltas");
                foreach (Empleado empleado in empleadoGroup) {

                    Console.WriteLine(empleado);

                }
            }

            var consulta2 = listaEmp.Where<Empleado>(emp => emp.Edad == 50 || emp.Edad == 48)
.OrderByDescending(emp =>
emp.Apellido).ThenByDescending(emp => emp.Nombre)
```

```

        emp.Apellido, emp.Edad } ).GroupBy(emp => emp.Edad);
        .Select(emp=> new { emp.Nombre,

foreach (var obj in consulta2)
{
    Console.WriteLine(obj);
}

IEnumerable<dynamic> consulta3 = listaEmp.Where<Empleado>(emp => emp.Edad == 50 ||
emp.Edad == 48)
.OrderByDescending(emp => emp.Apellido).ThenByDescending(emp => emp.Nombre)
.Select(emp => new { emp.Nombre, emp.Apellido, emp.Edad });

var consultaJoin = consulta.Join(consulta3, emp=> emp.Edad, emp3=>emp3.Edad ,
(emp, emp3)=> new { emp.Nombre, emp.Apellido, emp.Direccion });

foreach (var obj in consultaJoin)
{
    Console.WriteLine(obj);
}
}
}
}

```

3.3.6 Ejercicios

1. Modificar la consulta anterior para que obtenga todos los empleados cuya edad es distinta de 50.
2. Modificar la consulta para que ordene por Apellidos
3. Modificar la consulta para que obtenga todos los empleados distintos de 50 y cuyo nombre no empiece por "A".
4. Modificar la consulta para que obtenga todos los empleados distintos de 50 y cuyo nombre no empiece por "A". Ordenar por apellidos y agrupar por Nombre.

3.4 LINQ y XML

En primer lugar vamos a repasar brevemente las clases que se usan en C# y el framework .Net para manejar documentos xml. Empezaremos por XDocument.

XDocument

Representa un documento XML. Para ver los componentes y el uso de un objeto XDocument, consulte. El objetivo será explicar brevemente la clase si entrar en detalles, con el fin de poder usarlo con LINQ.

Constructores

SOBRECARGAS

XDocument()	Inicializa una nueva instancia de la clase XDocument.
XDocument(Object[])	Inicializa una nueva instancia de la clase XDocument con el contenido especificado.
XDocument(XDocument)	Inicializa una nueva instancia de la clase XDocument a partir de un objeto XDocument ya existente.
XDocument(XDeclaration, Object[])	Inicializa una nueva instancia de la clase XDocument con la clase XDeclaration y el contenido especificados.

Propiedades

Declaration: Obtiene o establece la declaración XML de este documento.

DocumentType: Obtiene la definición de tipo de documento (DTD) de este documento.

NodeType: Obtiene el tipo de nodo de este nodo.

Root: Obtiene el elemento raíz del árbol XML de este documento, de tipo XElement.

Métodos

Load: carga un xml de cadena, fichero,XMLReader, diferentes posibilidades

LoadAsync: igual, pero de manera asíncrona.

Parse

Save: Salva el XML a fichero

SaveAsync

WriteTo: Escribe un XML a partir de un XML Writer. No lo usaremos en clase en principio

WriteToAsync

La clase XElement

Nos permite crear un nodo XML

Constructores

SOBRECARGAS

XElement(XElement)	Inicializa una nueva instancia de la clase XElement a partir de otro objeto XElement.
XElement(XName)	Inicializa una nueva instancia de la clase XElement con el nombre especificado.
XElement(XStreamingElement)	Inicializa una nueva instancia de la clase XElement a partir de un objeto XStreamingElement.
XElement(XName, Object)	Inicializa una nueva instancia de la clase XElement con el nombre y el contenido especificados.

Propiedades

EmptySequence
FirstAttribute
HasAttributes
HasElements
IsEmpty
LastAttribute
Name
NodeType
Value

Métodos

AncestorsAndSelf: Devuelve la colección de los elementos que contienen este elemento y todos sus elementos ascendientes, en el orden del documento.
Attribute: devuelve un XAttribute, un atributo a partir de su nombre.
Attributes: Nos devuelve un inenumerable con todos los atributos XAttribute.
DescendantNodesAndSelf: Devuelve una colección de nodos que contienen este elemento y todos sus nodos descendientes, en el orden del documento.
DescendantsAndSelf: Devuelve la colección de los elementos que contienen este elemento y todos sus elementos descendientes, en el orden del documento.
GetDefaultNamespace: nos devuelve el Namespace del nodo si contiene

GetNamespaceOfPrefix
GetPrefixOfNamespace
Load
LoadAsync
Parse
RemoveAll
RemoveAttributes
ReplaceAll
ReplaceAttributes
Save
SaveAsync
SetAttributeValue: permite cambiar el valor de un atributo del nodo
SetElementValue: permite cambiar el valor de un elemento contenido en el nodo (un nodo hijo)
SetValue: permite cambiar el valor del nodo XML
WriteTo
WriteToAsync

Podemos crear XML a partir de una declaración, y definiendo un nodo raíz

```
XDocument doc = new XDocument(  
    new XDeclaration("1.0", "utf-8", "yes"),  
    new XElement("Root"));
```

Podemos crear nodos xml con la clase Element

```
XElement elemento = new XElement("Node1", "Prueba");
```

Podemos crear atributos y añadirlos a un elemento con el método Add the element.

```
XAttribute atr = new XAttribute("Atributo1", "Valor1");  
  
elemento.Add(atr);
```

Podemos añadir el nodo completo al nodo raíz con Add de la clase XElement.

```
doc.Root.Add(elemento);
```

Salvarlo en un fichero con el método save

```
doc.Save(@"C:\xmlcsharp\pruebacodigo.xml");  
  
Console.WriteLine(doc);
```

Resultado del XML creado

```
<Root>  
  <Node1 Atributo1="Valor1">Prueba</Node1>  
</Root>
```

Podemos recorrer todos los nodos que contienen el Root con el método Elements() de XElement.

```
Console.WriteLine("Recorriendo los nodos internos del Root Element");

foreach (XElement elem in doc.Root.Elements())
{
    Console.WriteLine(elem);
}
```

Resultado de la ejecución:

Recorriendo los nodos internos del Root Element
<Node1 Atributo1="Valor1">Prueba</Node1>

Recorrer todos los nodos descendientes y el mismo. Usamos var porque los descendientes pueden ser de tipo: XElement, XText, XAttribute y XComment.

```
Console.WriteLine("Recorriendo todos los nodos internos del Root Element y el mismo");
foreach (var elem in doc.Root.DescendantNodesAndSelf())
{
    Console.WriteLine(elem);
}
```

Resultado de la ejecución:

Recorriendo todos los nodos internos del Root Element y el mismo
<Root>
 <Node1 Atributo1="Valor1">Prueba</Node1>
</Root>
<Node1 Atributo1="Valor1">Prueba</Node1>
Prueba

Podemos cargar también con el método Estático Load de la Clase XDocument, xml desde un String o desde un fichero. Para el String es recomendable usar la clase TextReader.

Cargando el XML en un XDocument desde XML

```
TextReader tx = new StringReader(@"<MIXML><Node1>Prueba</Node1></MIXML>");

var doc2 = XDocument.Load(tx);
```

Añadir un espacio de nombres a un nodo concatenándoselo delante del nombre del nodo

```
XNamespace naps = "https://Miespaciodenombres";
```

```
doc2.Root.Add(new XElement(naps + "Node2", "prueba2"));
```

```
Console.WriteLine(doc2);
```

Resultado de la ejecución:

```
<MIXML>
  <Node1>Prueba</Node1>
  <Node2 xmlns="https://Miespaciodenombres">prueba2</Node2>
</MIXML>
```

Cargando el XML desde fichero con el método Load

```
var doc3 = XDocument.Load(@"C:\xmlcsharp\arboles.xml");

foreach (XElement elem in doc3.Root.Elements())
{
    Console.WriteLine(elem);
}
```

Resultado de la ejecución:

```
<ESPECIE>
  <n.latin>Acer monspessulanum</n.latin>
  <n.vulgar>Arce de Montpellier, Arce menor</n.vulgar>
  <hoja>Caducifolio</hoja>
  <altura>De 6 a 10 metro</altura>
  <estructura>Copa esférica. Tronco principal recto con bifurcaciones. Ramaje colgante</estructura>
  <color_prim>Haz verde brillante, envés verde blanquecino</color_prim>
  <color_otoño></color_otoño>
  <resistencia>Heladas fuertes (hasta -15°C)</resistencia>
</ESPECIE>
```

```

<ESPECIE>
  <n.latin>Olea europea</n.latin>
  <n.vulgar>Olivo</n.vulgar>
  <hoja>Perenne</hoja>
  <altura>De 8 a 15 metros</altura>
  <estructura>Copa irregular. Tronco principal irregular con bifurcaciones.</estructura>
  <color_prim></color_prim>
  <color_otoño></color_otoño>

```

3.5 Ejercicio

1. Crear un XML con la siguiente estructura usando las clases anteriormente explicadas.

```

<?xml version="1.0" encoding="UTF-8"?>
<empleados xmlns="https://Miespaciodenombres.com" >

  <empleado>
    <nombre>Juan</nombre>
    <apellidos>Ramirez</apellidos>
    <cedula>123</cedula>
  </empleado>

  <empleado>
    <nombre>Maria</nombre>
    <apellidos>Solis</apellidos>
    <cedula>452</cedula>
  </empleado>

</empleados>

```

2. Investigad para añadir un comentario al documento con la clase XComment.
 <!--Comentario de prueba -->

```

<empleados xmlns="https://Miespaciodenombres.com" >
  <!--Comentario de prueba -->

  <empleado>

```

3.6 Consultas LINQ en xml

Con diferentes ejemplos vamos a usar las clases C# para XML vistas anteriormente para realizar consultas con LINQ. Proporcionamos dos XML

empresas.xml

```
<empresas>
<empresa tipo="tecnologia">

<nombre>Equipos Digitales S.L.</nombre>
<dir>Av. Valladolid</dir>
<poblacion cod_postal="28043">Madrid</poblacion>
<provincia>Madrid</provincia>
<cif>222</cif>
<telefono/>
<ganancias>200000</ganancias>

</empresa>

<empresa tipo="construccion">

<nombre>Manos a la obra S.L.</nombre>
<dir>Calle retama 5</dir>
<poblacion cod_postal="28043">Caceres</poblacion>
<provincia>Caceres</provincia>
<cif>455</cif>
<telefono>45646456 </telefono>
<ganancias>300000</ganancias>

</empresa>

<empresa tipo="gestion">
<nombre>Consultora Esapas S.L.</nombre>
```

```
<dir>Calle la via 5</dir>
<poblacion cod_postal="28043">Tomelloso</poblacion>
<provincia>Ciudad Real</provincia>
<cif>333</cif>
<telefono>5643673 </telefono>
<ganancias>1000000</ganancias>
</empresa>
```

```
<empresa tipo="seguridad">
<nombre>SegCla S.L.</nombre>
<dir>Calle la constitución 17</dir>
<poblacion cod_postal="28043">Brihuega</poblacion>
<provincia>Brihuega</provincia>
<cif>777</cif>
<telefono>112233344 </telefono>
<ganancias>600000</ganancias>
</empresa>
```

```
</empresas>
```

empleadosemple.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<empleados>
```

```
  <empleado>
    <nombre>Juan</nombre>
    <apellidos>Ramirez</apellidos>
    <edad>45</edad>
    <salario>42000</salario>
    <cedula>123</cedula>
    <cifempresa>455</cifempresa>
```

```
  </empleado>
```

```
    <empleado>
      <nombre>Maria</nombre>
      <apellidos>Solis</apellidos>
      <edad>23</edad>
      <salario>22000</salario>
      <cedula>452</cedula>
      <cifempresa>222</cifempresa>
    </empleado>
```

```
    <empleado>
      <nombre>Alejandro</nombre>
      <apellidos>Ramirez</apellidos>
```

```
<edad>32</edad>
<salario>25000</salario>
<cedula>2547</cedula>
```

```
    <cifempresa>335</cifempresa>
</empleado>
```

```
    <empleado>
<nombre>Jenny </nombre>
<apellidos>Smith</apellidos>
<edad>42</edad>
<salario>50000</salario>
<cedula>54455</cedula>
</empleado>
```

```
    <empleado>
<nombre>Milena</nombre>
<apellidos>Marin</apellidos>
<edad>45</edad>
<salario>34000</salario>
<cedula>54534</cedula>
<cifempresa>335</cifempresa>
</empleado>
```

```
    <empleado>
<nombre>Alejandro</nombre>
<apellidos>Ramirez</apellidos>
<edad>29</edad>
<salario>60000</salario>
<cedula>2547</cedula>
    <cifempresa>222</cifempresa>
</empleado>
```

```
    <empleado>
<nombre>Luis</nombre>
<apellidos>Restrepo</apellidos>
<edad>39</edad>
<salario>33000</salario>

    <cedula>3534</cedula>
    <cifempresa>455</cifempresa>
</empleado>
```

```
    <empleado>
<nombre>Aura</nombre>
<apellidos>Navia</apellidos>
<edad>38</edad>
<salario>31000</salario>

    <cedula>754234</cedula>
```

```

    <cifempresa>222</cifempresa>
  </empleado>

  <empleado>
    <nombre>Milena</nombre>
    <apellidos>Marin</apellidos>
    <cedula>54534</cedula>
    <empresa>222</empresa>
    <edad>34</edad>
    <salario>35000</salario>

  </empleado>
</empleados>

```

Y vamos a ver diferentes ejemplos de consultas, explicando paso a paso el código. Lo primero es cargar los xml en XDocument. Podíamos haberlo hecho también en un XElement.

```

XDocument docEmple = XDocument.Load(@"C:\xmlcsharp\empleadosempre.xml");

XDocument docEmpre = XDocument.Load(@"C:\xmlcsharp\empresa.xml");

```

En una consulta con sintaxis de query seleccionamos empleados cuyo sueldo es mayor que 35000. Cogemos todos los empleados descendientes del nodo raíz de tipo empleado `docEmple.Root.Descendants("empleado")`. Nos quedamos con aquellos cuyo salario es mayor que 35000 con la clausula where, `double.Parse(empl.Element("salario").Value.ToString()) > 35000`. Para ello accedemos al valor del nodo “salario” y lo convertimos a double.

```

var emple1 = from empl in docEmple.Root.Descendants("empleado")
              where double.Parse(empl.Element("salario").Value.ToString()) > 35000
              select empl;

```

Las consultas en LINQ para xml son un poco más costosas porque requieren acceder a los nodos y luego a sus valores. Ya sabéis que se puede realizar la misma consulta con la sintaxis LINQ de métodos y expresiones lambda.

```

var emple = docEmple.Root.Descendants("empleado").Where(empl=>
double.Parse(empl.Element("salario").Value.ToString())>35000);

```

La siguiente es una consulta de agregación en la que calculamos la media de los salarios de los empleados con el método Average. `Average(empl => double.Parse(empl.Element("salario").Value.ToString()))`;

Otra vez tenemos que transformar el valor del nodo salario a double, antes de calcular la media. Al método Average se le pasa un delegado en forma de expresión lambda que proporciona el campo por el que realizaremos la media.

```
double media = docEmple.Root.Descendants("empleado").Average(emp =>
double.Parse(emp.Element("salario").Value.ToString()));

Console.WriteLine("La media de salarios es {0,10:#.000} ", media);
//Maxima edad

int edad = docEmple.Root.Descendants("empleado").Max(emp =>
int.Parse(emp.Element("edad").Value.ToString()));
```

Con dos queries obtenemos el empleado con el salario más alto. Podríamos haberlo hecho en una query anidando una dentro de otra, pero lo hacemos en dos por dar mayor claridad. Con el método Max calculamos el salario máximo de todos los empleados. Para ello pasamos una delegada que proporciona el campo sobre el que se calculará el máximo, salario

```
double salario = docEmple.Root.Descendants("empleado").Max(emp =>
double.Parse(emp.Element("salario").Value.ToString()));
```

De todos los empleados con la clausula where nos quedamos con aquellos que coinciden con el salario máximo, obtenido en la Query anterior.

```
var empSalarioMaxMethod = docEmple.Root.Descendants("empleado").Where(emp =>
double.Parse(emp.Element("salario").Value.ToString())==salario);
```

La version de sintaxis de método quedaría como sigue:

```
var empSalarioMaxMethod = docEmple.Root.Descendants("empleado").Where(emp =>
double.Parse(emp.Element("salario").Value.ToString())==salario);
```

La ultima Query de este ejemplo une los dos XML, el de empleados y empresas. Nos vamos a quedar los empleados que tienen un cifempresa, es decir, que trabajan para una empresa y además que ese cif aparece en el xml empresas.xml. Por ejemplo:

```
<empleado>
  <nombre>Alejandro</nombre>
  <apellidos>Ramirez</apellidos>
  <edad>29</edad>
  <salario>60000</salario>
  <cedula>2547</cedula>
  <cifempresa>222</cifempresa>
</empleado>
```

```

<empresa tipo="tecnologia">

<nombre>Equipos Digitales S.L.</nombre>
<dir>Av. Valladolid</dir>
<poblacion cod_postal="28043">Madrid</poblacion>
<provincia>Madrid</provincia>
<cif>222</cif>
<telefono/>
<ganancias>200000</ganancias>

</empresa>

```

El ejemplo va desarrollado bajo sintaxis de método. Hacemos un Join para unir docEmple y docEmpre, sobre el campo cifempresa y cif, y nos quedamos con el empleado.

```

var empleadoConEmpresaMethod =
docEmple.Root.Descendants("empleado").Join(docEmpre.Root.Descendants("empresa"),
    emp => emp.Element("cifempresa").Value, empre=> empre.Element("cif").Value,
    (emp,empre)=> emp);

```

Ejercicio

1. Intentar transformar este ejemplo a consulta con sintaxis de query. Es bastante sencillo
2. Modificar el ejemplo para crear un objeto anónimo que en el resultado final nos quedemos con el nombre, los apellidos del empleado, y el nombre de la empresa.

Ejemplo completo

```

using System;
using System.Linq;
using System.Xml.Linq;

namespace ConsultasLINQXML
{
    class Program
    {
        static void Main(string[] args)
        {
            XDocument docEmple = XDocument.Load(@"C:\xmlcsharp\empleadosempre.xml");
            XDocument docEmpre = XDocument.Load(@"C:\xmlcsharp\empresa.xml");

            //Empleados con salario mayor de 35000

            Console.WriteLine("Empleados con salarios mayor de 35000");

            var emple1 = from empl in docEmple.Root.Descendants("empleado")
                          where double.Parse(empl.Element("salario").Value.ToString()) > 35000
                          select empl;

```

```

        var emple = docEmple.Root.Descendants("empleado").Where(emp=>
double.Parse(emp.Element("salario").Value.ToString())>35000);

        foreach( var empleado in emple)
        {

            Console.WriteLine(empleado);

        }

        //Media de salarios

        double media = docEmple.Root.Descendants("empleado").Average(emp =>
double.Parse(emp.Element("salario").Value.ToString()));

        Console.WriteLine("La media de salarios es {0,10:#.000} ", media);
        //Maxima edad

        int edad = docEmple.Root.Descendants("empleado").Max(emp =>
int.Parse(emp.Element("edad").Value.ToString()));

        Console.WriteLine("La maxima edad es {0}", edad);

        // Empleado con mayor salario

        double salario = docEmple.Root.Descendants("empleado").Max(emp =>
double.Parse(emp.Element("salario").Value.ToString()));

        var empSalarioMaxMethod = docEmple.Root.Descendants("empleado").Where(emp =>
double.Parse(emp.Element("salario").Value.ToString())==salario);

        var empSalarioMaxQuery = (from emp in docEmple.Root.Descendants("empleado")
                                where
double.Parse(emp.Element("salario").Value.ToString()) == salario
                                select emp);

        foreach (var emp in empSalarioMaxMethod) {

            Console.WriteLine("Empleado mayor salario method syntax {0}",emp);

        }

        foreach (var emp in empSalarioMaxQuery)
        {

            Console.WriteLine("Empleado mayor salario query syntax {0} ", emp);

        }

        //Empleados en empresas agrupados por empresas con Join

        var empleadoConEmpresaMethod =
docEmple.Root.Descendants("empleado").Join(docEmple.Root.Descendants("empresa"),
        emp => emp.Element("cifempresa").Value, empre=> empre.Element("cif").Value,
        (emp,empre)=> emp);

```

```
}  
    }  
}
```

4 Entity Framework

Entity Framework Core (EF Core) es un asignador de base de datos de objeto moderno para .NET, lo que se conoce como ORM. Admite consultas LINQ, seguimiento de cambios, actualizaciones y migraciones de esquemas.

Entity Framework (EF) Core es una versión ligera, extensible, de código abierto y multiplataforma de la popular tecnología de acceso a datos Entity Framework.

EF Core puede actuar como asignador relacional de objetos, que se encarga de lo siguiente:

- Permite a los desarrolladores de .NET trabajar con una base de datos usando objetos .NET.
- Permite prescindir de la mayor parte del código de acceso a datos que normalmente es necesario escribir.
- EF Core es compatible con muchos motores de base de datos; vea Proveedores de bases de datos para más información.

4.1 *Que se un ORM (Objet Relational Model)*

Java: cómo comprobar si existe o no un archivo o una carpeta en el disco duro | 5 razones que convencerán a tu jefe que debes teletrabajar

¿Qué es un ORM?

Por José Manuel Alarcón

Domina C# 9 y .NET 5: Este es el curso que necesitas

Tradicionalmente, para realizar acceso a datos desde un lenguaje orientado a objetos (POO) como pueden ser .NET o Java, era necesario mezclar código y conceptos muy diferentes.

Por un lado, teníamos la aplicación en sí, escrita en alguno de estos lenguajes como C# o Java. Dentro de este programa hacíamos uso de ciertos objetos especializados en conectarse con bases de datos y lanzar consultas contra ellas. Estos objetos de tipo Connection, Query y similares, eran en realidad conceptos de la base de datos llevados a un programa orientado a objetos que no tenía nada que ver con ellos. Finalmente, para lanzar consultas (tanto de obtención de datos como de modificación o de gestión) se introducían instrucciones en lenguaje SQL, en forma de cadenas de texto, a través de estos objetos.

Además, estaba el problema de los tipos de datos. Generalmente el modo de representarlos y sus nombres pueden variar entre la plataforma de desarrollo y la base de datos. Por ejemplo, en .NET un tipo de datos que almacena texto es simplemente un string. En SQL Server para lo mismo podemos utilizar cadenas de longitud fija o variable, de tipo Unicode

o ANSI, etc... Y lo mismo con los otros tipos de datos. Algunos ni siquiera tienen por qué existir en el lenguaje de programación. Y tampoco debemos olvidarnos de los valores nulos en la base de datos, que pueden causar todo tipo de problemas según el soporte que tengan en el lenguaje de programación con el que nos conectamos a la base de datos.

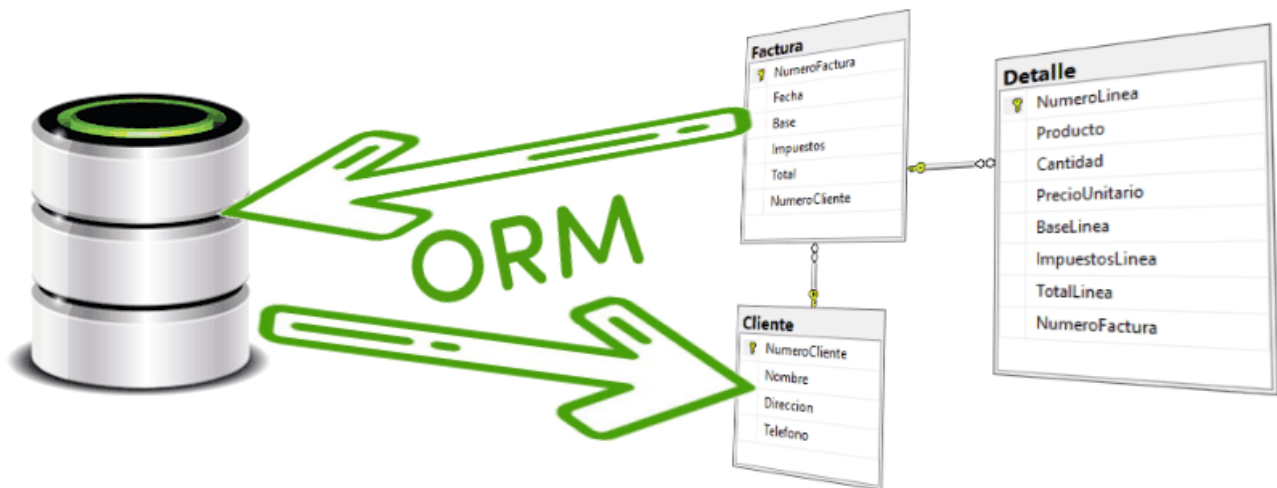
Otros posibles problemas y diferencias surgen del modo de pensar que tenemos en un lenguaje orientado a objetos y una base de datos. En POO, por ejemplo, para representar una factura y sus líneas de factura podrías definir un objeto Factura con una propiedad Lineas, y si quieres consultar las líneas de una factura solo debes escribir **factura.Lineas** y listo, sin pensar en cómo se relacionan o de dónde sale esa información. En una base de datos, sin embargo, esto se modela con dos tablas diferentes, una para las facturas y otra para las líneas, además de ciertos índices y claves externas entre ellas que relacionan la información. Si además se diese el caso de que una misma línea de factura pudiese pertenecer a más de una factura, necesitas una tercera tabla intermedia que se relaciona con las dos anteriores y que hace posible localizar la información. Como vemos formas completamente diferentes de pensar sobre lo mismo.

La complejidad no termina aquí ya que las bases de datos tienen procedimientos almacenados (que son como pequeños programas especializados que se ejecutan dentro de la base de datos), transacciones, y otros conceptos que desde el punto de vista de un lenguaje POO son completamente extraterrestres y ajenos.

Estas diferencias entre conceptos, tipos de datos, y modos de trabajar pueden causar muchos problemas de lo que se dio en llamar "desfase de impedancia" o "impedance mismatch" en inglés, en una clara analogía los circuitos eléctricos y al flujo eléctrico (en este caso aplicado a flujo de información). El concepto se refiere a la dificultad para hacer fluir la información entre la base de datos y las diferentes capas del programa en función de la diferencia existente entre cada una de estas partes.

Este desfase de impedancia hace que pueda llegar a ser muy complicado trabajar contra una base de datos desde un lenguaje POO si queremos sacar partido a los conceptos habituales que usamos en éstos, y huir de bibliotecas de funciones que nos fuerzan a trabajar con los conceptos de la base de datos.

4.1.1 Mapeadores



Como acabamos de ver, lo ideal en una aplicación escrita en un lenguaje orientado a objetos sería definir una serie de clases, propiedades de éstas que hagan referencia a las otras, y trabajar de modo natural con ellas.

¿Qué quieres una factura? Simplemente instancias un objeto Factura pasándole su número de factura al constructor. ¿Necesitas sus líneas de detalle? LLamas a la propiedad Lineas del objeto. ¿Quieres ver los datos de un producto que está en una de esas líneas? Solo lee la propiedad correspondiente y tendrás la información a tu alcance. Nada de consultas, nada de relaciones, de claves foráneas...

En definitiva, nada de conceptos "extraños" de bases de datos en tu código orientado a objetos. En la práctica para ti la base de datos es como si no existiera.

Eso precisamente es lo que intenta conseguir el software llamado ORM, del inglés Object-Relational Mapper o "Mapeador" de relacional a objetos (y viceversa). Un ORM es una biblioteca especializada en acceso a datos que genera por ti todo lo necesario para conseguir esa abstracción de la que hemos hablado.

Gracias a un ORM ya no necesitas utilizar SQL nunca más, ni pensar en tablas ni en claves foráneas. Sigues pensando en objetos como hasta ahora, y te olvidas de lo que hay que hacer "por debajo" para obtenerlos.

El ORM puede generar clases a partir de las tablas de una base de datos y sus relaciones, o hacer justo lo contrario: partiendo de una jerarquía de clases crear de manera transparente las entidades necesarias en una base de datos, ocupándose de todo (ni tendrás que tocar el gestor de bases de datos para nada).

Mapeadores disponibles para los diferentes lenguajes.

Qué ORMs tenemos disponibles?

En el mundo Java el ORM más conocido y utilizado es Hibernate que pertenece a Red Hat aunque es gratuito y Open Source. Hay muchos otros como Jooq, ActiveJDBC que trata de emular los Active Records de Ruby On Rails, o QueryDSL, pero en realidad ninguno llega ni por asomo al nivel de uso de Hibernate. Si necesitas un ORM en Java debes aprender Hibernate, y luego ya si quieres te metes con otro, pero este es indispensable.

En la plataforma .NET tenemos varios conocidos, pero el más popular y utilizado es Entity Framework o EF, que es el creado por la propia Microsoft y que viene incluido en la plataforma .NET (tanto en la "tradicional" como en .NET Core). También existe un "port" de Hibernate para .NET llamado NHibernate y que a mucha gente le gusta más que EF. Hay otros como Dapper que está creado por la gente de Stack Exchange y es mucho más sencillo que los anteriores, lo cual es considerado una gran virtud por mucha gente (entre los que me incluyo), y también es muy utilizado. Y es muy conocido Subsonic, que lleva casi diez años en activo pero que puede llegar a ser bastante complicado (a mí personalmente no me gusta nada).

En PHP tienes Doctrine, tal vez el más conocido y recomendado (utilizado por el framework Symfony), pero también se usan bastante Propel, RedbeanPHP y uno muy popular pero ya en desuso es Xyster (pero te lo encontrarás aún bastante por ahí).

En Python el hiper-conocido framework Django (así sinónimo de desarrollo web con este lenguaje) incluye su propio ORM, pero también tenemos SQLAlchemy por el que muchos beben los vientos y lo califican como el mejor ORM jamás hecho (no tengo experiencia con él como para saberlo). También están Peewee o Pony ORM entre otros.

Prácticamente todas las plataformas tienen el suyo, así que búscalos para la tuya y mira cuál es el más popular y el que más comunidad reúne.

Ventajas e inconvenientes de un ORM

Los ORM ofrecen enormes ventajas, como ya hemos visto, al reducir esa "impedancia" que impide el buen flujo de información entre los dos paradigmas POO-Relacional. Pero además:

- No tienes que escribir código SQL, algo que muchos programadores no dominan y que es bastante complejo y propenso a errores. Ya lo hacen por nosotros los ORM.
- Te dejan sacar partido a las bondades de la programación orientada a objetos, incluyendo por supuesto la herencia, lo cual da mucho juego para hacer cosas.
- Nos permiten aumentar la reutilización del código y mejorar el mantenimiento del mismo, ya que tenemos un único modelo en un único lugar, que podemos reutilizar en toda la aplicación, y sin mezclar consultas con código o mantener sincronizados los cambios de la base de datos con el modelo y viceversa.
- Mayor seguridad, ya que se ocupan automáticamente de higienizar los datos que llegan, evitando posibles ataques de inyección SQL y similares.
- Hacen muchas cosas por nosotros: desde el acceso a los datos (lo obvio), hasta la conversión de tipos o la internacionalización del modelo de datos.

Pero no todo va a ser alegría. También tienen sus inconvenientes:

- Para empezar pueden llegar a ser muy complejos. Por ejemplo, NHibernate tiene ya más de medio millón de líneas de código ahora mismo, y muchas clases y detalles que hay que saber. Eso hace que su aprendizaje sea complejo en muchos casos, y hay que invertir tiempo en aprenderlos y practicar con ellos hasta tener seguridad en su manejo diario.
- No son ligeros por regla general: añaden una capa de complejidad a la aplicación que puede hacer que empeore su rendimiento, especialmente si no los dominas a fondo. En la mayor parte de las aplicaciones probablemente no te importe, pero en ocasiones su impacto en el rendimiento es algo a tener muy en cuenta. En general una consulta SQL directa será más eficiente siempre.
- La configuración inicial que requieren se puede complicar dependiendo de la cantidad de entidades que se manejen y su complejidad, del gestor de datos subyacente, etc...
- El hecho de que te aísle de la base de datos y no tengas casi ni que pensar en ella es un arma de doble filo. Si no sabes bien lo que estás haciendo y las implicaciones que tiene en el modelo relacional puedes construir modelos que generen consultas monstruosas y muy poco óptimas contra la base de datos, agravando el problema del rendimiento y la eficiencia.

En definitiva, los ORM son una herramienta que puede llegar a ser maravillosa, pero que en aplicaciones pequeñas pueden ser como "matar moscas a cañonazos". En aplicaciones más grandes donde el mantenimiento y la homogeneidad sean importantes, son indispensables. Eso sí, no te eximen de aprender bien el lenguaje SQL o las maneras más tradicionales de realizar acceso a datos, ya que conocerlas puede marcar la diferencia cuando surjan problemas o haya que determinar por qué se produce una merma de rendimiento, etc...

4.2 Paquetes NuGet

Una herramienta esencial para cualquier plataforma de desarrollo moderno, es un mecanismo a través del cual los desarrolladores pueden crear, compartir y consumir código útil. A menudo, este código se integra en "paquetes" que contienen código compilado (como archivos DLL) y otro contenido necesario en los proyectos que utilizan estos paquetes.

En .NET (incluido .NET Core), el mecanismo compatible con Microsoft para compartir código es NuGet, que define cómo se crean, hospedan y consumen paquetes en .NET, y ofrece las herramientas para cada uno de esos roles.

Desde un punto de vista sencillo, un paquete NuGet es un archivo ZIP con la extensión .nupkg que contiene código compilado (archivos DLL), otros archivos relacionados con ese código y un manifiesto descriptivo que incluye información como el número de versión del paquete. Los programadores con código para compartir crean paquetes y los publican en un host público o privado. Los consumidores de paquetes obtienen esos paquetes de los hosts adecuados, los agregan a sus proyectos y, después, llaman a la funcionalidad de un paquete

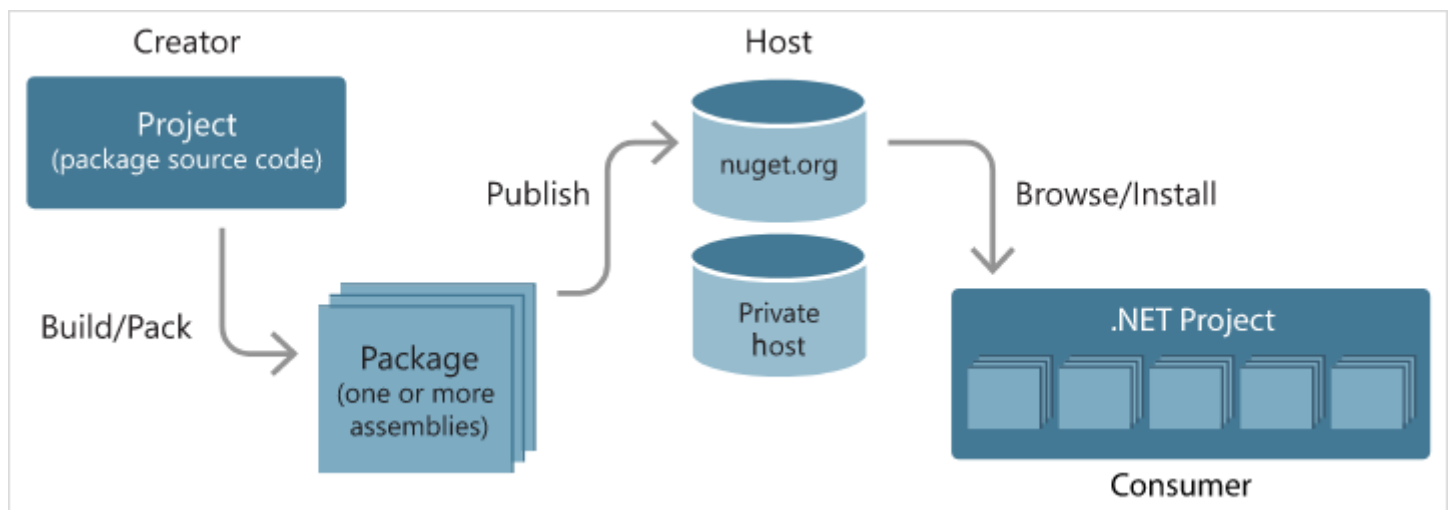
en el código del proyecto. Después, el propio NuGet controla todos los detalles intermedios.

Dado que NuGet admite hosts privados junto al host de nuget.org público, puede usar paquetes NuGet para compartir código que es exclusivo de una organización o un grupo de trabajo. También puede utilizar paquetes NuGet como una manera cómoda de tener su propio código para usarlo nada más que en sus propios proyectos. En resumen, un paquete NuGet es una unidad de código que se puede compartir, pero no requiere ni implica ningún medio concreto de uso compartido.

4.2.1 El flujo de paquetes entre creadores, hosts y consumidores

En su rol de host público, NuGet mantiene el repositorio central de más de 100 000 paquetes únicos en nuget.org. Millones de desarrolladores de .NET/.NET Core usan estos paquetes a diario. NuGet también permite hospedar paquetes de forma privada en la nube (como en Azure DevOps), en una red privada o incluso en el sistema de archivos local. Así, los paquetes solo están a disposición de los desarrolladores que tengan acceso al host, lo que le ofrece la posibilidad de poner paquetes a disposición de un grupo determinado de consumidores. Las opciones se explican en Hospedar sus propias fuentes de NuGet. Mediante opciones de configuración, puede también controlar exactamente qué hosts pueden tener acceso a un equipo determinado, lo que garantiza que los paquetes se obtienen de orígenes específicos y no de un repositorio público como nuget.org.

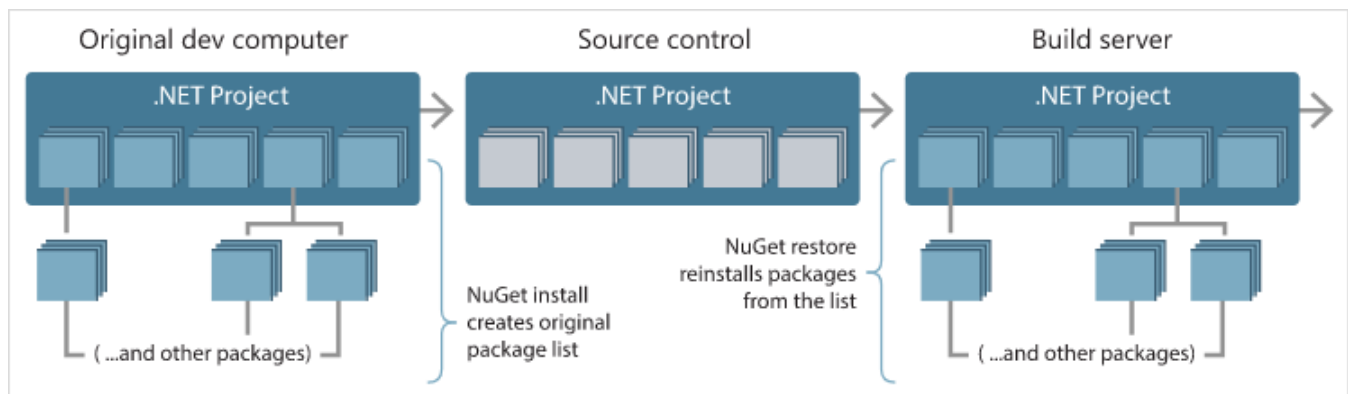
Con independencia de su naturaleza, un host actúa como un punto de conexión entre los *creadores* y los *consumidores* de paquetes. Los creadores compilan paquetes NuGet útiles y los publican en un host. Después, los consumidores buscan paquetes útiles y compatibles en hosts accesibles, los descargan y los incluyen en sus proyectos. Una vez instalados en un proyecto, las API de los paquetes están disponibles para el resto del código del proyecto.



4.2.2 Compatibilidad de destino de paquetes

Un paquete "compatible" implica que contiene ensamblados compilados para al menos una plataforma .NET de destino que es compatible con la plataforma de destino del proyecto de consumo. Los desarrolladores pueden crear paquetes que son específicos de una plataforma, como ocurre con los controles UWP, o pueden admitir una gama más amplia de destinos. Para maximizar la compatibilidad de un paquete, los desarrolladores lo destinan a .NET Standard, que todos los proyectos de .NET y .NET Core pueden consumir. Se trata del medio más eficaz tanto para los creadores como para los consumidores, ya que un único paquete (que normalmente contiene un único ensamblado) funciona para todos los proyectos de consumo.

Los desarrolladores de paquetes que requieren las API fuera de .NET Standard, por otra parte, crean ensamblados independientes para las diferentes plataformas de destino que van a admitir e incluyen todos los ensamblados en el mismo paquete (lo que se denomina "compatibilidad con múltiples versiones"). Cuando un consumidor instala ese paquete, NuGet extrae solo los ensamblados que son necesarios para el proyecto. Esto reduce la superficie del paquete en la aplicación final y los ensamblados que produce ese proyecto. Obviamente, un paquete de compatibilidad con múltiples versiones resulta más difícil de mantener para su creador.



Solo con la lista de referencias, NuGet puede reinstalar (es decir, restaurar) todos los paquetes de hosts públicos y privados en cualquier momento posterior. Al confirmar un proyecto en el control de código fuente o compartirlo de alguna otra manera, solo se incluye la lista de referencias, no los archivos binarios del paquete (vea Paquetes y control de código fuente).

El equipo que recibe un proyecto, como un servidor de compilación que obtiene una copia del proyecto como parte de un sistema de implementación automatizada, simplemente solicita a NuGet que restaure las dependencias cuando sea necesario. Los sistemas de compilación como Azure DevOps proporcionan pasos de "restauración de NuGet" para este propósito exacto. De forma similar, cuando los desarrolladores obtienen una copia de un proyecto (como al clonar un repositorio), pueden invocar un comando como `nuget restore` (CLI de NuGet), `dotnet restore` (CLI de dotnet), o `Install-Package` (consola del Administrador de paquetes) para obtener todos los paquetes necesarios. Visual Studio, por su parte, restaura automáticamente los paquetes al compilar un proyecto (siempre que la restauración

automática esté habilitada, tal y como se describe en Restauración de paquetes).

Claramente, el rol principal de NuGet que interesa a los desarrolladores es que mantenga esa lista de referencias en nombre del proyecto y que proporcione los medios para restaurar de forma eficaz (y actualizar) los paquetes a los que se hace referencia. Esta lista se mantiene en uno de los dos formatos de administración de paquetes, que se denominan:

PackageReference (o "referencias de paquetes en archivos de proyecto") | (NuGet 4.0 y versiones posteriores) mantiene una lista de las dependencias de nivel superior de un proyecto directamente en el archivo de proyecto, por lo que no se necesita un archivo independiente. Se genera dinámicamente un archivo asociado, `obj/project.assets.json`, que administra el gráfico de dependencias general de los paquetes que un proyecto utiliza con todas las dependencias de nivel inferior. Siempre se utiliza PackageReference en los proyectos de .NET Core.

packages.config: (NuGet 1.0 y versiones posteriores) un archivo XML que mantiene una lista plana de todas las dependencias del proyecto, incluidas las dependencias de otros paquetes instalados. Los paquetes instalados o restaurados se almacenan en una carpeta `packages`.

El formato de administración de paquetes que se usa en un proyecto determinado depende del tipo de proyecto y la versión disponible de NuGet (y/o Visual Studio). Para comprobar qué formato se usa, solo hay que buscar `packages.config` en la raíz del proyecto después de instalar el primer paquete. Si no ve ese archivo, busque directamente un elemento `<PackageReference>` en el archivo de proyecto.

Si se puede elegir, se recomienda utilizar PackageReference. `packages.config` se mantiene con fines de herencia y ya no está en desarrollo activo.

4.2.3 ¿Qué más hace NuGet?

Hasta ahora ha aprendido las siguientes características de NuGet:

NuGet ofrece el repositorio central `nuget.org` con compatibilidad de hospedaje privado. NuGet proporciona a los desarrolladores las herramientas que necesitan para crear, publicar y consumir paquetes.

Y lo más importante, NuGet mantiene una lista de referencias de los paquetes que se usan en un proyecto y permite restaurar y actualizar los paquetes de esa lista.

Para que estos procesos funcionen de forma eficaz, NuGet realiza algunas optimizaciones en segundo plano. En concreto, NuGet administra una caché de paquetes y una carpeta de paquetes globales para abreviar la instalación y reinstalación. La caché evita descargar un paquete que ya se ha instalado en el equipo. La carpeta de paquetes globales permite que varios proyectos compartan el mismo paquete instalado, lo que reduce el consumo general de NuGet en el equipo. Las carpetas de paquetes globales y de caché resultan muy útiles cuando a menudo se restaura un mayor número de paquetes, por ejemplo, en un servidor de compilación. Para obtener más detalles sobre estos mecanismos, vea Administración de paquetes globales y carpetas de caché.

Dentro de un proyecto individual, NuGet administra el gráfico general de dependencias, que

incluye volver a resolver varias referencias a las distintas versiones del mismo paquete. Es bastante común que un proyecto tenga una relación de dependencia con uno o varios paquetes que, a su vez, tienen las mismas dependencias. Algunos de los paquetes de utilidad más prácticos de nuget.org se usan en otros muchos paquetes. En el gráfico de dependencias completo, podría tener fácilmente diez referencias distintas a versiones diferentes del mismo paquete. Para no incluir varias versiones de ese paquete en la propia aplicación, NuGet determina la única versión que pueden usar todos los consumidores. (Para obtener más información, vea Inserción de dependencias).

Además, NuGet mantiene todas las especificaciones relacionadas con la estructura de los paquetes (incluida la localización y los símbolos de depuración) y cómo se hace referencia a ellos (incluidos los intervalos de versiones y las versiones preliminares). NuGet ofrece también varias API para trabajar con sus servicios mediante programación, así como compatibilidad para los desarrolladores que crean plantillas de proyecto y extensiones de Visual Studio.

Dedique un momento a examinar la tabla de contenido de esta documentación, y podrá ver todas estas funcionalidades representadas, junto con notas de la versión que se remontan a los inicios de NuGet.

4.2.4 Herramientas para usar NuGet.

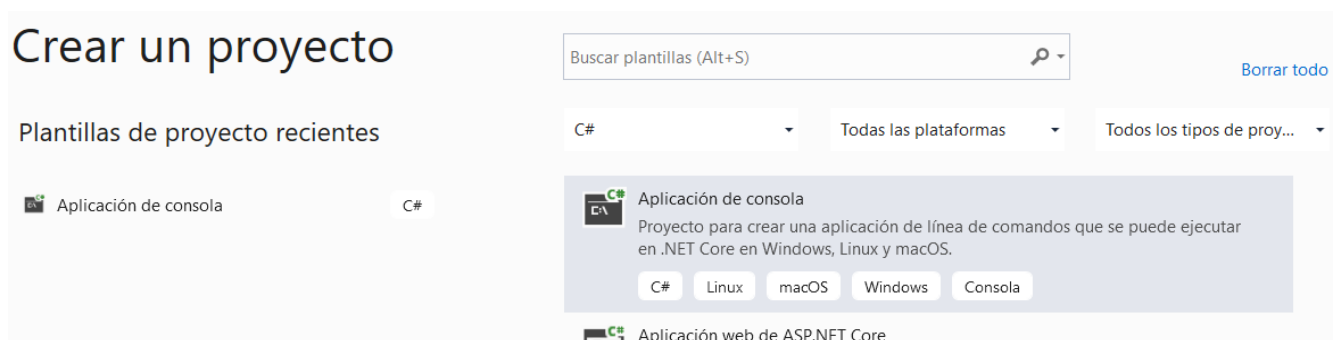
Hay un montón de herramientas para realizar la configuración e instalación de proyectos .Net y paquetes. Nosotros vamos a trabajar en entorno Microsoft con Visual Studio, y podemos usar la herramienta que lleva asociada para la gestión de paquetes NuGet, llamada Consola del Administrador de paquetes. En la siguiente tabla se ofrecen las posibles herramientas para trabajar con este entorno de paquetes.

CLI de dotnet	Todas	Creación, Consumo	Herramienta CLI para las bibliotecas .NET Core y .NET Standard y para los proyectos de estilo SDK que tienen como destino .NET Framework (consulte Atributo SDK). Ofrece determinadas funcionalidades de la CLI de NuGet directamente en la cadena de herramientas de .NET Core. Al igual que con la CLI de nuget.exe, la CLI de dotnet no interactúa con proyectos de Visual Studio.
CLI de nuget.exe	Todas	Creación, Consumo	Herramienta CLI para bibliotecas de .NET Framework y proyectos de estilo diferente de SDK que tienen como destino las bibliotecas de .NET Standard. Proporciona todas las funcionalidades de NuGet, con algunos comandos que se aplican de forma específica a los creadores del paquete, otros solo a los consumidores y otros a ambos. Por ejemplo, los creadores de paquetes usan el comando nuget pack para crear un paquete a partir de varios ensamblados y archivos relacionados, los consumidores de paquetes usan nuget install para incluir los paquetes en una carpeta de proyecto y todos

			usan nuget config para establecer variables de configuración de NuGet. Como herramienta independiente de la plataforma, la CLI de NuGet no interactúa con proyectos de Visual Studio.
Consola del Administrador de paquetes	Visual Studio en Windows	Consumo	Ofrece comandos de PowerShell para instalar y administrar paquetes en proyectos de Visual Studio.
Interfaz de usuario del administrador de paquetes	Visual Studio en Windows	Consumo	Ofrece una interfaz de usuario fácil de usar para instalar y administrar paquetes en proyectos de Visual Studio.
Administrar la interfaz de usuario de NuGet	Visual Studio para Mac	Consumo	Ofrece una interfaz de usuario fácil de usar para instalar y administrar paquetes en proyectos de Visual Studio para Mac.
MSBuild	Windows	Creación, Consumo	Ofrece la posibilidad de crear y restaurar los paquetes que se usan en un proyecto directamente a través de la cadena de herramientas de MSBuild.

4.3 Instalación de Entity Framework en los proyectos Visual Studio. Primer proyecto

Abrimos nuestro Visual Studio y crearemos un nuevo proyecto ModeloEntityFramework sencillo de consola para .Net 5.



Configure su nuevo proyecto

Aplicación de consola

C#

Linux

macOS

Windows

Consola

Nombre del proyecto

ModeloEntityFrameworkSencillo

Aplicación de consola

C#

Linux

macOS

Windows

Consola

Plataforma de destino ⓘ

.NET 5.0 (Actual)

Atrás

Crear

Creamos el proyecto y nuestro siguiente paso será instalar el EntityFramework para la base de datos que queremos crear. Empezaremos con una base de datos sencilla SQLite. Pero antes vamos a crear el modelo que vamos a usar para crear la base de datos. Vamos a usar el procedimiento que Microsoft llama Code First, primero se hace el modelo de datos en código luego se migrará o importa , creando la base de datos de manera automática.

4.3.1 Modelo de clases

Añadiremos la clase Empleado para empezar. Que será nuestro modelo de datos, lo que conocemos como modelo en programación orientada a objetos que representa la realidad sobre la que queremos crear una aplicación.

Nota: Muy importante es señalar aquí que la clase Empleado debe tener un constructor sin atributos. Los ORM suelen construir objetos vacíos y rellenar los datos que traemos de base de datos con setters. Es decir si el campo nombre no es null para colocar el valor que esta

en base de datos para ese empleado, Entity Framework hace un new de Empleado sin parámetros Empleado e= new Empleado(); y luego e.Nombre("Antonio") escribe el nombre traído de base de datos que puede ser Antonio. Constructor sin parámetros: `public Empleado()`

Igualmente, todas las propiedades que queramos guardar en base de datos de un objeto deben tener getter y setter accesible, para poder leer de ellas y escribir en ellas. No tenemos porque guardar , mapear todas las propiedades del objeto en base de datos.

La clase Persona

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace ModeloEntityFrameworkSencillo
{
    public class Empleado: IComparable<Empleado>
    {
        public string Id { get; set; }

        public String Apellido { get; set; }

        public String Nombre { get; set; }

        public String Direccion { get; set; }

        public int Edad { get; set; }
        public double Salario { get; set; }

        public Empleado()
        {
        }

        public Empleado(string id, string apellido, string nombre, string direccion, int edad,
double salario)
        {
            Id = id;
            Apellido = apellido;
            Nombre = nombre;
            Direccion = direccion;
            Edad = edad;
            Salario = salario;
        }

        public override string ToString()
        {
            return String.Format("Apellidos: {0}, Nombre: {1}, Direccion: {2}, Edad: {3:d},
Salario: {4} ", this.Apellido, this.Nombre, this.Edad, this.Direccion, this.Salario);
        }

        public int CompareTo(Empleado other)
        {
            return this.CompareTo(other);
        }
    }
}
```

```

        public override bool Equals(object obj)
        {
            return obj is Empleado empleado &&
                Id == empleado.Id &&
                Apellido == empleado.Apellido &&
                Nombre == empleado.Nombre &&
                Direccion == empleado.Direccion &&
                Edad == empleado.Edad &&
                Salario == empleado.Salario;
        }

        public override int GetHashCode()
        {
            return GetHashCode.Combine(Id, Apellido, Nombre, Direccion, Edad, Salario);
        }
    }
}

using System;
using System.Collections.Generic;
using System.Text;

namespace ModelEntityFrameworkSencillo.Debug
{
    class GeneraColeccionesAleatorias
    {
        public static List<Empleado> ListaEmpleados(int numEmpleados)
        {
            List<Empleado> listaEmpleados = new List<Empleado>();

            Random r = new Random();

            for (int i= 0; i < numEmpleados; i++ )
            {
                Empleado emp=null;

                emp = new Empleado(i.ToString(),
                                    GenerarCamposAleatorios.apellidosAleatorio(),
                                    GenerarCamposAleatorios.nombreAleatorio(),
                                    GenerarCamposAleatorios.direccionAleatoria(),
                                    GenerarCamposAleatorios.edadAleatoria(),
                                    GenerarCamposAleatorios.SalarioAleatorio());

                listaEmpleados.Add(emp);
            }

            return listaEmpleados;
        }
    }
}

```


Añadimos dos clases para generar Empleados con datos aleatorios para poder llenar la base de datos.

```
using System;
using System.Collections.Generic;
using System.Text;

namespace ModeloEntityFrameworkSencillo.Debug
{
    class GenerarCamposAleatorios
    {
        public static string[] nombres = {"ANTONIO", "MANUEL", "JOSE", "FRANCISCO", "DAVID",
"JUAN", "JOSE ANTONIO", "JAVIER", "DANIEL", "JOSE LUIS",
"FRANCISCO JAVIER", "CARLOS", "JESUS", "ALEJANDRO", "FRANCISCA", "LUCIA", "MARIA
ISABEL", "MARIA JOSE", "ANTONIA", "DOLORES", "SARA",
"PAULA", "ELENA", "MARIA LUISA", "RAQUEL"};

        public static string[] apellidos = {"García", "González", "Rodríguez", "Fernández",
"López", "Martínez", "Sánchez", "Pérez", "Gómez",
"Martín", "Jiménez", "Ruiz", "Hernández", "Díaz", "Moreno", "Muñoz", "Álvarez", "Romero",
"Alonso", "Gutiérrez", "Navarro",
"Torres", "Domínguez", "Vázquez", "Ramos", "Gil", "Ramírez", "Serrano", "Blanco", "Molina",
"Morales", "Suarez", "Ortega",
"Delgado", "Castro", "Ortiz", "Rubio", "Marín", "Sanz", "Núñez", "Iglesias", "Medina",
"Garrido", "Cortes", "Castillo", "Santos"};

        public static string[] direcciones = {"Alfonso López de Haro", "Calle de Alvarfáñez de
Minaya", "Calle de Arcipreste de Hita",
"Calle de Arrabal del Agua", "Plaza de Beladiez", "Plaza de Caídos en la Guerra civil", "Calle
Minaya", "Calle Hermanos Galiano"};

        private static Random rand = new Random();

        private static Func<int, int, int> randomFromTo = (ini, fin) => rand.Next() % (fin-
ini) +ini;

        private static Func<int, int, double> randomFromToDouble = (ini, fin) =>
rand.NextDouble() * (fin - ini) + ini;

        public static string nombreAleatorio()
        {
            return nombres[randomFromTo(0, nombres.Length - 1)];
        }

        public static string apellidosAleatorio()
        {
            return apellidos[randomFromTo(0, apellidos.Length - 1)] + " " +
apellidos[randomFromTo(0, apellidos.Length - 1)];
        }

        public static DateTime fechaNacimientoAleatoria()
        {
            return new DateTime(randomFromTo(1960, 2001), randomFromTo(1, 12), randomFromTo(1,
28));
        }
    }
}
```

```

    }

    public static string direccionAleatoria()
    {
        return direcciones[randomFromTo(0, direcciones.Length - 1)];
    }

    public static int edadAleatoria()
    {
        return randomFromTo(18, 67);
    }
    public static double SalarioAleatorio()
    {
        return randomFromToDouble(600, 4000);
    }
}
}

```

```

using System;
using System.Collections.Generic;
using System.Text;

```

```

namespace LinqConsultas.Debug

```

```

{
    class GeneraColeccionesAleatorias
    {

```

```

        public static List<Empleado> ListaEmpleados(int numEmpleados)
        {

```

```

            List<Empleado> listaEmpleados = new List<Empleado>();

```

```

            Random r = new Random();

```

```

            for (int i= 0; i < numEmpleados; i++ )
            {

```

```

                Empleado emp=null;

```

```

                switch (r.Next()%2)
                {

```

```

                    case 0:

```

```

                        emp = new Administrativo(GenerarCamposAleatorios.apellidosAleatorio(),
                                                GenerarCamposAleatorios.nombreAleatorio(),

```

```

                                                GenerarCamposAleatorios.fechaNacimientoAleatoria(),
                                                GenerarCamposAleatorios.direccionAleatoria(),
                                                GenerarCamposAleatorios.SalarioAleatorio());

```

```

                        break;

```

```

                    case 1:

```

```

        emp = new Ejecutivo(GenerarCamposAleatorios.apellidosAleatorio(),
                             GenerarCamposAleatorios.nombreAleatorio(),
                             GenerarCamposAleatorios.fechaNacimientoAleatoria(),
                             GenerarCamposAleatorios.direccionAleatoria(),
                             GenerarCamposAleatorios.SalarioAleatorio());

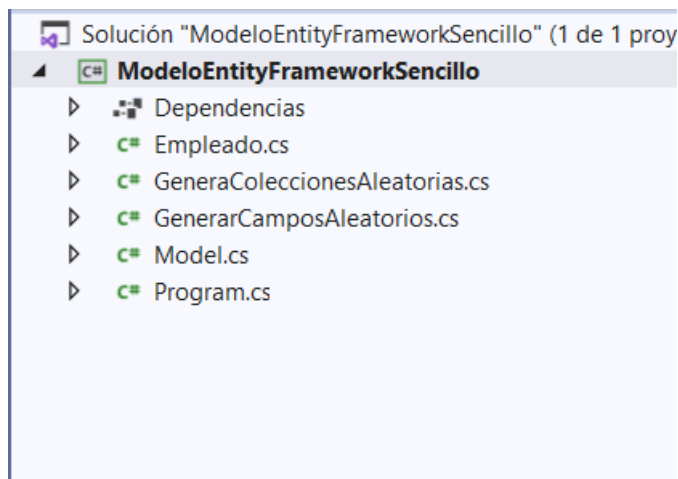
        break;
    }

    listaEmpleados.Add(emp);
}

return listaEmpleados;
}
}
}

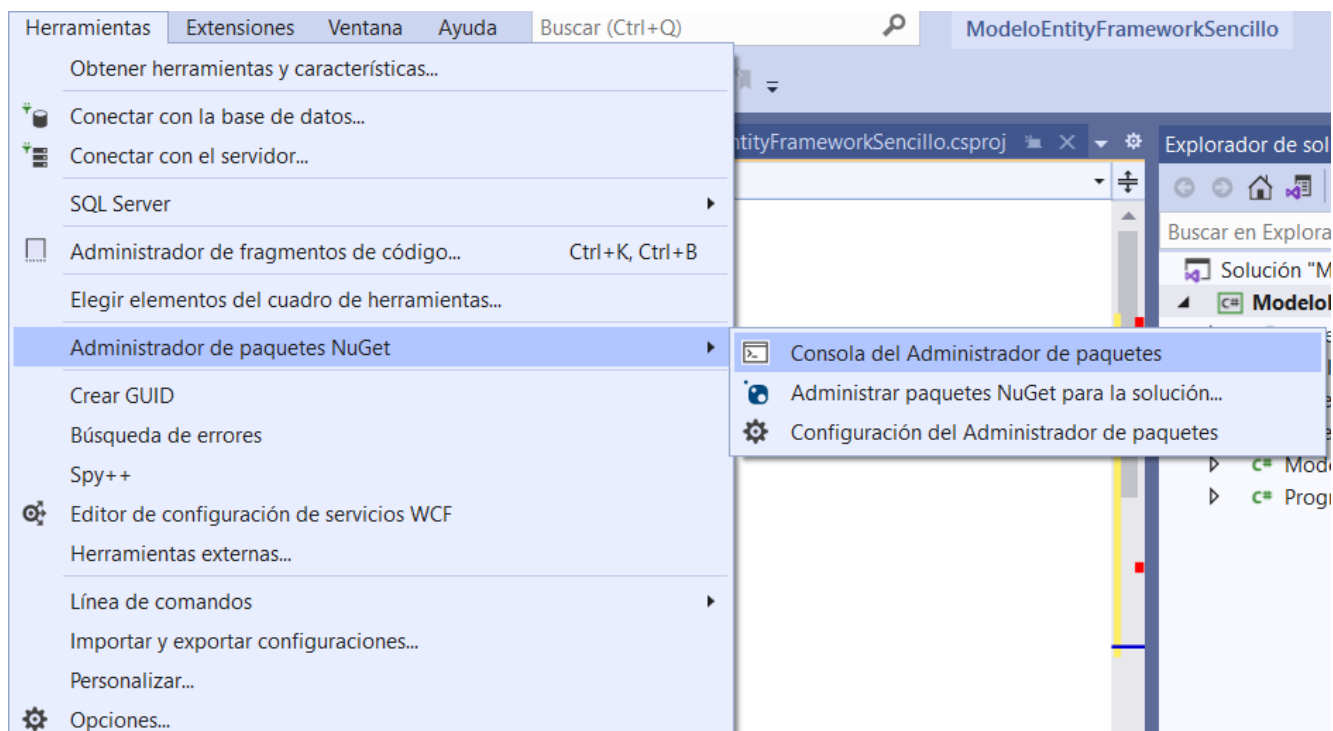
```

El proyecto va a quedar de la siguiente manera:

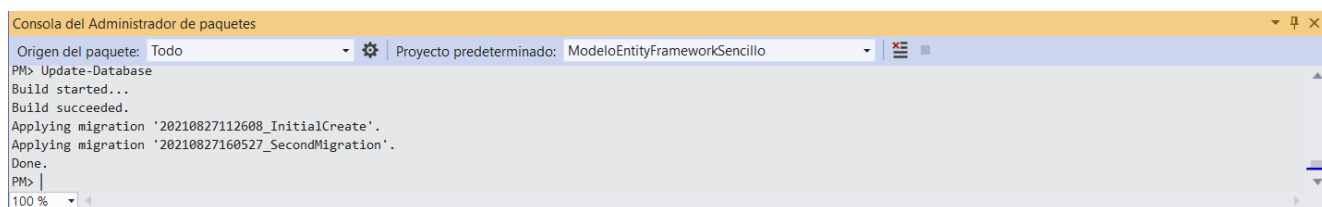


4.3.2 Modelo Entity. Code First. Creación de la base de datos con la API Fluent.

Continuamos añadiendo el fichero Model.cs. En este fichero usaremos las clases del EntityFramework necesarias para poder construir nuestra base de datos a partir del modelo de datos, es lo que se conoce como la API Fluent, que creará nuestra base de datos y que generará nuestro modelo ORM de transformación objeto-relacional. Para ello será necesario instalar ahora si los paquetes necesarios del Entity Framework en nuestro proyecto usando los paquetes NuGet. Abrimos en el menú Herramientas -> Administrador de paquetes NuGet -> Consola del Administrador de paquetes.



Se abrirá una consola Powershell como la siguiente:



Vamos a instalar los paquetes del Entity Framework para la base de datos SQLite, que es con la primera que trabajaremos en estos apuntes. Ejecutamos la orden:

`Install-Package Microsoft.EntityFrameworkCore.Sqlite`

```
Consola del Administrador de paquetes
Origen del paquete: Todo
PM> Update-Database
Build started...
Build succeeded.
Applying migration '20210827112608_InitialCreate'.
Applying migration '20210827160527_SecondMigration'.
Done.
PM> Install-Package Microsoft.EntityFrameworkCore.Sqlite
100 %
```

De esta manera se instalarán los paquetes NuGet necesarios para poder trabajar con la base de datos SQLite con el Entity Framework. El motor de base de datos SQLite está embebido en el Framework EntityFramework en los paquetes que instalamos.

```
Consola del Administrador de paquetes
Origen del paquete: Todo
Proyecto predeterminado: ModeloEntityFrameworkSencillo
'System.Collections.Immutable 5.0.0' se instaló correctamente en ModeloEntityFrameworkSencillo
'System.ComponentModel.Annotations 5.0.0' se instaló correctamente en ModeloEntityFrameworkSencillo
'System.Diagnostics.DiagnosticSource 5.0.1' se instaló correctamente en ModeloEntityFrameworkSencillo
'System.Memory 4.5.3' se instaló correctamente en ModeloEntityFrameworkSencillo
La ejecución de acciones de NuGet tardó 187 ms
Tiempo transcurrido: 00:00:02.1237603
PM>
100 %
```

Ahora vamos a añadir al fichero Model.cs el siguiente código que explicaremos paso a paso:

Una instancia de DbContext representa una combinación de los patrones de unidad de trabajo y repositorio, de modo que se puede usar para realizar consultas desde una base de datos y agrupar los cambios que se volverán a escribir en el almacén como una unidad.

DbContext se usa normalmente con un tipo derivado que contiene DbSet<TEntity> las propiedades de las entidades raíz del modelo. Estos conjuntos se inicializan automáticamente cuando se crea la instancia de la clase derivada. Al utilizar el enfoque Code First, las DbSet<TEntity> propiedades en el contexto derivado se utilizan para generar un modelo por Convención.

El EntityFramework creará y manipulará la base de datos a partir de DbContext y DbSet. Cuando hacemos heredar a nuestra clase EmpleadoContext de DbContext `public class EmpleadoContext : DbContext`, estamos indicando que es el punto de inicio de la definición de nuestra base de datos a partir de código. Utiliza lo que se conoce como la API Fluent para la creación de base de datos y mapeo-objeto relacional.

Al definir un Objeto DbSet de tipo Empleado, estamos indicando que se debe crear una tabla con los campos públicos de Empleado. Por defecto inferirá la clave de esa tabla con el primer campo de la clase Empleado, que es Id. `public DbSet<Empleado> Empleados { get; set; }`

En el constructor de `EmpleadoContext`, estamos colocando nuestro directorio de base de datos en el directorio Local de Windows. Podríamos elegir otro, en este primer ejemplo lo colocamos ahí, cogiéndolo de la variable de entorno del Sistema Operativo Windows con la clase `Environment` que accede a las variables de entorno del sistema.

```
var folder = Environment.SpecialFolder.LocalApplicationData;  
var path = Environment.GetFolderPath(folder);
```

Finalmente definimos nuestra propiedad `DbPath` añadiendo el nombre del fichero de la base de datos `Empleados.db` al path anterior.

```
DbPath = $"{path}{System.IO.Path.DirectorySeparatorChar}Empleados.db";
```

El método `onConfiguring` se ejecutará cuando ejecutamos el comando de migración de nuestro modelo y configura la fuente de base de datos en `DbPath`.

```
protected override void OnConfiguring(DbContextOptionsBuilder options)  
{  
    options.UseSqlite($"Data Source={DbPath}");  
}
```

```
using Microsoft.EntityFrameworkCore;  
using System;  
using System.Collections.Generic;  
using System.Linq;  
using System.Text;  
using System.Threading.Tasks;
```

```
namespace ModeloEntityFrameworkSencillo
```

```
{  
    public class EmpleadoContext : DbContext  
    {  
        public DbSet<Empleado> Empleados { get; set; }
```

```
        public string DbPath { get; private set; }
```

```
        public EmpleadoContext()  
        {
```

```
            var folder = Environment.SpecialFolder.LocalApplicationData;  
            var path = Environment.GetFolderPath(folder);  
            DbPath = $"{path}{System.IO.Path.DirectorySeparatorChar}Empleados.db";
```

```
        }
```

```

        protected override void OnConfiguring(DbContextOptionsBuilder options)
        {
            options.UseSqlite($"Data Source={DbPath}");
        }
    }
}

```

Instalamos a través de la consola las herramientas para crear la base de datos.

```
Install-Package Microsoft.EntityFrameworkCore.Tools
```

```

PM> Install-Package Microsoft.EntityFrameworkCore.Sqlite
Restaurando paquetes para C:\Users\carlo\OneDrive\Brianda20202021\interfaces\NET\Te
\ModeloEntityFrameworkSencillo.csproj...
Instalando el paquete NuGet Microsoft.EntityFrameworkCore.Sqlite 5.0.9.
Ejecutando restauración...

```

Construimos el esquema de migración de nuestro modelo de datos a base de datos SQLite. El nombre que le damos es el nombre de la migración. Si vamos haciendo mas migraciones o borramos el fichero con Remove-Migration o damos otro nombre y se creara una nueva versión de la migración incremental

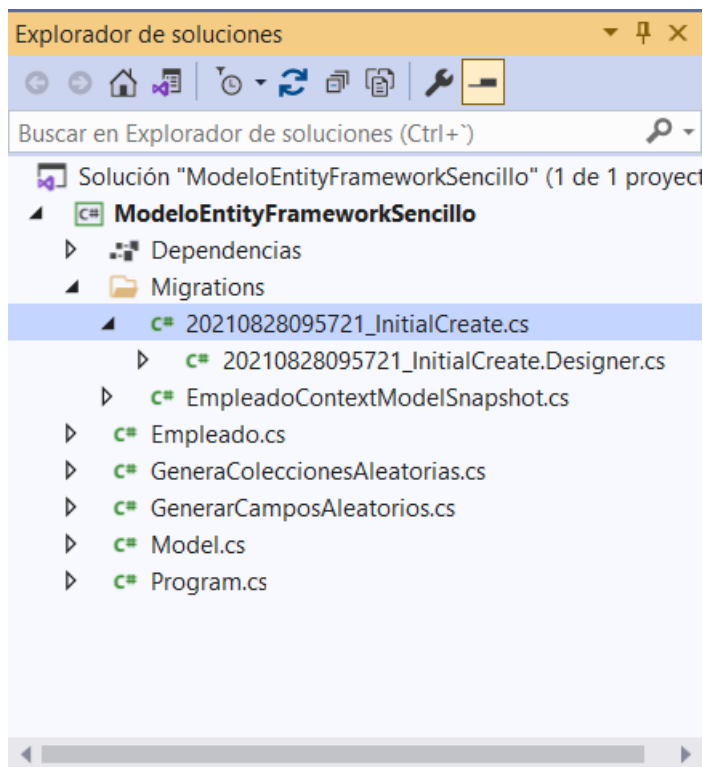
```
Add-Migration InitialCreate
```

```

Origen del paquete: Todo [v] [⚙] Proyecto predeterminad
PM> Add-Migration InitialCreate
Build started...
Build succeeded.
To undo this action, use Remove-Migration.
PM>

```

Aparecerá en el proyecto un fichero fecha_InitialCreate.cs con la migración que contiene un código como el siguiente:



Fichero de migración de creación de la base de datos

La clase que se ejecutara cuando hagamos el Update-Database para crear las Tablas en la base de datos, en este caso una tabla Empleados, es InitialCreate que hereda de la clase Migration.

El MigrationBuilder en el método Up se encarga de crear la tabla con sus columnas y restricciones, se ejecutará cuando hagamos el Update-Database. La tabla se borra dentro del método Down, al realizar un Remove_Migration.

Resaltar que tanto la creación de base de datos, como el mapeo se realiza todo a través de código C# usando clases del Entity Framework.

```
using Microsoft.EntityFrameworkCore.Migrations;

namespace ModeloEntityFrameworkSencillo.Migrations
{
    public partial class InitialCreate : Migration
    {
        protected override void Up(MigrationBuilder migrationBuilder)
        {
            migrationBuilder.CreateTable(
                name: "Empleados",
                columns: table => new
                {
                    Id = table.Column<string>(type: "TEXT", nullable: false),
                    Apellido = table.Column<string>(type: "TEXT", nullable: true),
                    Nombre = table.Column<string>(type: "TEXT", nullable: true),
                    Direccion = table.Column<string>(type: "TEXT", nullable: true),
                }
            );
        }
    }
}
```



```

        Edad = table.Column<int>(type: "INTEGER", nullable: false),
        Salario = table.Column<double>(type: "REAL", nullable: false)
    },
    constraints: table =>
    {
        table.PrimaryKey("PK_Empleados", x => x.Id);
    });
}

protected override void Down(MigrationBuilder migrationBuilder)
{
    migrationBuilder.DropTable(
        name: "Empleados");
}
}
}

```

Fijaos como la clase ModelBuilder será la encargada de definir el modelo que mapeara nuestros objetos, en este caso Persona a base de datos. Define una anotación, y define las columnas de la base de datos, clave Primaria y nombre de la tabla, para crear el Entity, que representa el modelo de base de datos en nuestro programa. De ahí el nombre Entity Framework.

```

modelBuilder.Entity("ModeloEntityFrameworkSencillo.Empleado", b =>
{
    b.Property<string>("Id")
        .HasColumnType("TEXT");

    b.Property<string>("Apellido")
        .HasColumnType("TEXT");

    b.Property<string>("Direccion")
        .HasColumnType("TEXT");

    b.Property<int>("Edad")
        .HasColumnType("INTEGER");

    b.Property<string>("Nombre")
        .HasColumnType("TEXT");

    b.Property<double>("Salario")
        .HasColumnType("REAL");

    b.HasKey("Id");

    b.ToTable("Empleados");
});

```

Fichero de migración Designer del modelo de datos

Mapea los datos de la base de datos a nuestra clase Empleado

```

modelBuilder.Entity("ModeloEntityFrameworkSencillo.Empleado",

```

```

// <auto-generated />
using Microsoft.EntityFrameworkCore;
using Microsoft.EntityFrameworkCore.Infrastructure;
using Microsoft.EntityFrameworkCore.Migrations;
using Microsoft.EntityFrameworkCore.Storage.ValueConversion;
using ModeloEntityFrameworkSencillo;

namespace ModeloEntityFrameworkSencillo.Migrations
{
    [DbContext(typeof(PersonaContext))]
    [Migration("20210827184124_InitialCreate")]
    partial class InitialCreate
    {
        protected override void BuildTargetModel(ModelBuilder modelBuilder)
        {
#pragma warning disable 612, 618
            modelBuilder
                .HasAnnotation("ProductVersion", "5.0.9");

            modelBuilder.Entity("ModeloEntityFrameworkSencillo.Empleado", b =>
            {
                b.Property<string>("Id")
                    .HasColumnType("TEXT");

                b.Property<string>("Apellido")
                    .HasColumnType("TEXT");

                b.Property<string>("Direccion")
                    .HasColumnType("TEXT");

                b.Property<int>("Edad")
                    .HasColumnType("INTEGER");

                b.Property<string>("Nombre")
                    .HasColumnType("TEXT");

                b.Property<double>("Salario")
                    .HasColumnType("REAL");

                b.HasKey("Id");

                b.ToTable("Empleados");
            });
#pragma warning restore 612, 618
        }
    }
}

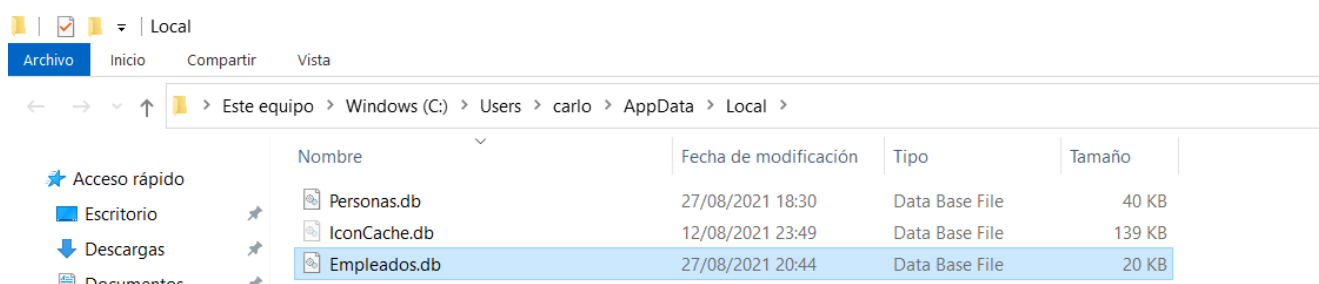
```

Creemos o actualizamos la base de datos con el siguiente comando que aplica la migración.

[Update-Database](#)

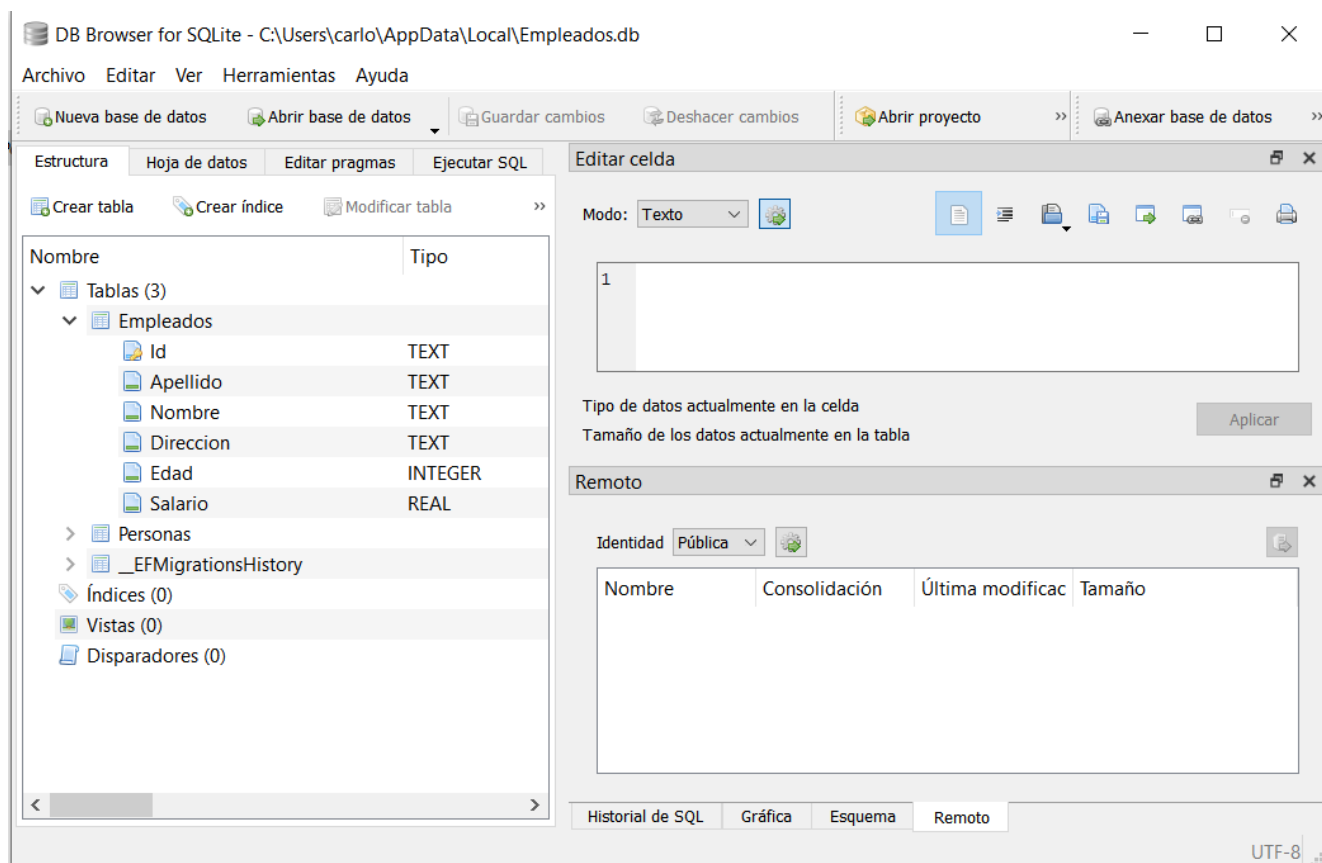
```
Consola del Administrador de paquetes
Origen del paquete: Todo [v] [g] Proyecto predeterminado: ModeloEntityFrameworkSenci
To undo this action, use Remove-Migration.
PM> Update-Database
Build started...
Build succeeded.
Applying migration '20210827184124_InitialCreate'.
Done.
PM> |
100 %
```

Encontrareis la base de datos sqlite Empleados.db en vuestro directorio Local:



Nombre	Fecha de modificación	Tipo	Tamaño
Personas.db	27/08/2021 18:30	Data Base File	40 KB
IconCache.db	12/08/2021 23:49	Data Base File	139 KB
Empleados.db	27/08/2021 20:44	Data Base File	20 KB

Instalando el SQLite Browser que se proporciona en los recursos, **SQLiteDatabaseBrowserPortable_3.12.0_English.paf.exe**, desplegarlo en c:\dev podéis ver como la base de datos se ha construido correctamente.



Hemos creado nuestra primera base de datos usando el EntityFramework

Vamos a hora al programa principal Program.cs para borrar, insertar, actualizar y realizar consultas de nuestra base de datos.

Hacemos que nuestro programa funciones si la conexión con la base de datos con EmpleadoContext se hace correctamente

```
using (var db = new EmpleadoContext())
{
```

Escribimos donde se haya la base de datos físicamente, en que ruta.

```
Console.WriteLine($"Database path: {db.DbPath}.");
```

Si la base de datos esta llena, borramos todos los datos con el método Remove. Hemos transformado nuestra Select a una lista con el método ToList por comodidad. Con el método Remove eliminamos cada empleado de la selección. Si la base de datos esta vacía no hará nada. El método SaveChanges confirmará las operaciones sobre la base de datos como un COMMIT.

Para acceder a los datos de la tabla empleado, usamos la propiedad de tipo DbSet de Empleados, que nos permitirá manipular los datos de la tabla Empleado. Sobre DbSet podemos realizar las consultas LINQ, en caso de realizar select recibiremos un objeto IQueryable. Recordar que implementa IEnumerable.

```
public DbSet<Empleado> Empleados { get; set; } de EmpleadoContext
```

```
db.Empleados
```

```
db.Empleados.Select(emp => emp).ToList().ForEach(emp => db.Remove(emp));  
db.SaveChanges();
```

Generamos una lista aleatoria de 200 empleados. E insertamos cada empleado con el método Add en la base de datos. Confirmamos los cambio al final.,

```
List<Empleado> empleados = GeneraColeccionesAleatorias.ListaEmpleados(200);  
  
empleados.ForEach(emp => db.Add(emp));  
  
db.SaveChanges();
```

Vamos a realizar nuestra primera consulta LINQ con sintaxis de query. Nos quedamos con todos los empleados cuyo nombre sea Antonio. Y los mostramos. Al operar con EntityFramework y una base de datos la consulta nos puede devolver un objeto de tipo IQueryable, que ya indicamos anteriormente en los apuntes. IQueryable<Empleado> query.

```
Console.WriteLine("Empleados de nombre Antonio");  
  
IQueryable<Empleado> query = from empleado in db.Empleados  
                               where empleado.Nombre == "Antonio"  
                               select empleado;  
  
foreach(Empleado empl in query)  
{  
    Console.WriteLine("Empleado : {0}", empl);  
}
```

Otro ejemplo de query es quedarse con los empleados mayores de 30 años. Usamos sintaxis de método.

```
Console.WriteLine("Empleados con edad mayor de 30");  
db.Empleados.Where(per => per.Edad > 30).Select(emp =>  
emp).ToList().ForEach(per => Console.WriteLine(per));
```

En la ultima query del ejemplo obtenemos todos los empleados con edad 30 y modificamos esa edad incrementando en 1. Finalmente confirmamos los cambios, con el método Update.

```
db.Empleados.Where(per => per.Edad == 30).Select(emp => emp).ToList()
```

```

        .ForEach(emp =>
        {
            emp.Edad++;
            db.Empleados.Update(emp);
        });

        db.SaveChanges();

using System;
using System.Linq;
using System.Collections.Generic;

using ModeloEntityFrameworkSencillo.Debug;

namespace ModeloEntityFrameworkSencillo
{
    class Program
    {
        static void Main(string[] args)
        {
            using (var db = new EmpleadoContext())
            {

                Console.WriteLine($"Database path: {db.DbPath}.");

                db.Empleados.Select(emp => emp).ToList().ForEach(emp => db.Remove(emp));
                db.SaveChanges();

                List<Empleado> empleados = GeneraColeccionesAleatorias.ListaEmpleados(200);
                empleados.ForEach(emp => db.Add(emp));
                db.SaveChanges();

                Console.WriteLine("Empleados de nombre Antonio");

                IQueryable<Empleado> query = from empleado in db.Empleados
                                              where empleado.Nombre == "Antonio"
                                              select empleado;

                foreach(Empleado empl in query)
                {
                    Console.WriteLine("Empleado : {0}", empl);
                }

                Console.WriteLine("Empleados con edad mayor de 30");
                db.Empleados.Where(per => per.Edad > 30).Select(emp =>
emp).ToList().ForEach(per => Console.WriteLine(per));

                Console.WriteLine("Empleados que contienen la letra A");
                db.Empleados.Where(emp => emp.Nombre.Contains("A")).Select(emp =>
emp).ToList().ForEach(emp => Console.WriteLine(emp));

```

```

Console.WriteLine("Lista de todos los empleados");
db.Empleados.Select(emp=>emp).ToList().ForEach(emp => Console.WriteLine(emp));

Console.WriteLine("Incrementamos en 1 la edad de todos los empleados cuya edad
sea 30");

db.Empleados.Where(per => per.Edad == 30).Select(emp => emp).ToList()
.ForEach(emp =>
    {
        emp.Edad++;
        db.Empleados.Update(emp);
    });

db.SaveChanges();
}
}
}
}

```

Nota: La clase DbSet<TEntity>

Un DbSet representa la colección de todas las entidades en el contexto, o que se puede consultar desde la base de datos, de un tipo determinado. Los objetos DbSet se crean a partir de DbContext mediante el método DbContext.set. **Fijaos que recibe un TEntity como tipo DbSet<TEntity>**, y que TEntity debe ser una clase **where TEntity : class**. **Esto nos indica que solo podemos mapear clases a base de datos.**

Implementaciones

```

public class DbSet<TEntity> : System.Data.Entity.Infrastructure.DbQuery<TEntity>,
System.Collections.Generic.IEnumerable<TEntity>, System.Data.Entity.IDbSet<TEntity>,
System.Linq.IQueryable<TEntity> where TEntity : class

```

Propiedades

Local

Métodos

Add
 AddRange
 Attach
 Create
 Equals
 Find
 FindAsync
 GetHashCode
 GetType
 Remove
 RemoveRange

SqlQuery: Crea una consulta SQL sin formato que devolverá entidades de este conjunto

Importante tener en cuenta aquí que tiene métodos heredados como

Update: Inserta el objeto en base de datos.

SaveChanges: confirma los cambios en base de datos.

AddOrUpdate: si existe el objeto lo modifica en base de datos, sino lo inserta.

Para que la modificación sea efectiva debemos realizar un SaveChanges()

Nota: Muchos de los métodos que nos son ofrecidos tienen su versión Async, para trabajar de manera asíncrona con bases de datos remotas, que es la forma habitual de trabajar cuando estamos atacando una base de datos de servidor.

4.3.3 Ejercicios

Al programar principal anterior añadir código para:

1. Calcular el máximo, mínimo, suma y media de los salarios, y mostrarlos por pantalla. Podéis construir un tipo struct para guardar toda la información
2. Obtener el empleado de mayor edad y mostrarlo por pantalla
3. Obtener el empleado de menor salario y mostrarlo por pantalla.
4. Obtener un listado de empleados con salario mayor de 40000 ordenado por nombre y apellidos
5. Obtener un listado de empleados con edad entre 25 y 45 ordenados por edad.

4.3.4 Modelos mas avanzados. Anotaciones.

Hasta ahora hemos visto que simplemente indicando el DbSet el modelo de migración Entity para C# reconoce la estructura de la base de datos. Pero podemos ser más específicos e indicar nombre de columnas, que se mapea a base de datos y que no, tipos y tamaños de nuestros datos. Para ello se usan una serie de anotaciones que veremos en el siguiente ejemplo.

Crear un proyecto ModeloEntityFramework exactamente igual al que hemos realizado en la versión anterior. Seguid los mismos pasos. Instalar el Entity Framework. Instalar el Entity Framework para SQLite

`Install-Package Microsoft.EntityFrameworkCore.Sqlite`

Empezaremos con la clase Persona. Vamos a almacenar personas en nuestro modelo para

empezar. Usaremos etiquetado para indicarle al proceso de migración Entity Framework como queremos que sea nuestra base de datos.

Añadimos la clase Persona y vamos a ver brevemente que anotaciones hemos usado para realizar el modelo. Fijaos que añadimos dos nuevas directrices using para las anotaciones.

```
using System.ComponentModel.DataAnnotations;  
using System.ComponentModel.DataAnnotations.Schema;
```

Además ahora controlamos el nombre de nuestra tabla con la anotación Table

```
[Table("Personas")]
```

```
[Table("Personas")]  
public class Persona
```

Indicamos para el atributo Id su tipo y tamaño con la anotación Column [Column(TypeName = "varchar(10)")] y que es clave con la anotación [Key]. Nótese que el TypeName tiene que ser un tipo aceptado por la base de datos destino en este caso SQLite. Para nombre, apellidos y otros atributos indicamos su tipo igualmente.

```
[Key]  
[Column(TypeName = "varchar(10)")]  
public string Id { get; set; }
```

El atributo Edad es en este caso un atributo autocalculado que depende de la fecha de nacimiento. Lo hacemos privado como propiedad y publico en su acceso para poder autocalcularlo. Nos aseguramos que el privado no se mapea con la etiqueta

```
[NotMapped]  
private int laEdad { get; set; }
```

El acceso publico lo mapeamos, indicando además el nombre que queremos que tenga esa columna. Fijaos como debemos darle un set en este caso a Edad, para poder traerlo de base de datos. Otra opción podría ser no mapear Edad, y calcular automáticamente el valor en fecha de nacimiento. Esto es así porque como muchos ORM, Entity Framework usar el set para escribir el valor de Edad cuando recupera de base de datos. En este caso hemos Etiquetado [Column("Edad")], lo que fuerza que el nombre de la columna en base de datos sea Edad.

```

[Column("Edad")]
public int Edad
{
    get { return laEdad; }

    set { this.laEdad = value; }
}

using System;
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;
using System.Text;

namespace ModeloEntityFramework
{
    [Table("Personas")]
    public class Persona
    {
        [Key]
        [Column(TypeName = "varchar(10)")]

        [Column(TypeName = "varchar(200)")]
        public String Apellido { get; set; }
        [Column(TypeName = "varchar(200)")]
        public String Nombre { get; set; }
        [Required]

        private DateTime FechaN;
        public DateTime FechaNac
        {
            get { return this.FechaN; }

            set { this.FechaN = value; this.laEdad = DateTime.Now.Year - FechaNac.Year; }
        }

        [NotMapped]
        private int laEdad { get; set; }

        [Column("Edad")]
        public int Edad
        {
            get { return laEdad; }

            set { this.laEdad = value; }
        }

        [Column(TypeName = "varchar(500)")]

```

```

        public string Direccion { get; set; }

        public Persona()
        {

        }

        public Persona(String Id, String Apellido, String Nombre, DateTime FechaNac, string
Direccion)
        {
            this.Id = Id;
            this.Apellido = Apellido;
            this.Nombre = Nombre;
            this.FechaNac= FechaNac;
            this.Direccion = Direccion;
            this.Edad = DateTime.Now.Year - FechaNac.Year; ;
        }

        public override string ToString()
        {
            return String.Format("Apellidos: {0}, Nombre: {1}, FechaNac: {2:d}, Edad: {3} ",
this.Apellido, this.Nombre, this.FechaNac, this.Edad);
        }
    }
}

```

4.3.5 Ejercicio Notas Cornell.

El fichero Model.cs ss muy parecido al anterior. Se pide en este caso que comentéis cada línea marcada indicando que significa y cual es su labor.

```

using Microsoft.EntityFrameworkCore;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace ModeloEntityFrameWork {

    public class PersonaContext : DbContext
    {
        public DbSet<Persona> Personas { get; set; }

        public string DbPath { get; private set; }

        public PersonaContext() {

```

```

        var folder = Environment.SpecialFolder.LocalApplicationData;
        var path = Environment.GetFolderPath(folder);
        DbPath = $"{path}{System.IO.Path.DirectorySeparatorChar}Personas.db";

    }

    protected override void OnConfiguring(DbContextOptionsBuilder options)
    {
        options.UseSqlite($"Data Source={DbPath}");
    }

}

}

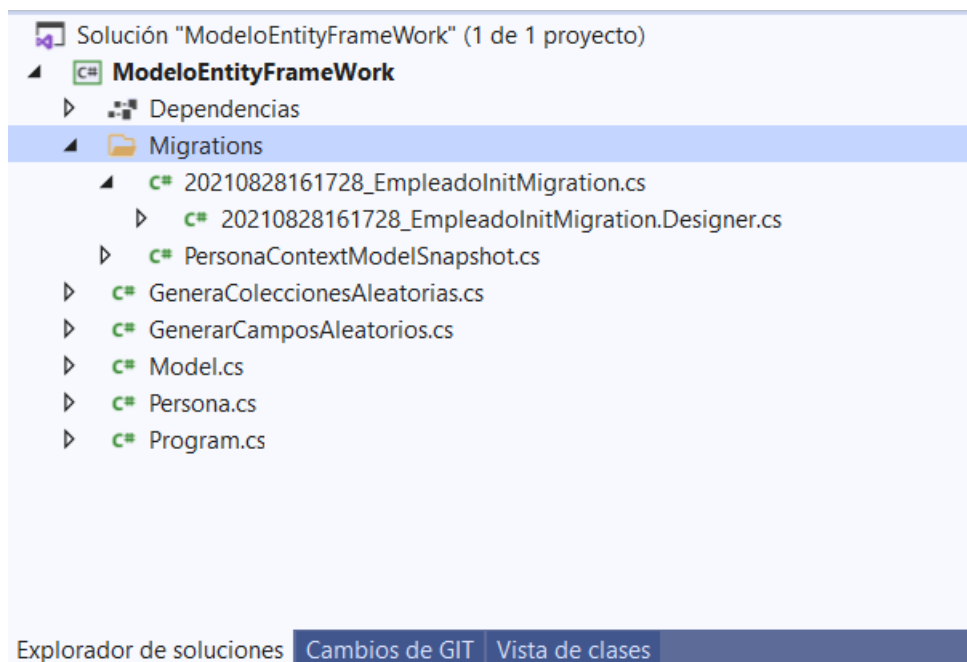
```

Realizar la migración en la consola de administración de paquetes.

`Install-Package Microsoft.EntityFrameworkCore.Sqlite`

`Add-Migration EmpleadoInitMigration`

`Update-Database`



4.3.6 Ejercicio

1. Comprobar que la base de datos se ha creado en la ruta indicada
2. Comprobar que los ficheros de migración se corresponde a las anotaciones que hemos establecido.

```
// <auto-generated />
using System;
using Microsoft.EntityFrameworkCore;
using Microsoft.EntityFrameworkCore.Infrastructure;
using Microsoft.EntityFrameworkCore.Migrations;
using Microsoft.EntityFrameworkCore.Storage.ValueConversion;
using ModeloEntityFramework;

namespace ModeloEntityFramework.Migrations
{
    [DbContext(typeof(PersonaContext))]
    [Migration("20210828161728_EmpleadoInitMigration")]
    partial class EmpleadoInitMigration
    {
        protected override void BuildTargetModel(ModelBuilder modelBuilder)
        {
#pragma warning disable 612, 618
            modelBuilder
                .HasAnnotation("ProductVersion", "5.0.9");

            modelBuilder.Entity("ModeloEntityFramework.Persona", b =>
            {
                b.Property<string>("Id")
                    .HasColumnType("varchar(10)");

                b.Property<string>("Apellido")
                    .HasColumnType("varchar(200)");

                b.Property<string>("Direccion")
                    .HasColumnType("varchar(500)");

                b.Property<int>("Edad")
                    .HasColumnType("INTEGER")
                    .HasColumnName("Edad");

                b.Property<DateTime>("FechaNac")
                    .HasColumnType("TEXT");

                b.Property<string>("Nombre")
                    .HasColumnType("varchar(200)");

                b.HasKey("Id");

                b.ToTable("Personas");
            });
#pragma warning restore 612, 618
        }
    }
}
```

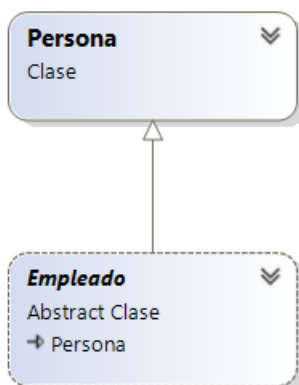
3. Investigad en internet porque la fecha se ha guardado como TEXT, y no como un tipo fecha.

```
b.Property<DateTime>("FechaNac")  
    .HasColumnType("TEXT");
```

4. Borrar la migracion desde la consola de comandos y volver a hacerla.

4.3.7 Añadiendo clases al modelo.

Normalmente cuando utilizamos orientación a objetos nuestro interés no es guardar en base de datos las clases que están en la parte alta de la jerarquía. Mas bien queremos guardar las que están en la base de la jerarquía. Por ejemplo si tenemos el siguiente modelo:



No almacenaremos Persona, nos interesa trabajar y almacenar Empleados. Vamos a añadir a nuestro proyecto la clase Empleado que hereda de Persona:

```
using System;  
using System.Collections.Generic;  
using System.Text;  
  
namespace ModeloEntityFramework  
{  
    public class Empleado : Persona  
    {  
        public double Salario { get; set; }  
    }  
}
```

```

    public Empleado()
    {

    }

    public Empleado(String Id, String Apellido, String Nombre, DateTime FechaNac, String
direccion, double Salario): base(Id,Apellido, Nombre, FechaNac,direccion) {

        this.Salario = Salario;

    }

    public double calculoImpuestos() {

        return Salario * 0.2;

    }

    public double salarioNeto()
    {

        return Salario - calculoImpuestos();

    }

    public override string ToString()
    {
        return String.Format("{0} Salario: {1}", base.ToString(), this.Salario);
    }
}

```

Nuevo GeneraColeccionesAleatorias.cs

```

using System;
using System.Collections.Generic;
using System.Text;

namespace ModeloEntityFrameWork.Debug
{
    class GeneraColeccionesAleatorias
    {

        public static List<Empleado> ListaEmpleados(int numEmpleados)
        {

            List<Empleado> listaEmpleados = new List<Empleado>();

            Random r = new Random();

            for (int i= 0; i < numEmpleados; i++ )
            {

```

```

        listaEmpleados.Add(new
Empleado(i.ToString(), GenerarCamposAleatorios.apellidosAleatorio(),
                                                GenerarCamposAleatorios.nombreAleatorio(),
                                                GenerarCamposAleatorios.fechaNacimientoAleatoria(),
                                                GenerarCamposAleatorios.direccionAleatoria(),
                                                GenerarCamposAleatorios.SalarioAleatorio()));

    }

    return listaEmpleados;
}
}
}

```

Nuevo Program.cs

```

using System;
using System.Linq;
using System.Collections.Generic;
using ModeloEntityFrameWork.Debug;

namespace ModeloEntityFrameWork
{
    class Program
    {
        static void Main(string[] args)
        {

            using (var db = new EmpleadoContext()) { {

                Console.WriteLine($"Database path: {db.DbPath}.");

                db.Empleados.Select(per => per).ToList().ForEach(per => db.Remove(per));
                db.SaveChanges();

                List<Empleado> empleados = GeneraColeccionesAleatorias.ListaEmpleados(200);

                empleados.ForEach(empleado => db.Add(empleado));

                db.SaveChanges();

                db.Empleados.Select(emp => emp).Where(emp =>
emp.Apellido.Contains("A")).ToList().ForEach(emp => Console.WriteLine(emp));
                db.Empleados.Select(emp => emp).Where(emp => emp.Edad>30).ToList().ForEach(emp =>
Console.WriteLine(emp));
                db.Empleados.Select(emp => emp).ToList().ForEach(emp => Console.WriteLine(emp));

            }
        }
    }
}

```



```

    }
}

```

Y vamos a modificar el modelo de nuestra base de datos para guardar Empleados. Se creará la base de datos EmpleadosPersona.db. Fijaos que ahora no incluimos Persona en el DbSet, sino Empleado. Al no incluirlo no se creará una tabla Personas. Sólo se creará la tabla Empleados. Como los campos de Empleados se heredan de Persona las anotaciones también se heredan y se aplican en la creación del modelo de datos y de la base de datos como veremos en los ficheros de migración.

```

using Microsoft.EntityFrameworkCore;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace ModeloEntityFramework {

    public class EmpleadoContext : DbContext
    {
        public DbSet<Empleado> Empleados { get; set; }

        public string DbPath { get; private set; }

        public EmpleadoContext() {

            var folder = Environment.SpecialFolder.LocalApplicationData;
            var path = Environment.GetFolderPath(folder);
            DbPath = $"{path}{System.IO.Path.DirectorySeparatorChar}EmpleadosPersona.db";

        }

        // The following configures EF to create a Sqlite database file in the
        // special "local" folder for your platform.
        protected override void OnConfiguring(DbContextOptionsBuilder options)
        {
            options.UseSqlite($"Data Source={DbPath}");
        }

    }
}

```

Ficheros de migración.

Fichero de creación de base de datos.

```
using System;
using Microsoft.EntityFrameworkCore.Migrations;

namespace ModeloEntityFramework.Migrations
{
    public partial class EmpleadoMigration : Migration
    {
        protected override void Up(MigrationBuilder migrationBuilder)
        {
            migrationBuilder.CreateTable(
                name: "Empleados",
                columns: table => new
                {
                    Id = table.Column<string>(type: "varchar(10)", nullable: false),
                    Salario = table.Column<double>(type: "DECIMAL", nullable: false),
                    Apellido = table.Column<string>(type: "varchar(200)", nullable: true),
                    Nombre = table.Column<string>(type: "varchar(200)", nullable: true),
                    FechaNac = table.Column<DateTime>(type: "TEXT", nullable: false),
                    Edad = table.Column<int>(type: "INTEGER", nullable: false),
                    Direccion = table.Column<string>(type: "varchar(500)", nullable: true)
                },
                constraints: table =>
                {
                    table.PrimaryKey("PK_Empleados", x => x.Id);
                });
        }

        protected override void Down(MigrationBuilder migrationBuilder)
        {
            migrationBuilder.DropTable(
                name: "Empleados");
        }
    }
}
```

El fichero de mapeo para el Modelo, el Designer.

```
// <auto-generated />
using System;
using Microsoft.EntityFrameworkCore;
using Microsoft.EntityFrameworkCore.Infrastructure;
using Microsoft.EntityFrameworkCore.Migrations;
using Microsoft.EntityFrameworkCore.Storage.ValueConversion;
using ModeloEntityFramework;

namespace ModeloEntityFramework.Migrations
{
    [DbContext(typeof(EmpleadoContext))]
    [Migration("20210829102857_EmpleadoMigration")]
    partial class EmpleadoMigration
    {
        protected override void BuildTargetModel(ModelBuilder modelBuilder)
        {
#pragma warning disable 612, 618
            modelBuilder

```

```

        .HasAnnotation("ProductVersion", "5.0.9");

modelBuilder.Entity("ModeloEntityFrameWork.Empleado", b =>
{
    b.Property<string>("Id")
        .HasColumnType("varchar(10)");

    b.Property<string>("Apellido")
        .HasColumnType("varchar(200)");

    b.Property<string>("Direccion")
        .HasColumnType("varchar(500)");

    b.Property<int>("Edad")
        .HasColumnType("INTEGER")
        .HasColumnName("Edad");

    b.Property<DateTime>("FechaNac")
        .HasColumnType("TEXT");

    b.Property<string>("Nombre")
        .HasColumnType("varchar(200)");

    b.Property<double>("Salario")
        .HasColumnType("DECIMAL");

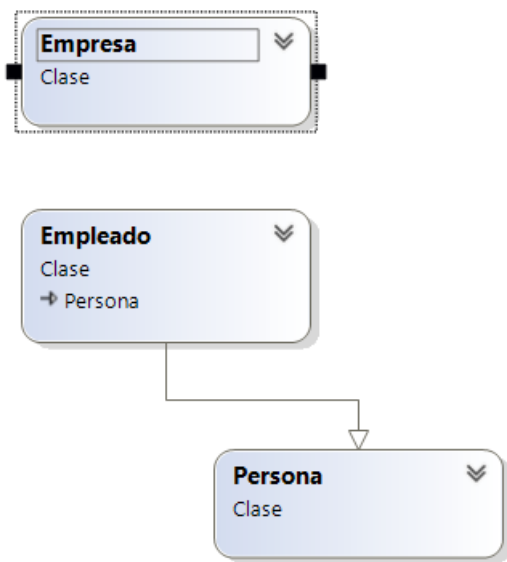
    b.HasKey("Id");

    b.ToTable("Empleados");
});
#pragma warning restore 612, 618
}
}
}

```

4.3.8 Varias clases relacionadas en nuestro modelo. Entity Framework

Vamos a realizar una serie de cambios en nuestro modelo de manera que añadimos la clase Empresa que contendrá Empleados.



Añadimos la clase Empresa que tendrá una lista de Empleados. `public List<Empleado> Empleados { get; set; }`

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.ComponentModel.DataAnnotations.Schema;
using System.ComponentModel.DataAnnotations;

namespace ModeloEntityFramework
{
    [Table("Empresas")]
    public class Empresa
    {
        [Key]
        [ForeignKey("IdEmpresa")]

        public string Id { get; set; }

        [Column(TypeName = "varchar(200)")]
        public string RazonSocial { get; set; }

        [Column(TypeName = "varchar(16)")]
        public string Telefono { get; set; }

        [Column(TypeName = "varchar(500)")]
        public string Direccion { get; set; }

        public List<Empleado> Empleados { get; set; }

        public Empresa()
    }
}
  
```

```

    {

    }

    public Empresa(string id, string razonsocial, string telefono, string direccion,
List<Empleado> empleados)
    {

        this.Id = id;
        this.RazonSocial = razonsocial;
        this.Telefono = telefono;
        this.Direccion = direccion;
        this.Empleados = empleados;
    }

    public string toString()
    {

        return String.Format("Id: {0} , Razon Social: {1}, Telefono: {2}, Dirección {3}",
this.Id, this.RazonSocial, this.Telefono, this.Direccion);
    }

    }
}

```

Igualmente haremos que empleado tenga como parte de su clave primary el Id de la Empresa. Añadimos `public string IdEmpresa {get;set;}` como nuevo atributo de Empleado. Igualmente añadiremos IdEmpresa al constructor `public Empleado(String Id, String IdEmpresa,....`

```

using System;
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;
using System.Text;

namespace ModeloEntityFramework
{

    [Table("Empleados")]
    public class Empleado : Persona
    {

        public string IdEmpresa {get;set;}
        [Column(TypeName = "DECIMAL")]
        public double Salario { get; set; }

        public Empleado()
        {

        }

        public Empleado(String Id, String IdEmpresa, String Apellido, String Nombre, DateTime
FechaNac, String direccion, double Salario): base(Id,Apellido, Nombre, FechaNac,direccion) {

```

```

        this.IdEmpresa = IdEmpresa;
        this.Salario = Salario;
    }

    public double calculoImpuestos() {
        return Salario * 0.2;
    }

    public double salarioNeto()
    {
        return Salario - calculoImpuestos();
    }

    public override string ToString()
    {
        return String.Format("{0} Salario: {1}", base.ToString(), this.Salario);
    }
}

```

El problema ahora es que las anotaciones no permiten clave con múltiples atributos en Entity Framework. Para establecer una clave con múltiples atributos debemos usar la API Fluent, modificando nuestro fichero Model.cs.

Añadimos el método `OnModelCreating`, que nos permitirá por un lado añadir una clave múltiple a La Entidad Empleado, y por otro crear una relación uno a muchos entre Empleado y Empresa. Hay que tener en cuenta que la API Fluent es prioritaria con respecto a las anotaciones, en caso de conflicto se aplica lo que hay en la API Fluent.

```

protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<Empleado>()
        .HasKey(e => new { e.Id, e.IdEmpresa });

    modelBuilder.Entity<Empresa>()
        .HasMany<Empleado>(g => g.Empleados)
        .WithOne()
        .HasForeignKey(emp => emp.IdEmpresa)
        .OnDelete(DeleteBehavior.Cascade);
}

```

Con el método `HasKey` nos permite crear una clave con múltiples atributos para Empleado.

```

modelBuilder.Entity<Empleado>()
    .HasKey(e => new { e.Id, e.IdEmpresa });

```

El método `HasMany` nos permite añadir una relación uno a muchos entre Empresa. y

Empleado. La relación se crearía de igual manera sin usar este método pues Empresa contiene la propiedad con una lista de Empleados. Lo hacemos para forzar el DeleteOnCascade.

```
modelBuilder.Entity<Empresa>()  
    .HasMany<Empleado>(g => g.Empleados)
```

WithOne hace que la relación sea uno a muchos

```
.WithOne()
```

Indicamos la clave foranea IdEmpresa con HasForeignKey.

```
.HasForeignKey(emp => emp.IdEmpresa)
```

Y establecemos que el DeleteOnCascade, de manera que si se borra una Empresa se borran todos sus Empleados asociados de la tabla Empleados.

```
.onDelete>DeleteBehavior.Cascade);
```

```
;
```

Model.cs

```
using Microsoft.EntityFrameworkCore;  
using System;  
using System.Collections.Generic;  
using System.Linq;  
using System.Text;  
using System.Threading.Tasks;  
  
namespace ModeloEntityFramework {  
  
    public class EmpresaContext : DbContext  
    {  
        public DbSet<Empleado> Empleados { get; set; }  
        public DbSet<Empresa> Empresas { get; set; }  
  
        public string DbPath { get; private set; }  
  
        public EmpresaContext() {  
  
            var folder = Environment.SpecialFolder.LocalApplicationData;  
            var path = Environment.GetFolderPath(folder);  
            DbPath = $"{path}{System.IO.Path.DirectorySeparatorChar}EmpleadosEmpresa.db";  
  
        }  
    }  
}
```

```

protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<Empleado>()
        .HasKey(e => new { e.Id, e.IdEmpresa });

    modelBuilder.Entity<Empresa>()
        .HasMany<Empleado>(g => g.Empleados)
        .WithOne()
        .HasForeignKey(emp => emp.IdEmpresa)
        .OnDelete(DeleteBehavior.Cascade);

    ;
}

// The following configures EF to create a Sqlite database file in the
// special "local" folder for your platform.
protected override void OnConfiguring(DbContextOptionsBuilder options)
{
    options.UseSqlite($"Data Source={DbPath}");
}

}
}

```

Vamos a modificar igualmente la clase que crean datos aleatorios, GeneraCamposAleatorios, para incluir la generación de datos para Empresa. Marcados en azul están indicados los nuevos métodos.

```

using System;
using System.Collections.Generic;
using System.Text;

namespace ModeloEntityFramework.Debug
{
    class GenerarCamposAleatorios
    {
        private static string[] nombreempresas = { "GAJE", "JOMA", "JILO", "SAGE", "SOLID",
            , "TECH", "MECHANI",
                "FRUTA", "TELEFO", "INFOR", "BANCO",
            "SUPERMERCA"};
        private static string[] ciudades = { "Madrid", "Barcelona", "Valencia", "Sevilla",
            "Zaragoza", "Málaga",
                "Murcia", "Palma de Mallorca", "Las Palmas de Gran
            Canaria", "Bilbao", "Alicante",
                "Córdoba", "Valladolid", "Vigo", "Gijón", "Albacete", "Coruna", "Vitoria",
                "Granada", "Elche"};

        private static string[] nombres = {"ANTONIO", "MANUEL", "JOSE", "FRANCISCO", "DAVID",
            "JUAN", "JOSE ANTONIO", "JAVIER", "DANIEL", "JOSE LUIS",
                "FRANCISCO JAVIER", "CARLOS", "JESUS", "ALEJANDRO", "FRANCISCA", "LUCIA", "MARIA
            ISABEL", "MARIA JOSE", "ANTONIA", "DOLORES", "SARA",
            "PAULA", "ELENA", "MARIA LUISA", "RAQUEL"};
        private static string[] apellidos = {"García", "González", "Rodríguez", "Fernández",
            "López", "Martínez", "Sánchez", "Pérez", "Gómez",

```



```

"Martin", "Jiménez", "Ruiz", "Hernández", "Diaz", "Moreno", "Muñoz", "Álvarez", "Romero",
"Alonso", "Gutiérrez", "Navarro",
"Torres", "Domínguez", "Vázquez", "Ramos", "Gil", "Ramírez", "Serrano", "Blanco", "Molina",
"Morales", "Suarez", "Ortega",
"Delgado", "Castro", "Ortiz", "Rubio", "Marín", "Sanz", "Núñez", "Iglesias", "Medina",
"Garrido", "Cortes", "Castillo", "Santos"};
    private static string[] direcciones = {"Alfonso López de Haro", "Calle de Alvarfáñez
de Minaya", "Calle de Arcipreste de Hita",
"Calle de Arrabal del Agua", "Plaza de Beladiez", "Plaza de Caídos en la Guerra civil", "Calle
Minaya", "Calle Hermanos Galiano"};

    private static Random rand = new Random();

    private static Func<int, int, int> randomFromTo = (ini, fin) => rand.Next() % (fin-
ini) +ini;

    private static Func<int, int, double> randomFromToDouble = (ini, fin) =>
rand.NextDouble() * (fin - ini) + ini;

    public static string nombreAleatorio()
    {
        return nombres[randomFromTo(0, nombres.Length - 1)];
    }

    public static string apellidosAleatorio()
    {
        return apellidos[randomFromTo(0, apellidos.Length - 1)] + " " +
apellidos[randomFromTo(0, apellidos.Length - 1)];
    }

    public static DateTime fechaNacimientoAleatoria()
    {
        return new DateTime(randomFromTo(1960, 2001), randomFromTo(1, 12), randomFromTo(1,
28));
    }

    public static string direccionAleatoria()
    {
        return direcciones[randomFromTo(0, direcciones.Length - 1)]+ ","+
ciudades[randomFromTo(0, ciudades.Length - 1)];
    }

    public static double SalarioAleatorio()
    {
        return randomFromToDouble(600, 4000);
    }

    public static string NombreEmpresa()
    {
        return nombreempresas[randomFromTo(0, nombreempresas.Length - 1)]
+ciudades[randomFromTo(0, ciudades.Length - 1)];
    }

```

```

public static string TelefonoAleatorio()
{
    return "+" + randomFromTo(1, 99) + randomFromTo(1, 999) + randomFromTo(1000000,
999999);

}

public static int NumeroAleatorio (int inicio, int fin)
{
    return randomFromTo(inicio, fin);
}
}

```

Y finalmente vamos a añadir la generación de una lista de Empresas en GeneraColeccionesAleatorias. En listaEmpleados están marcados los cambios. Añadimos el nuevo método `public static List<Empresa> listaEmpresas(int numEmpresas)` que generará un conjunto de empresas con número de empleados aleatorios entre 2 y 200.

```

listaEmpresa.Add(new Empresa(i.ToString(),
                                GenerarCamposAleatorios.NombreEmpresa(),
                                GenerarCamposAleatorios.TelefonoAleatorio(),
                                GenerarCamposAleatorios.direccionAleatoria(),
ListaEmpleados(GenerarCamposAleatorios.NumeroAleatorio(2, 200), i.ToString())));

```

```

using System;
using System.Collections.Generic;
using System.Text;

namespace ModeloEntityFramework.Debug
{
    class GeneraColeccionesAleatorias
    {
        public static List<Empleado> ListaEmpleados(int numEmpleados, string idEmpresa)
        {
            List<Empleado> listaEmpleados = new List<Empleado>();

            Random r = new Random();

            for (int i = 0; i < numEmpleados; i++)
            {

```

```

        listaEmpleados.Add(new Empleado(i.ToString(), idEmpresa,
GenerarCamposAleatorios.apellidosAleatorio(),
                                GenerarCamposAleatorios.nombreAleatorio(),
GenerarCamposAleatorios.fechaNacimientoAleatoria(),
                                GenerarCamposAleatorios.direccionAleatoria(),
                                GenerarCamposAleatorios.SalarioAleatorio()));

    }

    return listaEmpleados;
}

public static List<Empresa> listaEmpresas(int numEmpresas)
{
    List<Empresa> listaEmpresa = new List<Empresa>();

    Random r = new Random();

    for (int i = 0; i < numEmpresas; i++)
    {
        listaEmpresa.Add(new Empresa(i.ToString(),
                                GenerarCamposAleatorios.NombreEmpresa(),
                                GenerarCamposAleatorios.TelefonoAleatorio(),
                                GenerarCamposAleatorios.direccionAleatoria(),
ListaEmpleados(GenerarCamposAleatorios.NumeroAleatorio(2, 200), i.ToString())));

    }

    return listaEmpresa;
}
}
}

```

Ahora es momento de compilar y generar la nueva base de datos. Borrar la migración anterior, generar la migración nueva y hacer un Update-Database. Como resultado obtendréis:

La tabla Empresa

```

migrationBuilder.CreateTable(
    name: "Empresas",
    columns: table => new
    {
        Id = table.Column<string>(type: "TEXT", nullable: false),
        RazonSocial = table.Column<string>(type: "varchar(200)", nullable: true),
        Telefono = table.Column<string>(type: "varchar(16)", nullable: true),
        Direccion = table.Column<string>(type: "varchar(500)", nullable: true)
    }
)

```

```

    },
    constraints: table =>
    {
        table.PrimaryKey("PK_Empresas", x => x.Id);
    });

```

Y la nueva tabla empleados con clave de dos atributos y clave foránea con DeleteOnCascade:

```

migrationBuilder.CreateTable(
    name: "Empleados",
    columns: table => new
    {
        Id = table.Column<string>(type: "varchar(10)", nullable: false),
        IdEmpresa = table.Column<string>(type: "TEXT", nullable: false),
        Salario = table.Column<double>(type: "DECIMAL", nullable: false),
        Apellido = table.Column<string>(type: "varchar(200)", nullable: true),
        Nombre = table.Column<string>(type: "varchar(200)", nullable: true),
        FechaNac = table.Column<DateTime>(type: "TEXT", nullable: false),
        Edad = table.Column<int>(type: "INTEGER", nullable: false),
        Direccion = table.Column<string>(type: "varchar(500)", nullable: true)
    },
    constraints: table =>
    {
        table.PrimaryKey("PK_Empleados", x => new { x.Id, x.IdEmpresa });
        table.ForeignKey(
            name: "FK_Empleados_Empresas_IdEmpresa",
            column: x => x.IdEmpresa,
            principalTable: "Empresas",
            principalColumn: "Id",
            onDelete: ReferentialAction.Cascade);
    });

```

```

using System;
using Microsoft.EntityFrameworkCore.Migrations;

namespace ModeloEntityFramework.Migrations
{
    public partial class EmpleadoEmpresaMigration : Migration
    {
        protected override void Up(MigrationBuilder migrationBuilder)
        {
            migrationBuilder.CreateTable(
                name: "Empresas",
                columns: table => new
                {
                    Id = table.Column<string>(type: "TEXT", nullable: false),
                    RazonSocial = table.Column<string>(type: "varchar(200)", nullable: true),
                    Telefono = table.Column<string>(type: "varchar(16)", nullable: true),
                    Direccion = table.Column<string>(type: "varchar(500)", nullable: true)
                },
                constraints: table =>
                {
                    table.PrimaryKey("PK_Empresas", x => x.Id);
                });

            migrationBuilder.CreateTable(

```

```

name: "Empleados",
columns: table => new
{
    Id = table.Column<string>(type: "varchar(10)", nullable: false),
    IdEmpresa = table.Column<string>(type: "TEXT", nullable: false),
    Salario = table.Column<double>(type: "DECIMAL", nullable: false),
    Apellido = table.Column<string>(type: "varchar(200)", nullable: true),
    Nombre = table.Column<string>(type: "varchar(200)", nullable: true),
    FechaNac = table.Column<DateTime>(type: "TEXT", nullable: false),
    Edad = table.Column<int>(type: "INTEGER", nullable: false),
    Direccion = table.Column<string>(type: "varchar(500)", nullable: true)
},
constraints: table =>
{
    table.PrimaryKey("PK_Empleados", x => new { x.Id, x.IdEmpresa });
    table.ForeignKey(
        name: "FK_Empleados_Empresas_IdEmpresa",
        column: x => x.IdEmpresa,
        principalTable: "Empresas",
        principalColumn: "Id",
        onDelete: ReferentialAction.Cascade);
});

migrationBuilder.CreateIndex(
    name: "IX_Empleados_IdEmpresa",
    table: "Empleados",
    column: "IdEmpresa");
}

protected override void Down(MigrationBuilder migrationBuilder)
{
    migrationBuilder.DropTable(
        name: "Empleados");

    migrationBuilder.DropTable(
        name: "Empresas");
}
}
}

```

El programa principal ahora podrá quedar de la siguiente manera.

```

using System;
using System.Linq;
using System.Collections.Generic;
using ModeloEntityFramework.Debug;

namespace ModeloEntityFramework
{
    class Program
    {
        static void Main(string[] args)
        {

            using (var db = new EmpresaContext()) {

                Console.WriteLine($"Database path: {db.DbPath}.");
            }
        }
    }
}

```

```

        // db.Empleados.Select(emp => emp).ToList().ForEach(emp => db.Remove(emp));
        // db.SaveChanges();

        db.Empleados.Select(emp => emp).ToList().ForEach(emp =>
db.Remove(emp));
        db.SaveChanges();

        List<Empresa> empresas = GeneraColeccionesAleatorias.listaEmpresas(60);

        empresas.ForEach(empresa => db.Add(empresa));

        db.SaveChanges();
        //Listado de empresas

        Console.WriteLine("Listado de Empresas");
        db.Empleados.ToList().ForEach(emp => Console.WriteLine(emp.toString()));

        // Edad media de Empleados para todas las empresas cuyo nombre empieza por A

        double resultado = db.Empleados.Where(empresa =>
empresa.RazonSocial.StartsWith("J"))
            .Join(db.Empleados, empresa => empresa.Id
                , empleado => empleado.IdEmpresa, (empresa, empleado) =>
empleado).Average(empleado => empleado.Edad);

        Console.WriteLine("Media de edad de los empleados cuyo Razon social empieza
por J:" + resultado);

        //Obtener la media de sueldos de los empleados de las empresas que esten
ubicadas en Madrid
        // La ciudad esta contenida en la direccion.

    }

}
}
}

```

5 Proyecto Final de la Unidad.

Integrar las dos siguientes clases en el modelo de manera que se creen dos tablas nuevas para Administrativo y Ejecutivo. Realizar los cambios necesarios en las clases y en el modelo para que esto sea así. Debereis modificar la Api Fluent para que esto sea así, creando un enumerado TipoEmpleado.cs.

```
using System;
using System.Collections.Generic;
using System.Text;

namespace PrimeraClase
{
    class Administrativo : Empleado
    {
        private const double PORCENTAJEIMPUESTOS = 0.10;
        public override double calculoImpuestos()
        {
            return PORCENTAJEIMPUESTOS*this.Salario;
        }

        public override double salarioNeto()
        {
            return this.Salario -this.calculoImpuestos();
        }
    }
}
```

```
using System;
using System.Collections.Generic;
using System.Text;

namespace PrimeraClase {
    class Ejecutivo : Empleado, IBonus
    {
        private const double PORCENTAJEIMPUESTOS = 0.30;

        private const double BONUS = 400;

        public double calculoBonus()
        {

```

```
        return BONUS;
    }

    public override double calculoImpuestos()
    {
        return (this.Salario + BONUS) * PORCENTAJEIMPUESTOS;
    }

    public override double salarioNeto()
    {
        return Salario + calculoBonus() - calculoImpuestos();
    }
}
}
```