



E.T.S.
INGENIERÍA
INFORMÁTICA

Gestión de la Información

Grado en Ingeniería del Software



UNIVERSIDAD
DE MÁLAGA



LENGUAJES Y
CIENCIAS DE LA
COMPUTACIÓN
UNIVERSIDAD DE MÁLAGA

Tema 4

Aplicaciones .NET

José Luis Pastrana Brincones
pastrana@lcc.uma.es



E.T.S.
INGENIERÍA
INFORMÁTICA

Gestión de la Información

Grado en Ingeniería del Software



UNIVERSIDAD
DE MÁLAGA



LENGUAJES Y
CIENCIAS DE LA
COMPUTACIÓN
UNIVERSIDAD DE MÁLAGA

Tema 4.1

Introducción a la Plataforma .NET y al Lenguaje C#

¿Por qué .NET Framework?

- Motivado por el grado de complejidad que estaba tomando la programación Windows y el propio S.O.
 - Interfaces de los API's para los diferentes lenguajes
 - Multitud de servicios duplicados
 - Pocas posibilidades de reutilización del código
- Eliminar el “infierno de las DLL Win32”
 - Conflictos entre las aplicaciones con una librería en común en diferentes versiones
- SOLUCION: Plataforma .NET
 - Servicios universales (para todos los lenguajes)
 - Mantiene la compatibilidad hacia atrás
 - Hace posible la interoperabilidad entre lenguajes

¿Por qué .NET Framework?

- Soporte para múltiples lenguajes
 - Actualmente más de 26 lenguajes
 - C++ .NET, VB.NET, Python, Java, Fortran, Delphi, Ada, etc.
 - Introduce un nuevo lenguaje - C#
 - Lenguaje intrínseco, herencia, polimorfismo, encapsulación (orientación a objetos)
 - Servicios de la plataforma expuestos de forma idéntica a todos los lenguajes
 - Tanto Biblioteca de Componentes como Servicios básicos

¿Por qué .NET Framework?

- No exclusivamente para PC's con Windows
 - .NET Compact Framework para dispositivos móviles
 - PDA's, SmartPhones, etc.
 - Mono
 - .NET para Linux
- No solo para desarrollo de aplicaciones de escritorio
 - Aplicaciones y Servicios Web
 - Aplicaciones de consola
 - Bibliotecas de clases
 - Aplicaciones para Dispositivos Móviles
- Orientado a Componentes
 - Clases
 - Eventos y delegados
 - Propiedades
 - Adiós a los manejadores, punteros, gestión de bloques de memoria, etc.

.NET Framework

Funcionamiento

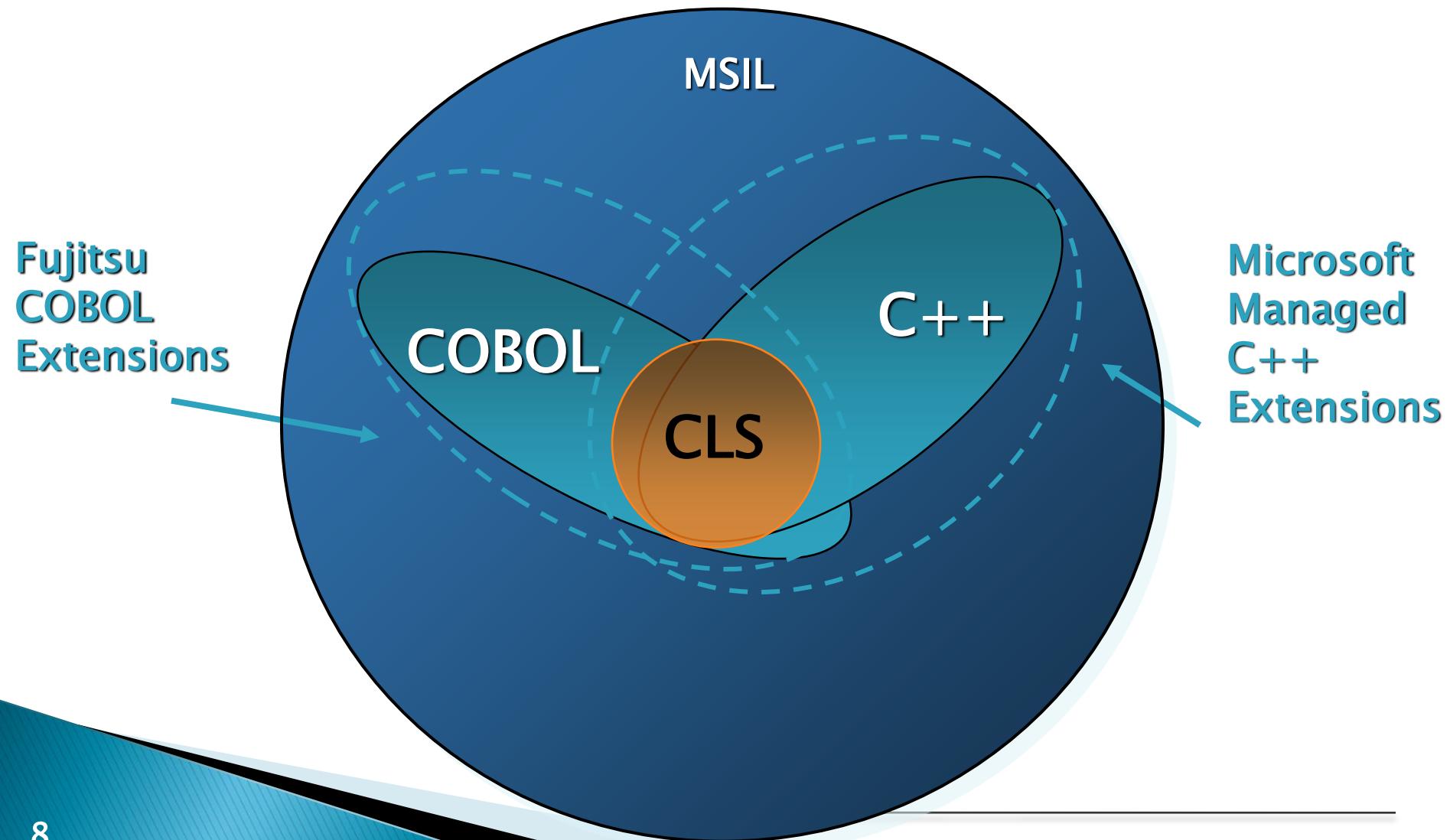
- Estructura de las aplicaciones
 - Archivos EXE y DLL's
 - Estructura interna distinta a la "tradicional"
 - Necesidad de tener instalado el .NET FrameWork
 - Contienen código independiente de la máquina
 - MSIL (Microsoft Intermediate Languaje)
 - Imposible ejecutarlo por si sólo
 - Necesidad de compilarlo previamente para el Procesador y Sistema Operativo en el que va a ejecutarse.
 - JIT (Just In-Time)
 - Compilador que realiza la compilación al vuelo del MSIL
 - Unidad mínima en memoria es el Assembly (ensamblado)

.NET Framework

Funcionamiento

- Aplicaciones -> Conjunto de ensamblados
- Application Domain (Dominio de la Aplicación)
 - Evolución del concepto de Proceso
 - Conjunto de ensamblados que comparten recursos
 - Espacio de direccionamiento
 - Modelo de hilos
 - etc.
- Generación del Código MSIL
 - Uso del CTS (Common Type System)
 - Recoge la definición de todos los tipos y sus operaciones de todos los lenguajes
 - Para la cooperación entre lenguajes se debe ajustar al CLS
 - CLS (Common Languaje Specification), subconjunto de la CTS común a todos los lenguajes

.NET Framework Funcionamiento



.NET Framework. Ensamblados

- “Piezas” con las que está constituida una aplicación
- Ensamblados se constituyen por módulos
- Manifiesto
 - Información dentro del ensamblado de la que se vale el sistema para:
 - Encontrar y diferenciar los diferentes módulos
 - Saber las dependencias del ensamblado
 - Herramienta *ildasm*
 - Nos permite examinar el contenido de un ensamblado

.NET Framework

Ensamblados

- Ensamblados privados y compartidos
 - Los ensamblados pueden ser compartidos (tipo shared)
 - Por defecto, al compilar todos son privados
 - Ensamblados .NET Vs. COM
 - Utilizando COM se deben registrar los módulos en el registro de Windows para que las aplicaciones los localicen.
 - Con .NET los ensamblados privados no precisan de registro y basta con distribuirlos junto a la aplicación
 - Ventajas:
 - Eliminación de problemas por conflicto de versiones (Infierno DLL)
 - Dejan de existir posibles entradas huérfanas por una mala desinstalación en el registro

.NET Framework

Compilación y Ejecución

- CTS y CLS
 - Permite generar un mismo código MSIL, independientemente del lenguaje de alto nivel utilizado
 - Gracias a CLS se puede utilizar código en otro lenguaje como si fuera del mismo.
 - Ej. Derivar desde C# una clase implementada en VisualBasic.NET
 - El utilizar o no el CLS depende de un atributo
 - Por defecto a “true” (permite la interoperabilidad entre lenguajes)
 - Una librería puede internamente no cumplir con CLS, pero si de cara al exterior, los tipos expuestos lo cumplen, se puede considerar que lo cumple.

.NET Framework

Compilación y Ejecución

- Ejecución supervisada (CLR)
 - Código MSIL se supervisa antes y después de ser compilado por el JIT
 - Comprobar permisos
 - Comprobar no existencia de código peligroso imposible de supervisar (punteros)
 - Eliminación de objetos (GC)
 - Etc.
 - El código bajo supervisión es llamado Manage Code o Código Gestionado.
 - Se mejora la estabilidad de las aplicaciones y del sistema
 - Se acaban los fallos incontrolados que afectan a la propia aplicación o a otras, incluso al propio sistema.

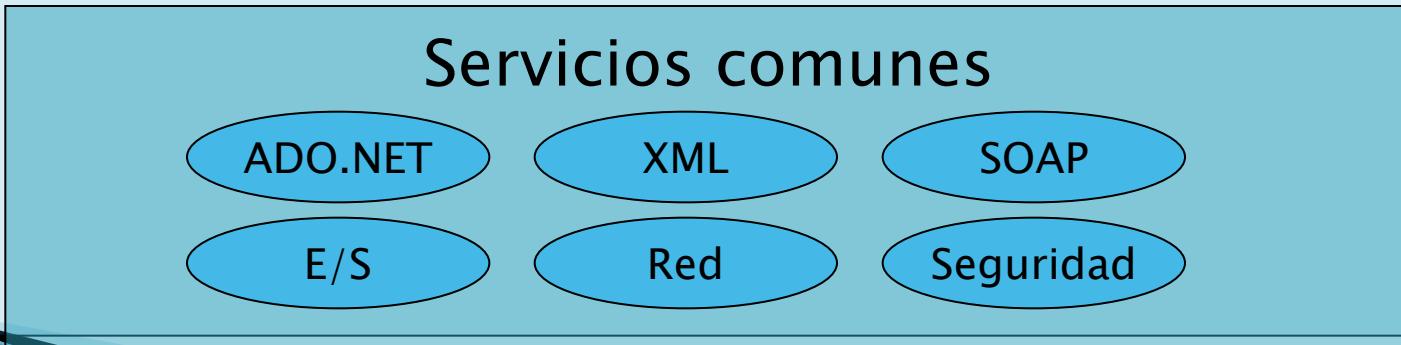
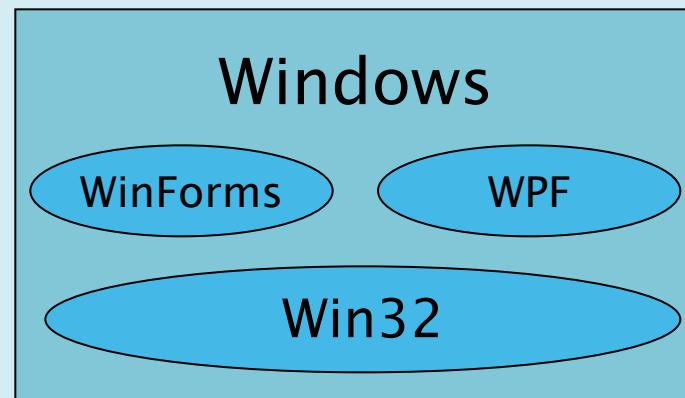
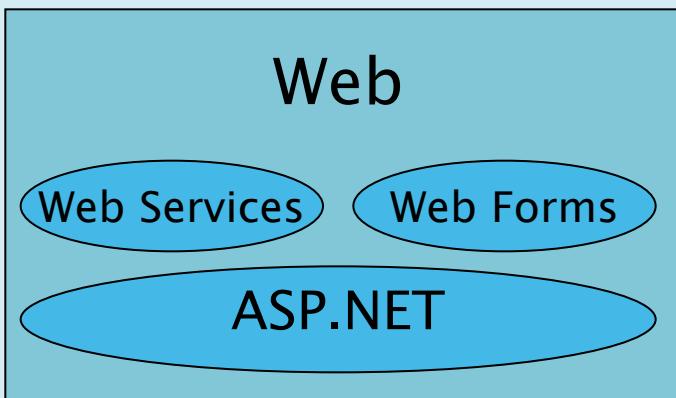
.NET Framework

Compilación y Ejecución

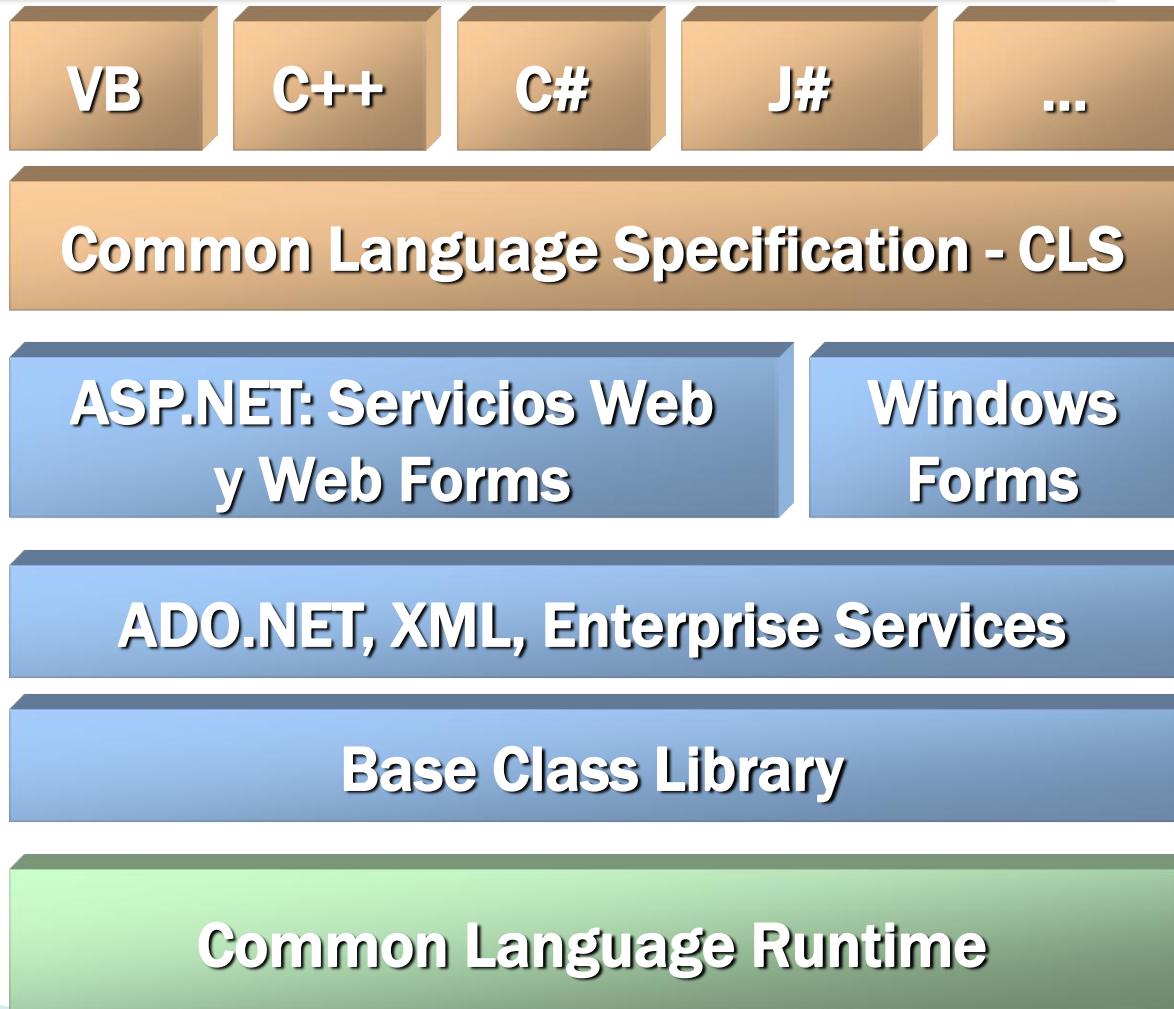
- Compilación JIT
 - Compila el código al vuelo según se necesite
 - También se conocen como *jitters*
 - Existe un compilador JIT para cada dispositivo y/o sistema
 - Código una vez compilado, no se vuelve a compilar
 - No se compila todo el código almacenado en memoria, sino solo el que se ejecuta
 - Existe la posibilidad de hacer código nativo, para cuando la aplicación solo va a ser ejecutada en un entorno determinado (dispositivo, sistema, etc).
 - Llamados *pre-jitter* pueden ser ejecutados desde la línea de comandos

.NET Framework Servicios

Servicios Microsoft .NET



.NET Framework Arquitectura



Características Principales de C#

- ▶ C# (leído en inglés “C Sharp” y en español “C Almohadilla”) es el lenguaje de propósito general diseñado por Microsoft para su plataforma .NET.
- ▶ Sus principales creadores son Scott Wiltamuth y Anders Hejlsberg, éste último también conocido por haber sido el diseñador del lenguaje Turbo Pascal y la herramienta RAD Delphi.
- ▶ La sintaxis y estructuración de C# es muy similar a la C++ o Java, lo que facilita la migración de códigos escritos en estos lenguajes a C# y su aprendizaje a los desarrolladores habituados a ellos.
- ▶ En resumen, C# es un lenguaje de programación que toma las mejores características de lenguajes preexistentes como Visual Basic, Java o C++ y las combina en uno solo.

Características Principales de C#

- ▶ C# (leído en inglés “C Sharp” y en español “C Almohadilla”) : nuevo lenguaje de propósito general de Microsoft para .NET
 - Principales creadores:
 - Scoot Wiltamuth
 - Anders Hejlsberg (Turbo Pascal y Delphi)
 - Versión 4.0
- ▶ Necesidad de un nuevo lenguaje para .NET con las siguientes características
 - Sintaxis y estructuración similar a C++
 - Sencillez y productividad equiparables a Visual Basic
 - Candidato ideal: Java
 - ¡Imposible!
- ▶ Surge C#
 - Nuevo lenguaje diseñado específicamente para .NET
 - Programación de .NET mucho más sencilla e intuitiva
 - Muy similar a Java y C++ con nuevas mejoras y orientado a componentes

Características Principales de C#

- ▶ Características de C#
 - Sencillez
 - Modernidad
 - Orientación a objetos
 - Orientación a componentes
 - Gestión automática de memoria
 - Seguridad de tipos
 - Instrucciones seguras
 - Sistema de tipo unificado
 - Extensibilidad de tipos básicos
 - Extensibilidad de operadores
 - Extensibilidad de modificadores
 - Versionable
 - Eficiente
 - Compatible

Características Principales de C#

- ▶ Características de C#
 - Sencillez
 - Modernidad
 - Orientación a objetos
 - Orientación a componentes
 - Gestión automática de memoria
 - Seguridad de tipos
 - Instrucciones seguras
 - Sistema de tipo unificado
 - Extensibilidad de tipos básicos
 - Extensibilidad de operadores
 - Extensibilidad de modificadores
 - Versionable
 - Eficiente
 - Compatible

Características Principales de C#

- ▶ Visiones tradicionales
 - C++, Java™: los tipos primitivos son “mágicos” y no interactúan con otros objetos
 - Smalltalk, Lisp: los tipos primitivos son objetos, pero con un coste en el rendimiento
- ▶ C# unifica tipos primitivos y objetos sin coste de rendimiento
 - Profunda simplicidad a través del sistema
- ▶ Extensibilidad y reutilización mejorada
 - Nuevos tipos primitivos: Decimal, SQL...
 - Colecciones, etc. trabajan para todos los tipos

Características Principales de C#

- ▶ Recolección de basura
 - No hay pérdidas de memoria ni punteros extraños
- ▶ Excepciones
- ▶ Seguridad en los tipos
- ▶ Evita errores comunes
 - Ej: if (x = y) ...

Características Principales de C#

- ▶ Recolección de basura
 - No hay pérdidas de memoria ni punteros extraños
- ▶ Excepciones
- ▶ Seguridad en los tipos
- ▶ Evita errores comunes
 - Ej: if (x = y) ...
- ▶ Interoperabilidad
 - C# se “habla” con XML, SOAP, COM, DLLs, y cualquier lenguaje de .NET
- ▶ Productividad incrementada
 - Curva de aprendizaje corta
 - Millones de líneas de C# ya escritas en .NET

Características Principales de C#

- ▶ Ejemplo:

```
using System;

class Hello
{
    static void Main( )
    {
        Console.WriteLine("Hola mundo");
        Console.ReadLine(); // Pulsar enter
    }
}
```

Características Principales de C#

- Parece Java, pero no lo es...
- Todo el código dentro de clases
- Utilización de “using” para utilizar clases no definidas en nuestro fichero
- El clásico método main
 - ¡¡¡ Ojo !!! Main()
 - Distintas posibilidades
 - static void Main()
 - static int Main()
 - ...
- La clase Console para entrada/salida por teclado

Características Principales de C#

- ▶ Un programa en C# es una colección de tipos
 - Clases, estructuras, enumeraciones, interfaces, delegados
- ▶ C# proporciona un conjunto de tipos predefinidos
 - Ej: int, byte, char, string, object, ...
- ▶ Se pueden crear nuevos tipos
- ▶ Todos los datos y el código están definidos dentro de un tipo
 - No hay variables globales
 - No hay funciones globales
 - Algunos tipos “más simples” como las enumeraciones sí que pueden ser definidos fuera de clases
 - Porque son un tipo como otro cualquiera

Características Principales de C#

- ▶ Los tipos pueden ser instanciados...
 - ...y ser entonces usados: Llamadas a métodos, obtener y establecer propiedades, etc.
- ▶ Se pueden realizar conversiones entre tipos
 - Implícita y explícitamente
- ▶ Los tipos están organizados
 - Namespaces, ficheros, ensamblados
- ▶ Hay dos categorías de tipos:
 - Valor y referencia

Características Principales de C#

- ▶ Valor
 - int
 - float
 - double
 - decimal
 - bool
 - char
- ▶ Referencias
 - object
 - string
 - ...

Características Principales de C#

- ▶ Tipos valor
 - Todos son estructuras predefinidas

Signed	sbyte, short, int, long
Unsigned	byte, ushort, uint, ulong
Character	char
Floating point	float, double, decimal
Logical	bool

Características Principales de C#

Tipo C#	Tipo System	Medida (bytes)	¿Signo?
sbyte	System.Sbyte	1	Sí
short	System.Int16	2	Sí
int	System.Int32	4	Sí
long	System.Int64	8	Sí
byte	System.Byte	1	No
ushort	System.UInt16	2	No
uint	System.UInt32	4	No
ulong	System.UInt64	8	No

Características Principales de C#

- ▶ Flotantes: siguen la especificación IEEE 754
- ▶ Soporta ± 0 , $\pm \text{Infinity}$, NaN

Tipo C#	Tipo System	Medida (bytes)
float	System.Single	4
double	System.Double	8

Características Principales de C#

- ▶ decimal: 128 bits
- ▶ Esencialmente un valor de 96 bits escalado por una potencia de 10
- ▶ Valores decimales representados con precisión
- ▶ No soporta ceros con signo, infinitos o NaN

Tipo C#	Tipo System	Medida (bytes)
decimal	System.Decimal	16

Características Principales de C#

- ▶ **bool:** Representa valores lógicos
- ▶ Puede ser true o false
- ▶ No se pueden utilizar 1 y 0 como valores booleanos
 - No hay una conversión estándar entre otros tipos y bool

Tipo C#	Tipo System	Medida (bytes)
bool	System.Boolean	1 (2 para arrays)

Características Principales de C#

- ▶ `char`: Representa un carácter Unicode
- ▶ Literales
 - ‘A’ // Carácter simple
 - ‘\u0041’ // Unicode
 - ‘\x0041’ // Unsigned short hexadecimal
 - ‘\n’ // Secuencia de escape

Tipo C#	Tipo System	Medida (bytes)
<code>char</code>	<code>System.Char</code>	2

Características Principales de C#

- ▶ `object`: Raíz de la jerarquía de objetos
- ▶ Sobrecarga de almacenamiento
 - 0 bytes para los tipos valor
 - 8 bytes para los tipos referencia

Tipo C#	Tipo System	Medida (bytes)
<code>object</code>	<code>System.Object</code>	0/8 overhead

Características Principales de C#

- ▶ public bool Equals(object)
 - ▶ protected void Finalize()
 - ▶ public int GetHashCode()
 - ▶ public System.Type GetType()
 - ▶ protected object MemberwiseClone()
 - ▶ public void Object()
 - ▶ public string ToString()
-
- ▶ Estos métodos pueden ser usados por todos los tipos, al heredar de System.Object

Características Principales de C#

- ▶ `string`: secuencia inmutable de caracteres Unicode
- ▶ Tipo referencia
- ▶ Ejemplo: `string s = "Soy un string";`

Tipo C#	Tipo System	Medida (bytes)
<code>string</code>	<code>System.String</code>	20 mínimo

Características Principales de C#

- ▶ Posible uso de secuencias de escape

```
string s1= “\\\\\\server\\\\fileshare\\\\filename.cs”;
```

- ▶ String textuales
 - La mayor parte de las secuencias de escape son ignoradas
 - Excepto para “”

```
string s2 = @“\\server\\fileshare\\filename.cs”;
```

- ▶ Numerosos métodos y propiedades
 - `Chars`, `Length`, `Compare`, `CompareTo`, `Concat`, `Contains`, `Copy`, `CopyTo`, `EndsWith`, `Equals`, `Format`, `IndexOf`, `IndexOfAny`, `Insert`, `IsNullOrEmpty`, `IsNullOrWhiteSpace`, `Join`, `LastIndexOf`, `PadLeft`, `PadRight`, `Remove`, `Replace`, `Split`, `StartsWith`, `Substring`, `ToCharArray`, `ToLower`, `ToUpper`, `Trim`, `TrimEnd`, `TrimStart`, etc.

Características Principales de C#

▶ Ejemplo:

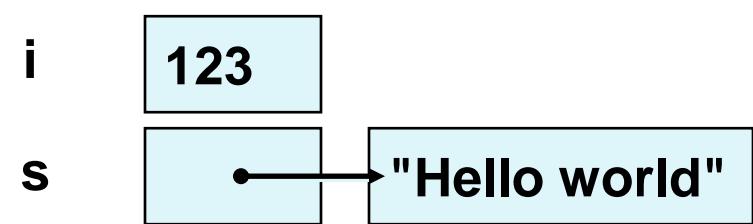
```
using System;

class StringExample
{
    public static void Main()
    {
        string s1 = "string a";
        string s2 = s1;
        Console.WriteLine("s1 es " + s1);
        Console.WriteLine("s2 es " + s2);
        s1 = "otro string";
        Console.WriteLine("s1 es ahora " + s1);
        Console.WriteLine("s2 es ahora " + s2);
    }
}
```

C#: El Sistema Unificado de Tipos

- ▶ Tipos valor
 - Directamente contienen datos
 - No pueden ser null
 - Se alojan en la pila
- ▶ Tipos referencia
 - Contienen referencias a objetos
 - Pueden ser null
 - Se alojan en el *heap manejado*

```
int i = 123;  
string s = "Hello world";
```



C#: El Sistema Unificado de Tipos

- ▶ Tipos valor
 - Primitivos `int i; float x;`
 - Enums `enum State { Off, On }`
 - Structs `struct Point {int x,y;}`
- ▶ Tipos referencia
 - String `string`
 - Clases `class Foo: Bar, IFoo {...}`
 - Interfaces `interface IFoo: IBar {...}`
 - Arrays `string[] a = new string[10];`
 - Delegados `delegate void Empty();`

C#: El Sistema Unificado de Tipos

	Valor	Referencia
Variable mantiene	Valor actual	Dirección de memoria
Alojada en	Stack, miembro	Heap
Posibilidad de null	Siempre tiene valor	Puede ser null
Valor por defecto	0	Null
Alias (en un ámbito)	No	Sí
La asignación significa...	Copiar datos	Copia de referencia

C#: El Sistema Unificado de Tipos

- ▶ Beneficios de los tipos valor
 - No hay alojamiento en el heap, menos presión para el GC
 - Uso más eficiente de memoria
 - Una indirección de memoria menos
 - Más eficiente
 - Sistema Unificado de Tipos
 - No existe la dicotomía tipo primitivo/objeto

C#: El Sistema Unificado de Tipos

- ▶ Conversiones implícitas
 - Ocurren automáticamente
 - Siempre funcionan
 - No hay pérdida de información (precisión)
- ▶ Conversiones explícitas
 - Requieren un cast
 - Pueden no funcionar
 - Puede haber pérdida de información (precisión)
- ▶ Ambos tipos de conversiones pueden ser definidas por el usuario

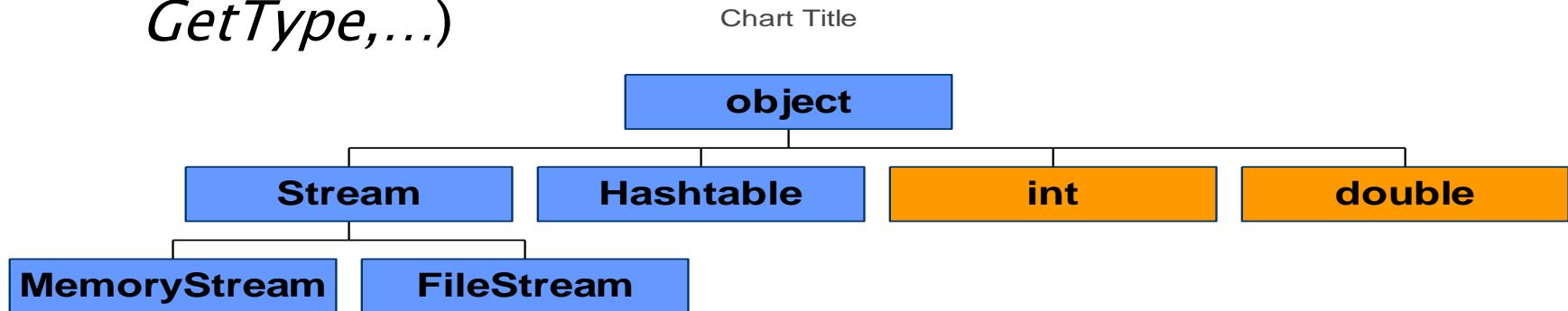
C#: El Sistema Unificado de Tipos

```
int x = 123456;                                // implicit
long y = x;                                     // explicit
short z = (short)x;

double d = 1.2345678901234;                     // explicit
float f = (float)d;
long l = (long)d;                               // explicit
```

C#: El Sistema Unificado de Tipos

- ▶ Todo es un objeto (Tipo referencia predefinido)
 - Todos los tipos predefinidos y creados por el usuario heredan de object
 - Muy similar a Java
 - Implementa métodos de propósito general (*Equals*, *GetType*,...)



C#: El Sistema Unificado de Tipos

▶ Polimorfismo

- La habilidad para realizar una operación sobre un objeto sin conocer el tipo preciso del mismo

```
void Pinta(object o)
{
    Console.WriteLine(o.ToString());
}
```

```
Pinta(42);
Pinta("abcd");
Pinta(12.345678901234m);
Pinta(new Point(23,45));
```

C#: El Sistema Unificado de Tipos

- ▶ Pregunta: ¿Cómo podemos tratar tipos valor y referencias de forma polimórfica?
 - ¿Cómo puede ser un int (tipo valor) ser convertido en un objeto (tipo referencia)?
- ▶ Respuesta: Boxing (embalado)
 - Sólo los tipos valor son embalados
 - Los tipos referencia no son embalados

C#: El Sistema Unificado de Tipos

- ▶ **Boxing**
 - Copia un valor dentro de un tipo referencia (object)
 - Cada tipo valor tiene su correspondiente tipo referencia "oculto"
 - Notar que se hace una copia de tipo referencia del tipo valor
 - No se está "apuntando" al tipo valor
 - El tipo valor es convertido ímplicitamente a object, un tipo referencia
 - Esencialmente un "up cast"

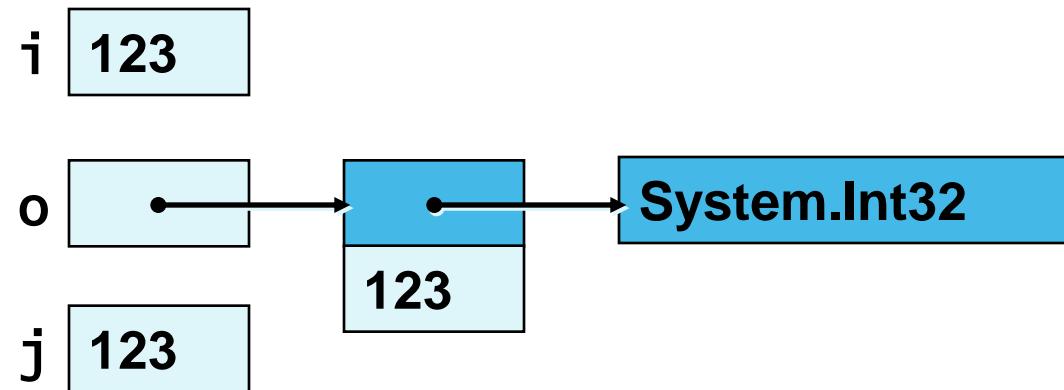
C#: El Sistema Unificado de Tipos

- ▶ **Unboxing**
 - Operación inversa del boxing
 - Copia el valor fuera de la “caja”
 - Copia desde un tipo referencia a tipo valor
 - Requiere una conversión explícita
 - Podría no funcionar (como todas las conversiones explícitas)
 - Esencialmente un “down cast”

C#: El Sistema Unificado de Tipos

▶ Boxing y Unboxing

```
int i = 123;  
  
object o = i;  
  
int j = (int)o;
```



C#: El Sistema Unificado de Tipos

- ▶ Beneficios del boxing
 - Permite poliformismo entre todos los tipos
 - Las clases de colecciones trabajan con todos los tipos
 - Elimina la necesidad de clases “wrapper”
- ▶ Desventajas del boxing
 - Coste en el rendimiento
- ▶ La necesidad para el boxing disminuye con el uso de “generics” (similar a los templates de C++ o Java)

C#: El Sistema Unificado de Tipos

▶ Tipos Definidos por el Usuario

Enumeraciones	enum
Arrays	int[], string[]
Interfaces	interface
Tipos referencia	class
Tipos valor	struct
Puntero a función	delegate

C#: El Sistema Unificado de Tipos

- ▶ Enumeraciones: Una enumeración define un nombre de tipo para un grupo de constantes simbólicas relacionadas
- ▶ Las constantes deben ser conocidas en tiempo de compilación
- ▶ Fuertemente tipadas
 - No hay conversiones implícitas a/desde int
 - Pueden ser convertidas explícitamente
 - Operadores: +, -, ++, --, &, |, ^, ~, ...
- ▶ Pueden especificar un tipo subyacente

C#: El Sistema Unificado de Tipos

▶ Ejemplo:

```
enum Color: byte
{
    Rojo = 1,
    Verde = 2,
    Azul = 4,
    Negro = 0,
    Blanco = Rojo | Verde | Azul
}

Color c = Color.Negro;
Console.WriteLine(c);          // 0
Console.WriteLine(c.Format()); // Negro
```

C#: El Sistema Unificado de Tipos

- ▶ Todas las enumeraciones derivan de `System.Enum`
 - Proporciona métodos para
 - Determinar el tipo subyacente
 - Comprobar si un valor es soportado
 - Inicializar desde una cadena de caracteres
 - Recuperar todos los valores de la enumeración

C#: El Sistema Unificado de Tipos

- ▶ ARRAYS: Los arrays permiten a un grupo de elementos de un tipo específico ser almacenados en un bloque de memoria contiguo
- ▶ Los arrays son tipos referencia
- ▶ Derivan de System.Array
- ▶ Primer elemento: 0
- ▶ Pueden ser multidimensionales
 - Los arrays conocen sus longitudes y rangos
- ▶ Chequeo de límites
 - Excepción System.OutOfBoundsException

C#: El Sistema Unificado de Tipos

- ▶ Declaración

```
int[] primes;
```

- ▶ Asignación de memoria

```
int[] primes = new int[9];
```

- ▶ Inicialización

```
int[] prime = new int[] {1,2,3,5,7,11,13,17,19};  
int[] prime = {1,2,3,5,7,11,13,17,19};
```

- ▶ Acceso y asignación

```
prime2[i] = prime[i];
```

```
foreach (int i in prime) Console.WriteLine(i);
```

C#: El Sistema Unificado de Tipos

▶ Trabajando con arrays

- Length: número de elementos

```
int num_elems = miarray.Length;
```

- Sort: Ordenación de arrays para tipos predefinidos

```
Array.Sort(miarray)
```

- Reverse: Inversión del orden de los elementos

```
Array.Reverse(miarray)
```

- Rank: Número de dimensiones de la tabla

```
int num_dims = miarray.Rank;
```

C#: El Sistema Unificado de Tipos

▶ Arrays multidimensionales

- Rectangulares

- `int[,] matR = new int[2,3];`
- Pueden inicializarse declarativamente
- `int[,] matR =
 new int[2,3] { {1,2,3}, {4,5,6} };`
- O con procedimientos

```
double [,] matriz = new double[10,10];  
  
for (int i=0; i<10; i++)  
    for (int j=0; j<10; j++)  
        matriz[i,j]=4;
```

C#: El Sistema Unificado de Tipos

- ▶ Arrays multidimensionales
 - Recortados (jagged)
 - Un array de arrays
 - No soportados por todos los lenguajes de .NET
 - `int[][] matJ = new int[2][];`
 - Deben inicializarse proceduralmente

```
int [][] A = new int[3][];  
  
A[0] = new int[4];  
A[1] = new int[3];  
A[2] = new int[1];
```

–Número de elementos de las dimensiones: **GetLength**

```
int dim = miarray.GetLength(0);
```

C#: El Sistema Unificado de Tipos

▶ Ejemplo:

```
string[][] novelistas = new string[3][];
novelistas[0] = new string[] {
    "Fyodor", "Mikhailovich", "Dostoyevsky"};
novelistas[1] = new string[] {
    "James", "Augustine", "Aloysius", "Joyce"};
novelistas[2] = new string[] {
    "Miguel", "de Cervantes", "Saavedra"};

for (i = 0; i < novelists.GetLength(0); i++)
{
    for (j = 0; j < novelists[i].GetLength(0); j++)
        Console.Write(novelistas[i][j] + " ");
    Console.Write("\n");
}
```

C#: Clases

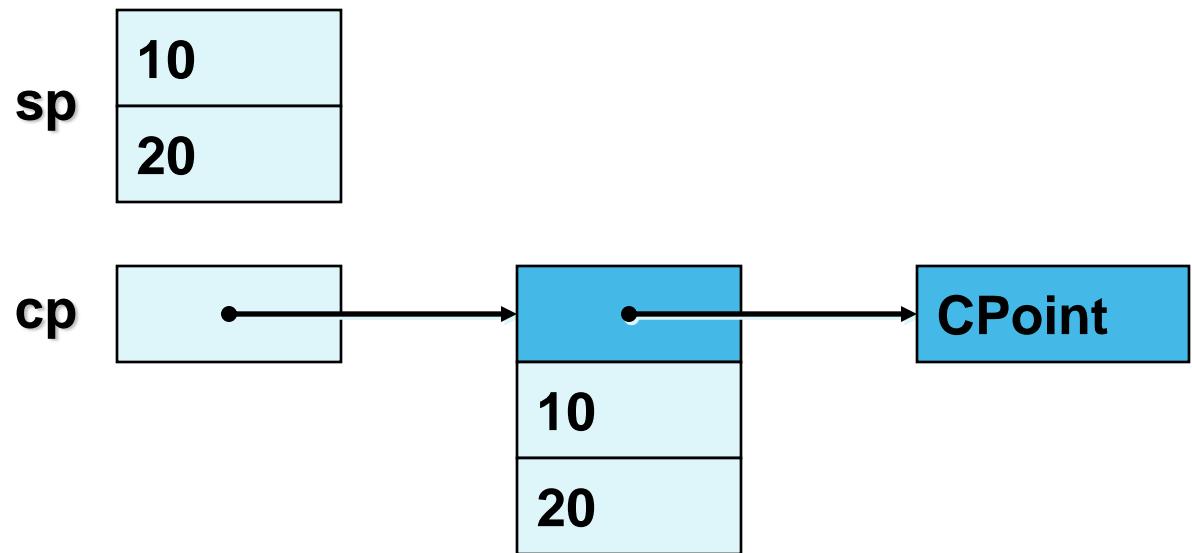
- ▶ Tipo referencia definido por el usuario
 - Similar a las clases de C++ y Java
- ▶ Herencia simple de clases
- ▶ Herencia múltiple de interfaces
- ▶ Miembros
 - Constantes, atributos, métodos, operadores, constructores, destructores
 - Propiedades, indexadores, eventos
 - Miembros de instancia y estáticos
- ▶ Tipos de acceso a los miembros
 - public, protected, private, internal, protected internal
 - private por defecto
- ▶ Instanciación con el operador new

C#: Structs

- ▶ Similar a las clases, pero...
 - Tipos valor definidos por el usuario
 - Siempre heredan de object (realmente de System.ValueType)
- ▶ Ideal para objetos “ligeros”
 - int, float, double, etc., todos son estructuras
 - Tipos “primitivos” definidos por el usuario
 - Complex, point, rectangle, color, rational
- ▶ Herencia múltiple de interfaces
- ▶ Los mismos miembros que las clases
- ▶ Acceso a los miembros
 - public, internal, private
- ▶ Instanciación con el operador new
 - Y con sintaxis C/C++ (new no es necesario)

C#: Clases y Structs

```
struct SPoint { int x, y; ... }  
class CPoint { int x, y; ... }  
  
SPoint sp = new SPoint(10, 20);  
CPoint cp = new CPoint(10, 20);  
SPoint sp2; sp2.x=10; sp2.y=20;
```

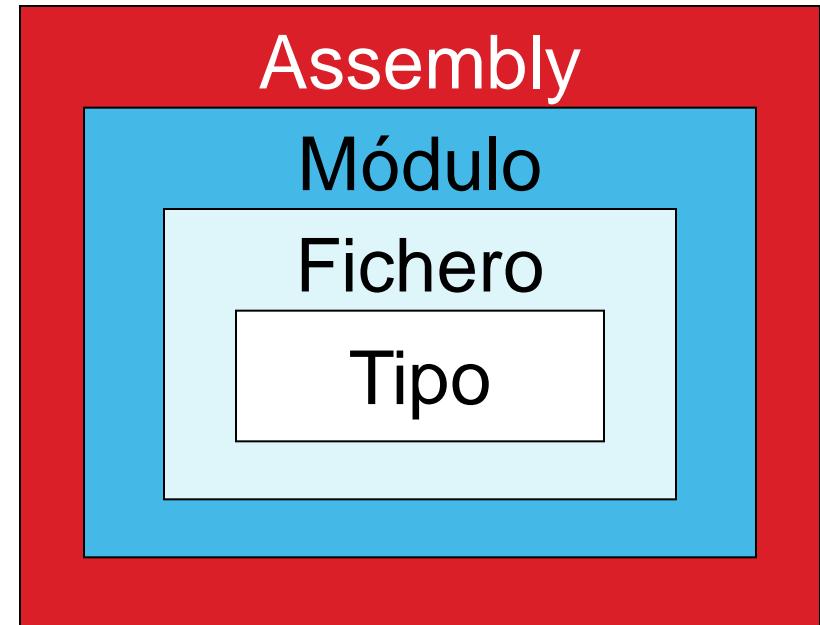


C#: Delegados

- ▶ Un delegado es un tipo referencia que define una signatura de un método
- ▶ Cuando se instancia, un delegado almacena uno o más métodos
 - Esencialmente un puntero a función orientado a objetos
- ▶ Base para los eventos del marco de trabajo

C#: Tipos

- ▶ Organización física
 - Los tipos están definidos en ficheros
 - Los ficheros son compilados en módulos
 - Los módulos están agrupados en Assemblies



C#: Tipos

- ▶ Los tipos están definidos en ficheros
 - Un fichero puede contener múltiples tipos
 - Cada tipo está definido en un solo fichero
¡NO!
 - En C# ya existen las clases parciales
- ▶ Los ficheros son compilados en módulos
 - Un módulo es una DLL o un EXE
 - Un módulo puede contener múltiples ficheros
- ▶ Los modulos están agrupados en Assemblies
 - Un Assembly puede contener múltiples módulos
 - Los Assemblies y los módulos tienen frecuentemente una relación 1:1

C#: Tipos

- ▶ No existen ficheros de cabecera
 - “Programación en un paso”
 - No es necesario sincronizar ficheros de cabecera y fuentes
 - El código es escrito “in-line”
 - La declaración y la definición son una y la misma
- ▶ No hay dependencias por el orden de declaración
 - No se necesitan referencias forward

C#: Espacios de Nombres

- ▶ Los espacios de nombres (Namespaces) proporcionan una forma para identificar únicamente un tipo
- ▶ Proporciona una organización lógica de tipos
- ▶ Los Namespaces pueden abarcar assemblies
- ▶ Puede haber namespaces anidados
- ▶ No hay relación entre los namespaces y la estructura de los directorios y ficheros (al contrario que Java)
- ▶ El nombre completo de un tipo incluye todos los namespaces
- ▶ Los namespaces proporcionan atajos de nombres a nivel de lenguaje
 - No se tiene que escribir un nombre completo calificado una y otra vez

C#

- ▶ Alta semejanza con C++ y Java
- ▶ if, while, do requieren una condición bool
- ▶ Los goto no pueden saltar dentro de bloques
- ▶ Sentencia switch
 - No hay caída por múltiples ramas
- ▶ Sentencia foreach
- ▶ Sentencias checked y unchecked
- ▶ Control de expresiones como sentencias

```
void Foo() {  
    i == 1;    // error  
}
```

C#

- ▶ Listas de sentencias
- ▶ Bloques de sentencias
- ▶ Sentencias etiquetadas
- ▶ Declaraciones
 - Constantes
 - Variables
- ▶ Sentencias de expresiones
 - checked, unchecked
 - lock
 - using
- ▶ Condicionales
 - if
 - switch
- ▶ Sentencias de bucle
 - while
 - do
 - for
 - foreach
- ▶ Sentencias de salto
 - break
 - continue
 - goto
 - return
- ▶ Manejo de excepciones
 - try / catch
 - throw

C#

- ▶ Las sentencias terminan con un punto y coma (;)
- ▶ Como C, C++ y Java
- ▶ Los bloques { ... } no necesitan punto y coma
- ▶ Comentarios
 - // Comentario en línea única, estilo C++
 - /* Comentario en múltiples
líneas,
Estilo C
*/

C#

- Declaración similar a C++ y Java
 - No hay variables globales

```
static void Main()
{
    const float pi = 3.14;
    const int r = 123;
    Console.WriteLine(pi * r * r);

    int a;
    int b = 2, c = 3;
    a = 1;
    Console.WriteLine(a + b + c);
}
```

C#

- ▶ El ámbito de una variable o constante comienza en el punto de declaración y finaliza al final de su bloque
- ▶ Dentro del ámbito de una variable o constante es un error declarar otra variable o constante con el mismo nombre

```
{  
    int x;  
    {  
        int x; // Error: ya existe x  
    }  
}
```

C# Sentencias condicionales

if

- ▶ Sintaxis similar a C++ y Java
- ▶ Requieren una expresión de tipo bool

```
int Test(int a, int b)
{
    if (a > b)      return 1;
    else if (a < b) return -1;
    else            return 0;
}
```

C# Sentencias condicionales switch

- ▶ Puede utilizarse sobre cualquier tipo predefinido (incluyendo string) o enum
- ▶ Deben indicar explícitamente cómo finalizar un caso

```
int Test(string label)
{
    int result;
    switch(label)
    {
        case null:                  result = 0; break;
        case "fastest":             result = 1; break;
        case "winner":              result = 1; break;
        case "runner-up":           result = 2; break;
        default:                    result = 0;
    }
    return result;
}
```

C# Sentencias Iteración. Bucles

- ▶ Sintaxis similar a C++ y Java
- ▶ Requiere una expresión booleana

```
int i = 0;  
while (i < 5)  
{  
    ...  
    i++;  
}
```

```
int i = 0;  
do  
{  
    ...  
    i++;  
} while (i < 5);
```

```
for (int i=0; i < 5; i++)  
{  
    ...  
}
```

C# Sentencias Iteración. Bucle foreach

- ▶ Sentencia muy útil para realizar iteraciones sobre colecciones
- ▶ Implementación de `IEnumerable`

```
public static void Main(string[] args)
{
    foreach (string s in args) Console.WriteLine(s);
}
```

```
foreach (Customer c in customers.OrderBy("name"))
{
    if (c.Orders.Count != 0)
    {
        ...
    }
}
```

C# Manejo de Excepciones

- ▶ Las excepciones son el mecanismo de C# para manejar condiciones de error inesperadas
- ▶ Son superiores a los valores de retorno
 - No pueden ser ignoradas
 - No tienen que ser manejadas en el punto en el que ocurren
 - Pueden ser usadas incluso en situaciones donde no se podrían retornar valores (ej: accediendo a una propiedad)
 - Se proporcionan excepciones estándares
 - El usuario puede definir sus propios tipos de excepciones

C# Manejo de Excepciones

- ▶ Sentencias try...catch...finally
- ▶ Los bloques try contienen código que podría elevar excepciones
- ▶ Los bloques catch manejan las excepciones
- ▶ Pueden haber múltiples bloques catch para manejar diferentes clases de excepciones
- ▶ Los bloques finally contienen código que siempre será ejecutado
 - No se pueden usar sentencias (ej: goto) para salir de un bloque finally

C# Manejo de Excepciones

- ▶ La sentencia throw eleva una excepción
- ▶ Una excepción está representada como una instancia de `System.Exception` o una clase derivada
 - Contiene información sobre la excepción
 - Propiedades
 - `Message`
 - `StackTrace`
 - `InnerException`
- ▶ Se puede reelevar una excepción, o capturar una excepción y elevar otra diferente

C# Manejo de Excepciones

```
try
{
    Console.WriteLine("try");
    throw new Exception("message");
}
catch (ArgumentNullException e)
{
    Console.WriteLine("null argument capturado");
}
catch
{
    Console.WriteLine("catch");
}
finally
{
    Console.WriteLine("finally");
}
```

C# Sincronización

- ▶ Utilización con aplicaciones multihebra
- ▶ Necesidad de preparación para el acceso concurrente a los datos
 - Deben prevenir la corrupción de los mismos
- ▶ La sentencia lock puede utilizarse para proporcionar exclusión mutua sobre una instancia
 - Solamente una sentencia lock puede tener acceso a la misma instancia en un momento determinado
 - Utiliza la clase System.Threading.Monitor del marco de trabajo .NET para proporcionar exclusión mutua

C# Sincronización

```
public class CheckingAccount
{
    decimal balance;
    public void Deposit(decimal amount)
    {
        lock (this) { balance += amount; }
    }
    public void withdraw(decimal amount)
    {
        lock (this) { balance -= amount; }
    }
}
```

C# Using como sentencia

- ▶ C# realiza una gestión automática de la memoria (recolección de basura)
 - Elimina la mayor parte de los problemas de gestión de memoria
- ▶ Sin embargo, esta gestión resulta en una finalización no determinista
 - No garantiza cuando son llamados los destructores de los objetos
- ▶ Los objetos que necesiten ser “limpiados” después de su utilización, deberían implementar la interfaz `System.IDisposable`
 - Sólo un método: `Dispose()`
- ▶ La sentencia `using` permite crear una instancia, usarla y asegurar que `Dispose` es llamado
 - Se garantiza que `Dispose` es llamado como si estuviera en un bloque `finally`

C# Using como sentencia

```
public class MyResource : IDisposable {  
    public void MyResource() {  
        // Adquiere un recurso valioso  
    }  
    public void Dispose() {  
        // Libera el recurso  
    }  
    public void DoSomething() {  
        ...  
    }  
}  
using (MyResource r = new MyResource()) {  
    r.DoSomething();  
} // r.Dispose() es llamado
```

C# checked y unchecked

- ▶ Las sentencias checked y unchecked permiten realizar chequeos de overflow para operaciones aritméticas de tipos enteros y conversiones
- ▶ checked fuerza el chequeo
- ▶ unchecked fuerza el no chequeo
- ▶ Se pueden usar como sentencias de bloque y como expresiones
- ▶ Por defecto: unchecked
- ▶ Usar la opción /checked del compilador para que checked sea el valor por defecto.
- ▶ Ejemplo

```
byte b = 255;  
checked { b++; }  
Console.WriteLine(b.ToString());
```

▶ Se eleva excepción OverflowException

C# Entrada/Salida

- ▶ Aplicaciones de consola
 - System.Console.WriteLine();
 - System.Console.Write();
 - System.Console.ReadLine();
 - System.Console.Read();
- ▶ Aplicaciones Windows
 - System.WinForms.MessageBox.Show();

```
string v1 = "Algún valor";
MyObject v2 = new MyObject();
Console.WriteLine("Primero {0}, segundo {1}", v1, v2);
```

C# Entrada/Salida

- ▶ Formato de cadenas de salida
 - {n}: muestra el valor del parámetro n-ésimo siguiendo a la cadena
 - Anchura y justificación: {n,wj}
 - wj: especificación de anchura
 - Justificación derecha: valores positivos
 - Justificación izquierda: valores negativos

```
int i=940;
int j=73;
Console.WriteLine("{0,4}\n+{1,4} ----\n {2,4}\n",
                  i,j,i+j);
```

C# Operadores

- ▶ C# proporciona un conjunto fijo de operadores, cuyo significado está definido para los tipos predefinidos
- ▶ Algunos operadores pueden ser sobrecargados (ej: +)
- ▶ La siguiente tabla resume los operadores de C# por categoría
 - Las categorías están en orden de precedencia decreciente
 - Los operadores de cada categoría tienen la misma precedencia

C# Operadores

▶ Precedencia

Categoría	Operadores
Primarios	Agrupamiento: (x) Acceso a miembros: x.y Llamada a métodos: f(x) Acceso a elementos: a[x] Post-incremento: x++ Post-decremento: x— Creación de instancias: new Recuperación de tipo: typeof check on: checked check off: unchecked

C# Operadores

▶ Precedencia

Categoría	Operadores
Unarios	Valor positivo de: + Valor negativo de: - Not: ! Negación (bits): ~ Pre-incremento: ++x Post-decremento: --x Casting: (T)x
Multiplicativos	Multiplicación: * División: / Resto: %

C# Operadores

▶ Precedencia

Categoría	Operadores
Aditivos	Sumar: + Restar: -
Desplazamiento	Shift bits left: << Shift bits right: >>
Lógicos	Menor que: < Mayor que: > Menor que o igual a: <= Mayor que o igual a: >= Igualdad/compatibilidad de tipos: is Conversión de tipos: as

C# Operadores

▶ Precedencia

Categoría	Operadores
Igualdad	Igual: == No igual: !=
AND (bits)	&
XOR (bits)	^
OR (bits)	
AND lógico	&&
OR lógico	

C# Operadores

▶ Precedencia

Categoría	Operadores
Condicional ternario	?:
Asignación	=, *=, /=, %=, +=, -=, <<=, >>=, &=, ^=, =

C# Operadores

- ▶ Los operadores de asignación y el condicional ternario son asociativos hacia la derecha
 - Las operaciones son realizadas de derecha a izquierda
 - $x = y = z$ evalua a $x = (y = z)$
- ▶ Todos los demás operadores binarios son asociativos hacia la izquierda
 - Las operaciones son realizadas de izquierda a derecha
 - $x + y + z$ evalua a $(x + y) + z$
- ▶ Utilizar paréntesis para controlar el orden

C# Clases VS Estructuras

- ▶ Conceptos CLAVE
 - Especialmente las clases
- ▶ Ambas son tipos definidos por el usuario
- ▶ Ambas pueden implementar múltiples interfaces
- ▶ Ambas pueden contener
 - Datos
 - Campos, constantes, eventos, arrays
 - Funciones
 - Métodos, propiedades, indexadores, operadores, constructores
 - Definiciones de tipos
 - Clases, estructuras, enumeraciones, interfaces, delegados

C# Clases VS Estructuras

Clase	Estructura
Tipo referencia	Tipo valor
Pueden heredar desde cualquier tipo referencia no-sealed	No hay herencia (únicamente de <code>System.valueType</code>)
Pueden tener un destructor	Sin destructor
Pueden tener constructores definidos por el usuario sin parámetros	No pueden tener constructores definidos por el usuario sin parámetros

C# Clases VS Estructuras

- ▶ Las estructuras C# SON muy diferentes de las estructuras C++

C++ Struct	C# Struct
Igual que las clases C++, pero todos los miembros son públicos	Tipo valor definido por el usuario
Pueden ser alojados en heap, stack o como miembros (pueden ser usados por valor o referencia)	Siempre son alojados en el stack o como un miembro
Los miembros son siempre public	Los miembros pueden ser public , internal or private

C# Clases VS Estructuras

```
public class Coche : Vehiculo
{
    public enum Marca { GM, Honda, BMW }
    Marca marca;
    string id;
    Point localizacion;
    Coche(Marca m, string id; Point loc)
    {
        this.marca = m;
        this.id = id;
        this.localizacion = loc;
    }
    public void Conducir()
    { Console.WriteLine("Broom"); }
}
```

```
Coche c =
    new Coche(Coche.Marca.BMW,
              "E-2380-CDB",
              new Point(3,7));
c.Conducir();
```

C# Clases VS Estructuras

```
public struct Point
{
    int x, y;
    public Point(int x, int y)
    {
        this.x = x;
        this.y = y;
    }
    public int X { get { return x; } set { x = value; } }
    public int Y { get { return y; } set { y = value; } }
}
```

```
Point p = new Point(2,5);
p.X += 100;
int px = p.X;      // px = 102
```

C# Miembros de instancia y Miembros estáticos

- ▶ Por defecto, los miembros son por instancia
 - Cada instancia mantiene sus propios campos
 - Los métodos se aplican a una instancia específica
- ▶ Los miembros estáticos son por tipo
 - Los métodos estáticos no pueden acceder a datos de instancias
 - No se puede utilizar “this” en métodos estáticos
- ▶ No abusar de los miembros estáticos
 - Son esencialmente datos y funciones globales orientados a objetos

C# Modificadores

- ▶ Los modificadores de acceso especifican quién puede usar un tipo o miembro
- ▶ Los modificadores de acceso controlan la encapsulación
- ▶ Los tipos de alto nivel (los accesibles directamente en un namespace) pueden ser `public` o `internal`
- ▶ Los miembros de clases pueden ser `public`, `private`, `protected`, `internal`, o `protected internal`
- ▶ Los miembros de estructuras pueden ser `public`, `private` o `internal`

C# Modificadores

Si el modificador de acceso es	Entonces un miembro definido en el tipo T y assembly A es accesible
<code>public</code>	A todo el mundo
<code>private</code>	Dentro de T únicamente (por defecto)
<code>protected</code>	a T o tipos derivados de T
<code>internal</code>	a tipos dentro de A
<code>protected internal</code>	a T o tipos derivados de T o a tipos dentro de A

C# Clases Abstractas

- ▶ Una clase abstracta es aquélla que no puede ser instanciada
 - Declaración con abstract
- ▶ Con propósito de ser usada como una clase base
 - Puede contener funciones miembros abstractas y no abstractas
- ▶ Similar a una interfaz
- ▶ Tiene que poderse heredar de ella
 - No puede ser sellada (sealed)

```
abstract class Edificio {  
    public abstract decimal CalculoCostes();  
    //No tiene implementacion y tiene que ser  
    //sobrecargada en una clase no abstracta  
}
```

C# Clases Selladas

- ▶ Una clase sellada es aquélla que no puede ser usada como una clase base
 - Declaración con sealed
- ▶ Las clases selladas no pueden ser abstractas
- ▶ Todas las estructuras son implícitamente selladas
- ▶ ¿Para qué sellar una clase?
 - Para prevenir herencias no deseadas
 - Optimización de código
 - Las llamadas a funciones virtuales pueden ser resueltas en tiempo de compilación

C# Clases Parciales

- La palabra clave “partial” permite dividir una clase, estructura o interfaz entre múltiples ficheros
 - Ej: Utilizado por el diseñador de formularios para separar código de usuario de código de interfaz
- Ej:

```
//ClaseGrande1.cs
partial class ClaseGrande  {
    public void MetodoUno()      { ... }
}

//ClaseGrande2.cs
partial class ClaseGrande  {
    public void MetodoDos()      { ... }
}
```

C# Clases Estáticas

- ▶ Una clase estática tiene el modificador static y no permite la creación de instancias
 - Ej: Clases de utilidades
- ▶ Ej:

```
static class StaticUtilities {  
    public static void HelperMethod() {...}  
}
```

Utilización:

```
StaticUtilities.HelperMethod();
```

C# this y base

- ▶ La palabra clave **this** es una variable predefinida disponible en funciones miembro no estáticas
 - Usada para acceder a datos y funciones miembro sin ambigüedad
- ▶ La palabra clave **base** es utilizada para acceder a miembros de clases que están ocultos por miembros de clase con nombres similares de la clase actual

C# this y base

```
class Persona
{
    string nombre;
    public Persona(string nombre) { this.nombre = nombre; }
    public void Introduce(Persona p)
    {
        if (p != this) Console.WriteLine("Soy " + nombre);
    }
}
```

```
class Shape
{
    int x, y;
    public override string ToString()
    { return "x=" + x + ",y=" + y; }
}
class Circle : Shape {
    int r;
    public override string ToString()
    { return base.ToString() + ",r=" + r; }
}
```

C# Constantes

- ▶ Una constante es un miembro dato que puede ser evaluada en tiempo de compilación y que es implícitamente estático (por tipo)
 - Ej: Math.PI

```
public class MyClass
{
    public const string version = "1.0.0";
    public const string s1 = "abc" + "def";
    public const int i3 = 1 + 2;
    public const double PI_I3 = i3 * Math.PI;
    public const double s = Math.Sin(Math.PI); //ERROR
    ...
}
```

C# Campos

- ▶ Un campo es una variable miembro
 - Atributo
- ▶ Mantiene datos para una clase o estructura
- ▶ Puede mantener:
 - Una instancia de clase (una referencia),
 - Una instancia de estructura (datos), o
 - Un array de instancias de clases o estructuras (un array es por sí mismo una referencia)

```
public class MiClase
{
    otraClase o;
    int x;
    string cad;
    ...
}
```

C# Campos

▶ Campos de sólo lectura

- Similar a constantes, pero son inicializados en tiempo de ejecución en su declaración o en un constructor
 - Una vez inicializados, no pueden ser modificados
- Difieren de las constantes
 - Inicializadas en tiempo de ejecución (vs. tiempo de compilación)
 - No tiene que recompilarse los clientes
 - Pueden ser estáticas o por instancia

```
public class MiClase {  
    public static readonly double d1 = Math.Sin(Math.PI);  
    public readonly string s1;  
    public MiClase(string s) { s1 = s; } }
```

C# Propiedades

- ▶ Una propiedad es como un campo virtual
- ▶ Parece un campo, pero está implementado con código.

```
public class Button: Control {  
    private string caption;  
    public string Caption {  
        get { return caption; }  
        set { caption = value;  
              Repaint(); }  
    }  
}
```

```
Button b = new Button();  
b.Caption = "OK";  
String s = b.Caption;
```

C# Métodos

- ▶ Todo el código se ejecuta en métodos
 - Constructores, destructores y operadores son tipos especiales de métodos
 - Las propiedades e indexadores son implementados con métodos get/set
- ▶ Los métodos tienen listas de argumentos
- ▶ Los métodos contienen sentencias
- ▶ Los métodos pueden retornar valores
 - Sólo si el tipo de retorno no es void

C# Métodos

- ▶ Por defecto, los datos son pasados por valor
- ▶ Una copia de los datos es creada y pasada al método
- ▶ Para los tipos valor
 - Las variables no pueden ser modificadas por una llamada a un método
- ▶ Para los tipos referencia
 - La instancia puede ser modificada por una llamada a un método, pero la variable pasada como argumento no puede ser modificada por una llamada a un método

C# Métodos

- ▶ El modificador `ref` permite pasar argumentos por referencia
- ▶ Permite modificar variables en llamadas a métodos
- ▶ Hay que usar el modificador `ref` en la definición del método y en el código que lo utiliza
- ▶ La variable tiene que tener un valor antes de la llamada

```
void RefFunction(ref int p) { p++; }
```

```
int x = 10;  
RefFunction(ref x);  
// x es ahora 11
```

C# Métodos

- ▶ El modificador out permite a los argumentos ser pasados por referencia al llamante
- ▶ Permite a un método inicializar una variable
- ▶ Hay que usar el modificador out en la definición del método y en el código que lo utiliza
- ▶ El argumento tiene que tener un valor antes de retornar

```
void OutFunction(out int p) { p = 22; }
```

```
int x;  
OutFunction(out x);  
// x es ahora 22
```

C# Métodos

▶ Sobrecarga de métodos

- Se pueden sobrecargar métodos, es decir, proporcionar múltiples métodos con el mismo nombre
- Cada definición debe tener una signatura única
- La signatura está basada solamente en los argumentos, el valor de retorno es ignorado

```
void Print(int i);
void Print(string s);
void Print(char c);
void Print(float f);
int Print(float f); // Error: signatura duplicada
```

C# Métodos

▶ Arrays de parámetros

- Los métodos pueden tener un número variable de argumentos, llamado un array de parámetros (parameter array)
- La palabra clave params declara un array de parámetros
- Debe ser el último argumento

```
int Sum(params int[] intArr) {  
    int sum = 0;  
    foreach (int i in intArr)  
        sum += i;  
    return sum;  
}
```

```
int sum = Sum(13,87,34);
```

C# Métodos Virtuales

- ▶ Los métodos pueden ser virtuales o no virtuales (por defecto)
- ▶ Los métodos no virtuales no son polimórficos
 - No pueden ser sobreescritos (override)
- ▶ Los métodos no virtuales no pueden ser abstractos
- ▶ Los métodos virtuales son definidos en una clase base
- ▶ Los métodos virtuales pueden ser sobreescritos en clases derivadas
 - Las clases derivadas proporcionan su propia implementación especializada
- ▶ Los métodos virtuales pueden contener una implementación por defecto
 - Utilizar un método abstracto si no hay una implementación por defecto
- ▶ Una forma de polimorfismo
- ▶ Propiedades, indexadores y eventos pueden ser también virtuales

C# Métodos Virtuales

```
class Shape {  
    public virtual void Draw() { ... }  
}  
class Box : Shape {  
    public override void Draw() { ... }  
}  
class Sphere : Shape {  
    public override void Draw() { ... }  
}
```

```
void HandleShape(Shape s) {  
    s.Draw();  
    ...  
}
```

```
HandleShape(new Box());  
HandleShape(new Sphere());  
HandleShape(new Shape());
```

C# Métodos Abstractos

- ▶ Un método abstracto es virtual y no tiene implementación
- ▶ Debe pertenecer a una clase abstracta
- ▶ Con intención de ser implementado en una clase derivada

```
abstract class Shape
{
    public abstract void Draw();
}

class Box : Shape
{
    public override void Draw()
    { ... }
}
```

```
void HandleShape(Shape s)
{
    s.Draw();
    ...
}
```

```
HandleShape(new Box());
HandleShape(new Shape()); // Error!
```

C# Constructores

- ▶ Los constructores son métodos especiales que son llamados cuando una clase o estructura es instanciada
- ▶ Realizan una inicialización a la medida
- ▶ Pueden ser sobrecargados
- ▶ Si una clase no define ningún constructor, se crea un constructor implícito sin parámetros
- ▶ No se pueden crear constructores sin parámetros para una estructura
 - Todos los campos son inicializados a cero/null

C# Constructores

- ▶ Un constructor puede llamar a otros con un inicializador de constructores
- ▶ Utilización de `this(...)` o `base(...)`
- ▶ El inicializador de constructores por defecto es `base()`

```
class B {  
    private int h;  
    public B() { h=0; }  
    public B(int h) { this.h = h; }  
}  
class D : B {  
    private int i;  
    public D() : this(24) { }  
    public D(int i) { this.i = i; }  
    public D(int h, int i) : base(h) { this.i = i; }  
}
```

C# Constructores

- ▶ Un constructor estático permite crear código de inicialización que es llamado una única vez para la clase
- ▶ Se garantiza su ejecución antes de la creación de la primera instancia de una clase o estructura y antes de que se acceda a cualquier miembro estático de la clase o estructura
- ▶ No hay otras garantías en el orden de ejecución
- ▶ Solamente un constructor estático por tipo
- ▶ Debe ser sin parámetros

```
class A {  
    static A() {  
        Console.WriteLine("Constructor de A");  
        // Inicialización de variables estáticas  
        ...  
    }  
}
```

C# Destructores

- ▶ Un destructor es un método que es llamado antes de que una instancia sea “recolectada”
- ▶ Utilización para liberar recursos mantenidos por la instancia
- ▶ Sólo para clases, las estructuras no pueden tener destructores
- ▶ No hay garantías de que sean llamados en un instante específico, pero se garantiza que son llamados antes de la finalización
- ▶ Utilizar la sentencia `using` junto con la interfaz `IDisposable` permite una finalización determinista

```
class Foo {  
    ~Foo() {  
        Console.WriteLine("Destruido {0}", this);  
    }  
}
```

C# Sobrecarga de Operadores

- ▶ Operadores definidos por el usuario
- ▶ Deben ser métodos estáticos

```
class Coche
{
    string matricula;
    public static bool operator ==(Coche x, Coche y)
    {
        return x.matricula == y.matricula;
    }
}
```

C# Sobrecarga de Operadores

- ▶ Operadores unarios sobrecargables

+	-	!	~
true	false	++	--

- Operadores binarios sobrecargables

+	-	*	/	!	~
%	&		^	==	!=
<<	>>	<	>	<=	>=

C# Sobrecarga de Operadores

- ▶ No se puede hacer sobrecarga en los siguientes casos:
 - Acceso a miembros, invocación de métodos, operadores de asignación
 - sizeof, new, is, as, typeof, checked, unchecked, &&, ||, y ?:
- ▶ Los operadores && y || son automáticamente evaluados desde & y |
- ▶ La sobrecarga de un operador binario (ej: *) implícitamente sobrecarga el correspondiente operador de asignación (ej: *=)

C# Conversiones

▶ Conversiones de usuario explícitas e implícitas

```
class Nota {  
    int valor;  
    // Convertir a hertz - sin pérdida de precisión  
    public static implicit operator double(Nota x) {  
        return ...;  
    }  
    // Convertir a la nota más cercana  
    public static explicit operator Nota(double x) {  
        return ...;  
    }  
}
```

```
Nota n = (Nota)442.578;  
double d = n;
```

C# Manejo de Excepciones

- ▶ Una excepción es cualquier condición de error o comportamiento inesperado encontrado en la ejecución de un programa
- ▶ Las excepciones pueden venir desde un programa en ejecución o desde el entorno de ejecución
- ▶ Para el entorno de ejecución, una excepción es un objeto que hereda de la clase `System.Exception`
- ▶ Existe toda una jerarquía de clases para el tratamiento de excepciones
 - `SystemException`
 - `ApplicationException`
 - `OverflowException`

...

C# Manejo de Excepciones

- ▶ Sentencias try...catch...finally
- ▶ Los bloques try contienen código que podría elevar excepciones
- ▶ Los bloques catch manejan las excepciones
- ▶ Pueden haber múltiples bloques catch para manejar diferentes clases de excepciones
- ▶ Los bloques finally contienen código que siempre será ejecutado

C# Manejo de Excepciones

```
try
{
    // Código para ejecución normal
}
catch
{
    // Manejo de errores
}
finally
{
    // Limpieza
}
```

C# Manejo de Excepciones

- ▶ La sentencia throw eleva una excepción
 - Las excepciones pueden ser elevadas por el propio sistema o elevadas por el usuario.
 - Ej: throw new OverflowException();
- ▶ Una excepción está representada como una instancia de System.Exception o una clase derivada
 - Contiene información sobre la excepción
 - Propiedades
- ▶ Se puede reelevar una excepción, o capturar una excepción y elevar otra diferente

C# Manejo de Excepciones

```
public static void Main()
{ string userInput;
    Console.WriteLine("Numero del 0 al 5 (o return para salir) > ");
    userInput = Console.ReadLine();
    while (userInput != "")
    { try
        { int index = int.Parse(userInput);
            if (index < 0 || index > 5)
                throw new IndexOutOfRangeException(
                    "Escribiste " + userInput);
            Console.WriteLine("Numero " + index);
        }
    }
```

C# Manejo de Excepciones

```
catch (IndexOutOfRangeException e)
{ Console.WriteLine("Exception: " + "Numero fuera de rango "
+ e.Message);
}
catch (Exception e)
{
    Console.WriteLine("Excepcion: " + e.Message);
}
catch { Console.WriteLine("Alguna otra excepcion"); }
Finally { Console.WriteLine("Hasta luego"); }
}
```

C# Manejo de Excepciones

▶ Propiedades:

- **HelpLink**: Enlace al fichero de ayuda con información sobre la excepción
- **Message**: Texto que describe la condición de error
- **Source**: El nombre de la aplicación u objeto que causó la excepción
- **StackTrace**: Secuencia de llamadas a métodos que provocaron la excepción
- **TargetSite**: Objeto .NET de reflexión que describe el método que arrojó la excepción
- **InnerException**: si la excepción fue arrojada dentro de un bloque catch, contiene el objeto excepción que llevó al código a dicho bloque catch

C# Manejo de Excepciones

- ▶ Se pueden crear nuevos tipos de excepciones
 - Permite personalizar nuestros tipos de errores
 - Un mejor control de errores
- ▶ Herencia de **Exception** o **ApplicationException**.
- ▶ **ApplicationException** extiende **Exception**, pero no agrega nueva funcionalidad. Esta excepción supone un medio para establecer diferencias entre excepciones definidas por aplicaciones y excepciones definidas por el sistema.

C# Genéricos

- ▶ Disponible a partir de .NET 2.0
 - En versiones anteriores había que “abusar” de la clase object para algunas tareas
 - Pérdida de eficiencia por castings
 - Inseguridad en la conversión de tipos
- ▶ Incorporación de tipos genéricos que son reemplazados con tipos específicos cuando se necesite
 - Esto permite seguridad de tipos
 - El compilador comprueba si un tipo específico no es soportado por la clase genérica
 - Mejora en el rendimiento
 - No es necesario realizar boxing y unboxing

C# Genéricos

- ▶ Existen construcciones similares en otros lenguajes
(Ej: java, C++)
- ▶ Ventajas de los genéricos
 - Inclusión del CLR
 - Permite utilizar tipos genéricos entre múltiples lenguajes .NET
- ▶ Clases de colección basada en genéricos
- ▶ ¿Cómo son implementados?
 - Instanciados en tiempo de ejecución, no en tiempo de compilación
 - Comprobados en la declaración, no en la instancia
 - Funcionan para tipos referencia y tipos valor

C# Genéricos

- ▶ Una clase genérica se define de forma similar a una clase normal con la declaración del tipo/tipos genérico/s
 - Este tipo genérico puede ser después utilizado dentro de la clase como atributo o como parámetro de métodos
- ▶ Ejemplo

```
public class MiGenerico<T>
{ private T miembro;
  public void metodo(T obj) { }
}
```

C# Genéricos

- ▶ Ejemplos: Pila y Diccionario

```
public class Pila<T>
```

```
{ T[] items;  
    int cuenta;
```

```
    public void Push(T item) {...}
```

```
    public T Pop() {...}
```

```
}
```

```
public class Diccionario<K,V>
```

```
{ public void Sumar(K clave, V valor) {...}
```

```
    public V this[K clave] {...}
```

```
}
```

C# Genéricos. Ejemplo Lista

```
public class ListaGenerica<T>
{
    private class Nodo
    {
        public Nodo(T t)
        {
            sig = null; elem = t;
        }
        private Nodo sig;
        public Nodo Siguiente
        {
            get { return sig; }
            set { sig = value; }
        }
        private T elem;
        public T Elem
        {
            get { return elem; }
            set { elem = value; }
        }
    }
}
```

```
private Nodo primero;
public ListaGenerica()
{
    primero = null;
}
public void MeterPrimero(T t)
{
    Nodo n = new Nodo(t);
    n.Siguiente = primero;
    primero = n;
}
```

C# Genéricos

```
class TestListaGenerica
{
    static void Main()
    {
        ListaGenerica<int> list = new ListaGenerica<int>();
        for (int x = 0; x < 10; x++)
        {
            list.MeterPrimero(x);
        }
    }
}
```

C# Genéricos. Restricciones

- ▶ A veces será necesario invocar métodos del tipo parámetro
 - Ej: public class Diccionario<K,V>

```
{ public void Sumar(K clave, V valor)
{
    ...
    if (clave.CompareTo(x) < 0) {...} // Error de compilación
    ...
}
```
- ▶ Posible solución: Realizar casting a alguna interfaz o clase
 - Ej: public class Diccionario<K,V>

```
{ public void Sumar(K clave, V valor)
{
    ...
    if (((IComparable)clave).CompareTo(x) < 0) {...}
    ...
}
```

C# Genéricos. Restricciones

- ▶ Problemas casting
 - Comprobación dinámica (sobrecarga)
 - Posible excepción InvalidCastException
- ▶ C# permite la indicación de restricción en los tipos parámetro
- ▶ Cláusula where para especificar restricciones
 - Clases de las que deben heredar los tipos genéricos
 - Interfaces que deben implementar
 - Para saber qué métodos se pueden utilizar
 - Necesidad de un constructor por defecto

C# Genéricos. Restricciones

- ▶ Ejemplo:

```
public class Diccionario<K,V> where K: IComparable
{
    public void Sumar(K clave, V valor)
    {
        ...
        if (clave.CompareTo(x) < 0) {...}
        ...
    }
}
```

- ▶ Ejemplo: Restricción de varias interfaces y de clase

```
public class TablaEntidad<K,E>
    where K: IComparable<K>, IPersistable
    where E: Entidad, new()
{
    public void Sumar(K clave, E entidad)
    {
        ...
        if (clave.CompareTo(x) < 0) {...}
        ...
    }
}
```

C# Genéricos. Restricciones

- ▶ Ejemplo:
 - Restricción con interfaces y constructor

```
public class ProcesarDocumentos<TDocumento,  
    TDocumentoManager>  
  
    where TDocumento : IDocumento , new()  
  
    where TDocumentoManager : IDocumentoManager<TDocumento>  
  
{  
    ...  
}
```

C# Genéricos. Restricciones

▶ Tipos de restricciones

where T: struct	El tipo argumento debe ser un tipo valor (excepto el tipo Nullable).
where T : class	El tipo argumento debe ser un tipo referencia: clase, interfaz, delegado o tipo array.
where T : new()	El tipo argumento debe tener un constructor público sin parámetros. Si se usa junto con otras restricciones debe indicarse la última.
where T : <base class name>	El tipo argumento debe ser o derivar de la clase base especificada.
where T : <interface name>	El tipo argumento debe ser o implementar la interfaz especificada. Se pueden indicar múltiples restricciones de interfaces.
where T : U	El tipo argumento T debe ser o derivar del tipo argumento U.

C# Interfaces Genéricas.

- ▶ Se pueden definir y/o utilizar interfaces genéricas
 - Ejemplo: nuevas clases para colecciones: `IEnumerable<T>`, `IComparable<T>`
- ▶ Las interfaces genéricas pueden utilizarse como restricciones
 - Ej: `class Stack<T> where T : System.IComparable<T>, IEnumerable<T>`
- ▶ Una interfaz puede definir más de un tipo parámetro
 - Ej: `interface IDictionary<K, V> { ... }`
- ▶ Clases concretas pueden heredar de interfaces construidas cerradas.
 - Ej: `interface IBaselInterface<T> { }`
`class ClaseEjemplo : IBaselInterface<string> { }`

C# Métodos Genéricos.

- ▶ Se pueden definir métodos genéricos
 - El tipo genérico es indicado en la declaración del método
 - En clases no necesariamente genéricas
- ▶ Ejemplo:

```
void Swap<T>(ref T x, ref T y)
```

```
{
```

```
    T temp;
```

```
    temp = x;
```

```
    x = y;
```

```
    y = temp;
```

```
}
```

C# Métodos Genéricos.

▶ Utilización

- Indicar el tipo al realizar la invocación

```
int i = 4;
```

```
int j = 5;
```

```
Swap<int>(ref i, ref j);
```

- Sin indicar el tipo al realizar la invocación

- El compilador de C# puede obtener los tipos

```
int i = 4;
```

```
int j = 5;
```

```
Swap(ref i, ref j);
```

C# Métodos Genéricos.

- ▶ Es posible la indicación de restricciones en los métodos genéricos

```
void SwapIfGreater<T>(ref T lhs, ref T rhs) where T :  
    System.IComparable<T>
```

```
{  
    T temp;  
    if (lhs.CompareTo(rhs) > 0)  
    {  
        temp = lhs;  
        lhs = rhs;  
        rhs = temp;  
    }  
}
```

C# Genéricos.

- ▶ ¿Cómo asignar un valor por defecto a un tipo genérico?
 - ¿null?
 - No es posible para los tipos valor
- ▶ C# proporciona la palabra clave default
 - Asigna null a tipos referencia y 0 a tipos valor
- ▶ Ejemplo:

```
public T GetSiguiente()  
{
```

```
    T temp = default(T);
```

```
    Nodo current = primero;
```

```
    if (current != null)
```

```
{
```

```
        temp = current.Elem;
```

```
        current = current.Sig;
```

```
}
```

```
    return temp;
```

```
}
```

C# Colecciones

- ▶ Las clases bases de .NET ofrecen estructuras de datos que agrupan objetos.
- ▶ La clase *system.Array* es un caso particular
 - Hay que definir su tamaño
 - No hay primitivas para añadir, borrar, insertar,....
 - Se necesitan índices
- ▶ *System.Collection* agrupa
 - Colecciones
 - Colas
 - Pilas
 - Listas ordenadas
 - Diccionarios(mapas)

C# Colecciones

- ▶ Se pueden acceder con el bucle *foreach*
- ▶ Qué es una colección?
 - Internamente un objeto es una colección si ofrece una referencia al objeto, conocido como enumerador (*enumerator*)
 - Es el que puede pasar por los items de la colección
 - Una colección debe implementar el interfaz *System.Collection.IEnumerable*

C# Colecciones Genéricas

- ▶ Colecciones utilizando tipos genéricos
 - System.Collections.Generic
- ▶ Clases de colección genéricas
 - List<T>
 - Equivalente de ArrayList
 - Dictionary < TKey, TValue >
 - SortedList < TKey, TValue >
 - LinkedList<T>
 - Queue<T>
 - Stack<T>

C# Colecciones Genéricas

▶ Ejemplo utilizando List<T>

```
List<Piloto> lpilotos = new List<Piloto>();  
lpilotos.Add(new Piloto("Fernando Alonso", "Ferrari"));  
lpilotos.Add(new Piloto("Lewis Hamilton", "McLaren-  
Mercedes"));  
lpilotos.Add(new Piloto("Mark Webber", "Red Bull"));  
lpilotos.Add(new Piloto("Pedro de la Rosa", "HRT"));  
  
foreach (Piloto r in lpilotos) { Console.WriteLine(r); }
```

C# Colecciones Genéricas

- ▶ Ejemplo ordenación con IComparer<T>

```
public class PilotoComparer : IComparer<Piloto>
{
    public enum TipoCompara { Nombre, Coche }
    private TipoCompara tipoCompara;

    public PilotoComparer(TipoCompara tipoCompara)
    {
        this.tipoCompara = tipoCompara;
    }
```

C# Colecciones Genéricas

```
public int Compare(Piloto x, Piloto y)
{  int result = 0;
   switch (tipoCompara) {
      { case tipoCompara.Nombre:
          result = x.Nombre.CompareTo(y.Nombre); break;
      case tipoCompara.Coches:
          result = x.Coches.CompareTo(y.Coches); break;
      }
   return result;
}
```

- ▶ Utilización
Ipilotos.Sort(new
PilotoComparer(PilotoComparer.TipoCompara.Coches));

C# Iteradores

- ▶ La sentencia foreach permite recorrer de manera cómoda los elementos de tipos enumerados
 - Para ser enumerable, una colección debe tener un método GetEnumerator que devuelve un enumerador.
 - Es necesario implementar la interfaz IEnumerable o IEnumerable<T>
- ▶ Generalmente, los enumeradores son difíciles de implementar
- ▶ Los iteradores de C# simplifican esta tarea mediante la utilización de la sentencia yield
 - Generan automáticamente los métodos Current, MoveNext y Dispose

C# Iteradores

▶ Características

- Un iterador es una sección de código que devuelve una secuencia ordenada de valores del mismo tipo.
- El código del iterador utiliza la instrucción **yield return** para devolver los elementos
- La instrucción **yield break** finaliza la iteración
- Se pueden implementar varios iteradores en una clase. Cada iterador debe tener un nombre único
 - Podrá utilizarse utilizando su nombre
 - Ej: `foreach(int x in miclase.Iterador2){}`
- El tipo de valor devuelto de un iterador debe ser **IEnumerable**, **IEnumerable<T>**, **IEnumerator** o **IEnumerator<T>**

C# Iteradores

- ▶ La manera más común de implementar iteradores es implementar el método GetEnumerator de la interfaz IEnumerable

- ▶ Ejemplo

```
public System.Collections.IEnumerator GetEnumerator()
{
    for (int i = 0; i < max; i++)
    {
        yield return i;
    }
}
```

- ▶ Utilización (si GetEnumerator fuera de la clase ListClass)
ListClass listClass1 = new ListClass();
foreach (int i in listClass1)

```
{  
    System.Console.WriteLine(i);  
}
```

C# Iteradores

- ▶ Ejemplo: Días de la semana

```
public class DiasSemana: System.Collections.IEnumerable {  
    string[] m_dias = { "Lun", "Mar", "Mier", "Jue", "Vier", "Sab" , "Dom"};  
  
    public System.Collections.IEnumerator GetEnumerator()  {  
        for (int i = 0; i < m_dias.Length; i++)  
        {  
            yield return m_dias[i];  
        }  
    }  
}
```

- ▶ Utilización

```
DiasSemana semana = new DiasSemana();  
  
foreach (string dia in semana)  
{  
    System.Console.Write(dia + " ");  
}
```

C# Iteradores

- ▶ Ejemplo: Clase genérica Pila

```
public class Pila<T>: IEnumerable<T>
{
    T[] items;
    int count;
    public void Push(T data) {...}
    public T Pop() {...}
    public IEnumerator<T> GetEnumerator()
    {
        for (int i = count - 1; i >= 0; --i)
            yield return items[i];
    }
}
```

C# Iteradores

- ▶ Ejemplo: Clase genérica Pila

```
using System;
class Test
{
    static void Main()
    {
        Pila<int> pila = new Pila<int>();
        for (int i = 0; i < 10; i++) pila.Push(i);
        foreach (int i in pila) Console.Write("{0} ", i);
        Console.WriteLine();
    }
}
```

C# Delegados

- ▶ Un delegado (delegate) es un tipo referencia que define una signatura de método
- ▶ Una instancia de un delegado mantiene uno o más métodos
 - Esencialmente es un “puntero de función orientado a objetos”
 - Los métodos pueden ser estáticos o no estáticos
 - Los métodos pueden retornar valores
- ▶ Proporciona polimorfismo para funciones individuales
- ▶ Base para el manejo de eventos

C# Delegados

- ▶ Funcionamiento:
 - Crear una instancia
 - Pasarle en el constructor el método de alguna instancia
 - Posteriormente hay que invocar la instancia como si fuera una función

C# Delegados

► Ejemplo: Ordenación

```
delegate bool CMP(object lhs, object rhs);
class ordenadora
{
    static public void Sort(object [] array, CMP cmp)
    {
        for (int i=0;i<array.Length;i++)
            for (int j=i+1;j<array.Length;j++)
                if (cmp(array[j],array[i]))
                {
                    object temp=array[i];
                    array[i]=array[j];
                    array[j]=temp;
                }
    }
}
```

C# Delegados

```
class Empleado
{
    private string nombre;
    private decimal sueldo;
    ...
    public static bool SueldoMenor(object lhs,
                                    object rhs)
    {
        Empleado empi = (Empleado) lhs;
        Empleado empd = (Empleado) rhs;
        return (empi.sueldo < empd.sueldo)?true:false;
    }
}
```

C# Delegados

```
class Program
{
    static void Main(string[] args)
    {
        Empleado [] empleados= { new Empleado(...), ... };
        CMP comp=new CMP(Empleado.SueldoMenor);
        Ordenadora.Sort(empleados,comp);
        ...
    }
}
```

C# Delegados VS Interfaces

- ▶ Se podrían usar siempre interfaces en lugar de delegados
- ▶ Las interfaces son más potentes
 - Múltiples métodos
 - Herencia
- ▶ Los delegados son más elegantes para los manejadores de eventos
 - Menos código
 - Pueden implementar fácilmente múltiples manejadores de eventos sobre una clase o estructura

C# Métodos Anónimos

- ▶ Permiten no tener que crear un método para que el delegado “funcione”
- ▶ Un método anónimo es un bloque de código que es utilizado como parámetro para el delegado

- Ej:

```
delegate string delegateTest(string val);
static void Main(string[] args)
{
    string mid = ", en medio,";
    delegateTest anonDel = delegate(string param)
    {
        param += mid;
        param += " y esto se suma.";
        return param;
    };
    Console.WriteLine(anonDel("Inicio"));
};
```



E.T.S.
INGENIERÍA
INFORMÁTICA

Gestión de la Información

Grado en Ingeniería del Software



UNIVERSIDAD
DE MÁLAGA



Tema 4.2 Acceso a datos con ADO.NET

José Luis Pastrana Brincones
pastrana@lcc.uma.es

ADO.NET

- ADO.NET es un conjunto de clases que exponen servicios de acceso a datos para el programador de .NET.
- ADO.NET ofrece abundancia de componentes para la creación de aplicaciones de uso compartido de datos distribuidas.
- Constituye una parte integral de .NET Framework y proporciona acceso a datos relacionales, XML y de aplicaciones.
- ADO.NET satisface diversas necesidades de desarrollo, como la creación de clientes de base de datos de aplicaciones para usuario y objetos empresariales de nivel medio que utilizan aplicaciones, herramientas, lenguajes o exploradores de Internet.

ADO.NET

- ▶ Soporta gran variedad de fuentes de datos:
 - bases de datos relacionales (SQL Server, Oracle, and Microsoft Access)
 - Otras fuentes de datos (Excel, Outlook, texto, xml).
- ▶ Se usa un proveedor de datos (data provider) para conectar a una fuente de datos, ejecutar comandos y obtener resultados.

.NET Framework data provider	Data source access
SQL Server	Microsoft SQL Server version 7.0 or later
OLE DB	Data sources using OLE DB
ODBC	Data sources using ODBC
Oracle	Oracle client software version 8.1.7 or later

ADO.NET

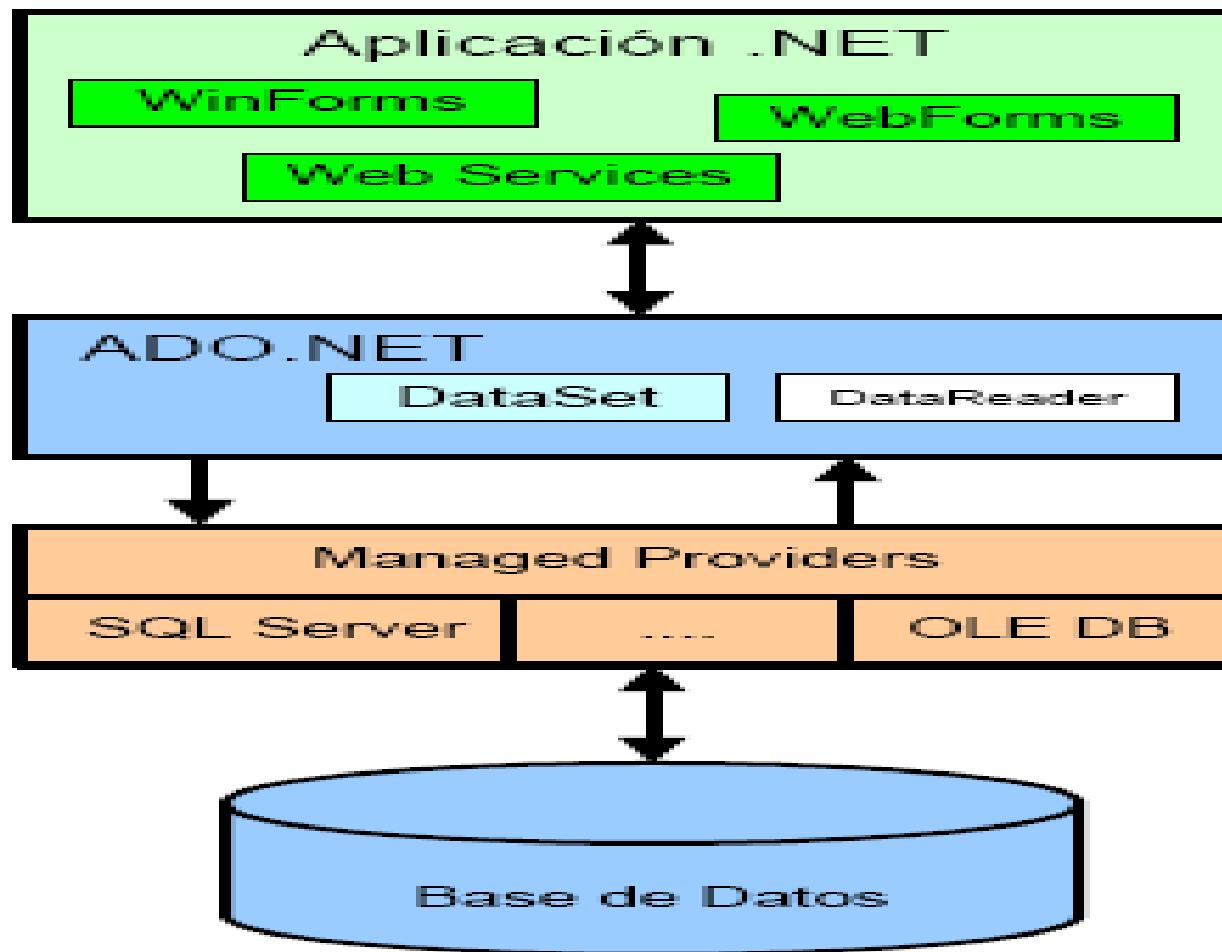
- ADO.NET proporciona acceso coherente a orígenes de datos como Microsoft SQL Server y XML, así como a orígenes de datos expuestos mediante OLE DB y ODBC.
- Las aplicaciones para usuarios que comparten datos pueden utilizar ADO.NET para conectar a estos orígenes de datos y recuperar, manipular y actualizar los datos contenidos.
- ADO.NET separa el acceso a datos de la manipulación de datos y crea componentes discretos que se pueden utilizar por separado o conjuntamente.

ADO.NET

- ADO.NET incluye proveedores de datos de .NET Framework para conectarse a una base de datos, ejecutar comandos y recuperar resultados.
- Los resultados se procesan directamente o se colocan en un objeto DataSet de ADO.NET con el fin de exponerlos al usuario para un propósito específico, combinados con datos de varios orígenes, o de utilizarlos de forma remota entre niveles.
- El objeto DataSet de ADO.NET también puede utilizarse independientemente de un proveedor de datos de .NET Framework para administrar datos que son locales de la aplicación o que proceden de un origen XML.

ADO.NET

- La arquitectura de ADO .NET es la siguiente:



ADO.NET

- Las clases de ADO.NET se encuentran en el archivo System.Data.dll y están integradas con las clases de XML que se encuentran en el archivo System.Xml.dll.
- Cuando se compila un código que utiliza el espacio de nombres System.Data, es necesario hacer referencia a los archivos System.Data.dll y System.Xml.dll.
- Para obtener un ejemplo de una aplicación de ADO.NET que se conecta a una base de datos, recupera datos de ésta y, a continuación, los muestra en el símbolo del sistema, vea Aplicación de ejemplo de ADO.NET.
- ADO.NET se basa en una arquitectura desconectada con una fuerte integración con XML y está diseñado para facilitar el desarrollo de aplicaciones débilmente acopladas.

ADO.NET

- El código ADO.NET es compatible hacia adelante, por tanto, todo código escrito usando el .NET Framework 1.1 o posterior funcionará en las siguientes versiones de .NET Framework.
- ADO.NET tiene ambos tipos de clases: conectadas (Proveedores de datos de .NET) y desconectadas (DataSet, DataTable).
 - Las clases conectadas obtiene y actualizan los datos en a través de una conexión a las fuentes de datos.
 - Las clases desconectadas acceden y manipulan “offline” los datos que han sido obtenidos a través de una clase conectada y posteriormente los sincronizan usando una clase conectada.

Componentes de ADO.NET

- Los proveedores de datos de .NET Framework
- El objeto *Connection* proporciona conectividad a un origen de datos.
- El objeto *Command* permite tener acceso a comandos de base de datos para devolver datos, modificar datos, ejecutar procedimientos almacenados, etc.
- El objeto *DataReader* proporciona una secuencia de datos de alto rendimiento desde el origen de datos.
- El objeto *DataAdapter* proporciona el puente entre el objeto *DataSet* y el origen de datos. El *DataAdapter* utiliza objetos *Command* para ejecutar comandos SQL en el origen de datos tanto para cargar el *DataSet* con datos como para reconciliar en el origen de datos los cambios aplicados a los datos incluidos en el *DataSet*.

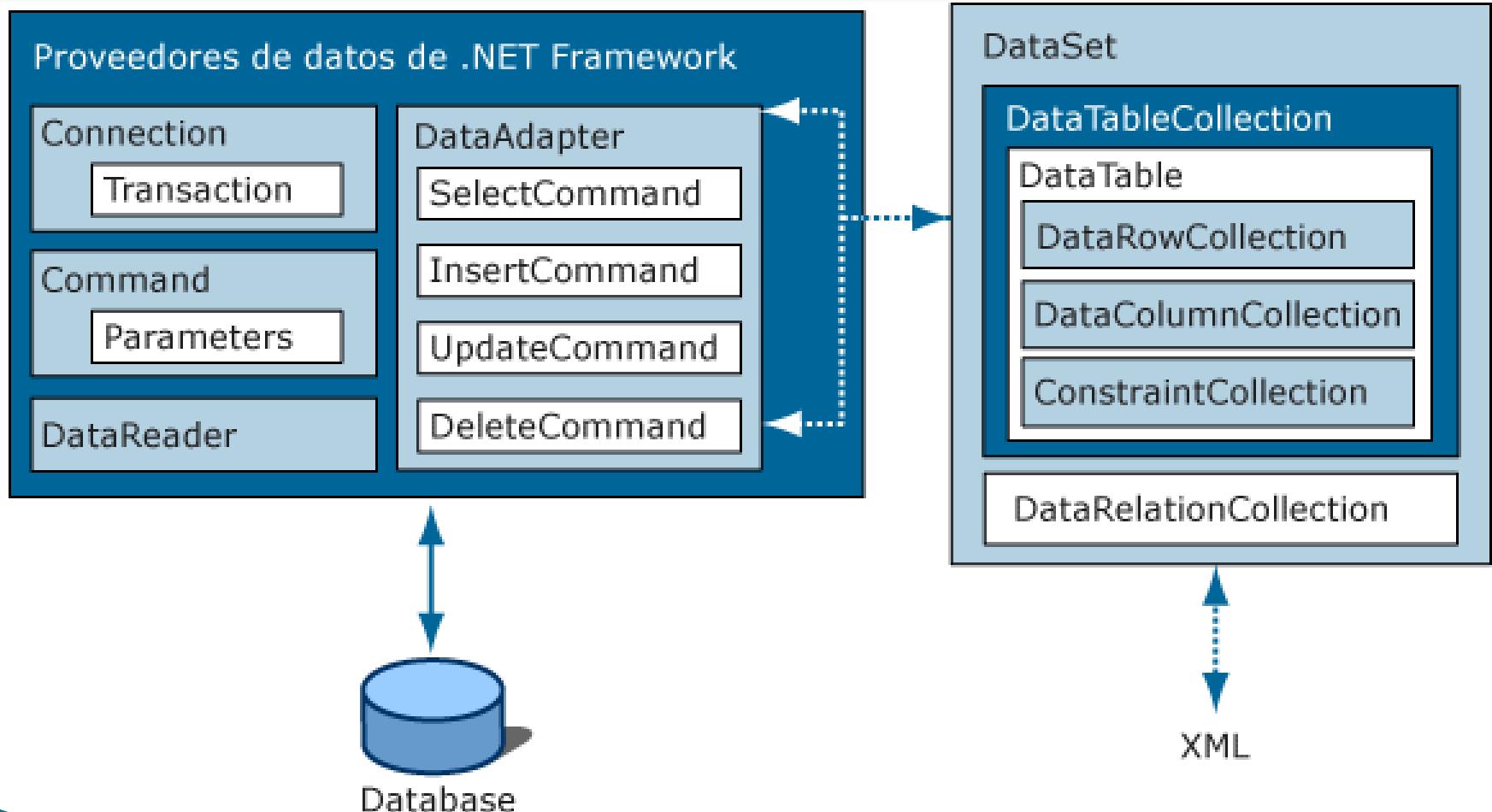
Componentes de ADO.NET

- Clases Desconectadas.
- *DataSet*: contiene una colección de uno o más objetos *DataTable* formados por filas y columnas de datos, así como información sobre claves principales, claves externas, restricciones y relaciones relativa a los datos incluidos en los objetos *DataTable*.
 - *DataTable*: Una tabla simple en memoria.
 - *DataColumn*: El esquema de una columna en un *DataTable*.
 - *DataRow*: Una fila de un *DataTable*.
 - *DataView*: Una vista de un *DataTable* usada para una ordenación personalizada, búsqueda, filtrado, etc.
 - *DataRelation*: Una relación padre/hijo entre dos *DataTable objects* de un *DataSet*.
 - *Constraint*: Una restricción sobre una o más columnas de un *DataTable* usada para mantener la integridad de los datos: *UniqueConstraint*, *ForeignKeyConstraint*.

Componentes de ADO.NET

NAMESPACE	Descripción
System.Data	Clases genéricas de acceso a datos.
System.Data.Common	Clases compartidas o redefinidas para cada proveedor de datos individual.
System.Data.EntityClient	Clases del Entity Framework
System.Data.Linq.SqlClient	Clases del proveedor de LINQ to SQL
System.Data.Odbc	Clases del proveedor de ODBC
System.Data.OleDb	Clases del proveedor de OLE DB
System.Data.ProviderBase	Clase base y factoría de conexiones para nuevos proveedores de datos.
System.Data.Sql	Clases e interfaces genéricos para SQL Server
System.Data.SqlClient	Clases del proveedor de SQL Server
System.Data.SqlTypes	Tipos de datos de SQL Server

Componentes de ADO.NET



Relación entre un proveedor de datos de .NET Framework y un DataSet

ADO.NET

Objetos principales de los proveedores de datos de .NET Framework (Clases Conectadas).

Objeto	Descripción
Connection	Establece una conexión a un origen de datos determinado. La clase base para todos los objetos Connection es DbConnection .
Command	Ejecuta un comando en un origen de datos. Expone Parameters y puede ejecutarse en el ámbito de un objeto Transaction de Connection. La clase base para todos los objetos Command es DbCommand .
DataReader	Lee una secuencia de datos de sólo avance y sólo lectura desde un origen de datos. La clase base para todos los objetos DataReader es DbDataReader .
DataAdapter	Llena un DataSet y realiza las actualizaciones necesarias en el origen de datos. La clase base para todos los objetos DataAdapter es DbDataAdapter .

Componentes de ADO.NET

- Los proveedores de datos de .NET Framework también incluyen:

Objeto	Descripción
Transaction	Permite incluir comandos en las transacciones que se realizan en el origen de datos. La clase base para todos los objetos Transaction es DbTransaction .
CommandBuilder	Un objeto auxiliar que genera automáticamente las propiedades de comando de un DataAdapter o que obtiene de un procedimiento almacenado información acerca de parámetros con la que puede llenar la colección Parameters de un objeto Command. La clase base para todos los objetos CommandBuilder es DbCommandBuilder .
ConnectionStringBuilder	Un objeto auxiliar que proporciona un modo sencillo de crear y administrar el contenido de las cadenas de conexión utilizadas por los objetos Connection. La clase base para todos los objetos ConnectionStringBuilder es DbConnectionStringBuilder .

Componentes de ADO.NET

Objeto	Descripción
Parameter	Define los parámetros de entrada, salida y valores devueltos para los comandos y procedimientos almacenados. La clase base para todos los objetos Parameter es DbParameter .
Exception	Se devuelve cuando se detecta un error en el origen de datos. En el caso de que el error se detecte en el cliente, los proveedores de datos de .NET Framework inicien una excepción de .NET Framework. La clase base para todos los objetos Exception es DbException .
Error	Expone la información relacionada con una advertencia o error devueltos por un origen de datos.
ClientPermission	Se proporciona para los atributos de seguridad de acceso a código de los proveedores de datos de .NET Framework. La clase base para todos los objetos ClientPermission es DBDataPermission .

Conexiones y Cadena de Conexión

- ▶ Las conexiones son recursos limitados y críticos. El tipo de conexión debe ser bien elegida según el SGBD.
 - SQL Server, Oracle, OLE DB ,ODBC.
 - Una conexión no debería estar abierta más allá de un método.
 - No se deben pasar conexiones como parámetros entre métodos.
- ▶ Los proveedores de datos usan una cadena de conexión que contiene una colección de pares.
- ▶ Ejemplo Access 2003

```
cadenaConexion = "Provider=Microsoft.Jet.OLEDB.4.0;Data Source=" + BD;
```

- ▶ Ejemplo Access 2007

```
cadenaConexion = "Provider=Microsoft.ACE.OLEDB.12.0;Data Source=" + BD;
```

- ▶ Ejemplo SQLSERVER

```
cadenaConexion = "Data Source=" + server + "\SQLEXPRESS;"  
+ "Integrated Security=SSPI;Initial Catalog=" + BD;
```

Conexiones y Cadena de Conexión

▶ SqlConnectionStringBuilder

```
SqlConnectionStringBuilder csb = new  
    SqlConnectionStringBuilder( );  
  
csb.DataSource = "localhost\\SQLEXPRESS";  
  
csb.Add("Initial Catalog", "EMPRESA") ;  
  
csb["Integrated Security"] = true;  
  
SqlConnection connection =  
    new SqlConnection(csb.ConnectionString);  
  
connection.Open( );  
  
    . . .  
  
connection.Close( );
```

Comandos

- ▶ Un comando se puede construir pasándole la sentencia SQL como parámetro del constructor.

```
string select = "SELECT COUNT(*) FROM Customers";  
SqlConnection conn = new SqlConnection(source);  
conn.Open();  
  
SqlCommand cmd = new SqlCommand(select, conn);
```

- ▶ *ExecuteNonQuery()*: Ejecuta un comando que no retorna nada (UPDATE, INSERT, DELETE).
- ▶ *ExecuteReader()*: Ejecuta un comando que retorna un *IDataReader* (*SELECT*).
- ▶ *ExecuteScalar()*: Ejecuta un comando que retorna un único valor. Si la consulta tiene varias tuplas, retorna el valor de la primera columna de la primera fila. (*SELECT COUNT ...*).

ADO .NET

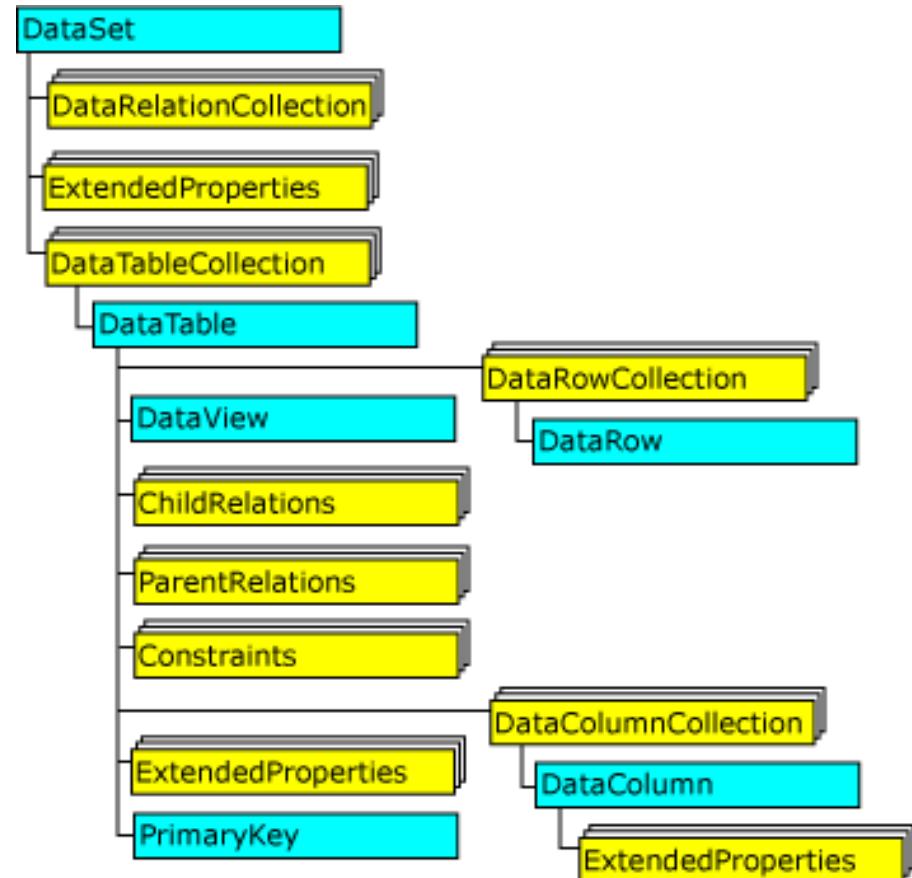
DataSet de ADO.NET

- El objeto *DataSet* es esencial para la compatibilidad con situaciones de datos distribuidos desconectados con ADO.NET.
- El *DataSet* es una representación de datos residente en memoria que proporciona un modelo de programación relacional coherente con independencia del origen de datos.
- Se puede utilizar con muchos y distintos orígenes de datos, con datos XML o para administrar datos locales de la aplicación.
- El DataSet representa un conjunto completo de datos que incluye tablas relacionadas y restricciones, así como relaciones entre las tablas. En la siguiente ilustración se muestra el modelo de objetos DataSet.

ADO .NET

Modelo de objetos DataSet

- ▶ Los métodos y objetos de un DataSet concuerdan con los del modelo de base de datos relacional.
- ▶ El DataSet también puede mantener y recargar su contenido como XML y su esquema como esquema de lenguaje de definición de esquemas XML (XSD).



ADO. NET

DataTableCollection

- Un *DataSet* de ADO.NET contiene una colección de cero o más tablas representadas por objetos *DataTable*.
- La propiedad *DataTableCollection* contiene todos los objetos *DataTable* de un *DataSet*.
- *DataTable* se define en el espacio de nombres *System.Data* y representa una única tabla de datos residentes en memoria.
- Contiene una colección de columnas representadas por una *DataColumnCollection*, así como restricciones representadas por una *ConstraintCollection*, que juntas definen el esquema de la tabla.
- Un *DataTable* también contiene una colección de filas representadas por la *DataRowCollection*, que contiene los datos de la tabla.
- Cada *DataRow* conserva sus versiones actual y original para identificar los cambios en los valores almacenados en la fila.
-

ADO. NET

DataRelationCollection

- Un DataSet contiene relaciones en su objeto *DataRelationCollection*.
- Una relación, representada por el objeto DataRelation, asocia las filas de un *DataTable* con las filas de otro *DataTable*.
- Los elementos esenciales de una *DataRelation* son el nombre de la relación, el nombre de las tablas que se relacionan y las columnas relacionadas de cada tabla.
- Es posible crear relaciones con más de una columna por tabla si se especifica una matriz de objetos *DataColumn* como columnas de claves.
- Cuando agrega una relación al *DataRelationCollection*, puede agregar opcionalmente una *UniqueKeyConstraint* y una *ForeignKeyConstraint* para imponer restricciones de integridad cuando se realizan cambios en valores de columna relacionados.

ADO. NET

ExtendedProperties

- *DataSet*, *DataTable* y *DataColumn* tienen todos una propiedad *ExtendedProperties*.
- *ExtendedProperties* es una *PropertyCollection* en la que puede colocar información personalizada, como la instrucción SELECT que se ha utilizado para generar el conjunto de resultados o la hora en que se generaron los datos.
- La colección *ExtendedProperties* se mantiene con la información de esquema del *DataSet*.

ADO. NET

Los pasos de creación y actualización de un DataSet son los siguientes:

1. Construir y llenar cada DataTable de un DataSet con datos desde un origen de datos mediante el método Fill de DataAdapter.
2. Cambiar los datos de los objetos DataTable individuales mediante la adición, actualización o eliminación de objetos DataRow.
3. Si se desea se puede:
 1. llamar al método GetChanges para crear un segundo DataSet que sólo incorpore los cambios realizados.
 2. Se invoca el método Merge para combinar los cambios del segundo DataSet con el primero.
 3. Invocar al método AcceptChanges de DataSet. O bien, invocar al método RejectChanges para cancelar los cambios.
4. Llame al método Update del DataAdapter correspondiente para reflejar los datos en la base de datos.

Ejemplo Clases Conectadas

```
string cadenaConexion = "Data Source=localhost\\SQLEXPRESS;"  
    + "Integrated Security=SSPI;Initial Catalog=hotel";  
  
string consultaSelect = "SELECT * FROM usuario;";  
  
SqlConnection connection = new SqlConnection(cadenaConexion);  
SqlCommand command = connection.CreateCommand();  
command.CommandText = consultaSelect;  
  
connection.Open();  
  
SqlDataReader reader = command.ExecuteReader();  
  
while (reader.Read())  
    Console.WriteLine("{0}\t{1}\t{2}", reader[0], reader[1], reader[2]);  
  
reader.Close();  
  
connection.Close();
```

Ejemplo Clases Desconectadas

```
string cadenaConexion = "Data Source=localhost\\SQLEXPRESS;"  
    + "Integrated Security=SSPI;Initial Catalog=hotel";  
  
string consultaSelect = "SELECT * FROM usuario;";  
  
DataTable myTable = new DataTable();  
  
SqlConnection conexion = new SqlConnection(cadenaConexion);  
  
SqlDataAdapter adaptador = new SqlDataAdapter(consultaSelect, conexion);  
adaptador.Fill(myTable);  
  
int num_filas = myTable.Rows.Count;  
  
int num_columnas = myTable.Columns.Count;  
  
for (int i = 0; i < num_filas; i++)  
{  for (int j = 0; j < num_columnas; j++) Console.WriteLine(myTable.Rows[i][j]+"\t");  
  Console.WriteLine();  
}  
}
```

Propiedades Públicas de un DataSet

Nombre	Descripción
CaseSensitive	Obtiene o establece un valor que indica si las comparaciones de cadena en los objetos DataTable distinguen entre mayúsculas y minúsculas.
DataSetName	Obtiene o establece el nombre del objeto DataSet actual.
EnforceConstraints	Obtiene o establece un valor que indica si se siguen las reglas de restricción al intentar realizar cualquier operación de actualización.
ExtendedProperties	Obtiene la colección de la información personalizada del usuario asociada a DataSet.
HasErrors	Obtiene un valor que indica si hay errores en alguno de los objetos DataTable de este DataSet.
IsInitialized	Obtiene un valor que indica si el objeto DataSet está inicializado.

Propiedades Públicas de un DataSet

Nombre	Descripción
Relations	Obtiene la colección de relaciones que vincula las tablas y permite el desplazamiento desde las tablas primarias a las secundarias.
RemotingFormat	Obtiene o establece una enumeración SerializationFormat para el objeto DataSet utilizado durante el funcionamiento remoto.
Site	Reemplazado. Obtiene o establece una interfaz System.ComponentModel.ISite para el objeto DataSet.
Tables	Obtiene la colección de tablas incluidas en DataSet.

Métodos Públicos de un DataSet

Nombre	Descripción
AcceptChanges	Confirma todos los cambios realizados en este DataSet desde que se ha cargado o desde la última vez que se ha llamado a AcceptChanges .
Clear	Borra cualquier dato de DataSet mediante el procedimiento de quitar todas las filas de todas las tablas.
Clone	Copia la estructura de DataSet, incluidos todos los esquemas, relaciones y restricciones de DataTable . No copia ningún dato.
Copy	Copia la estructura y los datos para este objeto DataSet.
GetChanges	Obtiene una copia del objeto DataSet que contiene todos los cambios que se le han realizado desde la última vez que se cargó o desde que se llamó a AcceptChanges.

Métodos Públicos de un DataSet

Nombre	Descripción
GetXml	Devuelve la representación XML de los datos almacenados en DataSet.
GetXmlSchema	Devuelve el esquema XML para la representación XML de los datos almacenados en DataSet.
HasChanges	Obtiene un valor que indica si DataSet presenta cambios, incluyendo filas nuevas, eliminadas o modificadas.
Load	Rellena un objeto DataSet con valores de un origen de datos utilizando la interfaz IDataReader proporcionada.

Métodos Públicos de un DataSet

Nombre	Descripción
Merge	Combina el objeto DataSet, el objeto DataTable o la matriz de objetos DataRow que se especifique en el objeto DataSet o DataTable actual.
ReadXml	Lee esquema y datos XML en el objeto DataSet.
ReadXmlSchema	Lee un esquema XML en el DataSet.
RejectChanges	Deshace todos los cambios realizados en el DataSet desde que se ha creado o desde que se ha llamado por última vez a DataSet.AcceptChanges.
Reset	Restablece el estado original del objeto DataSet.
WriteXml	Escribe datos XML y, de forma opcional, el esquema del DataSet.
WriteXmlSchema	Escribe la estructura del DataSet como un esquema XML.

ADO.NET

- Elegir un **DataReader** o un **DataSet**
- A la hora de decidir si su aplicación debe utilizar un *DataReader* o un *DataSet*, debe tener en cuenta el tipo de funcionalidad que su aplicación requiere.
- Use un **DataSet** para hacer lo siguiente:
 - Almacenar datos en la memoria caché de la aplicación para poder manipularlos. Si solamente necesita leer los resultados de una consulta, el *DataReader* es la mejor elección.
 - Utilizar datos de forma remota entre un nivel y otro o desde un servicio Web XML.
 - Interactuar con datos dinámicamente, por ejemplo para enlazar con un control de formularios Windows Forms o para combinar y relacionar datos procedentes de varios orígenes.
 - Realizar procesamientos exhaustivos de datos sin necesidad de tener una conexión abierta con el origen de datos, lo que libera la conexión para que la utilicen otros clientes.

ADO.NET

- Si no necesita la funcionalidad proporcionada por el *DataSet*, puede mejorar el rendimiento de su aplicación si utiliza el *DataReader* para devolver sus datos de sólo avance y de sólo lectura.
- Aunque el *DataAdapter* utiliza el *DataReader* para llenar el contenido de un *DataSet*, al utilizar el *DataReader* puede mejorar el rendimiento porque no usará la memoria que utilizaría el *DataSet*, además de evitar el procesamiento necesario para crear y llenar el contenido de *DataSet*.

Enlazar datos al control DataGridView

- Puede utilizar el diseñador para conectar un control DataGridView a distintos orígenes de datos, incluso bases de datos, objetos de negocio o servicios Web.
- Cuando enlaza el control a un origen de datos mediante el diseñador, el control se enlaza automáticamente a un componente BindingSource que representa el origen de datos.
- Además, las columnas se generan automáticamente en el control para coincidir con la información del esquema proporcionada por el origen de datos.

Enlazar datos al control DataGridView

- Después de generar las columnas, puede modificarlas para satisfacer sus necesidades.
- Por ejemplo, puede quitar u ocultar aquellas columnas que no desee mostrar, puede reorganizar las columnas o puede modificar los tipos de columna.
- También puede enlazar varios controles DataGridView a las tablas relacionadas para crear relaciones principal/detalle.
- En esta configuración, un control muestra una tabla primaria y otro control muestra sólo aquellas filas de una tabla secundaria que están relacionadas con la fila actual de la tabla primaria.

Enlazar datos al control DataGridView

Para enlazar el control a un origen de datos

1. En la etiqueta inteligente del control, haga clic en la flecha de lista desplegable para la opción Elegir origen de datos.
2. Si su proyecto aún no tiene un origen de datos, hace clic en Agregar origen de datos del proyecto y siga los pasos indicados por el asistente.
3. El nuevo origen de datos aparecerá en la ventana desplegable "Elegir origen de datos". Si el nuevo origen de datos contiene sólo uno miembro, por ejemplo, una tabla de base de datos única, el control se enlazará automáticamente a dicho miembro.
4. Expanda los nodos y seleccione el origen datos que deseé enlazar al control.

Enlazar datos al control DataGridView

5. Si el origen de datos contiene más de un miembro, por ejemplo, si ha creado un System.Data.DataSet que contiene varias tablas, expanda el origen de datos y, a continuación, seleccione el miembro específico que se va a enlazar.
6. Para crear una relación principal/detalle, en la ventana desplegable Elegir origen de datos de un segundo control DataGridView, expanda el BindingSource creado para la tabla primaria y, a continuación, seleccione la tabla secundaria relacionada en la lista que se muestra.



E.T.S.
INGENIERÍA
INFORMÁTICA

Gestión de la Información

Grado en Ingeniería del Software



UNIVERSIDAD
DE MÁLAGA



LENGUAJES Y
CIENCIAS DE LA
COMPUTACIÓN
UNIVERSIDAD DE MÁLAGA

Tema 4.3 LINQ

José Luis Pastrana Brincones
pastrana@lcc.uma.es

Introducción

- ▶ LINQ son las siglas de Language Integrated Query.
- ▶ Se pronuncia “link”, aunque hay mucha gente que incorrectamente lo llama “lin-queue”.
- ▶ Introduce las consultas en el lenguaje C# como ciudadanos de primera clase..
- ▶ El compilador comprueba las consultas LINQ.
 - Visual Studio las consultas resalta la sintaxis y ayuda con el IntelliSense-aware.
- ▶ Ofrece a los desarrolladores una sintaxis lógicamente estructurada y fuertemente tipada para la consulta de datos.
- ▶ Las consultas se realizan igual tanto para bases de datos, documentos XML o colecciones de datos como listas, colas, etc.
- ▶ LINQ puede ser extendido para permitir el acceso a cualquier fuente de datos.

Introducción

The LINQ Project

C#

VB

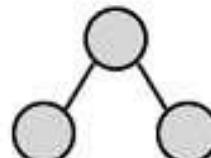
Others...

.NET Language-Integrated Query

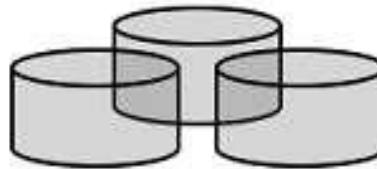
Standard
Query
Operators

LINQ to ADO.NET

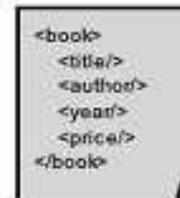
LINQ to XML
(System.XML)



LINQ to
Objects



LINQ
to
SQL LINQ
to
Entities



XML

Introducción

- ▶ **Integrado:** C# y VB
- ▶ **Aplicación:** múltiple fuentes de datos (datasources)
- ▶ **Extensible:** LINQ puede ser adaptado para trabajar con múltiples lenguajes y acceder a múltiples fuentes de datos. LINQ to Objects
 - LINQ to SQL
 - LINQ to XML
 - LINQ to Entities
 - Etc
- ▶ **Declarativo:** Se centra en qué hay que hacer y no en el cómo mo en qué orden debe ser realizado.
- ▶ **Jerárquico:** es capaz de generar y manipular grafos.
- ▶ **Componible:** Los resultados de una consulta pueden ser usados por una segunda consulta. Se pueden escribir 3 consultas separadas, pero relacionadas y LINQ notará las conexiones entre ellas y las combinará en una sólo y eficiente consulta que se ejecutará sólo 1 vez.
- ▶ **Transformativa:** El resultado de una consulta LINQ sobre una fuente de datos puede ser transformada en una segunda fuente de datos . Por ejemplo, el resultado de una consulta sobre una base de datos puede ser transformada en un documento XML

Integrado

- ▶ ADO:

```
SqlConnection sqlConnection = new SqlConnection(connectionString);
sqlConnection.Open();
System.Data.SqlClient.SqlCommand sqlCommand = new SqlCommand();
sqlCommand.Connection = sqlConnection;
sqlCommand.CommandText = "Select * from Customer";
return sqlCommand.ExecuteReader
    (CommandBehavior.CloseConnection)
```

- ▶ LINQ:

```
Northwind db = new Northwind(@"C:\Data\Northwnd.mdf");
var query = from c in db.Customers select c;
```

Múltiples Fuentes de Datos

Objects:

```
var query = from c in GetCustomers()
            where c.City == "Mexico D.F."
            select new { City = c.City, ContactName = c.ContactName };
```

SQL:

```
var query = from c in db.Customers
            where c.City == "Mexico D.F."
            select new { City = c.City, ContactName = c.ContactName };
```

XML:

```
var query = from x in customers.Descendants("Customer")
            where x.Attribute("City").Value == "Mexico D.F."
            select x;
```

Extensible

LINQ Extender	LINQ to Expressions	LINQ to LDAP	LINQ to Opf3
LINQ over C# project	LINQ to Flickr	LINQ to LLBLGen Pro	LINQ to Parallel (PLINQ)
LINQ to Active Directory	LINQ to Geo LINQ to Google	LINQ to Lucene	LINQ to RDF Files LINQ to Sharepoint
LINQ to Amazon	LINQ to Indexes	LINQ to Metaweb	LINQ to SimpleDB
LINQ to Bindable Sources	LINQ to IQueryable	LINQ to MySQL	LINQ to Streams
LINQ to CRM	LINQ to JavaScript	LINQ to NCover	LINQ to WebQueries
LINQ to Excel	LINQ to JSON	LINQ to NHibernate	LINQ to WMI

Declarativo

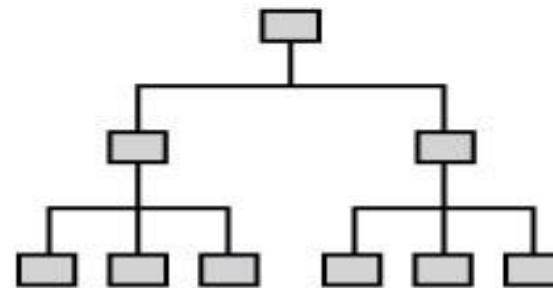
```
List<int> list01 = new List<int> { 1, 2, 3 };  
List<int> list02 = new List<int> { 4, 5, 6 };  
List<int> list03 = new List<int> { 7, 8, 9 };  
List<List<int>> lists = new List<List<int>>  
    { list01, list02, list03 };  
  
List<int> newList = new List<int>();  
foreach (var item in lists)  
{  
    foreach (var number in item) newList.Add(number);  
}  
  
var newList = from list in lists from num in list  
    where num % 2 == 0 orderby num select num;  
// Sólo pares y ordenados
```

Jerárquico

Grid versus Hierarchies

LINQ's hierarchical data model is more flexible than the grid-like data returned from a SQL query.

Name	Company	OrderId
John	Boring,inc	332121
Mary	RidgeCo, A.E.	322336



```
var query = from c in db.Customers  
select new { City = c.City,  
            orders = from o in c.Orders  
                      select new { o.OrderID }  
};
```

Componible

```
var query = from customer in db.Customers  
            where customer.City == "Paris"  
            select customer;
```

```
query2 = from customer in query  
            where customer.Address.StartsWith("25")  
            select customer;
```

Transformativo

```
var query =  
    new XElement("Orders", from c in  
    db.Customers where c.City == "Paris"  
  
    select new XElement("Order",  
        new XAttribute("Address", c.Address),  
        new XAttribute("City", c.City)));
```

- ▶ Transforma los resultados de una consulta LINQ to SQL Query en XML

Revisión Técnica

- ▶ Se puede usar para consultar colecciones genéricas del espacio de nombres `System.Collections.Generic`
 - `List<T>`, `Stack<T>`, `LinkedList<T>`, `Queue<T>`,
`HashSet<T>`, `Dictionary< TKey, Value >`.
- ▶ Inferencia de Tipos
 - ```
var i = 2;
i = "LINQ is strongly typed."; // Error
```
  - ```
IEnumerable<int> query = from number in list where number < 3
select number;
```
 - ```
var query = from number in list where number < 3 select number
```
- ▶ Tipos Anónimos
  - ```
var mountain =
new { Name = "Rainier", Height = 4392, State = "WA" };
```
- ▶ Expresiones Lambdas
 - ```
Func<int, int, int> myLambda = (a, b) => (a + b);
```

# Ejecución Postergada o Perezosa

---

```
List<int> list1 = new List<int> { 1,2,3,4,5 };
int num = 2;

var post = from n in list1 where n <= num select n;
num = 4;

foreach (var m in post) { Console.WriteLine(m); }

num = 2;

var post2 = (from n in list1 where n <= num
 select n).Count(); ;

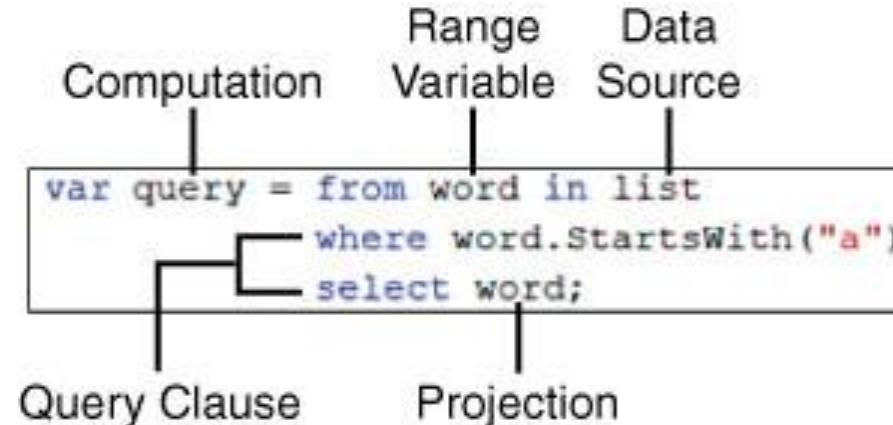
num = 4;

Console.WriteLine("cuenta {0}",post2);
```

# Essential LINQ

» LINQ to Objects

# Expresiones de Consulta



- ▶ La palabra reservada `var` le dice al compilador que infiera el tipo resultado de la consulta.
- ▶ Cuando una consulta LINQ to Objects devuelve algo que es `IEnumerable<T>` su ejecución es postergada hasta que sea necesaria su utilización.
- ▶ Los diferentes elementos de la consulta se llama cláusulas: cláusula `from`, cláusula `where` y cláusula `select`.
- ▶ La cláusula `select` al final de la consulta ayuda a definir el tipo devuelto por la consulta y se conoce como la proyección del resultado.

# Cláusulas

|               |                                                      |
|---------------|------------------------------------------------------|
| Primera Línea | Cláusula from con una variable y una fuente de datos |
| Líneas medias | Cláusulas where, orderby, join, let                  |
| Última Línea  | Cláusulas select y/o group-by                        |

```
var query = from word in list
where word.StartsWith("a")
select word;
```

Notación LINQ

- ▶ var query = list.Where(word => word.StartsWith("a"))  
.Select(word => word);

Notación  
“punto”

# Rango de las Variables

---

```
List<string> list = new List<string>
{ "LINQ", "query", "adventure" };
```

```
var query = from word in list
where word.StartsWith("a") select word;
```

- ▶ Se dice que **word** es introducida por la cláusula `from` usando la inferencia de tipos. El compilador sabe que **word** es un string porque lo deriva de `List<string>`.
- ▶ Puede haber casos donde sea necesario poner el tipo de forma explícita:

```
var query = from int num in ints
where num < 3 select num;
```

# Cláusulas Group-By al final de la Consulta

```
var query = from method in
 typeof(System.Linq.Enumerable).GetMethods()

 where method.DeclaringType == typeof(Enumerable)

 orderby method.Name

 group method by method.Name;

foreach (var item in query)

{
 Console.WriteLine(item.Key);

}

▶ El tipo del objeto devuelto por una llamada group-by es
System.Linq.IGrouping< TKey, TElement >. Este objeto implementa
IEnumerable< TElement > y provee una propiedad clave del tipo
TKey.
```

# Cláusulas Group-By y la palabra reservada INTO

---

```
var query = from method in
 typeof(System.Linq.Enumerable).GetMethods()
 where method.DeclaringType == typeof(Enumerable)
 orderby method.Name
 group method by method.Name
 into g select new { Name = g.Key, Overloads = g.Count() };
```

# Cláusula LET

---

```
var list = new
 List<int> { 1, 2, 3, 4, 5, 6, 7, 8, 9 };

var query = from n in list
 where (n > 3) & (n < 8)
 let g = n * 2
 where g % 2 == 0
 let newList = new List<int> { 2, 3 }
 from l in newList
 select new { l, r = g * l };
```

# JOINS

---

```
var query = from p in people
 join o in orders on p.MusicianId equals o.MusicianId
 select new { Musician = p.Name, OrderId = o.OrderId };

foreach (var item in query)
{
 Console.WriteLine(item);
}
```

- ▶ Al usar el operador `equals` en vez del operador `==` se espera recordar que sólo se permiten joins por igualdad.

```
var query1 = from p in people join o in orders on
 p.MusicianId equals o.MusicianId join i in instruments on
 o.InstrumentId equals i.InstrumentId
 select new { Musician = p.Name, OrderId = o.OrderId,
 Instrument = i.Name };
```

# JOINS Agrupados

---

```
var query = from p in people
 join o in orders on p.MusicianId equals o.MusicianId
 into orderGroups
 select new { Musician = p.Name, Orders = orderGroups };
var query1 = from p in people
 join o in orders on p.MusicianId equals o.MusicianId
 into orderGroups
 select new {
 Musician = p.Name,
 Orders = from o in
 orderGroups join i in instruments on o.InstrumentId
 equals i.InstrumentId
 select i.Name };

```

# Operadores de Consulta.

(\*: no postergados)(+: útiles)

| Operator Type | Operator Name     | Operator Type | Operator Name       | Operator Type | Operator Name       |
|---------------|-------------------|---------------|---------------------|---------------|---------------------|
| Partitioning  | Take              | Conversion    | AsEnumerable        | Generation    | Range <sup>+</sup>  |
|               | Skip              |               | ToDictionary*       |               | Repeat <sup>+</sup> |
|               | TakeWhile         |               | ToList*             |               | Empty <sup>+</sup>  |
|               | SkipWhile         |               | ToLookup*           |               | Any*, All*          |
| Join          | Join              |               | OfType              | Grouping      | Contains*           |
|               | GroupJoin         |               | ToLookup*           |               | GroupBy             |
| Ordering      | OrderBy           | Element       | Cast                | Equality      | SequenceEqual*      |
|               | OrderByDescending |               | First*              | Restriction   | Where               |
|               | ThenBy            |               | FirstOrDefault*     | Aggregate     | Count*              |
|               | Reverse           |               | Last*               |               | LongCount*          |
| Set           | Distinct          |               | LastOrDefault*      |               | Sum*                |
|               | Union             |               | Single*             |               | Min*                |
|               | Intersect         |               | SingleOrDefault*    |               | Max*                |
|               | Except            |               | ElementAt*          |               | Average*            |
|               | Concat            |               | ElementAtOrDefault* |               | Aggregate*          |
| Projection    | Select            |               | DefaultIfEmpty      |               |                     |
|               | SelectMany        |               |                     |               |                     |

# Operadores de Particionado

| Nombre Operador | Descripción                                                       |
|-----------------|-------------------------------------------------------------------|
| Take            | Toma los n primeros elementos de una secuencia                    |
| Skip            | Salta los n primeros elementos de una secuencia                   |
| TakeWhile       | Toma elementos de una secuencia mientas una condición sea cierta  |
| SkipWhile       | Salta elementos de una secuencia mientas una condición sea cierta |

```
var list = Enumerable.Range(1, 100);

var query4 = (from r in list where r % 11 == 0
 select r).TakeWhile(r => r < 50);
```

# Operadores de Elementos

| Nombre Operador     | Descripción                                                                                  |
|---------------------|----------------------------------------------------------------------------------------------|
| First*              | Recupera el primer elemento de una enumeración                                               |
| FirstOrDefault*     | Obtiene el primer elemento o el valor por defecto de un tipo lista.                          |
| Last*               | Recupera el último elemento de una enumeración                                               |
| LastOrDefault*      | Obtiene el último elemento o el valor por defecto de un tipo lista.                          |
| Single*             | Devuelve el único elemento de una secuencia que satisface una condición                      |
| SingleOrDefault*    | Devuelve el único elemento o el valor por defecto de un tipo lista.                          |
| ElementAt*          | Devuelve el elemento que ocupa una determinada posición en una lista                         |
| ElementAtOrDefault* | Devuelve el elemento que ocupa una determinada posición en una lista o el valor por defecto. |
| DefaultIfEmpty      | Recupera el valor por defecto si la lista está vacía o es null                               |

# Operadores de Conjuntos

| Nombre Operador | Descripción                                                                                        |
|-----------------|----------------------------------------------------------------------------------------------------|
| Distinct        | Muestra los elementos distintos en una secuencia                                                   |
| Union           | Muestra los elementos únicos obtenidos al combinar 2 conjuntos                                     |
| Intersect       | Muestra los elementos comunes a 2 conjuntos                                                        |
| Except          | Muestra los elementos de un conjunto excepto aquellos que pertenecen a un 2º conjunto (diferencia) |
| Concat          | Concatena 2 secuencias                                                                             |
| SequenceEqual   | Comprueba si 2 secuencias son iguales                                                              |

```
var listA = Enumerable.Range(1, 3);

var listB = new List<int> { 3, 4, 5, 6 };

var listC = listA.Union(listB);
```

# Operadores Agregados

| Nombre Operador | Descripción                                                     |
|-----------------|-----------------------------------------------------------------|
| Count*          | Cuenta los elementos de una secuencia                           |
| LongCount*      | Cuenta los elementos de una secuencia muy larga                 |
| Sum*            | Suma los elementos de una secuencia                             |
| Min*            | Encuentra el menor elemento de una secuencia                    |
| Max*            | Encuentra el mayor elemento de una secuencia                    |
| Average*        | Calcula la media de los elementos de una secuencia              |
| Aggregate*      | Realiza operaciones binarias con los elementos de una secuencia |

list.Aggregate((a, b) => (a \* b))

# Ejercicios

---

1. Obtener una lista con los primeros 100 números.
2. Usar linq para obtener los múltiplos de 7
3. Usar linq para obtener cuántos hay en esa lista (la del punto anterior)
4. Mostrar los múltiplos de 7 menores que un número leído de teclado (ejecución postergada)
5. Agrupar los números según el resto de la división por 7 (primero aquellos cuyo resto sea 1, luego 2, etc.)

# Essential LINQ

» LINQ to SQL

# Introducción

---

- ▶ LINQ to SQL permite a los desarrolladores acceder a datos relacionales como objetos fuertemente tipados traduciendo LINQ a SQL
- ▶ Se creó para puentear las diferencias entre los datos relacionales y los objetos del CLR.
  - Tuplas relacionales o registros frente a objetos fuertemente tipados
  - Identidad basada en valores frente a identidad basada en referencias
  - Claves foráneas frente a referencias a objetos
  - Resultados tabulares frente a grafos de objetos

# Creando la Clase Entidad

---

```
using System.Data.Linq;
using System.Data.Linq.Mapping;
[Table(Name="Customers")]
public class Customer {
 [Column(IsPrimaryKey=true)]
 public string CustomerID;
 [Column]
 public string City;
}
```



Entity Class

# DataContext

- ▶ Obtener objetos de la base de datos y actualizar los cambios.

```
// DataContext takes a connection string

DataContext db = new
 DataContext("c:\\\\northwind\\\\northwnd.mdf");

// Get a typed table to run queries

Table<Customer> Customers = db.GetTable<Customer>();

// Query for customers from London

IQueryable<Customer> CustomerQuery = from c in Customers
 where c.City == "London" select c;

foreach (var cust in CustomerQuery)
 Console.WriteLine("id = {0}, City = {1}",
 cust.CustomerID, cust.City);
```

# DataContext Fuertemente Tipado

```
public partial class NorthwindDataContext : DataContext
{
 public Table<Customer> Customers;
 public NorthwindDataContext(string connection):
 base(connection) { }

 NorthwindDataContext db = new
 NorthwindDataContext("c:\\\\northwind\\\\northwnd.mdf");
 IQueryable<Customer> CustomerQuery =
 from c in db.Customers
 where c.City == "London"
 select c;
```

# Relaciones

---

- ▶ Relaciones en Bases de datos relacionales → Valores de clave externa referidos a claves primarias en otras tablas → operación de join
- ▶ Objects → referencias a propiedades o colecciones.
- ▶ La notación punto es más sencilla que el uso de un join porque no se necesita especificar la condición de unión en cada vez que se recorre la colección.
- ▶ El definir relaciones le permite navegar por los datos usando la notación punto, por lo que no tiene que hacer uso explícito del operador JOIN de LINQ en la mayoría de los casos.

# Definiendo las Relaciones

---

```
[Table(Name="Customers")]
public class Customer
{
 [Column(IsPrimaryKey=true)]
 public string CustomerID;
 ...
 private EntitySet<Order> _Orders;
 [Association(Storage="_Orders",
 OtherKey="CustomerID")]
 public EntitySet<Order> Orders {
 get {return this._Orders; }
 set {this._Orders.Assign(value); }
 }
}
```

# Definiendo las Relaciones

---

```
[Table(Name="Orders")]
public class Order
{
 [Column(IsPrimaryKey=true)]
 public int OrderID;
 [Column]
 public string CustomerID;
 private EntityRef<Customer> _Customer;
 [Association(Storage="_Customer", ThisKey="CustomerID")]
 public Customer Customer {
 get{return this._Customer.Entity;}
 set{this._Customer.Entity = value;}
 }
}
```

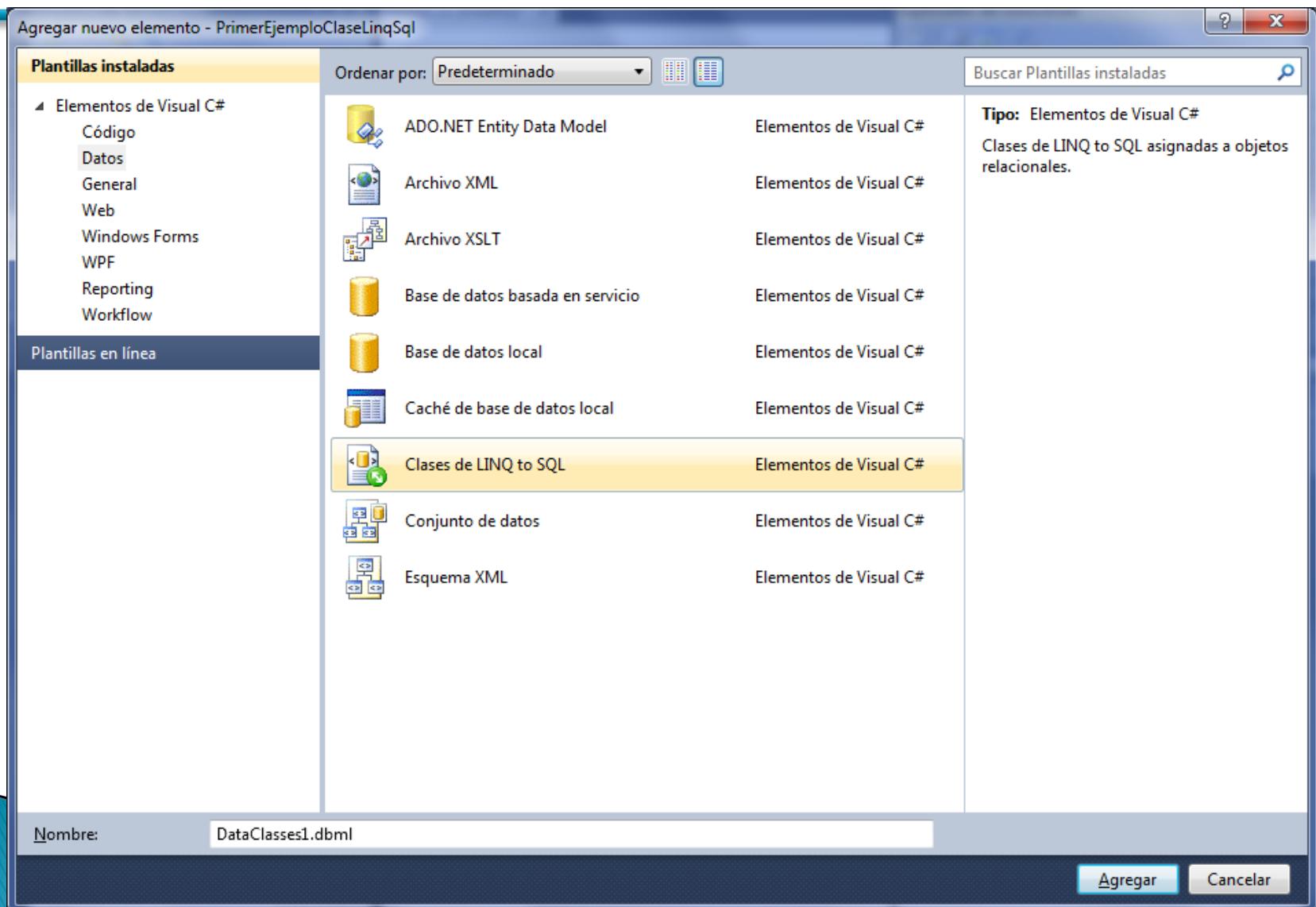
# Modificar/Salvar Entidades

```
NorthwindDataContext db = new NorthwindDataContext(...);
string id = "ALFKI";

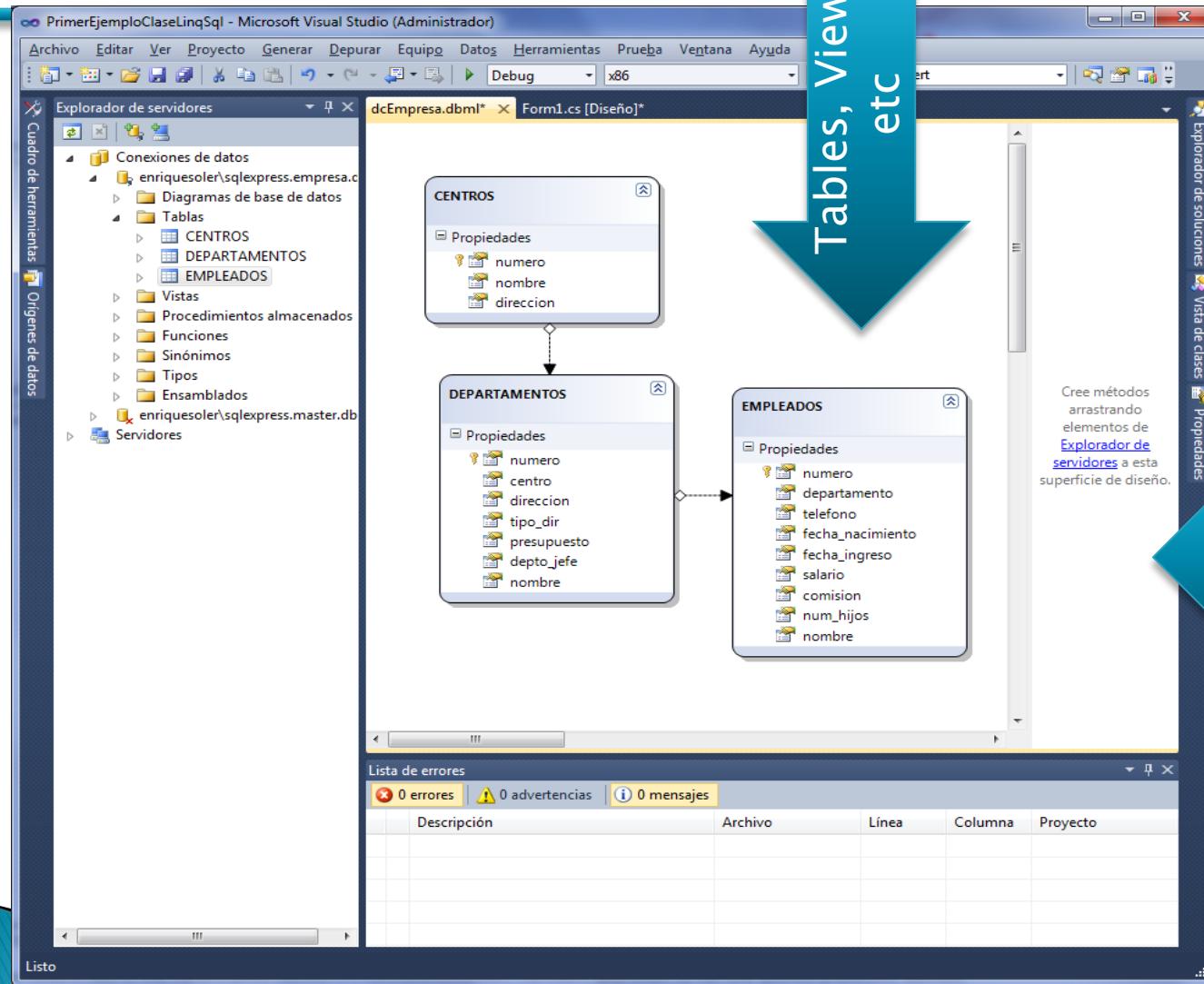
var cust = db.Customers.Single(c => c.CustomerID == id);
// Change the name of the contact
cust.ContactName = "New Contact";
// Delete an existing Customer
string id2 = "FISSA";

var cust2 = db.Customers.Single(c => c.CustomerID == id2);
db.Customers.DeleteOnSubmit(cust2);
// Create and add a new Order to Orders collection
Order ord = new Order { OrderDate = DateTime.Now };
cust.Orders.Add(ord);
// Ask the DataContext to save all the changes
db.SubmitChanges();
```

# Uso de la Herramienta Gráfica de Mapeo



# Uso de la Herramienta Gráfica de Mapeo



# Data Source: Object

Asistente para la configuración de orígenes de datos

Elegir un tipo de origen de datos

¿De dónde obtendrá la aplicación los datos?

Base de datos Servicio Objeto SharePoint

Permite elegir objetos que se pueden usar posteriormente para generar consultas.

< Anterior Siguiente >

Asistente para la configuración de orígenes de datos

Selecciónar los objetos de datos

Expanda los ensamblados y los espacios de nombres a los que se hace referencia para seleccionar objetos. Si falta un objeto en un ensamblado al que se hace referencia, cancele el asistente y recompile el proyecto que contiene el objeto.

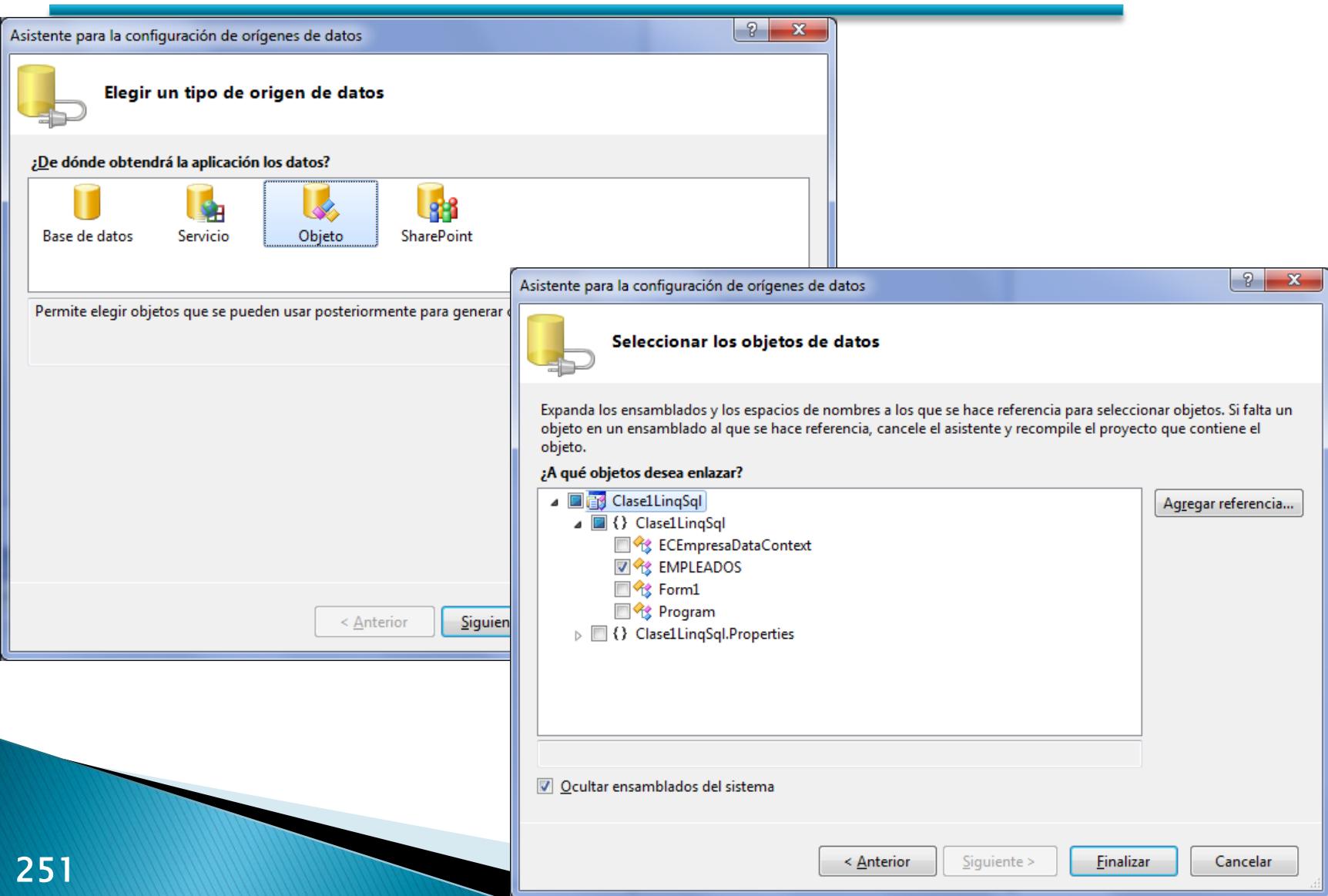
¿A qué objetos desea enlazar?

- Clase1LinqSql
  - {} Clase1LinqSql
    - EC EmpresaDataContext
    - EMPLEADOS
    - Form1
    - Program
  - {} Clase1LinqSql.Properties

Agregar referencia...

Ocultar ensamblados del sistema

< Anterior Siguiente > Finalizar Cancelar



# Uso de LINQ to SQL como fuente de Datos

---

```
private ECEmpresaDataContext dcE =
 new ECEmpresaDataContext();

public Form1()
{
 InitializeComponent();
 eMPLEADOSBindingSource.DataSource = dcE.EMPLEADOS;
}

private void buttonSAVE_Click(object sender, EventArgs e)
{
 dcE.SubmitChanges();
}
```

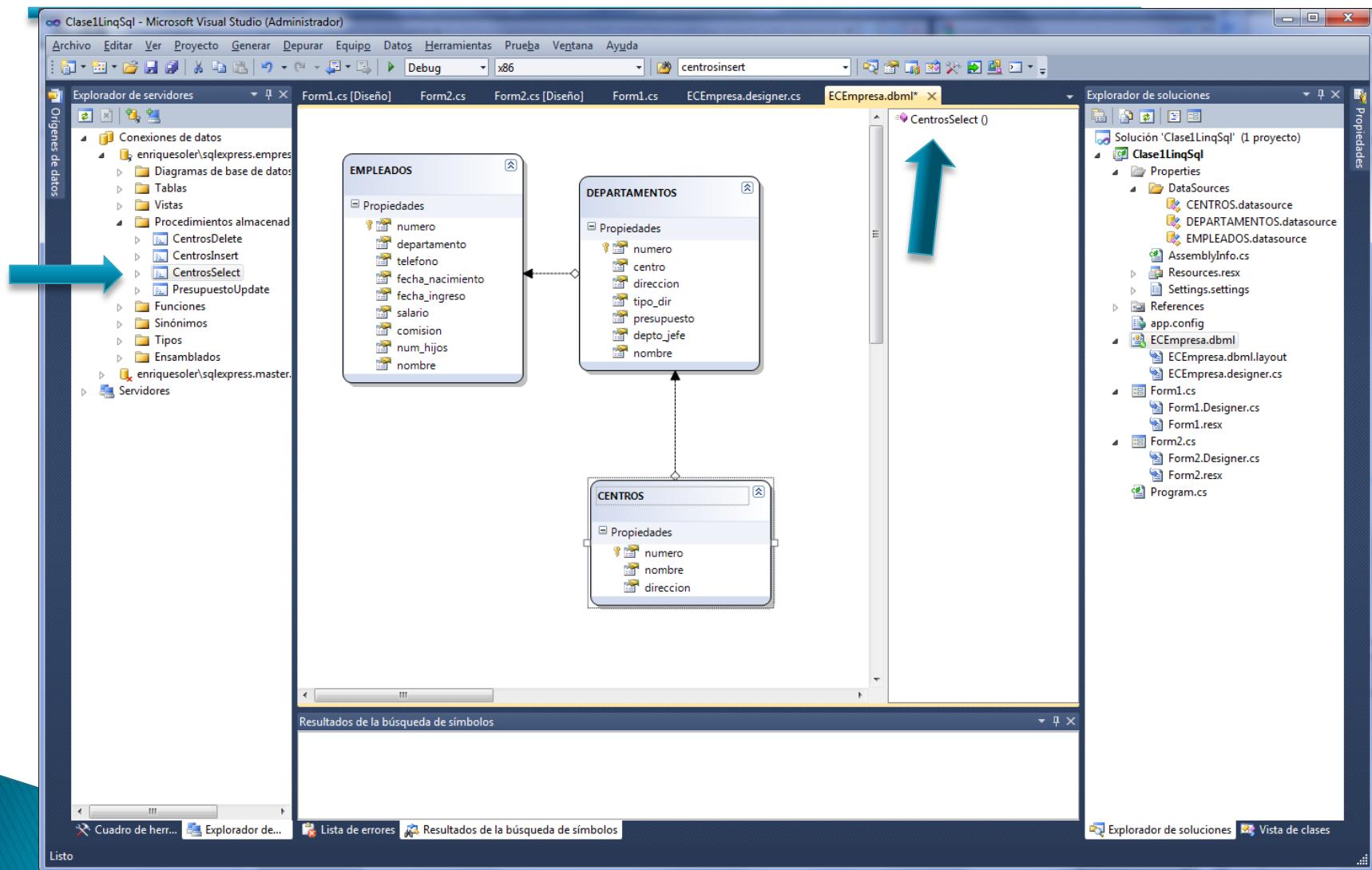
# Usando las Consultas

---

```
public Form1()
{
 InitializeComponent();
 EMPLEADOSBindingSource.DataSource =
 from e in dcE.EMPLEADOS
 where e.DEPARTAMENTOS.CENTROS.nombre == "SEDE CENTRAL"
 select e;

 var ricos = from e in dcE.EMPLEADOS where e.salario > 300
 select e;
 dataGridView1.DataSource = ricos;
}
```

# Uso de Procedimientos Almacenados en LINQ



# Uso de Procedimientos Almacenados en LINQ

---

```
var spCentros = dataContextE.CentrosSelect();
dgvCentros.DataSource = spCentros;
```

```
NorthwindDataContext db = new NorthwindDataContext();
var SalesResult = db.SalesByCategory("Produce", "1997");
```

# Entity Framework

---

- Entity Framework es un puente entre el mundo relacional y el mundo de los objetos (ORM).
- Entity Framework divide el modelo de datos en 3 modelos separados:
  - Modelo Conceptual
  - Modelo Físico
  - Modelo Lógico

# **Entity Framework**

---

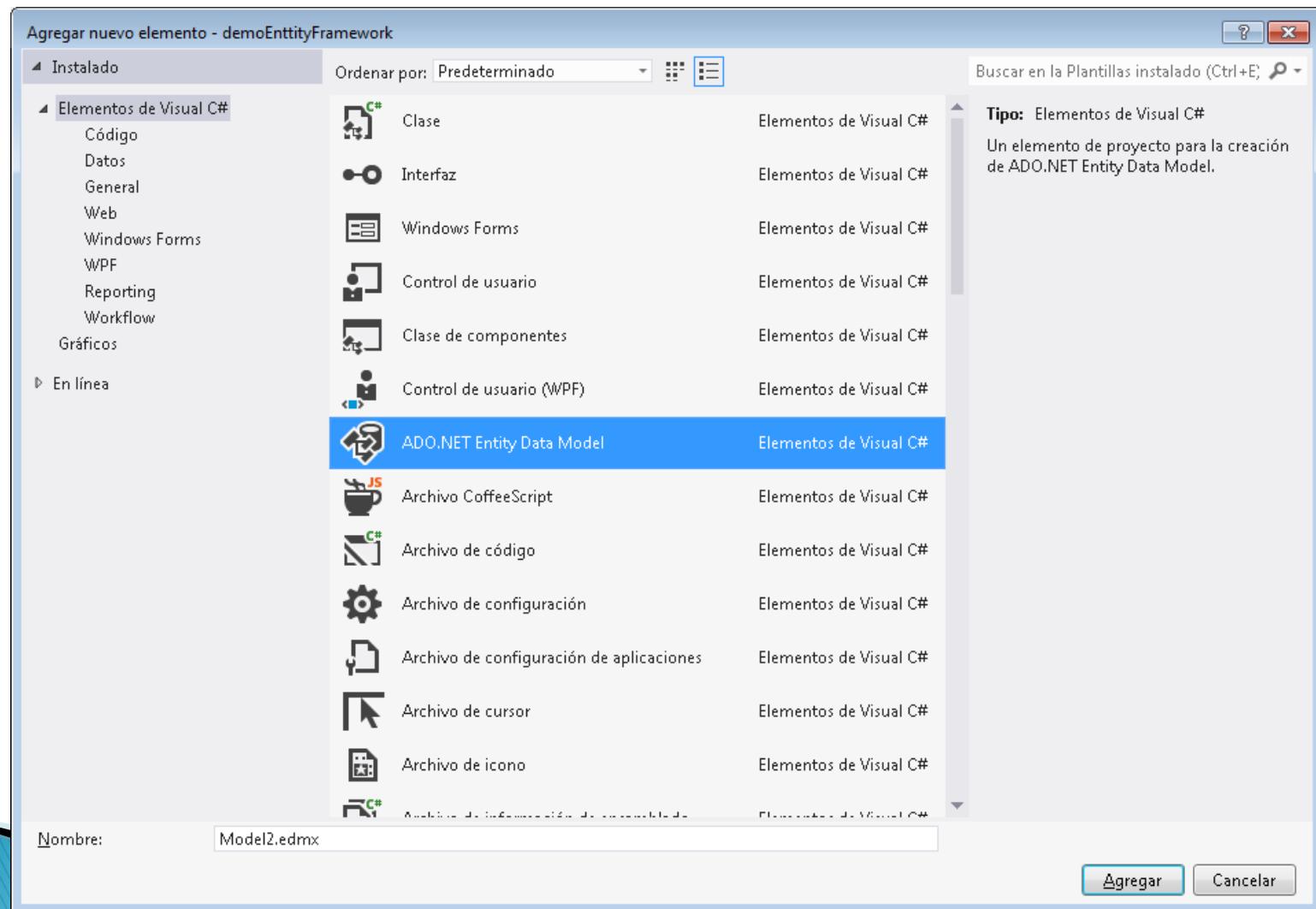
## **Primera Aproximación: BD -> Modelo**

Crear el Modelo a Partir de la Base de datos

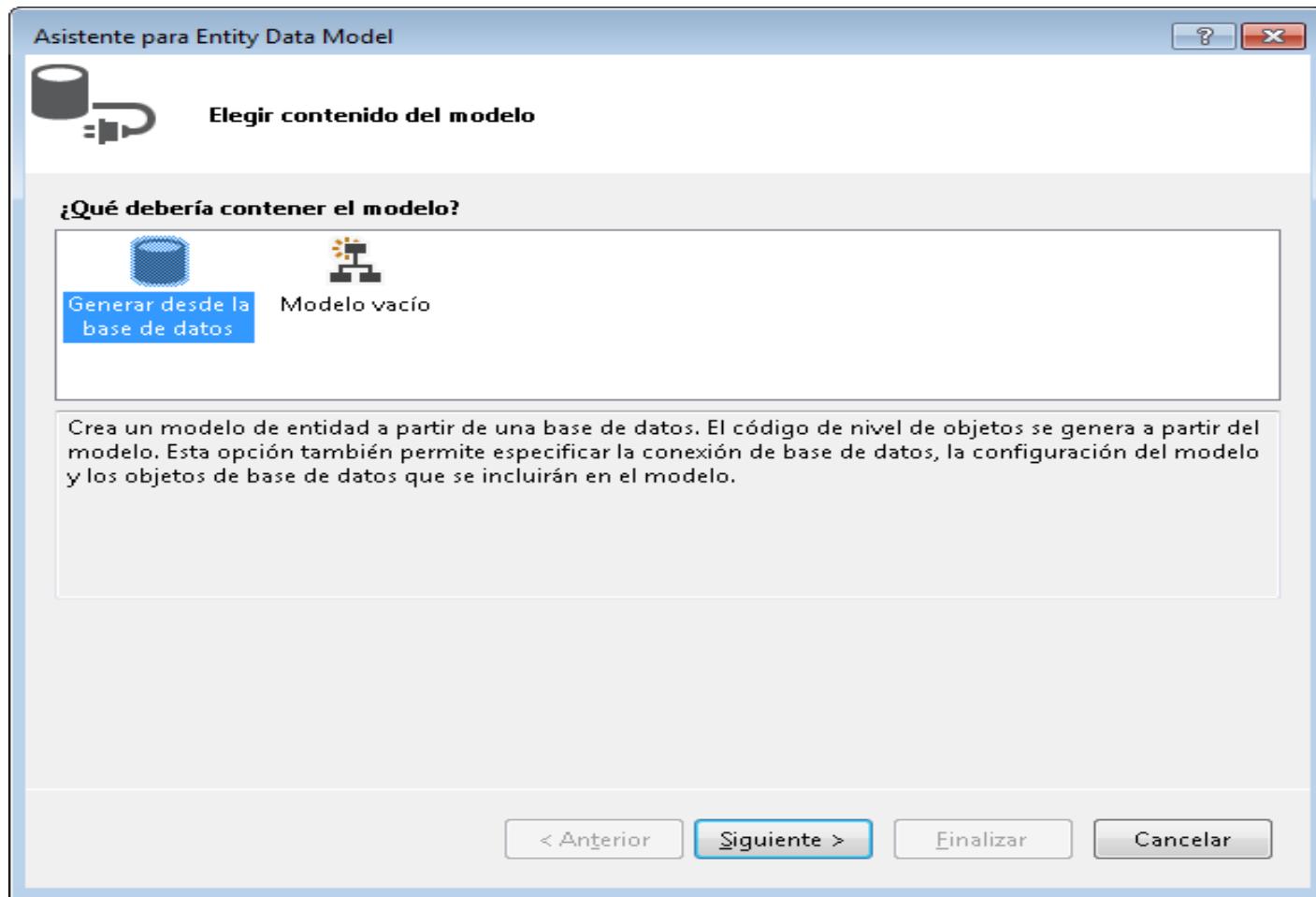
Seleccionamos la Base de datos de forma Similar a la realizada con ADO.NET

- Agregar Nuevo Elemento
- ADO.NET Entity Framework
- Generar desde la Base de Datos
- Seleccionamos la conexión

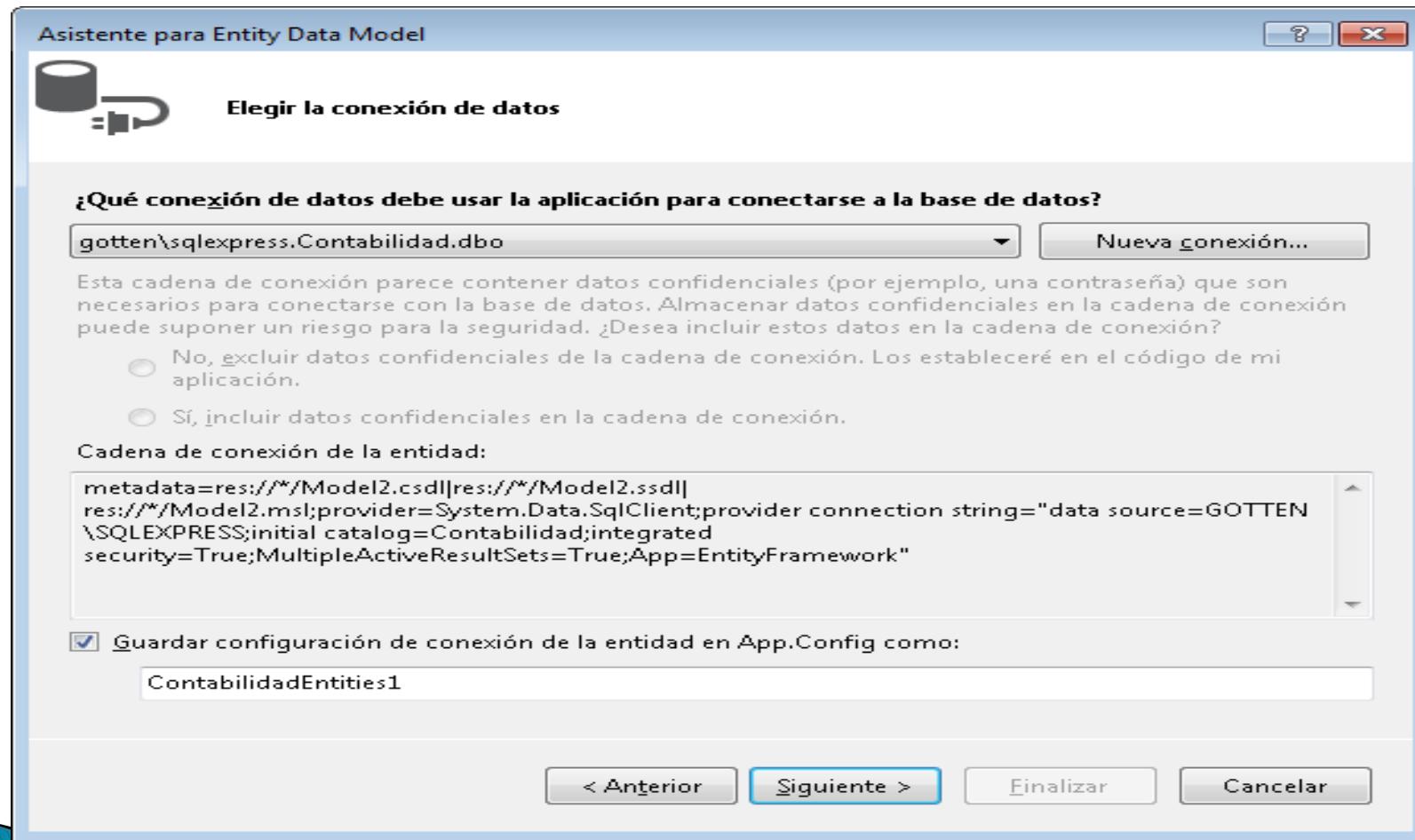
# Entity Framework



# Entity Framework

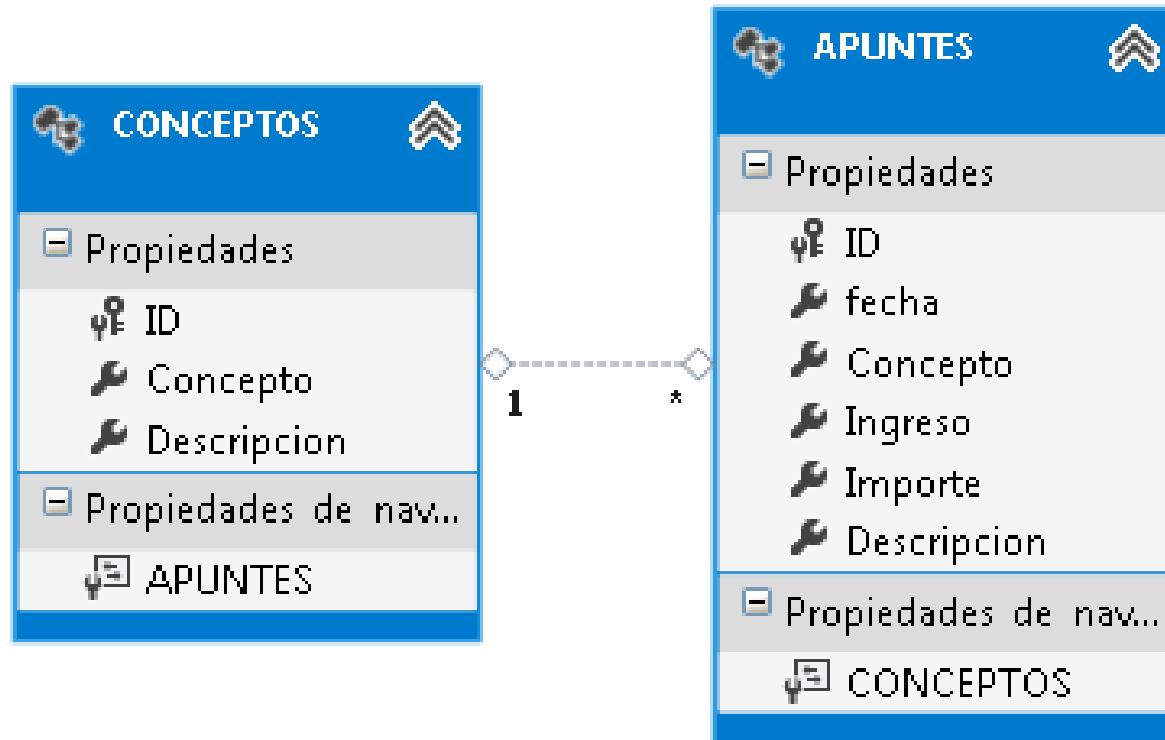


# Entity Framework



# Entity Framework

Y se han generado nuestras entidades y asociaciones



# **Entity Framework**

---

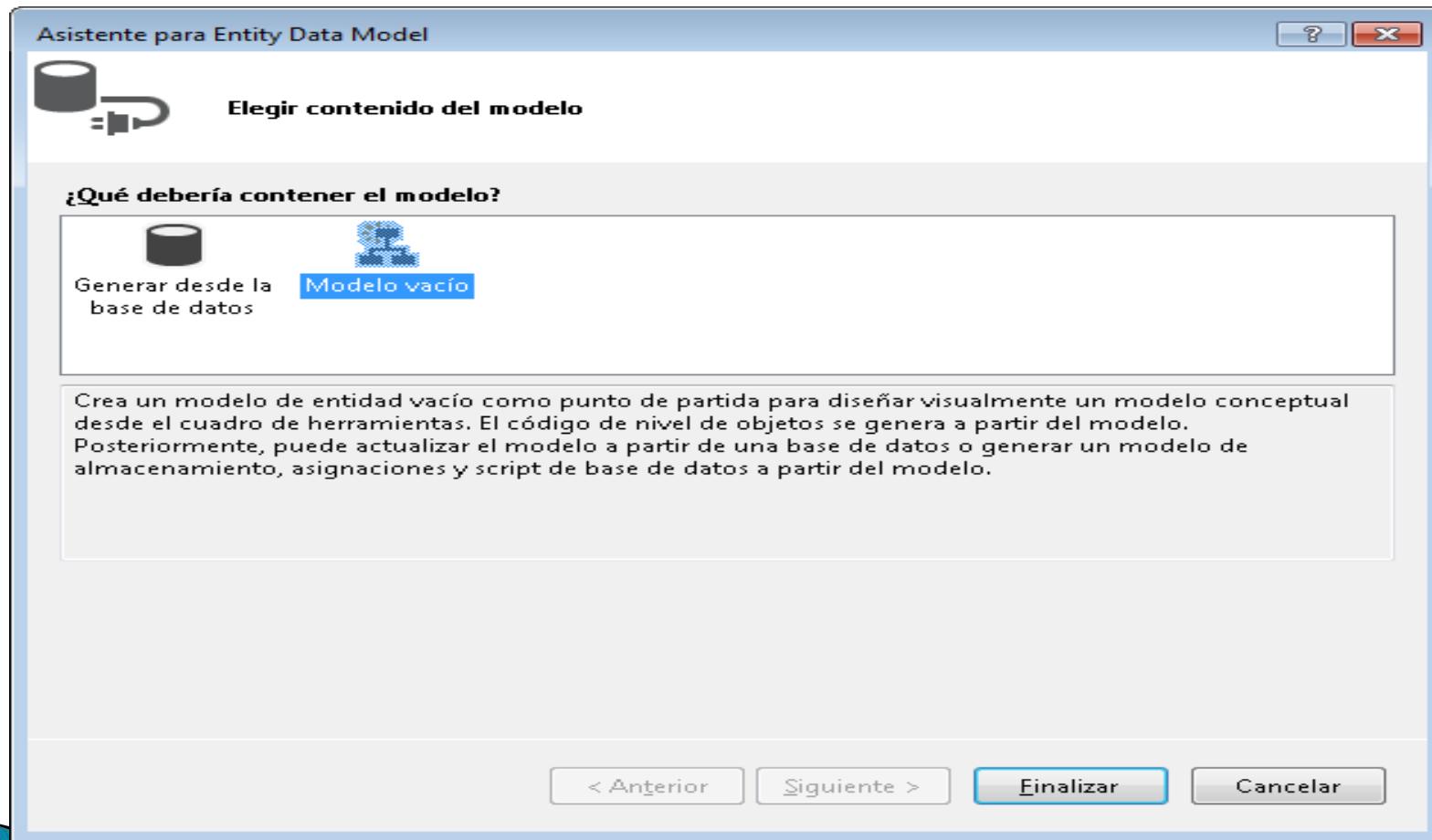
## **Segunda Aproximación: Modelo -> BD**

Crear el Modelo a generar la Base de datos a partir de éste.

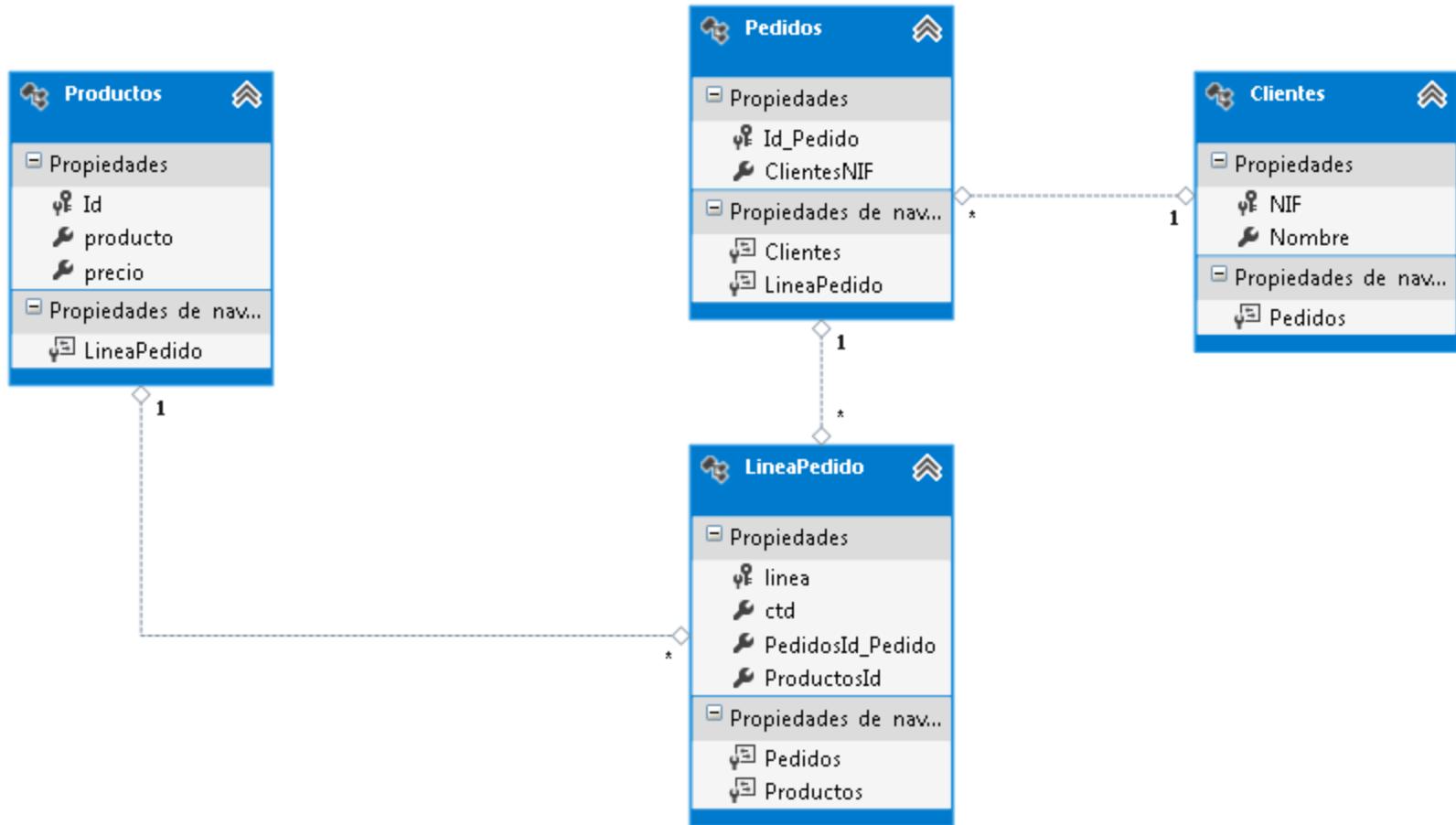
Procedemos de forma Similar a la anterior:

- Agregar Nuevo Elemento
- ADO.NET Entity Framework
- Modelo Vacío
- Diseñamos nuestra Entidades y Asociaciones
- Generamos la Base de Datos

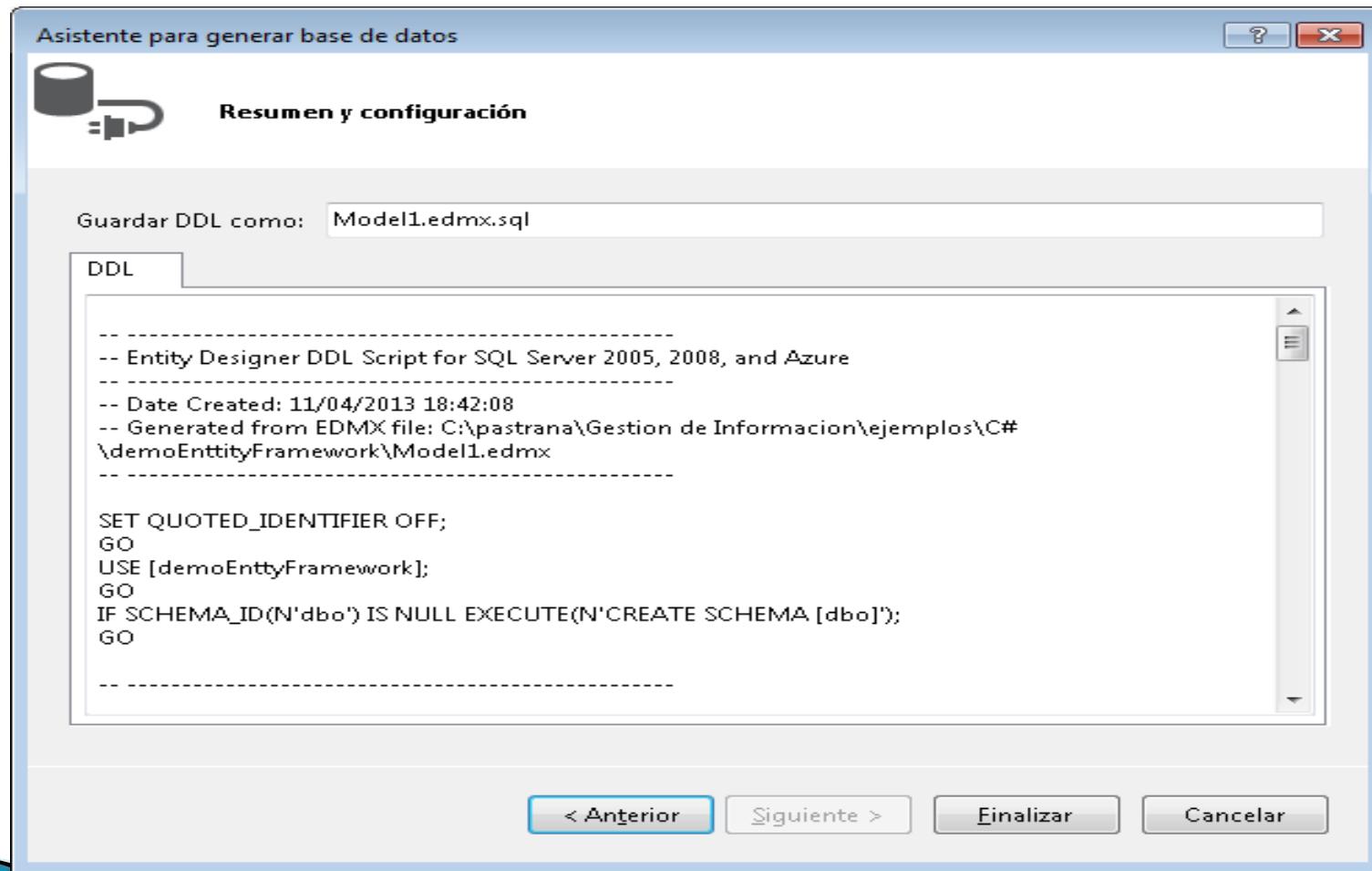
# Entity Framework



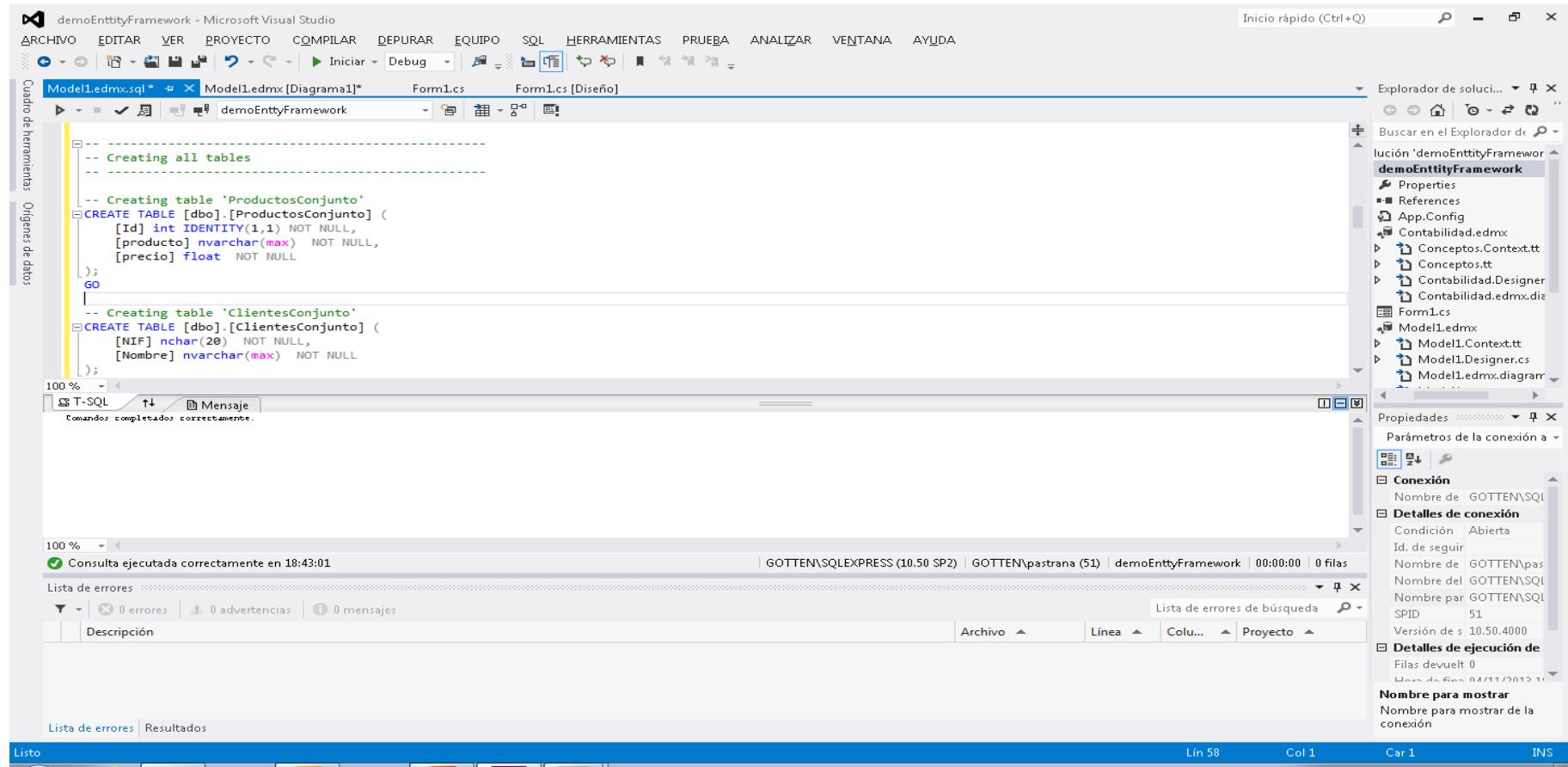
# Entity Framework



# Entity Framework



# Entity Framework



# Entity Framework

---

## Trabajando con los Datos. LINQ2Entities

### Usar un Contexto

```
private ContabilidadEntities contexto;
contexto = new ContabilidadEntities();
```

### Realizar una Consulta

```
var listaConceptos = from d in
 contexto.CONCEPTOS
 orderby d.ID
 select d;
```

# Entity Framework

---

## Enlazar una consulta con un control

```
private ContabilidadEntities contexto;
contexto = new ContabilidadEntities();

var listaConceptos = from d in
 contexto.CONCEPTOS
 orderby d.ID
 select d;

this.comboBox1.DisplayMember = "Concepto";
this.comboBox1.DataSource =
 listaConceptos.ToList();
```

# Entity Framework

---

## Trabajar con todos los elementos de la Colección

```
private ContabilidadEntities contexto;
contexto = new ContabilidadEntities();

var listaApuntes = contexto.APUNTES;
foreach(var ap2 in listaApuntes)
{
 MessageBox.Show(ap2.ID + " / "
 + ap2.Concepto + " / " + ap2.Ingreso
 + " / " + ap2.Importe + " / "
 + ap2.Descripcion);
}
```

# Entity Framework

---

## Modificar un Objeto

```
private ContabilidadEntities contexto;
contexto = new ContabilidadEntities();
```

```
APUNTES ap = contexto.APUNTES.
FirstOrDefault (x => x.ID == 1);
```

```
ap.Importe = 1000000;
contexto.SaveChanges();
```

# Entity Framework

---

## Añadir elementos de la Colección

```
ContabilidadEntities contexto = new
 ContabilidadEntities();
APUNTES ap3 = new APUNTES();
ap3.ID = 2;
ap3.fecha = DateTime.Now;
ap3.Importe = 2;
ap3.Ingreso = true;
ap3.Concepto = 2;
ap3.Descripcion = "C2";
contexto.APUNTES.Add(ap3);
contexto.SaveChanges();
```

# Entity Framework

---

## Eliminar elementos de la Colección

```
ContabilidadEntities contexto = new
 ContabilidadEntities();
var listaApuntes = contexto.APUNTES;

foreach(var ap2 in listaApuntes)
{
 if (ap2.ID == 2) listaApuntes.Remove(ap2);
}

contexto.SaveChanges();
```