Hexacta Labs

Introducción a .Net

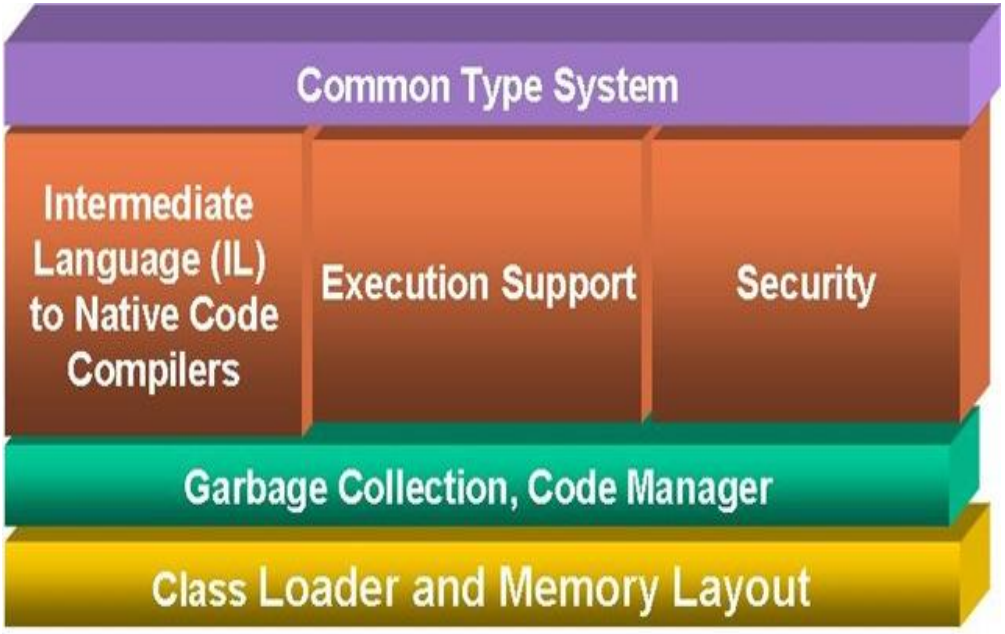
- Bases de la plataforma y C#

Agenda

- .Net
- C#
- Estructuras de control
- Trabajando con Clases
- Value Types y Reference Types
- Conversión de tipos
- Generics
- Collections
- Exceptions

.Net
¿Qué es?

- Common Language Runtime (CLR): provee una capa abstracción del sistema operativo.
- Base Class Libraries: código de bajo nivel para tareas comunes listo para usar.



.Net
Arquitectura Global

*.exe .dll

- Aplicaciones (*.exe) (tienen un único punto de entrada)
- Librerías (.dll)

.Net

¿Qué es un assemby?

- Un assembly contiene 4 cosas

  - Assembly manifest
    Tiene información para el runtime de .net , como nombre del assemby, versión, permisos requeridos y referencias a otros assemblies.

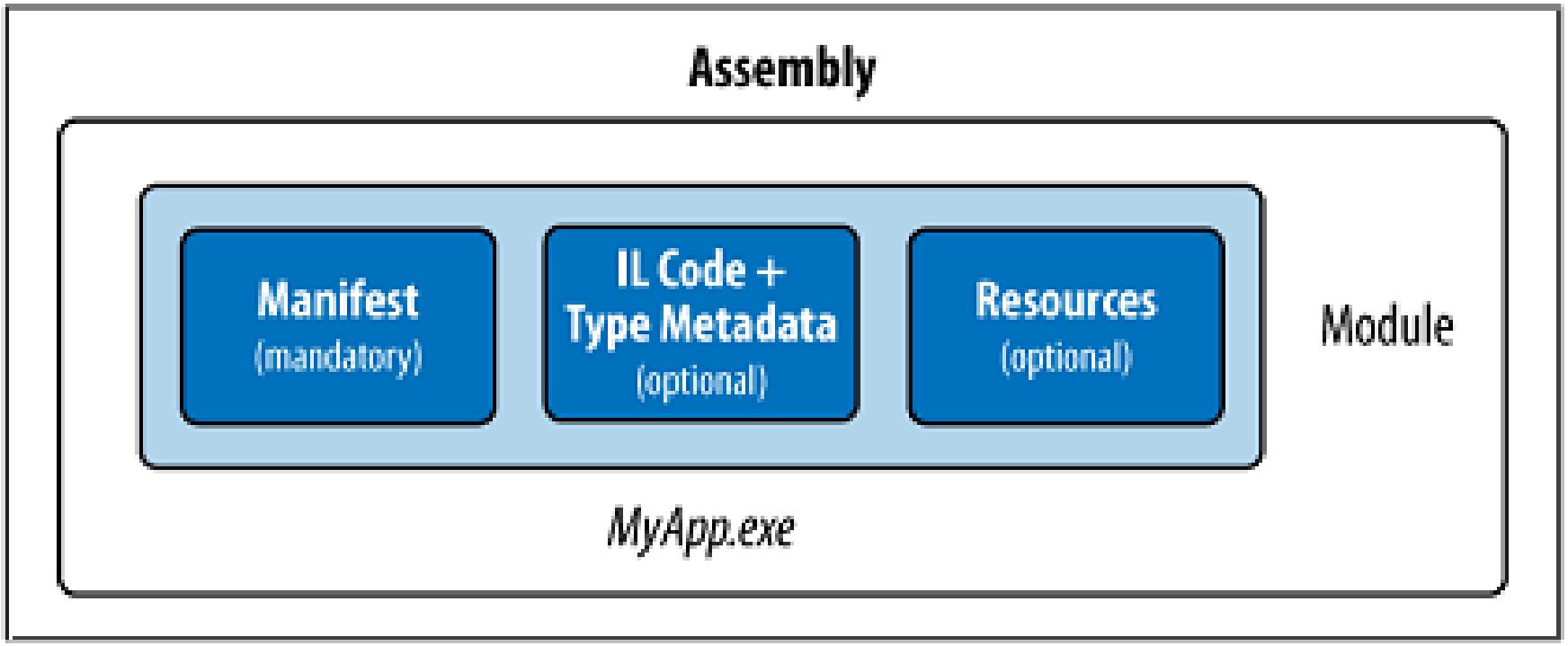
  - Application manifest
    Tiene información para el sistema operativo, como qué assembly debe ser desplegado y cuándo requiere elevar permisos.

  - Compiled types
    Código IL compilado y metadata de los tipos definidos en el assembly.

  - Resources
    Datos embebidos en el assembly, como imágenes y texto localizado.

.Net

¿Qué es un assemby?



- Un Strong Named Assembly es un assembly que tiene un identificador único compuesto por:
  - Un número único que le indicamos
  - Un hash que permite asegurar la identidad única

.Net
Global Assembly Cache

- Es un repositorio central.
- Almacena Assemblies.
- Versionado: la versión en la GAC se controla a nivel equipo y la puede controlar el administrador del mismo.

.Net

Namespaces

```
namespace Hexacta.HumanResources.Model.UnitTest.Builders
{
    public class HumanResourcesDepartmentBuilder
```

- Es un dominio dentro del cual el nombre de un Type debe ser único.
- Es parte del nombre completo de un Type.
- Sistema de organización.
- Indica jerarquía.
- Directorio lógico.
- Evitan problemas de nombres.
- Se pueden importar utilizando la directiva Using.
- Se pueden utilizar alias.

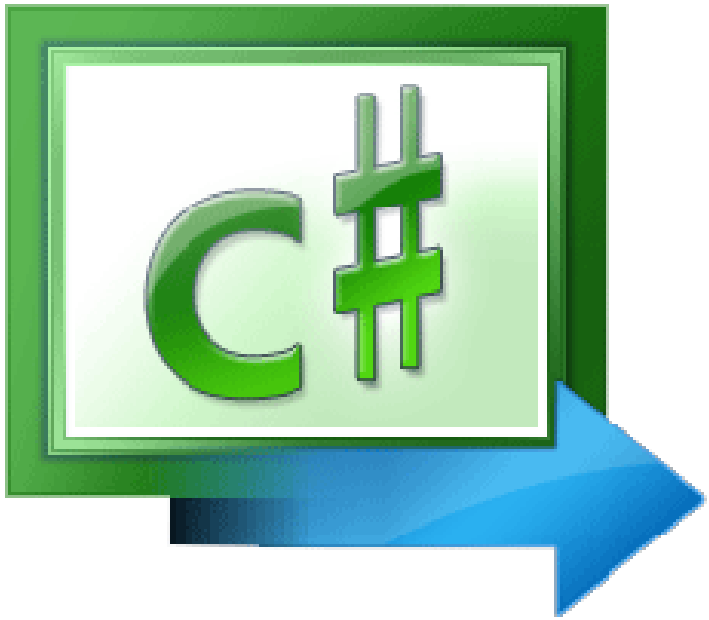
```
using Project = PC.MyCompany.Project; //alias

System.Console.WriteLine("Hola Mentor");

using System;
…
Console.WriteLine("Hola Mentor");
```

.Net

C#



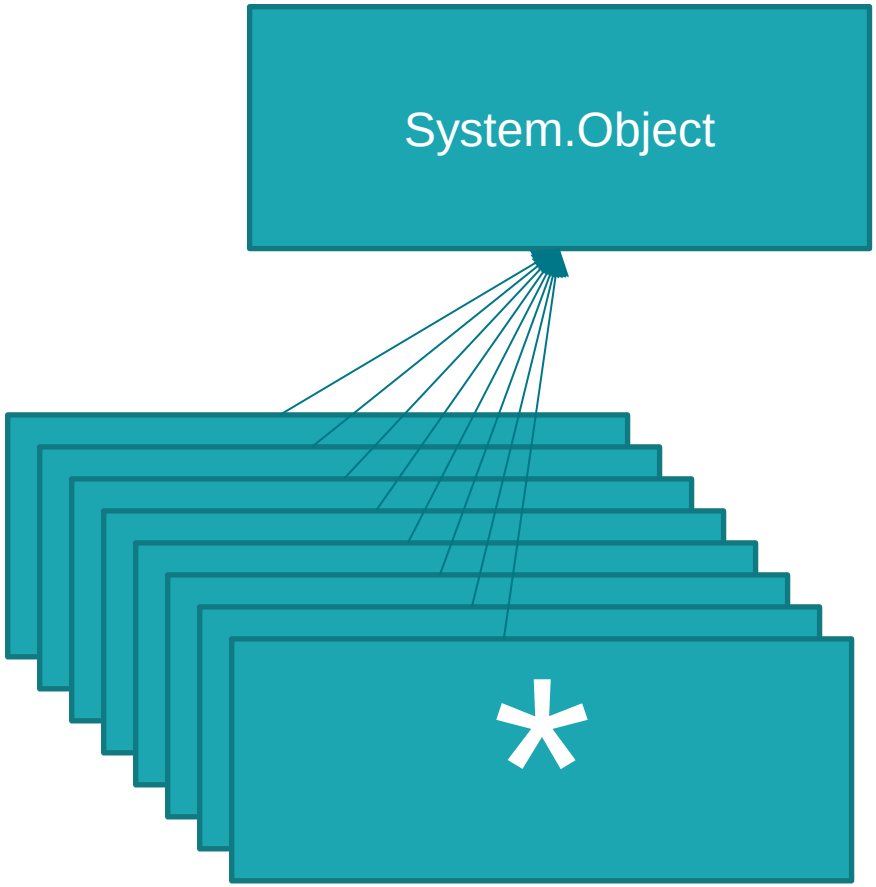
.Net

C#

- Evolución de C y C++.
- Autocontenido: no necesita adicionales al .cs (por ejemplo, headers).
- Tipos básicos independientes del compilador, SO o hardware.
- No tiene herencia múltiple
- OO
- Los métodos por defecto son sellados y no redefinibles.
- El compilador toma los archivos con extensión cs y los empaqueta en assemblies.

.Net

C# - Todo es un object



.Net

C# - Mi primera aplicación

```
1  using System;                                  // Importación del namespace
2
3  namespace Hexacta.MiProyecto                   // Declaración del namespace
4  {
5      public class Testa                         // Declaración de la clase
6      {
7          private static void Main()             // Declaración del método
8          {
9              int x = 12 * 30;                   // Sentencia 1
10             Console.WriteLine(x);              // Sentencia 1
11         }                                      // Fin del método
12     }                                          // Fin de la clase
13 }                                              // Fin del namespace
```

.Net

Estructuras de control

```
if (true)
{
}
else
{
}
```

```
while (true)
{

}
```

```
do
{

}
while (true);
```

```
switch (cardNumber)
{
    case 13:
        Console.WriteLine("King");
        break;
    case 12:
        Console.WriteLine("Queen");
        break;
    case 11:
        Console.WriteLine("Jack");
        break;
    case -1:
        goto case 12;
    default:
        Console.WriteLine(cardNumber);
        break;
}
```

```
for (int index = 0; index < length; index++)
{

}
```

```
foreach (var item in collection)
{

}
```

Jump statements: break, continue, goto, return, and throw

- Declaración

```
18      public class Mentor
19      {
20          //Fields, properties, methods and events go here...
21      }
```

.Net

Modificadores de acceso de clase

- Public: la visibilidad general, se puede acceder sin restricciones.
- Private: sólo puede accederse desde adentro del tipo.
- Internal: acceso público sólo desde adentro del assembly.

Campos	Fields
Constantes	Constants
Propiedades	Properties
Métodos	Methods
Eventos	Events
Operadores	Operators
Indizadores	Indexers
Constructores	Constructor
Destructores	Destructor
Tipos anidados	Nested types

- De instancia

```
public bool Exist()
```

- De clase

```
public static bool Exist(string filename)
```

.Net

Modificadores de acceso a miembros

```
public class Mentor
{
    // protected method:
    protected void AddMentoring() { }

    // private field:
    private List<Mentoring> mentoring;

    // protected internal property:
    protected internal int MentoringCount
    {
        get { return mentoring.Count; }
    }
}
```

.Net

Modificadores de acceso a miembros

- Public: la visibilidad general, se puede acceder sin restricciones.
- Private: sólo puede accederse desde adentro del tipo.
- Protected: será accesible sólo en una especialización de la clase (Heredera)
- Internal: acceso público sólo desde dentro del assembly (combinable con Protected).

- Un campo es una variable.

```
public class Mentor
{
        private int id;
}
```

- Una constante es una variable cuyo valor no puede cambiar nunca.

```
public class Mentor
{
    const int DaysBetweenMeetings = 30;
}
```

- Son miembros que ofrecen un mecanismo flexible para leer, escribir o calcular los valores de campos privados.
- El descriptor de acceso de una propiedad get se utiliza para devolver el valor de la propiedad y el descriptor de acceso set se utiliza para asignar un nuevo valor.
- La palabra clave value se usa para definir el valor asignado por el descriptor de acceso set.
- Las propiedades que no implementan un descriptor de acceso set son de sólo lectura.
- Posibilidad de utilizar propiedades auto implementadas.

```
private double seconds;

public double Hours
{
    get { return seconds / 3600; }
    set { seconds = value * 3600; }
}
```

```
public int Age { get; set; }
```

```
public int Age { get; protected set; }
```

- Un método realiza una acción en una serie de sentencias.
- Los métodos se declaran dentro de una clase con la siguiente estructura:

```
<access><optional modifier><return_type> <method_name>(<parameters>)
{

    <method_body>

}
```

```
private DateTime GetDayLastMeeting(int personId, DateTime? from = null)
```

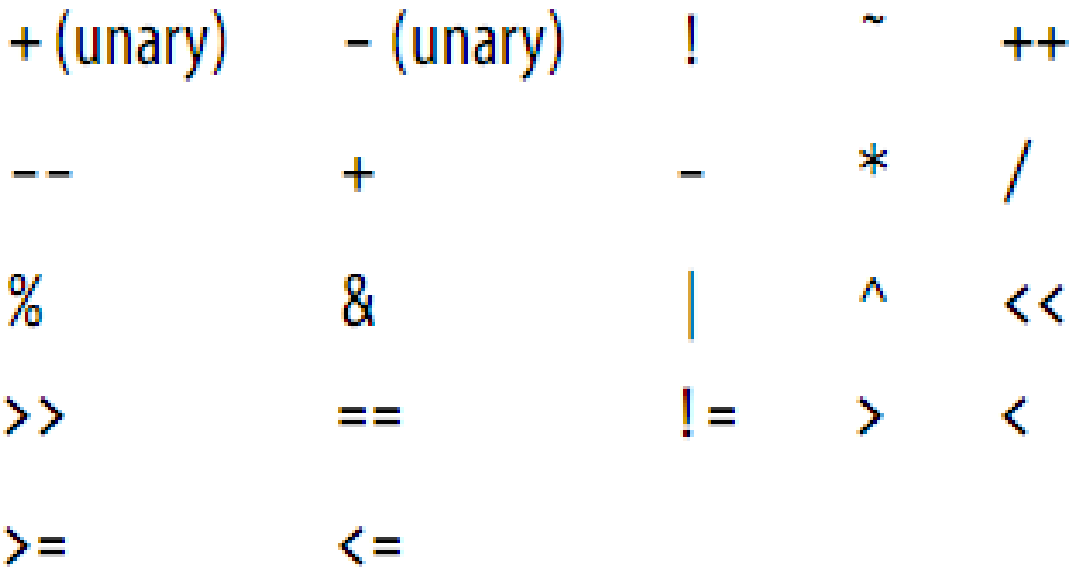
```
GetDayLastMeeting(from: DateTime.Now, personId: 45);

GetDayLastMeeting(45);
```

- Cada vez que se crea una instancia de una clase se llama al constructor.
- Los constructores ejecutan código de inicialización.
- Pueden existir n constructores que tienen parámetros diferentes (sobrecarga)
- Permiten modificadores de acceso.
- Siempre hay al menos un constructor. Si no se pone ninguno automáticamente habrá un “default constructor”: no recibe parámetros y no contiene ninguna sentencia.

```
public class Model
{
    public Model()
    {
        // Ctor Logic
    }
```

- Un operador es un término o símbolo que acepta como entrada expresiones u operados y devuelve un valor.
- Operadores que requieren sólo un operando se denominan unarios (ej: ++)
- Los operados que requieren 2 operandos son binarios (ej: +, -, *) y los de 3 son ternarios.
- Hay operadores que se pueden sobrecargar (+, -, *)
- Operador binario que no está en la imagen: **??**.



- Creación de objetos / Instanciación de clases

  - Palabra clave **new** seguida del nombre de la clase en la que se basará el objeto, de la siguiente manera:

```
Mentor object1 = new Mentor();
```

```
var object1 = new Mentor();
```

- Se crean dos cosas:
  - El objeto
  - La referencia al objeto

- Las clases pueden heredar de otras clases (que no sean sealed).
- De la clase de la cual se hereda se la denomina clase base.
- A la clase heredera se la denomina derivada o subclase.
- La clase derivada tiene dos tipos efectivos.

```
6  namespace Hexacta.Mentor.Model
7  {
8      public class Employee
9      {
0          public string Name() { }
1      }
2
3      public class Mentor : Employee
4      {
5          public void AddMentoring() { }
6      }
7  }
```

.Net
Interfaces

- Describen un grupo de comportamientos relacionados
- Pueden estar compuestas de métodos, propiedades, eventos, indizadores.
- No puede contener campos.
- Sus miembros son automáticamente públicos.
- Pueden heredar de otras interfaces
- Las clases pueden implementar interfaces:
  - Una clase puede implementar más de una interfaz.
  - Cuando una clase implementa una interfaz, toma de ella sólo los nombres de método y las firmas, ya que la propia interface no contiene ninguna implementación.

```
public class Minivan : Car, IComparable
{
    public int CompareTo(object obj)
    {
        //implementation of CompareTo
        return 0;  //if the Minivans are equal
    }
}
```

.Net
Clases parciales

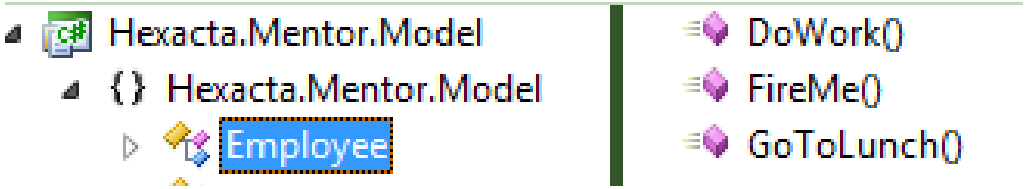
- Se puede dividir la definición de una clase en dos o más archivos de código fuente.
- Los archivos se combinan cuando se compila la aplicación.

```
public partial class Employee
{
    public void DoWork()
    {
    }
}

public partial class Employee
{
    public void GoToLunch()
    {
    }
}
```

```
public partial class Employee
{
    public virtual void FireMe()
    {

    }
}
```



.Net
Atributos

- Los atributos constituyen un medio apropiado para asociar información declarativa con código de C#.
- Se consulta en tiempo de build o ejecución mediante reflection.
- Agregan metadatos al programa.

```
[Serializable]
public class Mentor
{
    // Objects of this type can be serialized.
}
```

.Net

Variables locales implícitamente tipadas

- Es común declarar una variable e inicializarla en un mismo paso.
- Si el compilador es capaz de inferir el tipo de la expresión de inicialización, es posible usar la palabra clave **var** en lugar de la declaración de tipo.

```
var someInt = 0;
```

```
var someString = "Hola mundo!";
```

.Net
var y tipos anónimos

- En muchos casos el uso de *var* es opcional
- Cuando una variable es inicializada con un tipo anónimo es requerido.

```
var monja = new { Saludo="No tengo tipo" };

Console.WriteLine(monja.Saludo);
```

¿Qué hacer cuando se quiere reutilizar código sin forzar herencia?

La solución usando clases estáticas nos deja un código difícil de leer

Llamar a métodos estáticos de una clase como si fueran métodos de instancia de otra.

.Net
Extension methods

- El método debe tener las siguientes características:

  - La clase contenedora debe ser una clase static.
  - El método debe tener al menos un parámetro.
  - El primer parámetro debe tener la palabra clave this.

```
public static class MyExtensions
{
   public static int WordCount(this String str)
   {
      return str.Split(new char[] { ' ', '.', '?' }, StringSplitOptions.RemoveEmptyEntries).Length;
   }
}
```

```
string phrase = "Hello Extension Methods";
int wordQuantity =  phrase.WordCount();
```

.Net
Operador Is

- Prueba si se puede realizar una conversión por referencia.

```
if (aObject is Stock)
{
   Console.WriteLine (((Stock)a).SharesOwned);
}
```

.Net
Operador As

- Realiza un downcast y, en vez de arrojar una excepción, devuelve null si no pudo.

```
Asset a = new Asset();
Stock stock = a as Stock;       // stock is null; no exception thrown
if (stock != null) Console.WriteLine (stock.SharesOwned);
```

.Net
Value Types

- Los tipos más simples en .Net
- Contienen el valor y no una referencia al valor
- Las instancias de estos se guardan en un area de memoria de acceso mas veloz
- Se pueden crear Value Type por medio de Enums y Structs

- Constructor implicito:

```
- <tipo de dato> nombreVariable = valor inicial;
```

- *Net Framework provee los siguientes value types por default:*
  - *Numéricos (int, decimal, float, double, sbyte, short, long, ushort, uint, ulong, byte, double)*
  - *Alfabéticos (char)*
  - *Lógicos (bool)*

Value Types – ¿Cuándo usarlos?

- Representan un solo valor.
- Tienen como tamaño de instancia menos de 16 bytes.
- No cambiarán luego de la creación.
- No serán casteados a un reference type.

.Net
Enums

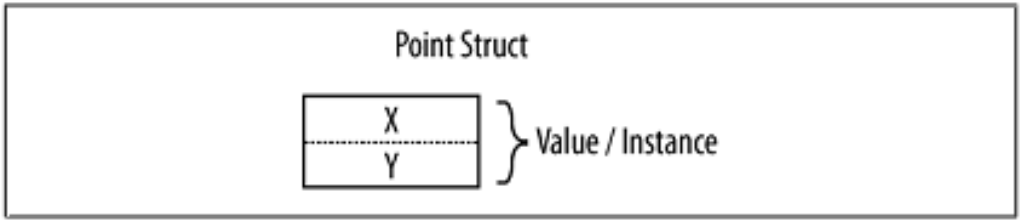
- Símbolos relacionados que contienen valores fijos.
- Utilizados para proveer una lista de opciones.
- Simplifican el desarrollo.
- Mejoran la claridad del código.

```
14      public enum Roles : int
15      {
16          Junior = 0,
17          SemiSenior = 1,
18          Senior = 2
19      };
```

.Net
Structs

- Encapsula pequeños grupos de variables.
- Eficientes.
- Representan un solo valor.
- 16 bytes máximo.
- Inmutables.
- No serán casteados a un reference type.

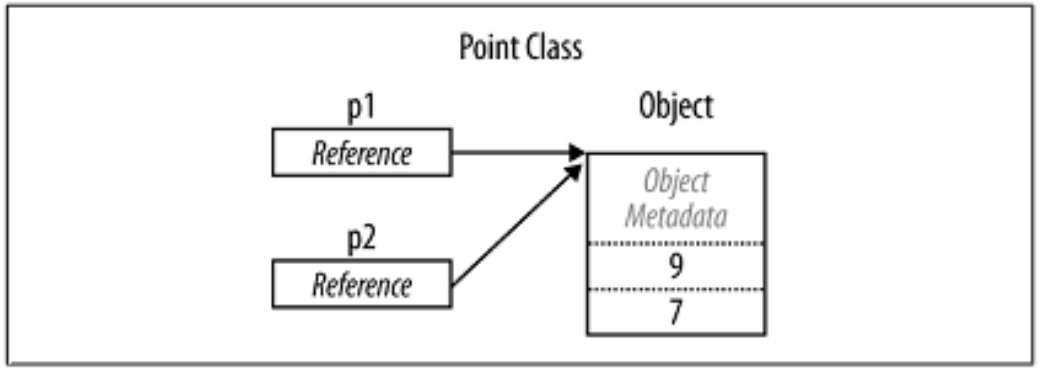
```
13      public struct Point
14      {
15          public int X, Y;
16      }
```



.Net
Reference Types

- Guardan la dirección del valor en el stack y el valor en el heap.
- Es decir, tiene un objeto y una referencia a éste.
- Muchas variables pueden tener la referencia al mismo objeto.
- Pueden no referenciar nada, o sea, ser “null”.

```
12      public class Point
13      {
14          public int X, Y;
15      }
```



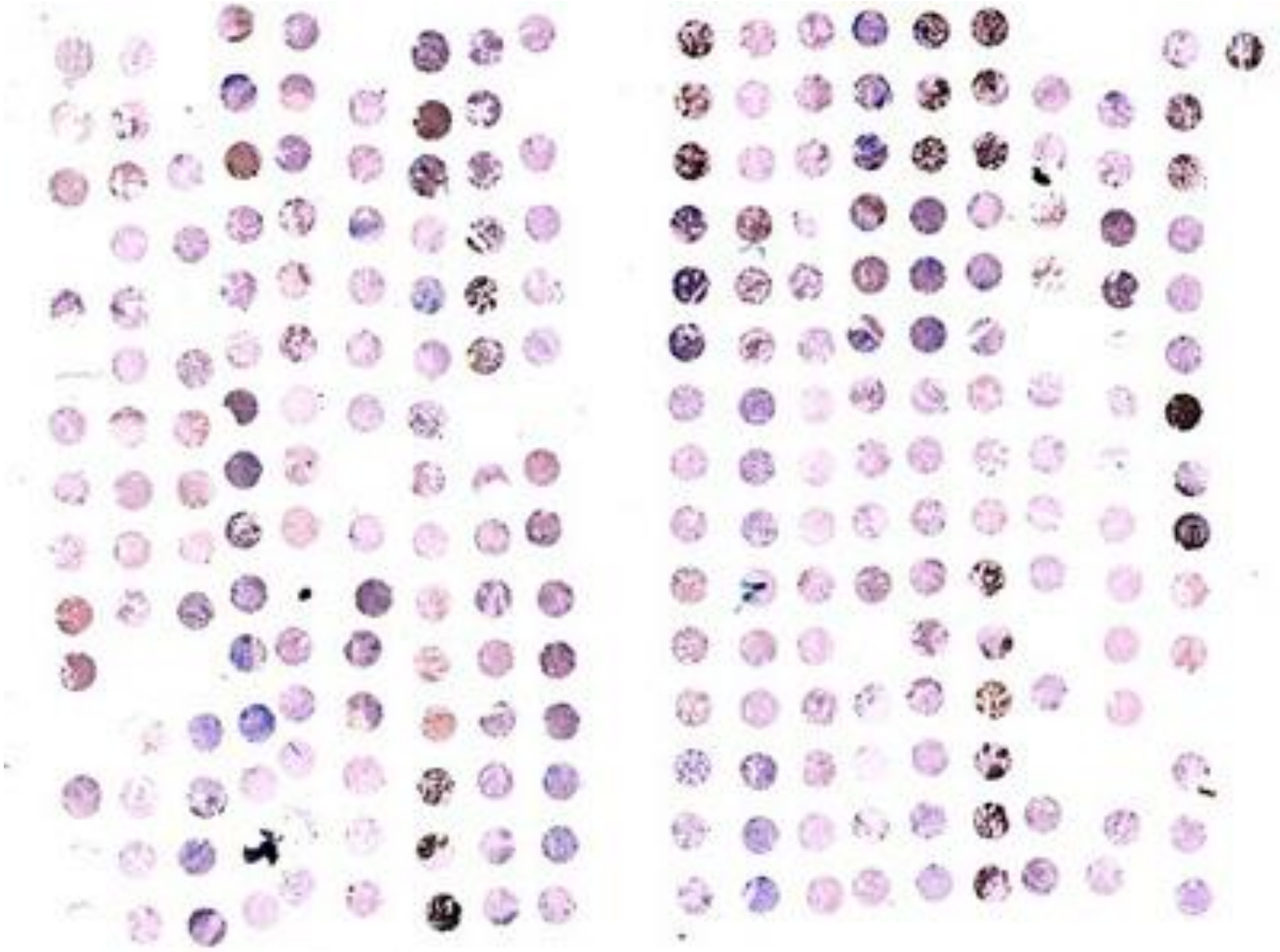
.Net
Boxing / Unboxing

- Boxing es la accion mediante la cual se convierte un value-type en un reference type
- Unboxing es la accion mediante la cual se convierte un reference type a un value-type

```
int x = 9;
object obj = x;            // Box the int

int y = (int) obj;          // Unbox the int
```

.Net
Collections



- Los tipos para representar colecciones pueden ser divididos en 3 categorías:

  - Interfaces para definir el comportamiento estándar de una colección.
  - Colecciones: listas para usar.
  - Clases base para escribir colecciones específicas en una aplicación.

```
public interface IEnumerator
{
    bool MoveNext();
    object Current { get; }
    void Reset();
}
```

```
public interface IEnumerable
{
    IEnumerator GetEnumerator();
}
```

.Net

ICollection, IList e IDictionary

- IEnumerable<T> (and IEnumerable): mínima funcionalidad (solo enumeración).
- ICollection<T> (and ICollection): funcionalidad media (ej: la propiedad Count).
- IList <T>/IDictionary <K,V>: gran funcionalidad (incluye “random access by index/key”).

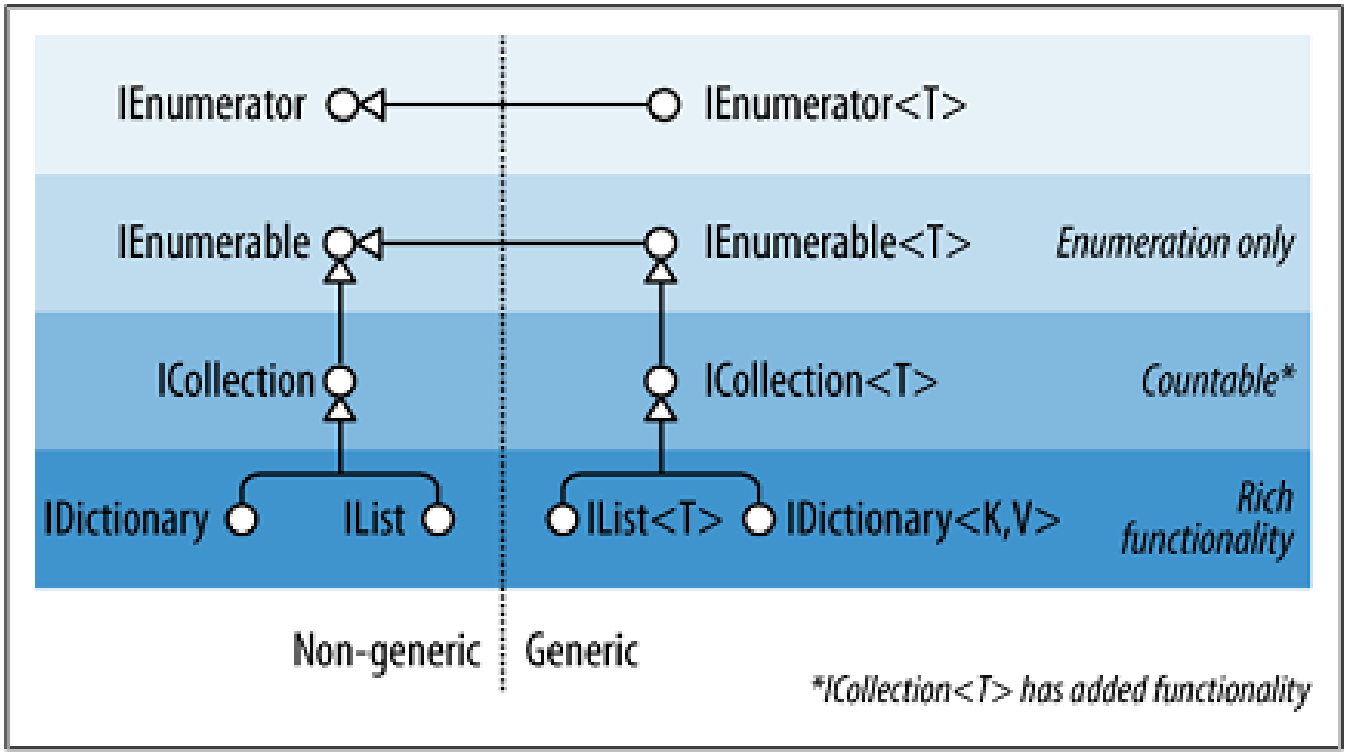
```
public interface ICollection<T> : IEnumerable<T>
{
    int Count { get; }
    bool Contains(T item);
    void CopyTo(T[] array, int arrayIndex);
    bool IsReadOnly { get; }
    void Add(T item);
    bool Remove(T item);
    void Clear();
}
```

.Net

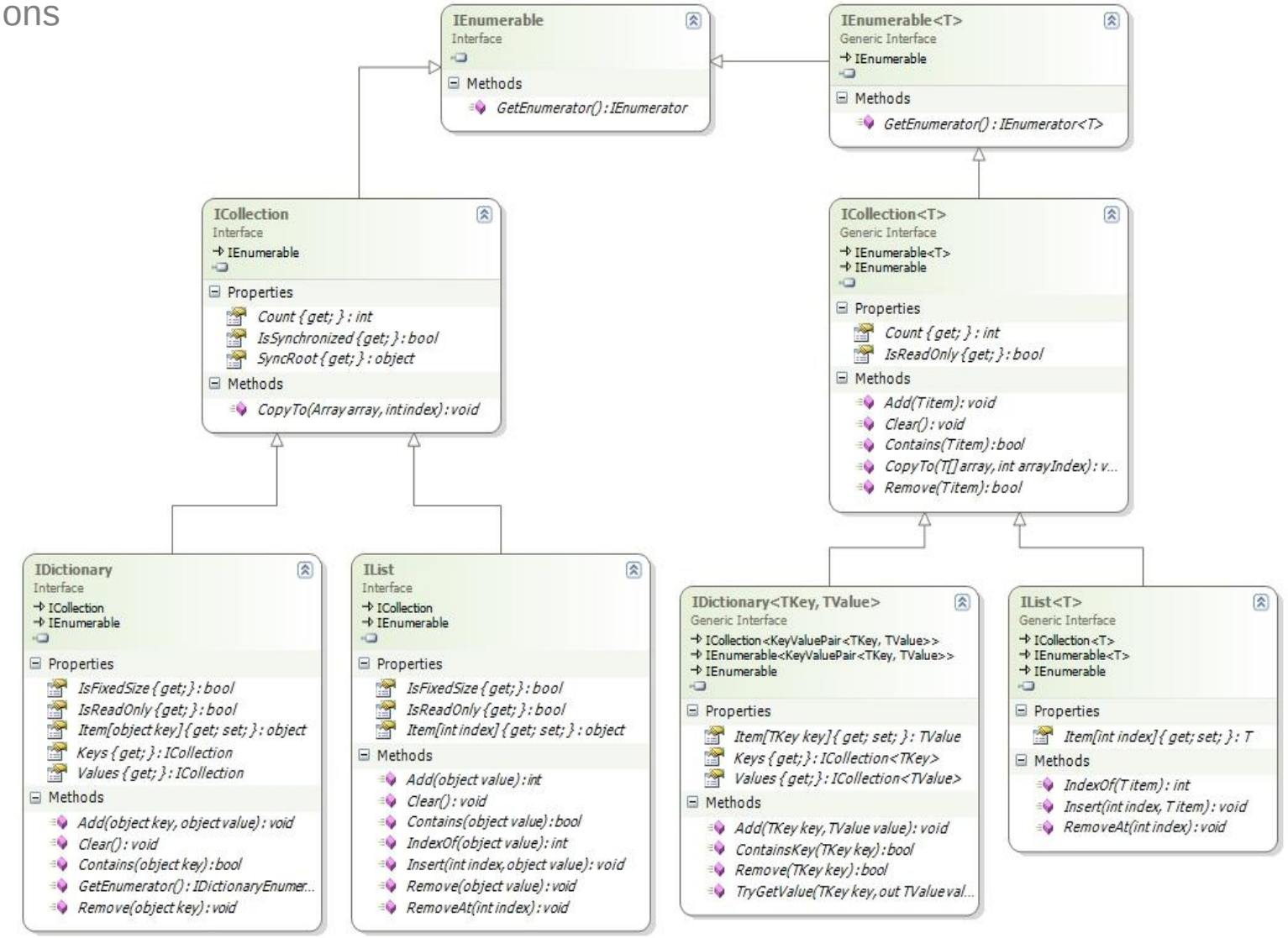
ICollection, IList e IDictionary

```
public interface IList<T> : ICollection<T>, IEnumerable<T>, IEnumerable
{
   T this[int index] { get; set; }
   int IndexOf(T item);
   void Insert(int index, T item);
   void RemoveAt(int index);
}
```

.Net
Enumeration



.Net
Collections



.Net
Array

- Es una clase.
- Implícita para todos los arrays simples o multidimensionales.
- Utilizada para implementar las colecciones más básicas y fundamentales dentro de .Net.
- C# provee sintaxis explicita para su declaración e instanciación.

```
int[] myArray = new int[] { 1, 2, 3 };

int[] myArray = { 1, 2, 3 };

int first = myArray[0];

int last = myArray[myArray.Length - 1];

int[,] twoD = { { 5, 6 }, { 8, 9 } };
```

Stack<T>

LinkedList<T>

Queue<T>

HashSet<T>

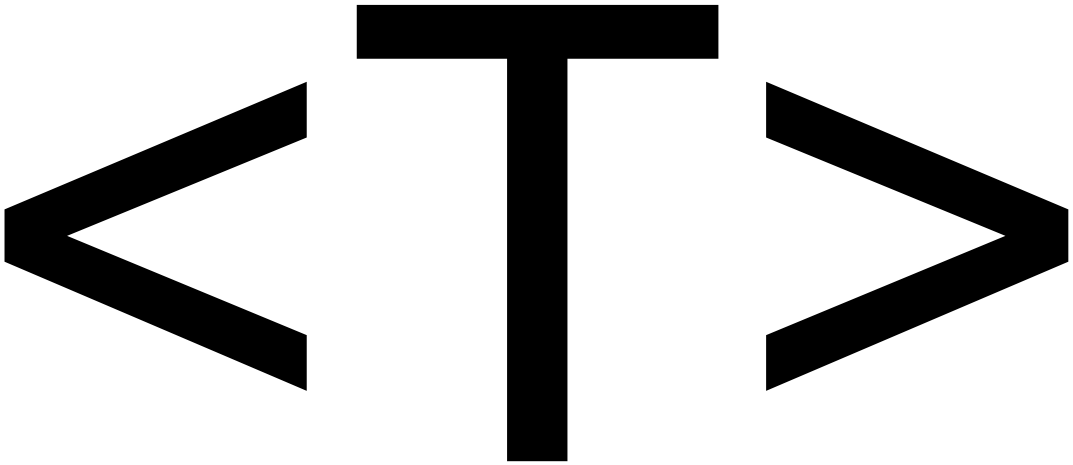
```
var letters = new HashSet<char> ("the quick brown fox");
 // the quickbrownfx
```

SortedSet<T>

```
var letters = new SortedSet<char> ("the quick brown fox");
//  bcefhiknoqrtuw
```

- Diccionarios: son colecciones con elementos KeyValuePair.

```
public struct KeyValuePair<TKey, TValue>
{
    public TKey Key { get; }
    public TValue Value { get; }
}
```


<T>

- Parámetro de tipo.
- Clases, interfaces o métodos, aplazan la especificación de los tipos hasta que un cliente los declara.
- Seguridad de tipo: evitan costo y riesgo de conversión de tipos.
- Rendimiento.
- Maximizan reutilización de código.

```
public class GenericList<T>
{
    void Add(T input) { }
}
```

```
// The .NET Framework 1.1 way to create a list:
System.Collections.ArrayList list1 = new System.Collections.ArrayList();
list1.Add(3);
list1.Add(105);
```

```
System.Collections.ArrayList list = new System.Collections.ArrayList();
list.Add(3);
list.Add("It is raining in Redmond.");
```

```
// The .NET Framework 2.0 way to create a list
List<int> list1 = new List<int>();

// No boxing, no casting:
list1.Add(3);

// Compile-time error:
list1.Add("It is raining in Redmond.");
```

- Denomine los parámetros de tipo genérico con nombres descriptivos, a menos que un nombre de una sola letra sea muy fácil de entender y un nombre descriptivo no agregue ningún valor.
- Considere el uso de T como nombre del parámetro de tipo para los tipos con un parámetro de tipo de una sola letra.
- Añada el prefijo "T" a los nombres de parámetros de tipo descriptivos.
- Considere indicar las restricciones de un parámetro de tipo en el nombre del parámetro.

```
public interface ISessionChannel<TSession> { /*...*/ }

public delegate TOutput Converter<TInput, TOutput>(TInput from);

public class List<T> { /*...*/ }
```

```
class Base { }
class Test<T, U>
    where U : class
    where T : Base, new() { }
```

```
where T : base-class   // Base class constraint
where T : interface    // Interface constraint
where T : class        // Reference-type constraint
where T : struct       // Value-type constraint (excludes Nullable types)
where T : new()        // Parameterless constructor constraint
where U : T            // Naked type constraint
```

Como regla general, cuantos más tipos se puedan parametrizar, más flexible y reutilizable será el código.

Sin embargo, un exceso de generalización puede producir código difícil de leer o comprender para otros programadores.

- Bloque try / catch / finally

```
try
{
  ... // exception may get thrown within execution of this block
}
catch (ExceptionA ex)
{
  ... // handle exception of type ExceptionA
}
catch (ExceptionB ex)
{
  ... // handle exception of type ExceptionB
}
finally
{
  ... // cleanup code
}
```

- Determina si hay un bloque try o catch para atrapar el error.
- Si es así, para la ejecución del programa al bloque catch.
- Cuando finaliza la ejecución del catch se continua con la ejecución fuera del bloque.
- Si no es así, la ejecución del programa salta al llamado del miembro que ocasiono el error.
- Y se vuelve a evaluar si existe un bloque catch para manejar el error.
- Si nunca lo encuentra, un mensaje de error se le muestra al usuario y se finaliza el programa.

.Net
Exceptions

- Un bloque try especifica un bloque de código en el cual si existe un error este podrá ser manejado.

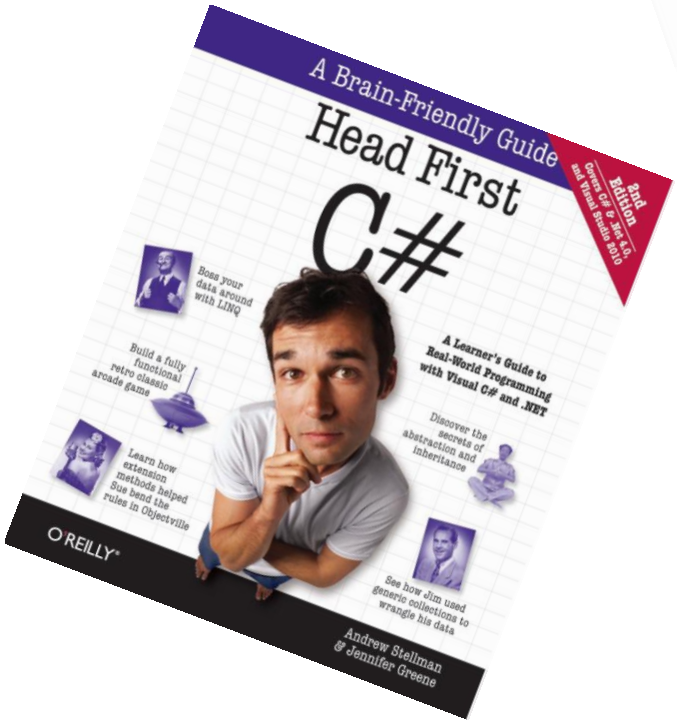
```
try
{
    byte b = byte.Parse(args[0]);
    Console.WriteLine(b);
}
catch (IndexOutOfRangeException ex)
{
    Console.WriteLine("Please provide at least one argument");
}
catch (FormatException ex)
{
    Console.WriteLine("That's not a number!");
}
catch (OverflowException ex)
{
    Console.WriteLine("You've given me more than a byte!");
}
catch (Exception ex)
{
    Console.WriteLine("General Error");
}
finally
{
    Console.WriteLine("Finally code");
}
```

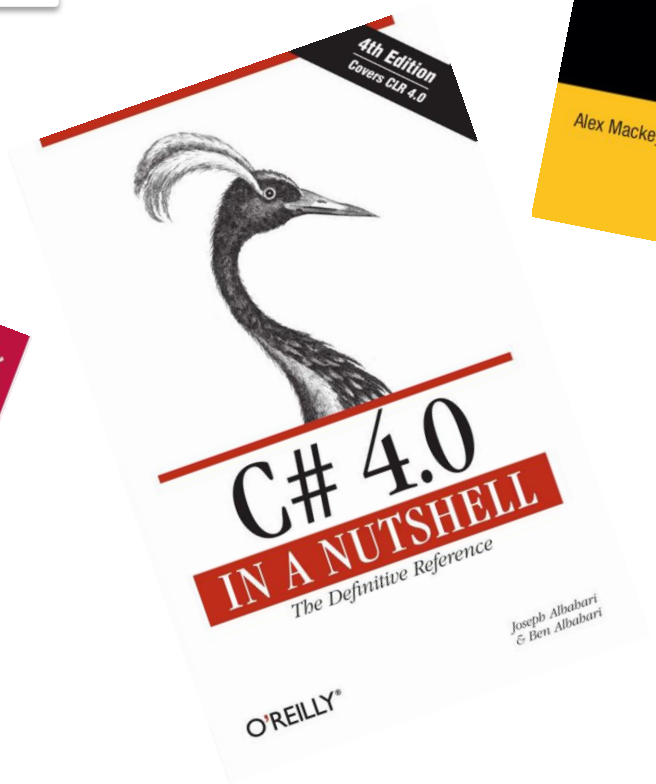
.Net

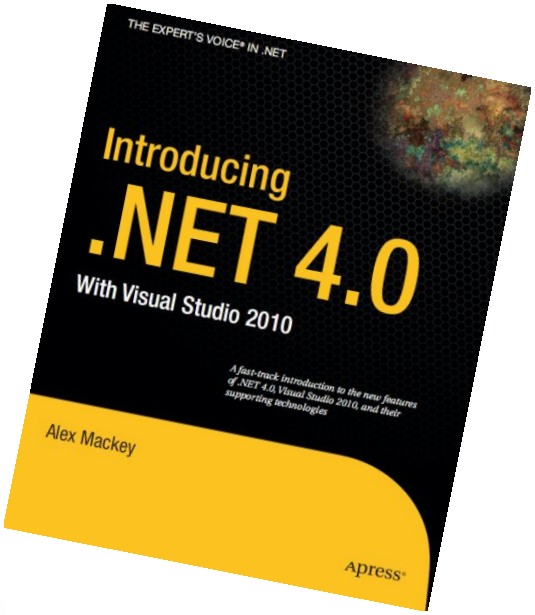
¿Consultas?

.Net
Bibliografia

msdn.microsoft.com







ARGENTINA
Arguibel 2860
Buenos Aires (C1426DKB)
tel: 54+11+4779 6400

BRASIL
Cardoso de Melo 1470 – 8, Vila Olimpia
San Pablo (04548004)
tel: 55+11+3045 2193

URUGUAY
Roque Graseras 857
Montevideo (11300)
tel: 598+2+7117879

USA
12105 Sundance Ct.
Reston (20194)
tel:+703 842 9455

www.hexacta.com

hexacta