Metode avansate de programare

Informatică Româna, 2020-2021, Curs 11-12

Generics, Collections, Delegates, Events, Lambda expressions, LINQ

Genericitate

- Incepand cu versiunea 2.0
- Spre deosebire de Java, in .NET genericitatea are suport nativ in IL (intermediate language) si in CLR.
- Prin urmare, limitarile de la Java generics dispar:
  - Se poate instantia parametru generic cu tipuri valoare
  - La executie se face diferenta intre:

```
List<Integer> and List<String>
```

  - Clasele de exceptii pot fi generice
  - Arrays pot avea elemente de tip generic

Metode generice

Java: `public <T> T foo (T x);` C#: `public T foo <T> (T x);`

```
static void Swap<T>(ref T a, ref T b)
{
            T temp = b;
            a = b;
            b = temp;
}
```

```
int x = 5, y = 10;
Swap(ref x, ref y); //without parameters type
Swap<int>(ref x, ref y); //with parameters type
```

- Metodele și declaratiile de tipuri sunt singurele constructii care pot introduce parametri generici.
- Proprietățile, indexatorii, câmpurile, operatorii, etc. nu pot să declare parametrii generici

```
class Vector<T>
{
    public T this[int index]
    {
        get { return data[index]; }
    }
}
```

Declararea parametrilor generici

- Parametrii generici pot fi introdusi in declaratii de clasa, structura, interfata, delegati si metode.

```
public struct AStruct<T>
{
    public T Value { get; }
}

class Dictionary<TKeyType, TValueType> {...}

public interface IComparable<T>
{
    int CompareTo(T other);
}
```

default() operator

- Returneaza valoarea implicita a tipului argument
- Daca T este un tip referinta `default(T)` va avea valoare null
- Daca T este un tip valoare `default(T)` va avea valoarea default specifica tipului corespunzator

```
static void init<T>(T[] array)
{
      for (int i = 0; i < array.Length; i++)
            array[i] = default(T);
}
```

Constrângeri

- Exista 3 tipuri de constrangeri:
  - *De derivare:* care indica compilatorului că parametrul de tip generic (T) este derivat dintr-un tip de bază.
  - *Constructor fara parametri:* se indica faptul ca (T) trebuie sa aiba un constructor fara parametri.
  - *Reference/Valoare:* T este class/struct
- Constrangerile pot fi applicate atat la nivel de tip cat si la nivel de metoda!

https://msdn.microsoft.com/en-us/library/d5x73970.aspx

Constrângeri

Constraint	Description
where T: struct	The type argument must be a value type. Any value type except Nullable can be specified. See Using Nullable Types for more information.
where T : class	The type argument must be a reference type; this applies also to any class, interface, delegate, or array type.
where T : new()	The type argument must have a public parameterless constructor. When used together with other constraints, the `new()` constraint must be specified last.
where T : <base class name>	The type argument must be or derive from the specified base class.
where T : <interface name>	The type argument must be or implement the specified interface. Multiple interface constraints can be specified. The constraining interface can also be generic.
where T : U	The type argument supplied for T must be or derive from the argument supplied for U.

Constrângeri

```
public class Employee {… }
public class GenericList<T> where T : Employee { … }
```

```
public interface IComparable<T>
{
    int CompareTo(T other);
}

static T Max<T>(T a, T b) where T : IComparable<T>
{
    return a.CompareTo(b) > 0 ? a : b;
}
```

```
class Stack<T>
{
    Stack<U> FilteredStack<U>() where U : T {...}
}
```

Generics and Inheritance

- Fie S un subtip al lui T
- Nu exista relatie de mostenire intre:

```
SomeGenericClass<S> and SomeGenericClass<T>
```

Înlocuirea Wildcards in C#

- Consideram urmatoare ierarhie de clase:
  - `interface Shape { }`
  - `class Circle : Shape {...}`
  - `class Rectangle : Shape {...}`
  - `IList<Circle> circles  IList<Shape> shapes`
- Care ar trebui sa fie signatura metodei **drawShapes** care primeste o lista de obiecte de tipul **Shape** si le deseneaza?

```
DrawShapes(IList <Shape> shapes) ??? List <?> shapes (Java)
```

  - aceasta metoda nu se poate apela pentru o lista de obiecte de tipul Circle

(`DrawShapes(IList <Circle> circles)`

Înlocuirea Wildcards in C#

- Solutia: folosirea unei clase auxiliare sau a unei metode generice cu constrangeri

```
class DrawHelper<T> where T : Shape
{
    public static void DrawShapes(IList<T> shapes);
}
```

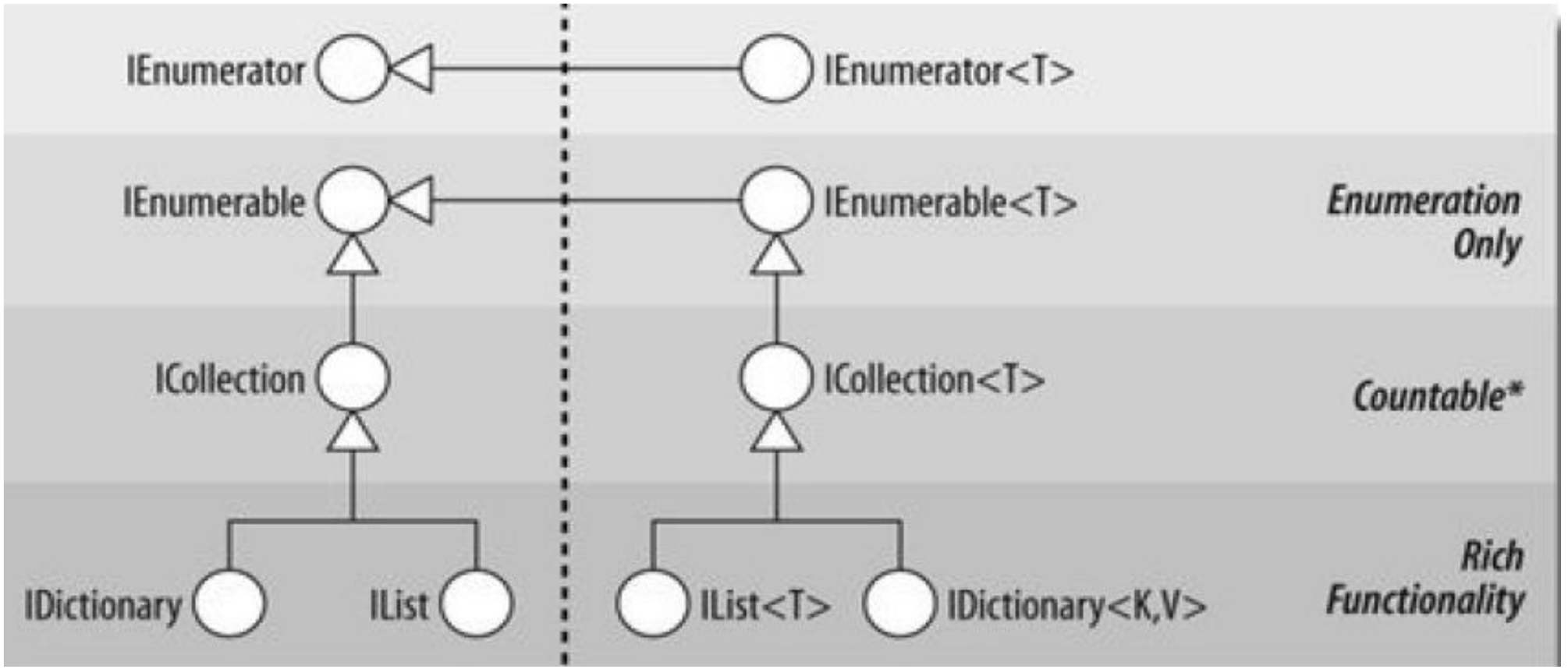
apelul metodei DrawShapes:

```
DrawHelper<Shape>.DrawShapes(listOfShapes);
DrawHelper<Circle>.DrawShapes(listOfCircles);
```

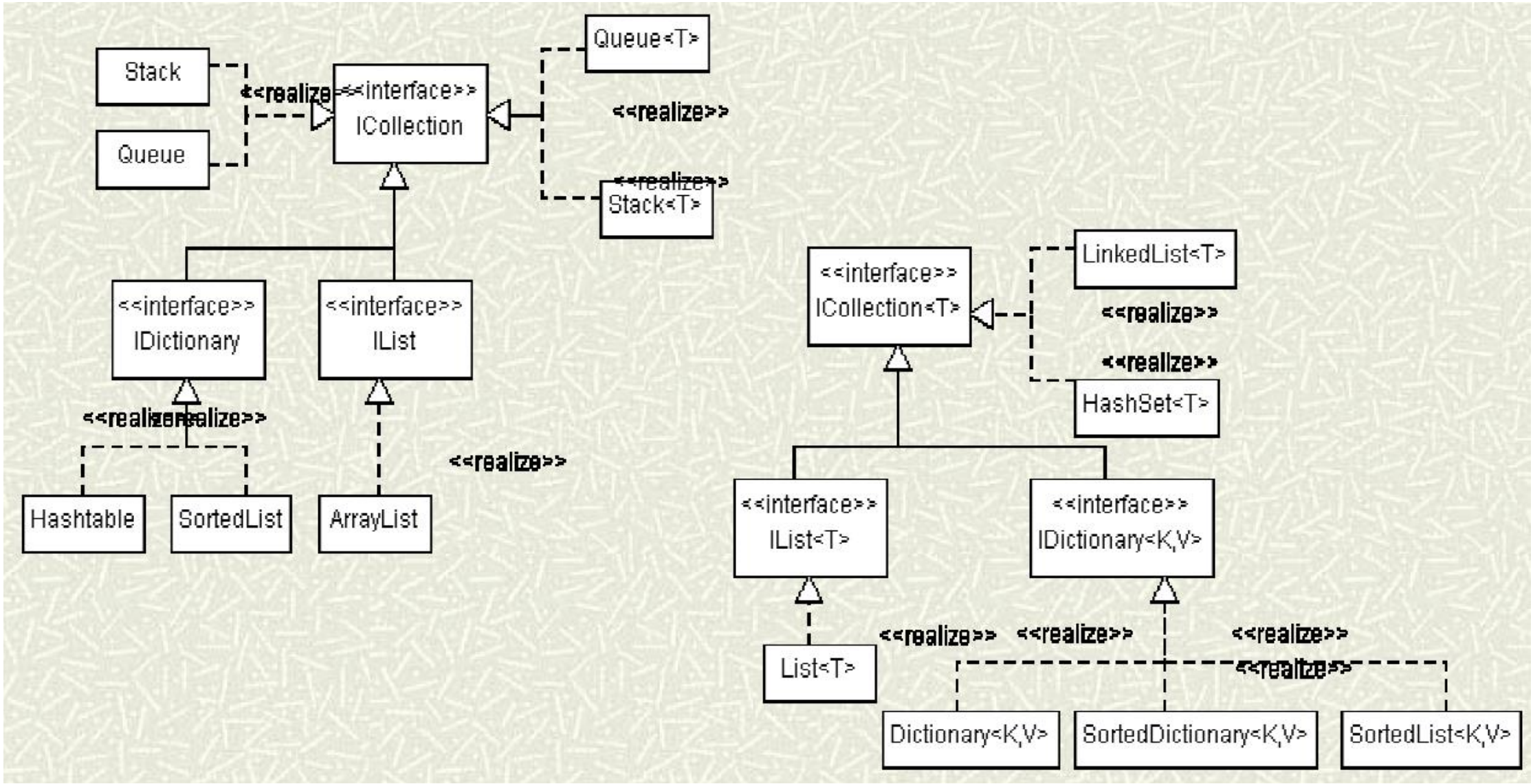
```
public static void DrawShapes<T>(IList<T> shapes) where T : Shape
{

}
```

Collections

- `System.Collections.Generic;`
- `System.Collections;`



Ierarhia structurilor de date



IEnumerable<T>

(Iterable<T> -Java)

```
interface IStack<T>: IEnumerable<T>
 {
     int Count { get; }
     void Push(T element);
     T Pop();
     T Peek();
     T[] GetAllStackElements();
 }

 class Stack<T> :  IStack<T>
 {
     private int capacity;
     private T[] elems;
     private int top;

     public Stack(int maxCapacity)
     {
         capacity = maxCapacity;
         elems = new T[capacity];
         top = -1;
         //initialize top with -1
     }
```

https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/covariance-contravariance/index

```
public interface IEnumerable<out T> : IEnumerable
```

```
public IEnumerator<T> GetEnumerator()
{
    return new MyIterator(this);
}

IEnumerator IEnumerable.GetEnumerator()
{
    return GetEnumerator();
}
```

IEnumerator<T> (Iterator<T>

```
public class MyIterator : IEnumerator<T>  //MyIterator – clasa interna in Stack
{
    private Stack<T> stack;
    int current;
    T[] elems;

    public MyIterator(Stack<T> stack)
    {
        this.stack = stack;
        current = stack.Count - 1;
        elems = stack.GetAllStackElements();
    }

    public T Current { get; }

    public bool MoveNext() { }
}
```

In C# clasele interne nu au acces la membri clasei outer!!! Din acest motiv MyIterator are o referinta la Stack! Vezi constructorul MyIterator(Stack<T> stack)

Delegates

- Problema: trebuie sa se execute o anumita actiune, dar nu se stie dinainte care anume, sau ce obiect va trebui efectiv utilizat.
- Un **delegat** este un tip referinta folosit pentru a incapsula o metoda cu un anumit antet
- Orice metoda care are acest antet poate fi legata la un anumit delegat.

Delegates (delegați)

- *Declarare:*

```
delegate <return type > DelegateName(< list of parameters >);
```

- *Example:*

```
delegate int ArithmeticOperation(int a, int b);
```

- Delegatii definiti de utilizator sunt subclase ale clasei System.Delegate. Acestea sunt automat generate de compilator si nu pot fi create explicit de catre utilizator.

Exemplu

```
class Student :IComparable<Student>
{
        public int StudentID { get; set; }
        public String StudentName { get; set; }
        public int Age { get; set; }

        public override string ToString()
        {
            return StudentID + " " + StudentName + " " + Age;
        }

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

Problema

- Consideram entitatea Student, definita anterior.
- Definiti urmatoarele metode:
  - `TeenAgerStudent`
  - `AdultStudent`
  - `RetiredStudent`
  - `StudentPredicate (delegate)`
- Creati o lista de student si retineti: doar studentii adulti, apoi cei pensionati, apoi adolescentii
- Folositi multicast delegate

Exemplu

```
delegate bool StudentPredicate(Student s);

public static bool TeenAgerStudent(Student s)
{
        return s.Age > 12 && s.Age < 20;
}

public static bool AdultStudent(Student s)
{
        return s.Age >= 20;
}

Student s = new Student() { StudentID = 1, StudentName = "Cris", Age = 19 };

StudentPredicate testIfTeenAger = new StudentPredicate(TeenAgerStudent);
Console.WriteLine(testIfTeenAger(s));

StudentPredicate testIfAdult = new StudentPredicate(AdultStudent);
Console.WriteLine(testIfAdult(s));

//metode anonime
StudentPredicate testIfRetired = delegate (Student stud) { return stud.Age > 60; };
Console.WriteLine(testIfRetired(s));
//lambda
StudentPredicate testIfChild =  stud => stud.Age <=12 ;
Console.WriteLine(testIfChild(s));
```

Multicast Delegates

```
delegate bool StudentPredicate(Student s);


StudentPredicate testStudent = AdultStudent; // sau new StudentPredicate(AdultStudent);
Console.WriteLine(testStudent(s));

//multicast delegate
testStudent += TeenAgerStudent;
Console.WriteLine(testStudent(s));


public static bool TeenAgerStudent(Student s)
{
        return s.Age > 12 && s.Age < 20;
}

public static bool AdultStudent(Student s)
{
        return s.Age >= 20;
}
```

Generics and Delegates filter students

```
delegate bool Predicate<T>(T entity);
```

```
public List<T> Filter<T>(List<T> list, Predicate<T> test)
{
    List<T> res = new List<T>();
    foreach (var entity in list)
        if (test(entity))
            res.Add(entity);
    res.Sort();
    return res;
}
```

```
public List<Student> filterTeenAgerStudents()
{
     return Filter(studentList, TeenAgerStudent);
}
```

```
public static bool TeenAgerStudent(Student s)
{
    return s.Age > 12 && s.Age < 20;
}
```

```
studentList=InitStudentList();

List<Student> list=filterTeenAgerStudents();
Console.WriteLine("TeenAger Students Ascending by Name");
foreach (var s in list)
    Console.WriteLine(s);
```

Generics and Delegates Filter&Sorter

```
delegate bool Predicate<T>(T entity);


public List<T> FilterAndSorter<T>(List<T> list, Predicate<T> test, Comparison<T> comp)
{
    List<T> res = Filter(list, test);
    res.Sort(comp);
    return res;
}


public List<Student> FilterTeenAgerStudentsDescByAge()
{
        return FilterAndSorter(studentList, TeenAgerStudent, (x, y) => { return -(x.Age - y.Age); });
}


studentList=InitStudentList();

list = FilterTeenAgerStudentsDescByAge();
Console.WriteLine("TeenAger Students Descending by Age");
foreach (var s in list)
        Console.WriteLine(s);
```

Evenimente

- Events are a language feature that formalizes the Publisher/Subscriber (Observer) pattern.
- An *event* is a wrapper for a delegate that exposes just the subset of delegate features required for the publisher/subscriber model.
- The main purpose of events is to prevent subscribers from interfering with each other.
- To declare an event member, the **event** keyword is put in front of a delegate member.

Evenimente

1. Definirea unei metode delegate publica

```
public delegate void DelegateEvent(Object sender, EventArgs args);
```

2. Definirea unei clase care lanseaza evenimentul (Publisher)

```
class Publisher
{
    public event DelegateEvent eventName;
            …
    someMethod(…)
    {
        EventArgsSubClass args =
              new EventArgsSubClass(some data);

        //code that raises an event
        eventName(this, args);
        //or eventName(this, null);
    }
}
```

Primul parametru este sursa evenimentului (cine lanseaza evenimentul), iar cel de-al doilea pastreaza anumite informatii care sunt transmise consumatorului de eveniment (event handler) sau metodei care trateaza evenimentul.

Evenimente

3. Definirea unei clase care trateaza aparitia evenimentului (observer/consumer)

```
class Observer
{
    //the methods that matches the delegate signature
    public void OnEventName(Object sender, EventArgs args)
    {
        //event handling code
    }
    …
}
//create the publisher(subject)
Publisher pub = new Publisher(…);
//create the observers
Observer obs1 = new Observer(…);
Observer obs2 = new Observer(…);
//subscribe the observers to the event
pub.eventName += new DelegateEvent(obs1.OnEventName);
pub.eventName += new DelegateEvent(obs2.OnEventName);
pub.someMethod(…); //explicit call of the method that raises the event
```

Extensions methods

```
public static class MyExtensionMethods
{
    public static bool IsNumeric(this string s)
    {
        float output;
        return float.TryParse(s, out output);
    }
}

string test = "4";
if (test.IsNumeric())
      Console.WriteLine("Yes");
else
      Console.WriteLine("No");
```

LINQ

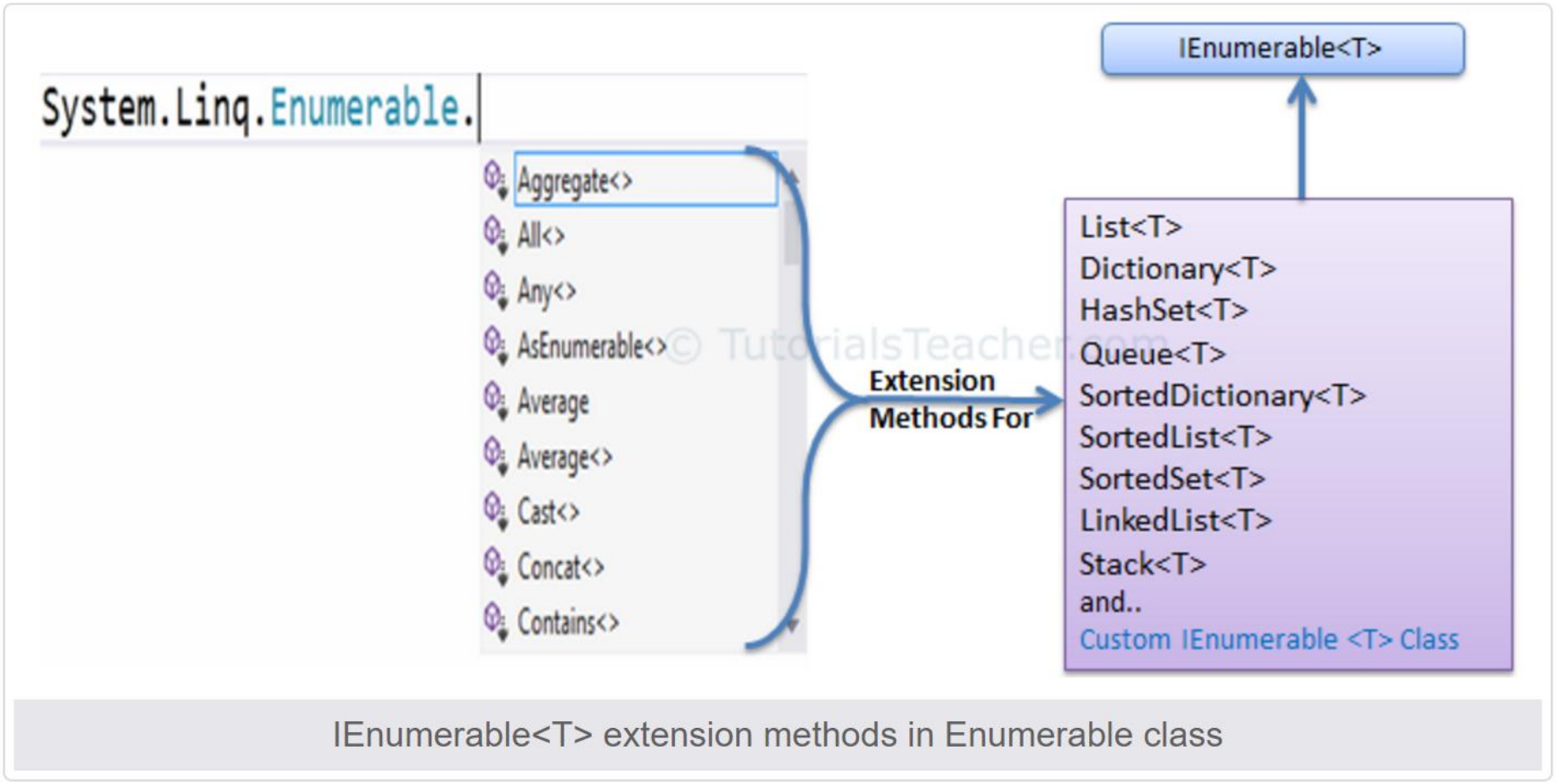
```
List<Student> studentList = new List<Student>();
studentList.Add(new Student() { StudentID = 1, StudentName = "John", Age = 18 })
studentList.Add(new Student() { StudentID = 2, StudentName = "Steve", Age = 21 });
studentList.Add(new Student() { StudentID = 3, StudentName = "Bill", Age = 25 });
studentList.Add(new Student() { StudentID = 4, StudentName = "Ram", Age = 20 });
studentList.Add(new Student() { StudentID = 5, StudentName = "Ron", Age = 31 });
studentList.Add(new Student() { StudentID = 6, StudentName = "Chris", Age = 17 });
studentList.Add(new Student() { StudentID = 7, StudentName = "Rob", Age = 19 });
```

Interogarea unei liste de studenti…..

```
List<Student> teenAgerStudents = studentList.Where(s => s.Age > 12 && s.Age <
20).ToList<Student>();

List<Student> studentsStartWith = studentList.Where(s =>
s.StudentName.StartsWith("R")).ToList<Student>();
```

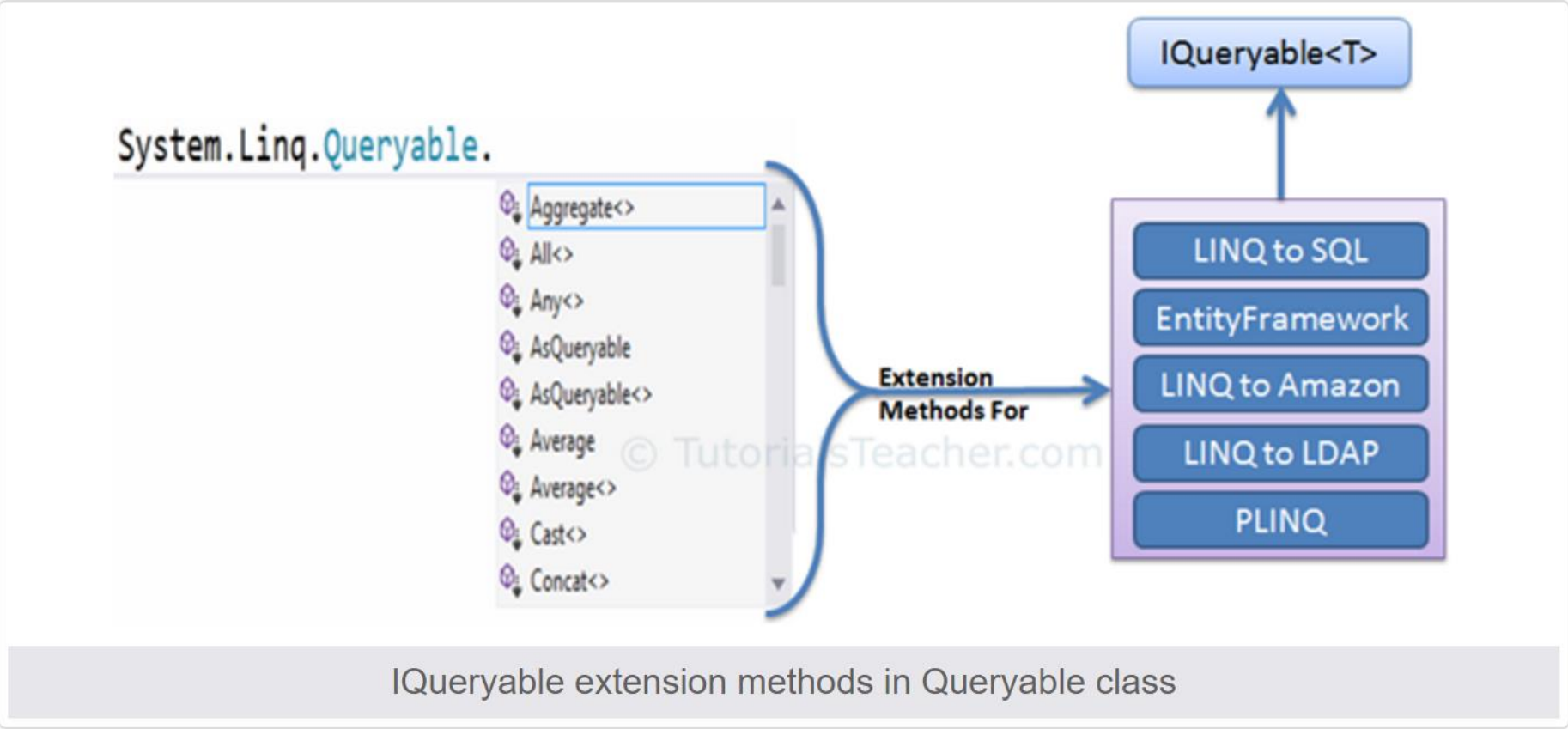
LINQ API

- LINQ is nothing but the collection of extension methods for classes that implements IEnumerable and IQueryable interface. (namespace *System.Linq*)



IEnumerable<T> extension methods in Enumerable class

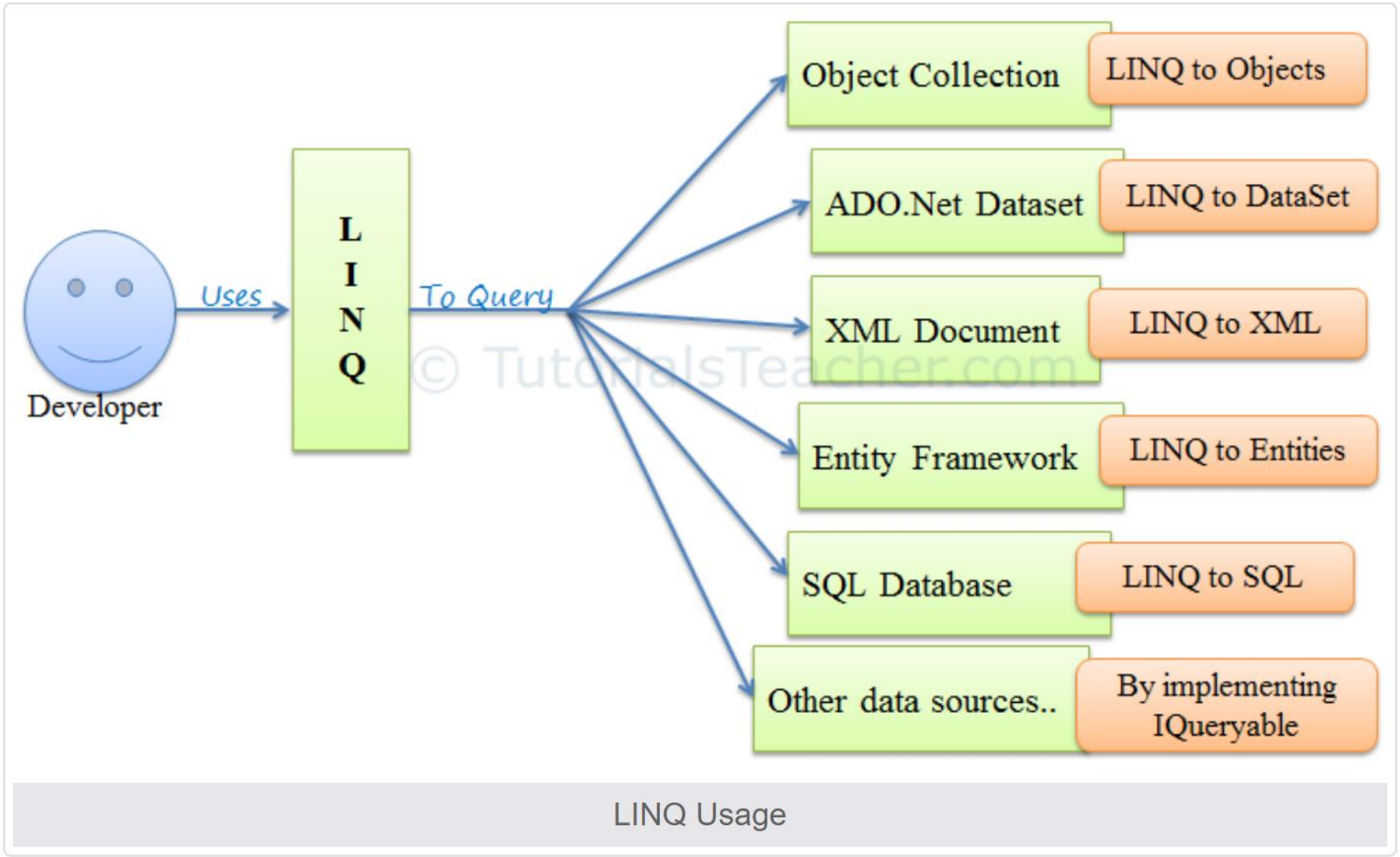
LINQ Queryable



IQueryable extension methods in Queryable class

For example, Entity Framework api implements IQueryable<T> interface to support LINQ queries with underlaying database like SQL Server.

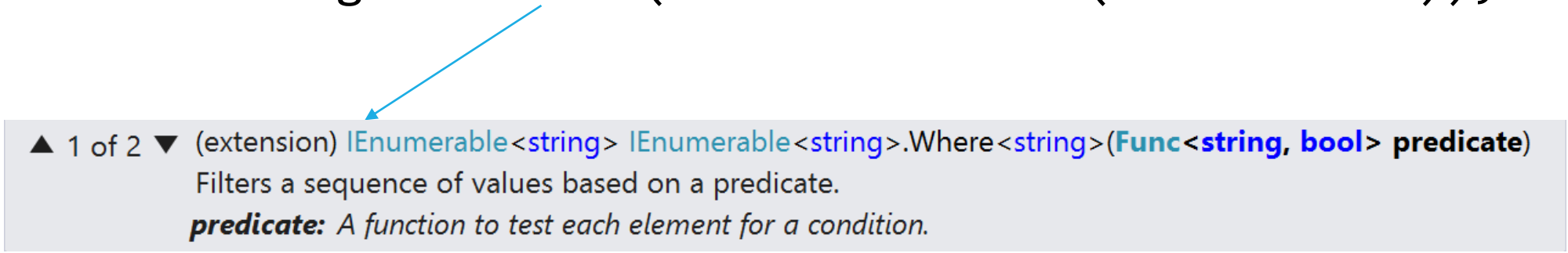
LINQ



LINQ Usage

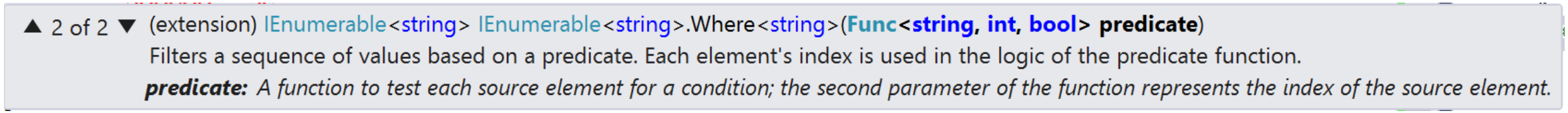
LINQ Method Syntax

```
IList<string> stringList = new List<string>() {
    "C# Tutorials",
    "VB.NET Tutorials",
    "Learn C++",
    "MVC Tutorials" ,
    "Java"
};
var result = stringList.Where(s => s.Contains("Tutorials"));
```



LINQ Method Syntax

```
public static void Linq5()
{
    string[] digits = { "zero", "one", "two", "three", "four", "five", "six", "seven",
                        "eight", "nine" };

    var shortDigits = digits.Where((digit, index) => digit.Length < index);
    //digits.Where()
    Console.WriteLine("Short digits:");
    foreach (var d in shortDigits)
    {
        Console.WriteLine("The word {0} is shorter than its value.", d);
    }
}
```



LINQ Query Syntax

- Query syntax is similar to SQL (Structured Query Language) for the database.

```
// string collection
IList<string> stringList = new List<string>() {
    "C# Tutorials",
    "VB.NET Tutorials",
    "Learn C++",
    "MVC Tutorials" ,
    "Java"
};

// LINQ Query Syntax
var result = from s in stringList
             where s.Contains("Tutorials")
             select s;
```

LINQ Query Syntax

```
public void Linq2()
{
    List<Product> products = GetProductList();

    var soldOutProducts =
        from p in products
        where p.UnitsInStock == 0
        select p;

    Console.WriteLine("Sold out products:");
    foreach (var product in soldOutProducts)
    {
        Console.WriteLine("{0} is sold out!", product.ProductName);
    }
}
```

LINQ Language Integrated Query

- **Standardized way of querying multiple data sources:** The same LINQ syntax can be used to query multiple data sources.
- **Less coding:** It reduces the amount of code to be written as compared with a more traditional approach.
- **Readable code:** LINQ makes the code more readable so other developers can easily understand and maintain it.
- **Compile time safety of queries:** It provides type checking of objects at compile time.
- **IntelliSense Support:** LINQ provides IntelliSense for generic collections.

LINQ

- https://code.msdn.microsoft.com/101-LINQ-Samples-3fb9811b

C# Version History

- http://www.tutorialsteacher.com/csharp/csharp-version-history