



Блок 4. Расширенный C#

4.4 — Методы расширения

План занятия

- Методы расширения
- Анонимные типы
- LINQ
- Отложенные вычисления



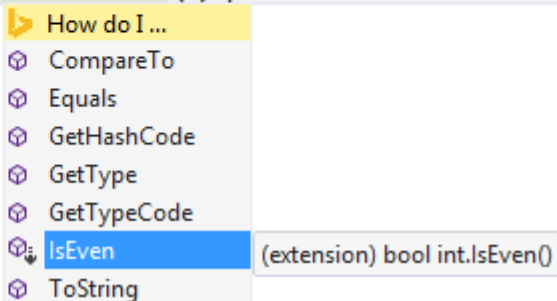
Методы расширения

- Вспомогательные методы, которые могут быть добавлены к объекту любого типа и даже интерфейса;
- Реализуются как статические методы, принимающие в качестве первого параметра объект целевого типа;
- Методы расширения обязаны располагаться в статических классах;
- Для превращения обычного метода в метод расширения применяется ключевое слово `this`;
- Методы расширения можно вызывать как обычные статические методы.

Расширяющие методы

```
public static class DataHelper
{
    public static bool IsEven(this int param)
    {
        return (param % 2 == 0);
    }
}

internal class Program
{
    private static void Main(string[] args)
    {
        int x = 4;
        bool isEven = x.IsEven();
    }
}
```



The image shows a Visual Studio code editor with a C# file. The code defines a static class `DataHelper` with a static method `IsEven` that takes an `int` parameter and returns a `bool`. The `IsEven` method is highlighted in the code. Below it, in the `Program` class, the `Main` method is shown. Inside `Main`, a variable `x` of type `int` is assigned the value 4. Then, a variable `isEven` of type `bool` is assigned the result of `x.IsEven()`. The `IsEven` method call is highlighted, and an IntelliSense dropdown menu is visible. The menu lists several methods: `CompareTo`, `Equals`, `GetHashCode`, `GetType`, `GetTypeCode`, `IsEven`, and `ToString`. The `IsEven` method is selected, and a tooltip shows the signature `(extension) bool int.IsEven()`.

Расширение коллекций

```
public static List<int> GetAllEven(this IEnumerable<int> collection)
{
    var evenItems = new List<int>();
    foreach (var item in collection)
    {
        if (item.IsEven())
        {
            evenItems.Add(item);
        }
    }
    return evenItems;
}
```

```
int[] arr = new int[] { 4, 2, 7, 4, 23, 6, 9 };
var evenItems = arr.GetAllEven();
```

Расширение коллекций

```
public static List<int> GetByCondition(this IEnumerable<int> collection,
    Predicate<int> condition)
{
    var items = new List<int>();
    foreach (var item in collection)
    {
        if (condition(item))
        {
            items.Add(item);
        }
    }
    return items;
}
```

Также вместо Predicate<int> можно использовать Func<int, bool>



Демонстрация

Проблемы при работе с большими объёмами данных

- Возраст самого старшего сотрудника
- Список ФИО всех сотрудников отдела X
- Список имён, фамилий и фото новых сотрудников

```
public class Employee
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public string Patronymic { get; set; }
    public DateTime BirthDate { get; set; }
    public int Age { get; private set; }
    public byte[] Photo { get; set; }
    public byte[] SmallPhoto { get; private set; }
    public string JobTitle { get; set; }
    public string Department { get; set; }
    public DateTime StartDate { get; set; }
    public int Standing { get; private set; }
    public decimal Salary { get; set; }
}
```


Анонимные типы

- Объявление и использование объекта анонимного типа:

```
var x = new
{
    a = 3,
    b = 4.81,
    c = "string data",
};

Console.WriteLine("a = {0}\tb = {1}\tc = {2}", x.a, x.b, x.c);
Console.WriteLine(x.GetType());
```

```
a = 3    b = 4.81    c = string data
<>f__AnonymousType0`3[System.Int32,System.Double,System.String]
```

Ограничения анонимных типов

- Анонимный тип не может быть предком или потомком какого-либо класса.
- Анонимный тип не может реализовать интерфейс.
- Анонимные типы нельзя использовать в сигнатурах методов (как на вход, так и на выход)/свойств и т.п.
- Коллекцию или массив анонимного типа объявить нельзя.

Коллекция анонимного типа

```
List<object> GetEmployees(List<Employee> lst, string department)
{
    var res = new List<object>();
    foreach (var item in lst)
    {
        if (item.Department == department)
        {
            res.Add(new
            {
                FirstName = item.FirstName,
                LastName = item.LastName
            });
        }
    }
    return res;
}
```

- LINQ (**L**anguage **I**Ntegrated **Q**ueries) — язык запросов к наборам данных;
- Реализован в виде методов, расширяющих коллекции (как правило, через `IEnumerable<T>`);
- Для работы LINQ необходимо подключить через **using** пространство имён `System.Linq`;
- Результатом каждого запроса является либо объект, либо его перечисление (`IEnumerable<T>`);
- Существует две формы записи:
 - Специальный LINQ-синтаксис;
 - Standard Query Operators (обычные методы).

<http://msdn.microsoft.com/en-us/library/bb397926.aspx>

Сравнение классического алгоритма и LINQ

```
static int[] GetPositive(int[] array)
{
    var lst = new List<int>();
    foreach (var item in array)
    {
        if (item > 0)
        {
            lst.Add(item);
        }
    }
    return lst.ToArray();
}
```

```
static int[] GetPositive(int[] array)
{
    var lst = from item in array
               where item > 0
               select item;
    return lst.ToArray();
}
```

Шаблон from-where-select

```
// получение данных
int[] numbers = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 0 };

// формирование запроса
var lowNums = from n in numbers
              where n <= 5
              select n;

// выполнение запроса
foreach (var x in lowNums)
{
    Console.Write(x.ToString() + " ");
}
```

Шаблон from-where-select: объекты

```
public class Student
{
    public int Id { get; set; }
    public string Name { get; set; }

    public Student(int id, string name)
    {
        Id = id;
        Name = name;
    }
}
```

Шаблон from-where-select: объекты

```
var source = new List<Student>
{
    new Student(1, "Иванов"),
    new Student(8, "Петров"),
    new Student(6, "Сидоров"),
    new Student(3, "Ткачёв"),
    new Student(9, "Смирнов"),
    new Student(2, "Николаев"),
};

var students = from st in source
               where st.Id <= 5
               select st;

foreach (var st in students)
{
    Console.WriteLine("{0} {1}", st.Id.ToString(), st.Name);
}
```


Шаблон from-where-select: новые объекты

```
var source = new List<Student>
{
    new Student(1, "Иванов"),
    new Student(8, "Петров"),
    new Student(6, "Сидоров"),
    new Student(3, "Ткачёв"),
    new Student(9, "Смирнов"),
    new Student(2, "Николаев"),
};

var students = from st in source
               where st.Id <= 5
               select new Student(st.Id + 1000, st.Name);

foreach (var st in students)
{
    Console.WriteLine("{0} {1}", st.Id.ToString(), st.Name);
}
```

Перечисление анонимного типа

```
var res = from employee in employees
           where employee.Department == "Отдел 1"
           select new
           {
               employee.FirstName,
               employee.LastName,
           };

foreach (var item in res)
{
    Console.WriteLine("{0} {1}", item.FirstName, item.LastName);
}
```

В данном случае объект **res** имеет тип **IEnumerable<T>** (перечисление), где типом элемента является анонимный тип.

Инструкция orderby

- Упорядочивание элементов

```
var students = from st in source
                where st.Id <= 5
                orderby st.Id
                select st;
```

```
1 Иванов
2 Николаев
3 Ткачёв
```

```
var students = from st in source
                where st.Id <= 5
                orderby st.Id descending
                select st;
```

```
3 Ткачёв
2 Николаев
1 Иванов
```

Список файлов, упорядоченных по размеру

```
var dir = new DirectoryInfo("C:\\");  
var files = from file in dir.GetFiles()  
            orderby file.Length  
            select file;  
  
foreach (var file in files)  
{  
    Console.WriteLine("{0} - {1}", file.Name, file.Length);  
}
```

```
BOOTNXT - 1  
bootmgr - 398156  
swapfile.sys - 268435456  
hiberfil.sys - 3261009920
```

Группировка (инструкция group by)

```
var students = new List<Student>
{
    new Student { Name = "Иванов", GroupId = 1 },
    new Student { Name = "Петров", GroupId = 2 },
    new Student { Name = "Сидоров", GroupId = 2 },
    new Student { Name = "Смирнов", GroupId = 2 },
    new Student { Name = "Ткачёв", GroupId = 3 },
    new Student { Name = "Николаев", GroupId = 3 },
    new Student { Name = "Токарев", GroupId = 1 },
    new Student { Name = "Оганесян", GroupId = 2 },
};

var query = from st in students
            group st by st.GroupId;

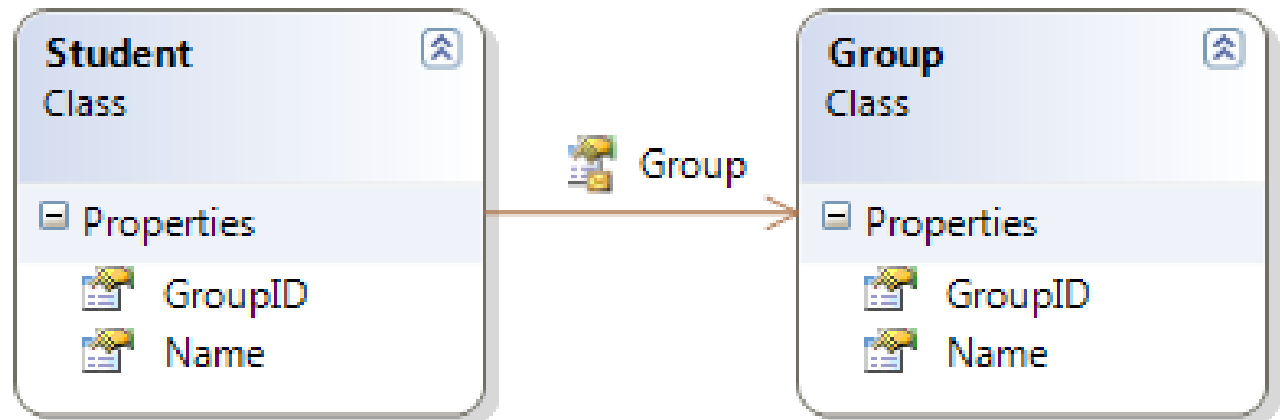
foreach (var group in query)
{
    Console.WriteLine("Группа {0}: ", group.Key);
    foreach (var st in group)
    {
        Console.Write("{0} ", st.Name);
    }
    Console.WriteLine();
}
```

```
Группа 1:
Иванов Токарев
Группа 2:
Петров Сидоров Смирнов Оганесян
Группа 3:
Ткачёв Николаев
```

Соединение по ключу (инструкция join)

```
public class Student
{
    public string Name { get; set; }
    public int GroupID { get; set; }
}

public class Group
{
    public int GroupID { get; set; }
    public string Name { get; set; }
}
```



Соединение по ключу (инструкция join)

```
// Формируем источники данных
var groups = new List<Group>
{
    new Group { GroupID = 1, Name = "Начальный уровень" },
    new Group { GroupID = 2, Name = "Базовый уровень" },
    new Group { GroupID = 3, Name = "Продвинутый уровень" },
};

var students = new List<Student>
{
    new Student { Name = "Иванов", GroupID = 1 },
    new Student { Name = "Петров", GroupID = 2 },
    new Student { Name = "Сидоров", GroupID = 2 },
    new Student { Name = "Смирнов", GroupID = 2 },
    new Student { Name = "Ткачёв", GroupID = 3 },
    new Student { Name = "Николаев", GroupID = 3 },
    new Student { Name = "Токарев", GroupID = 1 },
    new Student { Name = "Оганесян", GroupID = 2 },
};
```

Внутреннее соединение

```
// inner join
var query = from st in students
             join gr in groups
             on st.GroupID equals gr.GroupID
             select new
             {
                 Student = st.Name,
                 Group = gr.Name,
             };

foreach (var item in query)
{
    Console.WriteLine("{0} - {1}", item.Student, item.Group);
}
```

Иванов - Начальный уровень
Петров - Базовый уровень
Сидоров - Базовый уровень
Смирнов - Базовый уровень
Ткачёв - Продвинутый уровень
Николаев - Продвинутый уровень
Токарев - Начальный уровень
Оганесян - Базовый уровень

Групповое соединение

```
// group join
var query = from gr in groups
             orderby gr.GroupID
             join st in students
             on gr.GroupID equals st.GroupID
             into newGroup
             select new
             {
                 Group = gr.Name,
                 Students = from x in newGroup
                           orderby x.Name
                           select x,
             };

foreach (var item in query)
{
    Console.WriteLine("Группа: {0}: ", item.Group);
    foreach (var st in item.Students)
    {
        Console.WriteLine("{0} ", st.Name);
    }
    Console.WriteLine();
}
```

Группа: Начальный уровень: Иванов Токарев
Группа: Базовый уровень: Оганесян Петров Сидоров Смирнов
Группа: Продвинутый уровень: Николаев Ткачёв

Standard Query Operators

- Набор методов расширения, лежащих в основе LINQ;
- Обладают большими гибкостью и возможностями, чем синтаксис LINQ.

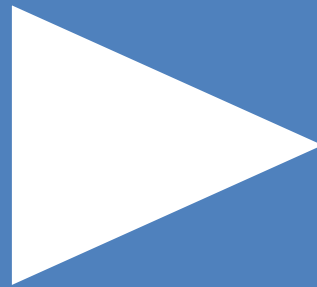
```
int[] arr = { 1, 2, -3, -4, 5, 6, -7, 8, 9, 0 };  
  
// Синтаксис LINQ  
var positive1 = from i in arr  
                where i > 0  
                select i;  
  
// Standard Query Operators  
var positive2 = arr.Where(i => i > 0);
```

Standard Query Operators

- Where() — выборка;
- Select() — проекция;
- SelectMany() — проекция со слиянием;
- Cast() — приведение типа;
- OrderBy() — упорядочивание по возрастанию;
- OrderByDescending() — упорядочивание по убыванию;
- OfType() — приведение типа с пропуском неподходящих;
- Count() — объём выборки;
- Any() — проверка непустоты выборки;
- Group() — группировка;
- Join() — соединение;
- Union() — слияние;
- Except() — исключение;
- ...

- Некоторые LINQ-методы имеют реализации, совмещённые с Where:
 - `.Where(condition).Count()` → `.Count(condition)`
 - `.Where(condition).Any()` → `.Any(condition)`
 - `.Where(condition).Sum()` → `.Sum(condition)`
 - ...

```
var positiveCount1 = arr.Count(n => n.IsEven());  
var positiveCount2 = arr.Count(IntExtension.IsEven);
```



Демонстрация

Отложенные (Lazy) вычисления

- Результатом работы большинства LINQ инструкций является механизм получения результата, а не сам результат.
- При изменении исходных данных результат выполнения также изменится.

```
int[] arr = { -5, 7, -4, 6, 2, 8, 9, -1 };  
  
var res = from n in arr  
          where n > 0  
          select n;  
  
Console.WriteLine(res.Count()); // 5  
  
arr[0] = 1;  
Console.WriteLine(res.Count()); // 6
```

Формирование дерева вычислений

```
var employees = new Employee[10];  
  
var ed = employees.Where(p => p.Department == "D1")  
    .Select(p => new  
    {  
        p.FirstName,  
        p.LastName,  
        p.Age  
    });  
  
var sed = ed.Where(p => p.Age > 21);  
  
foreach (var item in sed)  
{  
    Console.WriteLine("{0} {1}", item.FirstName,  
        item.LastName);  
}
```

Реализация отложенных вычислений

- Ядром отложенных вычислений в C# является специальный оператор `yield return`.

```
public static IEnumerable<int> GetPositive(int[] arr)
{
    foreach (var item in arr)
    {
        if (item > 0)
        {
            yield return item;
        }
    }
}
```


Оператор **yield return**

- Оператор **yield return** позволяет вернуть очередной элемент перечисления по запросу;
- Досрочное прекращение работы метода реализуется путём вызова **yield break**;
- Оператор **yield return** совместим только с интерфейсами-итераторами:
 - IEnumerator
 - IEnumerator<T>
 - IEnumerable
 - IEnumerable<T>;
- Метод не может содержать в себе **return** и **yield return** одновременно.

Спасибо за внимание!

Контактная информация:

Дмитрий Верескун

Инструктор

EPAM Systems, Inc.

Адрес: Саратов, Рахова, 181

Email: Dmitry_Vereskun@epam.com

<http://www.epam.com>