

ПЛАТФОРМА MICROSOFT.NET И ЯЗЫК ПРОГРАММИРОВАНИЯ

C#



Урок №9

Делегаты, события. LINQ

Содержание

1. Делегаты.....	4
Понятие делегата	4
Базовые классы для делегатов	4
<i>System.Delegate</i>	4
<i>System.MulticastDelegate</i>	5
Синтаксис объявления делегата	6
Цели и задачи делегатов.....	8
Вызов нескольких методов через делегат (<i>multicasting</i>).....	12
Создание generic делегатов	16
2. События.....	28
Понятие события.....	28
Синтаксис объявления события	28

Необходимость и особенности применения событий	32
Применение события для многоадресатного делегата	39
Использование событийных средств доступа	42
3. Анонимные методы.....	47
4. Лямбда выражения	51
Реализация тела метода в виде выражения.....	51
5. Extension методы	60
6. LINQ to Object.....	63
Роль LINQ	63
Исследование операций запросов LINQ	64
Возврат результата запроса LINQ. Анонимные типы.....	76
Применение запросов LINQ к объектам коллекций	83
Домашнее задание	90

1. Делегаты

Понятие делегата

Прежде чем дать определение делегата хотелось бы напомнить Вам об указателях на функции, используемых в C++. Как Вы помните, указателю на функцию можно присвоить адрес функции, тип возврата и параметры которой совпадают с типом возврата и параметрами самого указателя. Но существует в C++ еще и некая конструкция, называемая массивом указателей на функцию, которая позволяет хранить несколько адресов однотипных функций при выполнении вышеописанных условий. Так вот делегаты в C# являются аналогом массивов указателей на функцию в C++, но на более высоком уровне.

Делегат — это типобезопасный ссылочный **тип**, позволяющий хранить ссылки на методы заданной сигнатуры и с их помощью вызывать эти методы.

Базовые классы для делегатов

System.Delegate

Базовым для всех типов делегатов является класс `System.Delegate` он обеспечивает основную функциональность при работе с делегатами. Рассмотрим основные методы и свойства этого класса:

- `object Clone()` — создает неполную копию вызывающего делегата.
- `Combine(Delegate, Delegate)` — сцепляет списки вызовов заданных делегатов.

- `CreateDelegate(Type, MethodInfo)` — создает делегат указанного типа.
- `object DynamicInvoke(params object[] args)` — динамически вызывает метод, представленный текущим делегатом.
- `Delegate[] GetInvocationList()` — возвращает список вызовов делегата.
- `int GetHashCode()` — возвращает хэш-код текущего делегата.
- `Delegate Remove(Delegate, Delegate)` — удаляет последнее вхождение списка вызовов делегата из списка вызовов другого делегата.
- `bool operator == (Delegate, Delegate)` — перегруженный оператор «строгое равенство», возвращает `true`, если делегаты равны, иначе — `false`.
- `bool operator != (Delegate, Delegate)` — перегруженный оператор «не равно», возвращает `true`, если делегаты неравны, иначе — `false`.
- `MethodInfo Method { get; }` — свойство, которое возвращает метод, представленный текущим делегатом.
- `object Target { get; }` — свойство, которое возвращает экземпляр класса, метод которого вызывает текущий делегат.

System.MulticastDelegate

На самом деле все делегаты наследуются от класса `System.MulticastDelegate`, который в свою очередь наследуется от `System.Delegate`. Этот класс обеспечивает многоадресность делегатов, то есть способность сохранять ссылки на произвольное количество методов.

Многоадресность обеспечивается внутренним списком, в котором хранятся ссылки на методы, соответствующие заданной сигнатуре делегата.

Синтаксис объявления делегата

Как Вы уже поняли из вышесказанного — у делегатов существуют базовые классы, однако прямое наследование от этих классов запрещено. Попытка создания класса-наследника от кого-то из них приведет к ошибке на этапе компиляции (Рисунок 1.1).

```
class MyDelegate : MulticastDelegate
{
}
```

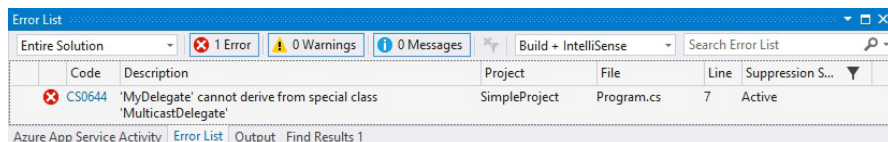


Рисунок 1.1. Ошибка: класс не может быть производным от MulticastDelegate

Для создания делегатов используется ключевое слово `delegate` и общий синтаксис объявления делегата выглядит следующим образом:

```
[модификатор доступа] delegate тип_данных ИмяДелегата
    (параметры);
```

Где:

- модификатор доступа — любой из существующих в C# модификаторов доступа (необязательная часть);
- `delegate` — ключевое слово для объявления делегата;

- тип данных и параметры — определяют сигнатуру методов, ссылки на которые могут храниться в данном делегате (параметры могут отсутствовать).

Приведем примеры объявления делегатов:

```
public delegate int IntDelegate(double d);
```

В данном случае объявлен общедоступный делегат по имени `IntDelegate`, позволяющий хранить ссылки на методы, принимающие в качестве параметра вещественное и возвращающие целочисленное значение.

```
delegate void VoidDelegate(int i);
```

Этот пример демонстрирует объявление закрытого делегата `VoidDelegate`, в котором могут храниться ссылки на методы, принимающие целочисленное значение и ничего не возвращающие.

Где следует размещать объявление делегатов? На самом деле при объявлении делегата компилятор генерирует класс, производный от `System.MulticastDelegate`, из этого следует, что объявленный делегат автоматически становится **классом**. Поэтому делегаты можно прописывать в виде поля другого класса (вложенный класс) или же в блоке `namespace` в виде отдельного класса, тем самым доступ к делегату из вызывающего кода будет различный. Решение о месте объявления делегата Вам следует принимать самостоятельно, исходя из логики Вашего приложения.

Цели и задачи делегатов

Используя один и тот же делегат, можно осуществлять вызов различных методов, при этом не имеет значения какие это методы экземплярные или статические, главное чтобы сигнатура методов совпадала с сигнатурой делегата, при этом методы определяются не на этапе компиляции, а на этапе выполнения. То есть на момент создания программы Вы не знаете, какой конкретно метод вызовет пользователь при выполнении Вашей программы, но Вы можете предоставить ему эту возможность в виде методов определенной сигнатуры.

Благодаря этой своей особенности делегаты широко применяются в различных конструкциях языка C#. Делегаты:

- являются основой для событий (раздел 2 текущего урока);
- являются основой для анонимных методов и лямбда-выражений (разделы 3 и 4);
- используются при определении методов обратного вызова;
- могут быть вызваны как в синхронном, так и в асинхронном режиме, то есть в другом потоке одновременно с каким-либо кодом (будет рассмотрено в последующих курсах).

В качестве примера использования делегатов приведем класс `Calculator`, который содержит методы, позволяющие выполнять элементарные арифметические операции. Эти методы принимают два параметра типа `double`

и возвращают значение типа `double`. В дополнении к классу объявим делегат, тип данных и параметры которого совпадают с сигнатурой методов класса `Calculator`. Возможный результат работы программы представлен на рисунке 1.2.

```
using System;
using static System.Console;

namespace SimpleProject
{
    public delegate double CalcDelegate(double x, double y);
    public class Calculator
    {
        public double Add(double x, double y)
        {
            return x + y;
        }
        public static double Sub(double x, double y)
        {
            return x - y;
        }
        public double Mult(double x, double y)
        {
            return x * y;
        }
        public double Div(double x, double y)
        {
            if (y != 0)
            {
                return x / y;
            }
            throw new DivideByZeroException();
        }
    }

    class Program
    {
```

```

static void Main(string[] args)
{
    Calculator calc = new Calculator();
    Write("Enter an expression: ");
    string expression = ReadLine();
    char sign = ' ';
    // определения знака арифметического действия
    foreach (char item in expression)
    {
        if (item == '+' || item == '-' || item ==
            '*' || item == '/')
        {
            sign = item;
            break;
        }
    }
    try
    {
        // получение значений операндов
        string[] numbers = expression.Split(sign);
        CalcDelegate del = null;
        switch (sign)
        {
            case '+':
                del = new CalcDelegate(calc.Add);
                break;
            case '-':
                del = new CalcDelegate(Calculator.
                    Sub);
                break;
            case '*':
                del = calc.Mult; // групповое
                                // преобразование методов
                break;
            case '/':
                del = calc.Div;
                break;
        }
    }
}

```

```

        default:
            throw new
                InvalidOperationException();
    }
    WriteLine($"Result: {del(double.
        Parse(numbers[0]),
        double.Parse(numbers[1]))}");
}
catch (Exception ex)
{
    WriteLine(ex.Message);
}
}
}
}

```

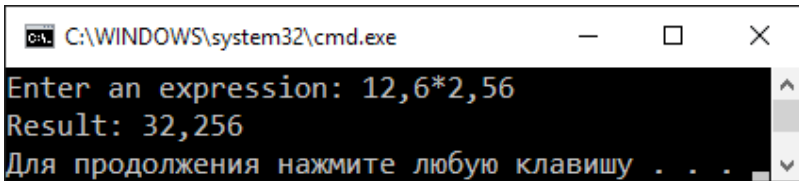


Рисунок 1.2. Пример использования делегата

Следует обратить Ваше внимание, что при создании объекта типа делегат, в конструктор передается **имя метода**, сигнатура которого соответствует сигнатуре делегата, независимо от того является этот метод экземплярным или статическим.

Также в предыдущем примере показан более простой способ инициализации делегата, при котором указывается только название метода с требуемой сигнатурой, без явного вызова конструктора делегата. Такая возможность называется **групповым преобразованием методов**.

Независимо от способа инициализации делегата, попытка указать в качестве параметра имя метода с неправильной сигнатурой приведет к ошибке на этапе компиляции (Рисунок 1.3).

```
using static System.Console;

namespace SimpleProject
{
    public delegate double CalcDelegate(double x, double y);
    class Program
    {
        static void Mult(double x, double y)
        {
            WriteLine(x * y);
        }
        static void Main(string[] args)
        {
            CalcDelegate del = Program.Mult; // Error
        }
    }
}
```

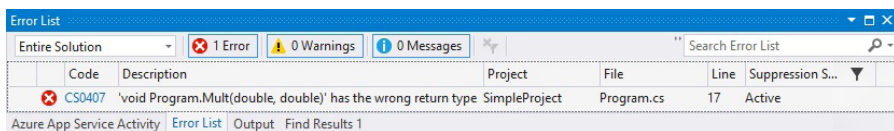


Рисунок 1.3. Ошибка: неправильный тип возвращаемого значения

Вызов нескольких методов через делегат (*multicasting*)

В предыдущем разделе при помощи делегата вызывался один определенный метод, что не полностью раскрывает

все возможности использования делегатов. Ведь по своей природе делегат может содержать множество ссылок на методы определенной сигнатуры, тем самым в делегате можно сформировать список методов, которые будут вызываться автоматически при вызове самого делегата, эта возможность делегатов называется многоадресатной передачей (*multicasting*).

Такая способность делегата обеспечивается благодаря наличию перегруженных операторов `+` и `-` для добавления в список и удаления из списка вызовов указанной ссылки на метод, соответственно, которые, как правило, применяются в сокращенной форме `+=` и `-=`. Использование этих операторов приводит к вызову методов `Combine(Delegate, Delegate)` и `Remove(Delegate, Delegate)` класса `System.Delegate`, Вы можете легко в этом убедиться, посмотрев CIL-код Вашего приложения при помощи утилиты `ildasm.exe` (Рисунок 1.4).

```
using System;

namespace SimpleProject
{
    public delegate double CalcDelegate(double x, double y);

    // класс Calculator остался прежним

    class Program
    {
        static void Main(string[] args)
        {
            Calculator calc = new Calculator();

            CalcDelegate delAll = null;
```

```

CalcDelegate delDiv = calc.Div;

delAll += delDiv; // добавления в список вызовов
delAll -= delDiv; // удаления из списка вызовов
    }
}
}

```

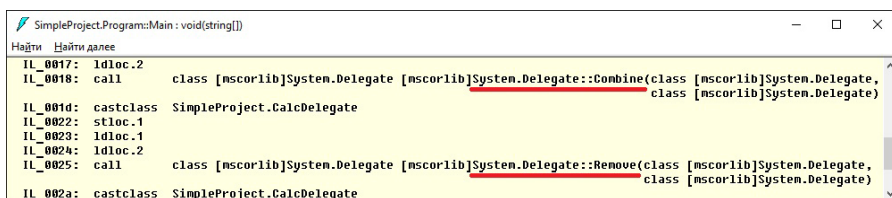


Рисунок 1.4. Перегрузка операторов + и – в делегатах

Использование многоадресатной передачи традиционно и более наглядно демонстрируется на примере делегата возвращающего тип `void`. Однако мы отойдем от сложившегося стереотипа и продемонстрируем multicasting, используя методы класса из предыдущего раздела, только внесем для этого необходимые изменения.

Если Вы сформируете цепочку вызовов из методов, которые возвращают тип, отличный от `void` (в нашем случае `double`), то вызвав эти методы через вызов делегата, Вы получите результат, который возвращает последний метод из этого списка вызовов (в этом Вы можете убедиться самостоятельно). Для того чтобы получить результат работы всех методов списка, необходимо использовать метод `GetInvocationList()`, который возвращает массив всех методов, ссылки на которые содержатся в текущем делегате (Рисунок 1.5).

```

using System;
using static System.Console;

namespace SimpleProject
{
    public delegate double CalcDelegate(double x, double y);

    // класс Calculator остался прежним

    class Program
    {
        static void Main(string[] args)
        {
            Calculator calc = new Calculator();

            CalcDelegate delAll = calc.Add; // групповое
                                           // преобразование методов
            delAll += Calculator.Sub;
            delAll += calc.Mult;
            delAll += calc.Div;

            foreach (CalcDelegate item in delAll.
                GetInvocationList()) // массив делегатов
            {
                try
                {
                    // ВЫЗОВ
                    WriteLine($"Result: {item(5.7, 3.2)}");
                }
                catch (Exception ex)
                {
                    WriteLine(ex.Message);
                }
            }
        }
    }
}

```

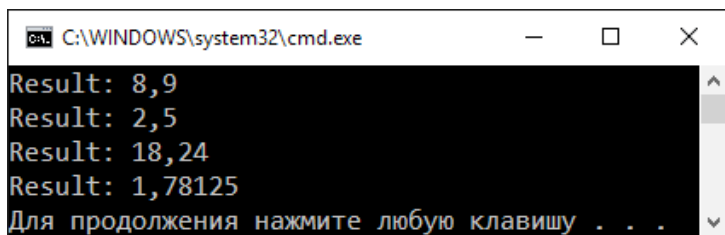


Рисунок 1.5. Вызов нескольких методов через делегат

Создание generic делегатов

В уроке №8 упоминалось о возможности создания обобщенных делегатов, в этом разделе мы рассмотрим этот вопрос более подробно.

Обобщенные делегаты позволяют безопасно вызывать любые методы, соответствующие обобщенной форме делегата, которая задается при его объявлении. Общая форма записи generic делегата выглядит следующим образом:

```
[модификатор доступа] delegate тип_возврата
    ИмяДелегата<типы_параметров> (параметры);
```

Под типами параметров подразумеваются любые типы данных, которые соответствуют типам параметрам методов при их вызове.

Пример использования обобщенных делегатов продолжит тему вычислений, но будет значительно проще предыдущего. Мы создали обобщенный делегат и класс с тремя методами, которые позволяют получить сумму символов, целых и вещественных чисел. Обращаем Ваше внимание на явное приведение результата в методе сложения двух символов к типу `char`, что необходимо из-за неявного приведения `char` к `int` (Рисунок 1.6).


```

using static System.Console;

namespace SimpleProject
{
    public delegate T AddDelegate <T>(T x, T y);
    public class ExampleClass
    {
        public int AddInt(int x, int y)
        {
            return x + y;
        }
        public double AddDouble(double x, double y)
        {
            return x + y;
        }
        public static char AddChar(char x, char y)
        {
            return (char)(x + y);
        }
    }
    class Program
    {
        static void Main(string[] args)
        {
            ExampleClass example = new ExampleClass();
            AddDelegate<int> delInt = example.AddInt;
            WriteLine($"The sum of integers: {delInt(8, 6)}");
            AddDelegate<double> delDouble =
                example.AddDouble;
            WriteLine($"The sum of real numbers:
                {delDouble(45.67, 62.81)}");
            AddDelegate<char> delChar = ExampleClass.AddChar;
            WriteLine($"The sum characters:
                {delChar('S', 'h')}");
        }
    }
}

```

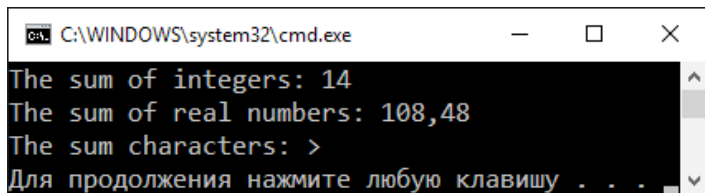


Рисунок 1.6. Использование обобщенного делегата

Как Вы заметили, в данном примере вместо создания трех делегатов для вызова каждого метода достаточно было создать один обобщенный делегат.

В пространства имён `System` существует ряд стандартных обобщенных делегатов, рассмотрим некоторые из них.

Обобщенный делегат `Action<T>` обеспечивает вызов методов, которые не возвращают значение и могут принимать до 16 параметров. Данный делегат необходим при вызове различных методов стандартных классов пространства имён `System`. Одним из таких классов является коллекция `List<T>`, метод `ForEach()` которой обеспечивает вызов, указанного в качестве параметра, метода для каждого элемента списка. Сигнатура вызываемого метода соответствует делегату `Action<T>`. В следующем примере при помощи метода `ForEach()` мы вызываем метод `FullName()` для каждого студента из списка, получая тем самым его фамилию и имя (Рисунок 1.7).

```
using System;
using System.Collections.Generic;
using static System.Console;

namespace SimpleProject
{
```

```

class Student
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public DateTime BirthDate { get; set; }
}
class Program
{
    static void FullName(Student student)
    {
        WriteLine($" {student.LastName}\t{student.
            FirstName}");
    }
    static void Main(string[] args)
    {
        List<Student> group = new List<Student> {
            new Student {
                FirstName = "John",
                LastName = "Miller",
                BirthDate = new DateTime(1997,3,12)
            },
            new Student {
                FirstName = "Candice",
                LastName = "Leman",
                BirthDate = new DateTime(1998,7,22)
            },
            new Student {
                FirstName = "Joey",
                LastName = "Finch",
                BirthDate = new DateTime(1996,11,30)
            },
            new Student {
                FirstName = "Nicole",
                LastName = "Taylor",
                BirthDate = new DateTime(1996,5,10)
            }
        };
    }
}

```

```

        WriteLine("List of students:");
        group.ForEach(FullName);
    }
}

```

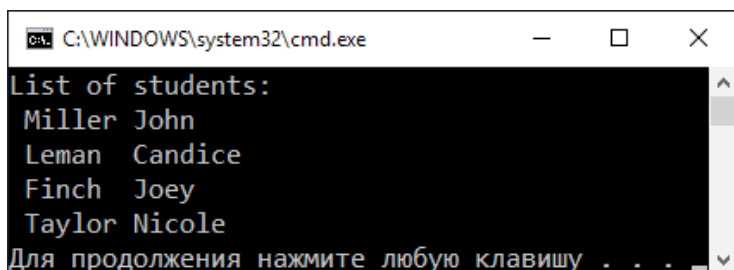


Рисунок 1.7. Применение обобщенного делегата `Action<T>`

Обобщенный делегат `Func<TResult>` обеспечивает вызов методов, которые могут принимать до 16 параметров и возвращают значение, тип которого указывается при объявлении делегата. Делегат `Func<TResult>` также довольно часто используется, например большинство методов класса `Enumerable` принимают этот делегат в качестве параметра. Мы еще будем рассматривать эти методы в шестом разделе текущего урока, а сейчас мы рассмотрим обобщенный метод `Select<TSource, TResult>()`, при помощи которого формируется результирующая последовательность элементов, полученная путем выполнения метода преобразования для каждого элемента исходной последовательности. Метод преобразования как раз и задается при помощи делегата `Func<TResult>`. В текущем примере мы при помощи метода `FullName()` сформируем последовательность строк — фамилия и имя студента,

для корректной работы метода необходимо подключить пространство имен `System.Linq` (Рисунок 1.8).

```
using System;
using System.Collections.Generic;
using System.Linq;
using static System.Console;

namespace SimpleProject
{
    class Student
    {
        public string FirstName { get; set; }
        public string LastName { get; set; }
        public DateTime BirthDate { get; set; }
    }

    class Program
    {
        static string FullName(Student student)
        {
            return $" {student.LastName}\\t
                    {student.FirstName}";
        }
        static void Main(string[] args)
        {
            List<Student> group = new List<Student> {
                new Student {
                    FirstName = "John",
                    LastName = "Miller",
                    BirthDate = new DateTime(1997,3,12)
                },
                new Student {
                    FirstName = "Candice",
                    LastName = "Leman",
                    BirthDate = new DateTime(1998,7,22)
                },
            }
        }
    }
}
```

```

        new Student {
            FirstName = "Joey",
            LastName = "Finch",
            BirthDate = new DateTime(1996,11,30)
        },
        new Student {
            FirstName = "Nicole",
            LastName = "Taylor",
            BirthDate = new DateTime(1996,5,10)
        }
    };

    WriteLine("List of students:");
    IEnumerable<string> students =
        group.Select(FullName);
    foreach (string item in students)
    {
        WriteLine(item);
    }
}

```

```

C:\WINDOWS\system32\cmd.exe
List of students:
Miller John
Leman Candice
Finch Joey
Taylor Nicole
Для продолжения нажмите любую клавишу . . .

```

Рисунок 1.8. Использование обобщенного делегата `Func<TResult>`

Как Вы заметили, в данном примере метод `Select()` возвращает последовательность, реализующую обобщенный

интерфейс `IEnumerable<string>`, также при вызове этого метода не указаны типы параметров (`group.Select<Student, string>(FullName)`), их можно опустить, так как они определяются автоматически.

Следующий обобщенный делегат `Predicate<T>` обеспечивает вызов методов, которые принимают один параметр и в качестве результата возвращают логическое значение — результат проверки переданного параметра заданным критериям. Данный делегат используется в методах осуществляющих действия по определенному условию, например методы поиска элементов коллекции классов `Array` и `List<T>`. В следующем примере мы при помощи метода `FindAll()` получим список всех студентов родившихся весной, критерии поиска определяем при помощи метода `OnlySpring()` (Рисунок 1.9).

```
using System;
using System.Collections.Generic;
using static System.Console;

namespace SimpleProject
{
    class Student
    {
        public string FirstName { get; set; }
        public string LastName { get; set; }
        public DateTime BirthDate { get; set; }
        public override string ToString()
        {
            return $"Surname: {LastName}, Name: {FirstName},
                    Born: {BirthDate.ToLongDateString()}";
        }
    }
}
```

```

class Program
{
    static bool OnlySpring(Student student)
    {
        return student.BirthDate.Month >=
            3 && student.BirthDate.Month <= 5;
    }

    static void Main(string[] args)
    {
        List<Student> group = new List<Student> {
            new Student {
                FirstName = "John",
                LastName = "Miller",
                BirthDate = new DateTime(1997,3,12)
            },

            new Student {
                FirstName = "Candice",
                LastName = "Leman",
                BirthDate = new DateTime(1998,7,22)
            },

            new Student {
                FirstName = "Joey",
                LastName = "Finch",
                BirthDate = new DateTime(1996,11,30)
            },

            new Student {
                FirstName = "Nicole",
                LastName = "Taylor",
                BirthDate = new DateTime(1996,5,10)
            }
        };

        WriteLine("Born in the spring:");
    }
}

```



```

        List<Student> students =
            group.FindAll(OnlySpring);

        foreach (Student item in students)
        {
            WriteLine(item);
        }
    }
}

```

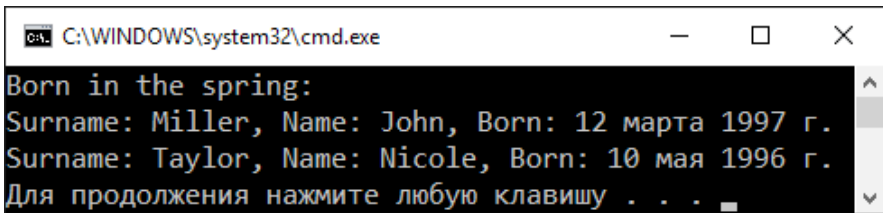


Рисунок 1.9. Применение обобщенного делегата `Predicate<T>`

Еще один обобщенный делегат `Comparison<T>` обеспечивает вызов методов, которые принимают два параметра одного типа и в качестве результата возвращают целочисленное значение: отрицательное, если первый параметр меньше второго, положительное, если первый параметр больше второго и ноль, если значения параметров равны. В частности данный делегат может применяться при сортировке элементов коллекций, для этого используется перегруженный метод `Sort(Comparison<T>)`. Отсортируем нашу коллекцию студентов по дате рождения, параметры сортировки задаем при помощи метода `SortBirthDate()` (Рисунок 1.10).

```

using System;
using System.Collections.Generic;
using static System.Console;

namespace SimpleProject
{
    class Student
    {
        public string FirstName { get; set; }
        public string LastName { get; set; }
        public DateTime BirthDate { get; set; }
        public override string ToString()
        {
            return $"Surname: {LastName},
                    Name: {FirstName},
                    Born: {BirthDate.ToLongDateString()}";
        }
    }

    class Program
    {
        static int SortBirthDate(Student student1,
                                  Student student2)
        {
            return student1.BirthDate.CompareTo(student2.
                                                  BirthDate);
        }

        static void Main(string[] args)
        {
            List<Student> group = new List<Student> {
                new Student {
                    FirstName = "John",
                    LastName = "Miller",
                    BirthDate = new DateTime(1997,3,12)
                },
            },

```

```

        new Student {
            FirstName = "Candice",
            LastName = "Leman",
            BirthDate = new DateTime(1998,7,22)
        },
        new Student {
            FirstName = "Joey",
            LastName = "Finch",
            BirthDate = new DateTime(1996,11,30)
        },
        new Student {
            FirstName = "Nicole",
            LastName = "Taylor",
            BirthDate = new DateTime(1996,5,10)
        }
    };
    WriteLine("Sort by date of birth:");
    group.Sort(SortBirthDate);
    foreach (Student item in group)
    {
        WriteLine(item);
    }
}
}

```

```

C:\WINDOWS\system32\cmd.exe
Sort by date of birth:
Surname: Taylor, Name: Nicole, Born: 10 мая 1996 г.
Surname: Finch, Name: Joey, Born: 30 ноября 1996 г.
Surname: Miller, Name: John, Born: 12 марта 1997 г.
Surname: Leman, Name: Candice, Born: 22 июля 1998 г.
Для продолжения нажмите любую клавишу . . .

```

Рисунок 1.10. Использование обобщенного делегата `Comparison<T>`

2. СОБЫТИЯ

Понятие события

Чтобы Вы лучше поняли события, опишем некую жизненную ситуацию. У каждого человека раз в году гарантировано происходит в жизни одно событие — День рождения. Люди обычно в этот день принимают поздравления, и довольно часто приглашают **определенное** количество людей отметить это событие за праздничным столом. Здесь ключевое слово «определенное», ведь Вы все равно празднуете в «узком кругу» и точно не пригласите на свой праздник человека, которого вообще не знаете.

Аналогичным образом при работе программы происходят те или иные события (ввод текста, нажатие кнопки и т.д.) и существует **определенное** количество объектов, которые должны как-то отреагировать на это событие, поэтому регистрируют свой метод для обработки этого события. Так же как Вы не будете приглашать незнакомого человека к себе за стол, так и события в программе не будут уведомлять о своем возникновении незаинтересованные объекты. Когда в программе происходит событие, то вызываются на выполнение все зарегистрированные методы-обработчики объектов, событие как бы говорит им: «Прошу за стол».

Синтаксис объявления события

При объявлении события используется ключевое слово `event` и общая форма записи выглядит следующим образом:

```
[модификатор доступа] event ИмяДелегата ИмяСобытия;
```

Как Вы видите, при объявлении события указывается делегат, который связан с этим событием. Поэтому прежде чем объявить событие, необходимо создать делегат, который будет содержать ссылки на методы, вызываемые при возникновении данного события.

В следующем примере мы смоделируем событие, которое уже не раз происходило с Вами в академии «ШАГ» — экзамен. В этом примере класс `Teacher` при помощи метода `Exam()` вызывает событие `examEvent`. Тем самым он оповещает о нем всех, студентов, которые выразили свою заинтересованность экзаменом через подписку на событие `examEvent` (операция `+=`). После этого у каждого элемента класса `Student` вызывается соответствующий метод, как реакция на событие `examEvent` (Рисунок 2.1)

```
using System;
using System.Collections.Generic;
using static System.Console;

namespace SimpleProject
{
    public delegate void ExamDelegate(string t);

    class Student
    {
        public string FirstName { get; set; }
        public string LastName { get; set; }
        public DateTime BirthDate { get; set; }

        public void Exam(string task)
        {
```

```

        WriteLine($"Student {LastName} solved
                    the {task}");
    }
}

class Teacher
{
    public event ExamDelegate examEvent;

    public void Exam(string task)
    {
        if (examEvent != null)
        {
            examEvent(task);
        }
    }
}

class Program
{
    static void Main(string[] args)
    {
        List<Student> group = new List<Student> {
            new Student {
                FirstName = "John",
                LastName = "Miller",
                BirthDate = new DateTime(1997,3,12)
            },
            new Student {
                FirstName = "Candice",
                LastName = "Leman",
                BirthDate = new DateTime(1998,7,22)
            },
            new Student {
                FirstName = "Joey",
                LastName = "Finch",
                BirthDate = new DateTime(1996,11,30)
            }
        };
    }
}

```

```

    },
    new Student {
        FirstName = "Nicole",
        LastName = "Taylor",
        BirthDate = new DateTime(1996,5,10)
    }
};

Teacher teacher = new Teacher();
foreach (Student item in group)
{
    teacher.examEvent += item.Exam;
}

teacher.Exam("Task");
}
}

```

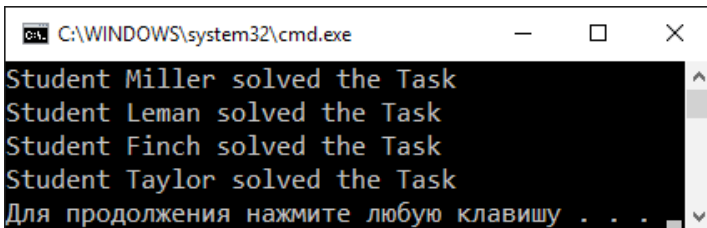



Рисунок 2.1. Пример использование события

Класс `Teacher` инкапсулирует весь механизм работы с событием, такого рода класс называется классом-диспетчером, основное предназначение которого — обеспечение удобства при работе с событием. В этом классе содержится метод `Exam()`, так называемый метод диспетчеризации, в котором проверяется существование методов-обработчиков подписавшихся на данное событие и только если

они есть, осуществляется вызов этих методов. IntelliSense предлагает нам убрать эту проверку, тем самым упростив вызов делегата `ExamDelegate` (Рисунок 2.2).

```
public void Exam(string task)
{
    if (examEvent != null)
    {
        examEvent(task);
    }
}
```

 `ExamDelegate Teacher.examEvent`

Delegate invocation can be simplified.

Рисунок 2.2. Вызов делегата может быть упрощен

Однако этого делать не стоит, ведь возможна ситуация когда ни один метод не подпишется на Ваше событие и тогда при его вызове сгенерируется исключительная ситуация (предлагаем Вам убедиться в этом самостоятельно).

Необходимость и особенности применения событий

К этому моменту Вы, может быть, уже задаете себе вопрос: «Зачем использовать события, если есть делегаты?». В этом разделе Вы получите ответ на свой вопрос.

На самом деле применение делегатов в чистом виде сопряжено с рядом трудностей, которые связаны с обеспечением инкапсуляции при использовании переменных-членов типа делегата. Продемонстрируем это на предыдущем примере, внося в него соответствующие коррективы (Рисунок 2.3).


```

using System;
using System.Collections.Generic;
using static System.Console;

namespace SimpleProject
{
    public delegate void ExamDelegate(string t);

    class Student
    {
        public string FirstName { get; set; }
        public string LastName { get; set; }
        public DateTime BirthDate { get; set; }

        public void Exam(string task)
        {
            WriteLine($"Student {LastName}
                        solved the {task}");
        }
    }

    class Teacher
    {
        public ExamDelegate examEvent;

        public void Exam(string task)
        {
            if (examEvent != null)
            {
                examEvent(task);
            }
        }
    }

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

```

```

List<Student> group = new List<Student> {
    new Student {
        FirstName = "John",
        LastName = "Miller",
        BirthDate = new DateTime(1997,3,12)
    },
    new Student {
        FirstName = "Candice",
        LastName = "Leman",
        BirthDate = new DateTime(1998,7,22)
    },
    new Student {
        FirstName = "Joey",
        LastName = "Finch",
        BirthDate = new DateTime(1996,11,30)
    },
    new Student {
        FirstName = "Nicole",
        LastName = "Taylor",
        BirthDate = new DateTime(1996,5,10)
    }
};

Teacher teacher = new Teacher();
foreach (Student item in group)
{
    teacher.examEvent += item.Exam;
}
// обращение напрямую
teacher.examEvent.Invoke("Overall rating 2!");

teacher.examEvent = null; // новое значение

teacher.Exam("Task"); // ни к чему не приведет
}
}
}

```

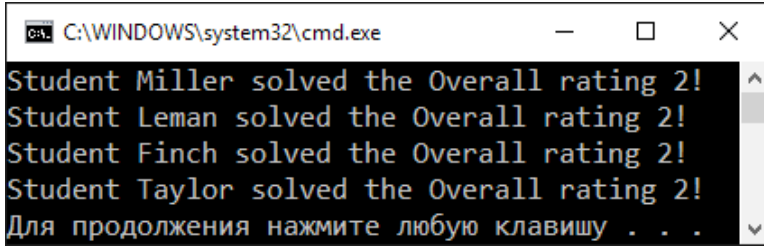


Рисунок 2.3. Проблемы применения делегатов
в чистом виде

В классе `Teacher` поле `examEvent` теперь типа делегата `ExamDelegate` и имеет модификатор доступа `public`, поэтому существует возможность обратиться к этому полю из вызывающего кода. При этом вызывающий код может, как присвоить этому полю новое значение, так и обратиться к списку вызовов делегата напрямую (показано в коде). После присвоения полю `examEvent` значения `null`, список вызовов не содержит ни одной ссылки на методы и любое обращение к этому списку ни к чему не приведет.

Решить эту проблему можно путем указания полю `examEvent` модификатора доступа `private` и создания дополнительных методов для формирования списка вызовов. Поэтому создатели языка C#, понимая сколько времени будет тратиться на написание однотипного кода, создали специальную конструкцию `event`.

На самом деле ключевое слово `event` служит оболочкой для делегата с модификатором доступа `private` и дополнительных методов, которые обеспечивают взаимодействие с этим делегатом и начинаются с `add` и `remove`, для добавления и удаления обработчиков события соответственно.

В этом легко убедиться, если при помощи утилиты `ildasm.exe` посмотреть строение любого класса, содержащего `event` (Рисунок 2.4).

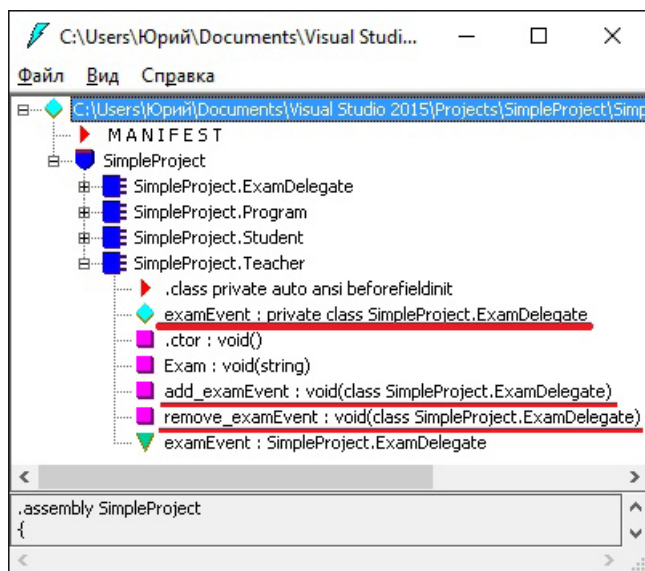


Рисунок 2.4. Класс `Teacher` после компиляции

При создании событий можно вообще обойтись без создания специального делегата, а использовать для этого обобщенный делегат `EventHandler<T>`, в качестве параметра `T` которого указывается тип, наследник от класса `EventArgs`. При этом обработчики события не должны возвращать значение и должны принимать два параметра: первый параметр — это ссылка на объект, сгенерировавший это событие, второй — объект типа `EventArgs`, содержащий необходимую дополнительную информацию о событии. Продемонстрируем применение делегата `EventHandler` на нашем примере, для получения

идентичного результата мы создали дополнительный класс `ExamEventArgs`, наследник от `EventArgs`, который содержит свойство `Task` (Рисунок 2.5).

```
using System;
using System.Collections.Generic;
using static System.Console;

namespace SimpleProject
{
    class ExamEventArgs : EventArgs
    {
        public string Task { get; set; }
    }

    class Student
    {
        public string FirstName { get; set; }
        public string LastName { get; set; }
        public DateTime BirthDate { get; set; }
        public void Exam(object sender, ExamEventArgs e)
        {
            WriteLine($"Student {LastName} solved the {e.Task}");
        }
    }

    class Teacher
    {
        public EventHandler<ExamEventArgs> examEvent;
        public void Exam(ExamEventArgs task)
        {
            if (examEvent != null)
            {
                examEvent(this, task);
            }
        }
    }
}
```

```

class Program
{
    static void Main(string[] args)
    {
        List<Student> group = new List<Student> {
            new Student {
                FirstName = "John",
                LastName = "Miller",
                BirthDate = new DateTime(1997,3,12)
            },
            new Student {
                FirstName = "Candice",
                LastName = "Leman",
                BirthDate = new DateTime(1998,7,22)
            },
            new Student {
                FirstName = "Joey",
                LastName = "Finch",
                BirthDate = new DateTime(1996,11,30)
            },
            new Student {
                FirstName = "Nicole",
                LastName = "Taylor",
                BirthDate = new DateTime(1996,5,10)
            }
        };
        Teacher teacher = new Teacher();
        foreach (Student item in group)
        {
            teacher.examEvent += item.Exam;
        }
        ExamEventArgs eventArgs =
            new ExamEventArgs { Task = "Task" };
        teacher.Exam(eventArgs);
    }
}

```

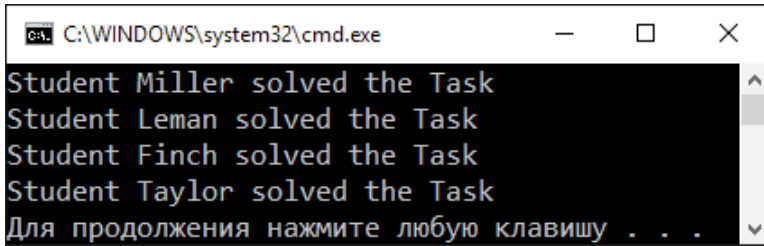


Рисунок 2.5. Применение обобщенного делегата EventHandler<T>

Применение события для многоадресатного делегата

События уже при создании задумывались для использования многоадресатного делегата, что было продемонстрировано в предыдущих примерах. На самом деле при осуществлении подписки на событие применяются только операции += и -= для добавления и удаления соответственно, операция = событиями не поддерживается. Следующий пример демонстрирует эту особенность событий (Рисунок 2.6).

```
using System;
using static System.Console;

namespace SimpleProject
{
    public delegate void ExamDelegate(string t);

    class Student
    {
        public string FirstName { get; set; }
        public string LastName { get; set; }
        public DateTime BirthDate { get; set; }
    }
}
```

```

        public void Exam(string task)
        {
            WriteLine($"Student {LastName} solved the {task}");
        }
    }

    class Teacher
    {
        public event ExamDelegate examEvent;

        public void Exam(string task)
        {
            if (examEvent != null)
            {
                examEvent(task);
            }
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
            Teacher teacher = new Teacher();

            Student student = new Student();

            teacher.examEvent += student.Exam;

            teacher.examEvent -= student.Exam;

            teacher.examEvent = student.Exam; // Error

            teacher.Exam("Task");
        }
    }
}

```

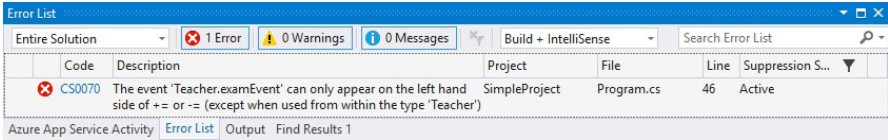



Рисунок 2.6. Ошибка: с событием используются только += и -=

Вы, наверное, обратили внимание на то, что после написания += IntelliSense предлагает Вам нажать клавишу Tab (Рисунок 2.7).

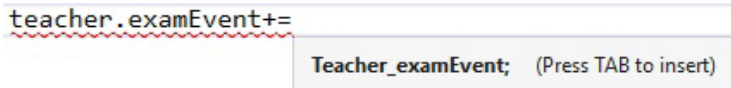


Рисунок 2.7. Подсказка IntelliSense

И если Вы прислушаетесь к его рекомендациям, то после нажатия на клавишу, в коде автоматически будет создана заготовка метода вызова, соответствующей сигнатуры, а после конструкции += пропишется название этого метода (Рисунок 2.8).

```

teacher.examEvent += Teacher_examEvent1;

teacher.Exam("Task");
}

1 reference
private static void Teacher_examEvent1(string t)
{
    throw new NotImplementedException();
}

```

Рисунок 2.8. Автоматическое создание заготовки метода вызова

Использование событийных средств доступа

Обычно использование стандартных операций `+=` и `-=` полностью удовлетворяют потребностям в управлении списком вызовов методов. Однако в этом случае список формируется по мере добавления методов, и вызов этих методов осуществляется в прямой последовательности от начала до конца списка, что не всегда соответствует поставленной задаче. Поэтому если необходимо изменить порядок вызова методов используют развернутую форму `event`, которая содержит методы для добавления и удаления обработчиков события `add` и `remove` соответственно, общая форма записи выглядит следующим образом:

```
[модификатор доступа]event ИмяДелегата ИмяСобытия
{
    add
    {
        //код добавления события в цепочку событий
    }
    remove
    {
        //код удаления события в цепочку событий
    }
}
```

Тем самым мы прописываем собственную реализацию методов, которая при сокращенной форме `event` создается автоматически (Рисунок 2.4).

Для примера использования событийных средств доступа воспользуемся знакомым Вам примером со сдачей

экзамена студентами, но теперь методы будут вызываться случайным образом. Чтобы этого добиться, в методе `add` для каждого обработчика события генерируется уникальное значение, которое выступает в роли ключа для коллекции `SortedList<int, ExamDelegate>` при добавлении каждого метода в эту коллекцию. Для этих целей в классе `Teacher` создали поле `_sortedEvents` типа `SortedList<int, ExamDelegate>`, чтобы сортировка методов осуществлялась автоматически при добавлении их в эту коллекцию. Удаление обработчика события из коллекции осуществляется в методе `remove` на основании индекса присвоенного этому обработчику (Рисунок 2.9).

```
using System;
using System.Collections.Generic;
using static System.Console;

namespace SimpleProject
{
    public delegate void ExamDelegate(string t);

    class Student
    {
        public string FirstName { get; set; }
        public string LastName { get; set; }
        public DateTime BirthDate { get; set; }

        public void Exam(string task)
        {
            WriteLine($"Student {LastName} solved the {task}");
        }
    }

    class Teacher
    {
```

```

        SortedList<int, ExamDelegate> _sortedEvents =
            new SortedList<int, ExamDelegate>();
        Random _rand = new Random();

        public event ExamDelegate examEvent
        {
            add
            {
                for (int key; ;)
                {
                    key = _rand.Next();
                    if (!_sortedEvents.ContainsKey(key))
                    {
                        _sortedEvents.Add(key, value);
                        break;
                    }
                }
            }
            remove
            {
                _sortedEvents.RemoveAt(_sortedEvents.
                    IndexOfValue(value));
            }
        }

        public void Exam(string task)
        {
            foreach (int item in _sortedEvents.Keys)
            {
                if (_sortedEvents[item] != null)
                {
                    _sortedEvents[item](task);
                }
            }
        }
    }

```

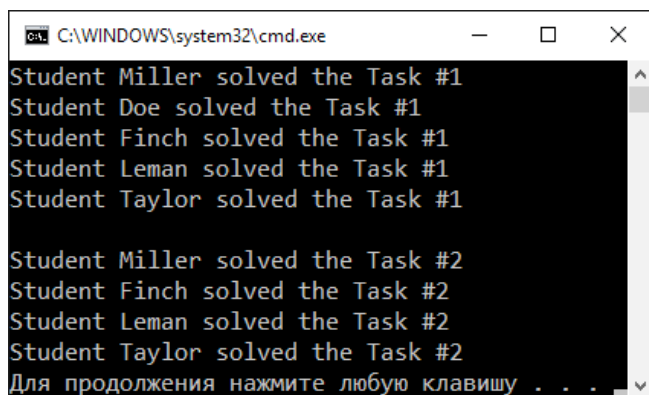
```
class Program
{
    static void Main(string[] args)
    {
        List<Student> group = new List<Student> {
            new Student {
                FirstName = "John",
                LastName = "Miller",
                BirthDate = new DateTime(1997,3,12)
            },
            new Student {
                FirstName = "Candice",
                LastName = "Leman",
                BirthDate = new DateTime(1998,7,22)
            },
            new Student {
                FirstName = "Joey",
                LastName = "Finch",
                BirthDate = new DateTime(1996,11,30)
            },
            new Student {
                FirstName = "Nicole",
                LastName = "Taylor",
                BirthDate = new DateTime(1996,5,10)
            }
        };
        Teacher teacher = new Teacher();

        foreach (Student item in group)
        {
            teacher.examEvent += item.Exam;
        }

        Student student = new Student
        {
            FirstName = "John",
            LastName = "Doe",
        }
    }
}
```

```
        BirthDate = new DateTime(1998, 10, 12)
    };

    teacher.examEvent += student.Exam;
    teacher.Exam("Task #1");
    WriteLine();
    teacher.examEvent -= student.Exam;
    teacher.Exam("Task #2");
}
}
```



```
C:\WINDOWS\system32\cmd.exe
Student Miller solved the Task #1
Student Doe solved the Task #1
Student Finch solved the Task #1
Student Leman solved the Task #1
Student Taylor solved the Task #1

Student Miller solved the Task #2
Student Finch solved the Task #2
Student Leman solved the Task #2
Student Taylor solved the Task #2
Для продолжения нажмите любую клавишу . . .
```

Рисунок 2.9. Использование событийных средств доступа

3. Анонимные методы

Во всех предыдущих примерах в качестве, вызываемых через делегаты или события, методов указывались методы классов. Однако если эти методы вызываются только при помощи делегатов и нигде не вызываются напрямую, то имеет смысл использовать специальный блок кода, который называется **анонимным методом**. Общая форма записи представлена ниже.

```
delegate [(параметры)] {  
    // выполняемый код  
};
```

Как Вы видите, при создании анонимного метода используется ключевое слово `delegate`, после которого указываются параметры, являющиеся необязательными, если Вы не планируете использовать их в выполняемом коде. В фигурных скобках прописывается исполняемый код и наконец, после закрывающейся фигурной скобки обязательно должна присутствовать точка с запятой, иначе будет сгенерирована ошибка на этапе компиляции. Следующий пример демонстрирует использование анонимных методов (Рисунок 3.1).

```
using System;  
using static System.Console;  
  
namespace SimpleProject  
{  
    public delegate double AnonimDelegateDouble(double x,  
        double y);  
}
```

```

public delegate void AnonimDelegateInt(int n);

public delegate void AnonimDelegateVoid();

class Dispatcher
{
    public event AnonimDelegateDouble eventDouble;
    public event AnonimDelegateInt eventVoid;

    public double OnEventDouble(double x, double y)
    {
        if (eventDouble != null)
        {
            return eventDouble(x, y);
        }
        throw new NullReferenceException();
    }

    public void OnEventVoid(int n = 0)
    {
        if (eventVoid != null)
        {
            eventVoid(n);
        }
    }
}

class Program
{
    static void Main(string[] args)
    {
        WriteLine("\tThe use of events");
        Dispatcher dispatcher = new Dispatcher();
        // анонимный метод
        dispatcher.eventDouble +=
            delegate (double a, double b)
            {

```



```

        if (b != 0)
        {
            return a / b;
        }

        throw new DivideByZeroException();
    };

    double n1 = 5.7, n2 = 3.2;

    WriteLine($"{n1} / {n2} =
        {dispatcher.OnEventDouble(n1, n2)}"); // ВЫЗОВ
    WriteLine("    Using a local variable");
    int number = 5;

    dispatcher.eventVoid += delegate (int n)// анонимный
                                   // метод
    {
        WriteLine($"{number} + {n} = { number + n}");
    };

    dispatcher.OnEventVoid(); // ВЫЗОВ
    dispatcher.OnEventVoid(6);

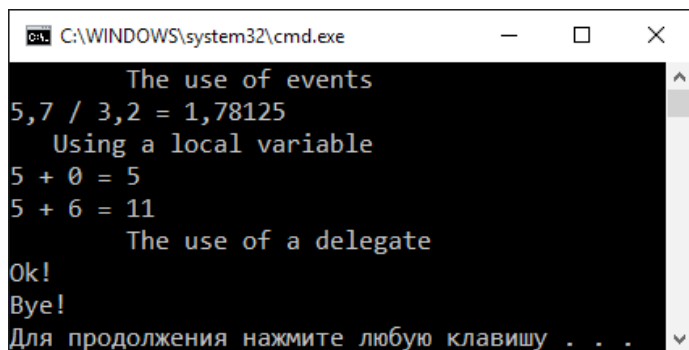
    WriteLine("\tThe use of a delegate");

    AnonimDelegateVoid voidDel =
        new AnonimDelegateVoid(delegate
            { WriteLine("Ok!"); });

    // анонимный метод
    voidDel += delegate { WriteLine("Bye!"); };

    voidDel(); // ВЫЗОВ
}
}
}

```



```
C:\WINDOWS\system32\cmd.exe

The use of events
5,7 / 3,2 = 1,78125
Using a local variable
5 + 0 = 5
5 + 6 = 11
The use of a delegate
Ok!
Bye!
Для продолжения нажмите любую клавишу . . .
```

Рисунок 3.1. Использование анонимных методов

В данном примере продемонстрирована возможность использования анонимных методов, как с событиями, так и с делегатами. Обращаем Ваше внимание на возможность использования в анонимных методах локальных переменных метода, в котором определены эти анонимные методы. Как Вы заметили, анонимные методы можно использовать даже при объявлении делегатов.

Использование анонимных методов никак не сказывается на скорости выполнения кода, главное их преимущество — упрощение кода, отсутствие необходимости создания дополнительных методов.

4. Лямбда выражения

Если анонимные методы появились в версии C# 2.0, то уже в версии C# 3.0 была представлена конструкция, которая позволяет в более сокращенной форме записывать сами анонимные методы — лямбда выражение.

Лямбда выражение состоит из двух частей, разделенных лямбда оператором (\Rightarrow). Левая часть лямбда выражения должна содержать список параметров, а правая часть — исполняемый код (тело лямбда выражения), который воздействует на параметры указанные в левой части.

```
(параметры) => { // код, использующий параметры }
```

Для лучшего понимания Вами лямбда выражений продемонстрируем сравнительный анализ между ними и анонимными методами.

Запись анонимного метода в общем виде выглядит так:

```
delegate (параметры) { // выполняемый код};
```

А вот общий вид лямбда выражения:

```
(параметры) => { // выполняемый код}
```

Как Вы видите, лямбда выражение состоит из тех же частей, что и анонимный метод только выглядит компактнее.

Реализация тела метода в виде выражения

В зависимости от количества операторов в теле лямбда выражений, они разделяются на одиночные и блочные.

Как понятно из названия, одиночные лямбда выражения состоят из одного оператора, а блочные из множества операторов, которые должны быть заключены в фигурные скобки.

Применение любого лямбда выражения можно разбить на три этапа:

- Объявление делегата необходимой сигнатуры.
- Создание экземпляра этого делегата и его инициализация совместимым лямбда выражением.
- Обращение к экземпляру делегата, при котором вычисляется лямбда выражение.

Чтобы продемонстрировать это, перепишем пример предыдущего раздела, но с использованием лямбда выражений и небольшими изменениями (Рисунок 4.1).

```
using System;
using static System.Console;

namespace SimpleProject
{
    public delegate double AnonimDelegateDouble(double x,
                                                double y);
    public delegate int AnonimDelegateInt(int n);
    public delegate void AnonimDelegateVoid();

    class Dispatcher
    {
        public event AnonimDelegateDouble eventDouble;
        public event AnonimDelegateInt eventInt;
        public double OnEventDouble(double x, double y)
        {
            if (eventDouble != null)
            {
```

```

        return eventDouble(x, y);
    }
    throw new NullReferenceException();
}

public int OnEventInt(int n = 0)
{
    if (eventInt != null)
    {
        return eventInt(n);
    }
    throw new NullReferenceException();
}
}

class Program
{
    static void Main(string[] args)
    {
        WriteLine("\tBlock lambda expression");
        Dispatcher dispatcher = new Dispatcher();

        // явная типизация
        dispatcher.eventDouble += (double a, double b) =>
        {
            if (b != 0)
            {
                return a / b;
            }
            throw new DivideByZeroException();
        };

        double n1 = 5.7, n2 = 3.2;
        WriteLine($"{n1} / {n2} =
            {dispatcher.OnEventDouble(n1, n2)}"); // вызов
        WriteLine("\tSingle lambda expression");
        int number1 = 5, number2 = 6;
    }
}

```

```

    dispatcher.eventInt += n => number1 + n;
    // неявная типизация

    WriteLine($"{number1} + {number2} =
        {dispatcher.OnEventInt(number2)}"); // вызов

    WriteLine("\tThe use of a delegate");

    AnonimDelegateVoid voidDel =
        new AnonimDelegateVoid(() =>
            { WriteLine("Ok!"); });

    voidDel += () => { WriteLine("Bye!"); };
    voidDel(); // вызов
}
}
}

```

```

C:\WINDOWS\system32\cmd.exe
Block lambda expression
5,7 / 3,2 = 1,78125
Single lambda expression
5 + 6 = 11
The use of a delegate
Ok!
Bye!
Для продолжения нажмите любую клавишу . . .

```

Рисунок 4.1. Использование лямбда выражений

Обращаем Ваше внимание на явную и неявную типизацию параметров лямбда выражений. При неявной типизации тип параметров лямбда выражения определяется исходя из сигнатуры делегата, если используется один параметр, то скобки списка параметров можно опустить.

Ценность лямбда выражений состоит в том, что они могут применяться в любом месте, где возможно использовать делегаты. Например, в некоторых методах коллекции `List` необходимо использовать делегат `Predicate<T>`, то есть подразумевается применение метода определенной сигнатуры, но теперь в создании такого метода даже анонимного отпала необходимость, ведь Вы можете использовать лямбда выражение. Изменим, знакомый Вам пример из первого раздела, теперь в методе `FindAll()` применяется лямбда выражение (Рисунок 4.2).

```
using System;
using System.Collections.Generic;
using static System.Console;

namespace SimpleProject
{
    class Student
    {
        public string FirstName { get; set; }
        public string LastName { get; set; }
        public DateTime BirthDate { get; set; }
        public override string ToString()
        {
            return $"Surname: {LastName},
                    Name: {FirstName},
                    Born: {BirthDate.ToLongDateString()}";
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
            List<Student> group = new List<Student> {
```

```

        new Student {
            FirstName = "John",
            LastName = "Miller",
            BirthDate = new DateTime(1997,3,12)
        },
        new Student {
            FirstName = "Candice",
            LastName = "Leman",
            BirthDate = new DateTime(1998,7,22)
        },
        new Student {
            FirstName = "Joey",
            LastName = "Finch",
            BirthDate = new DateTime(1996,11,30)
        },
        new Student {
            FirstName = "Nicole",
            LastName = "Taylor",
            BirthDate = new DateTime(1996,5,10)
        }
    };

    WriteLine("Born in the spring:");

    List<Student> students = group.FindAll(s =>
        s.BirthDate.Month >=
        3 && s.BirthDate.Month <= 5);

    foreach (Student item in students)
    {
        WriteLine(item);
    }
}

```

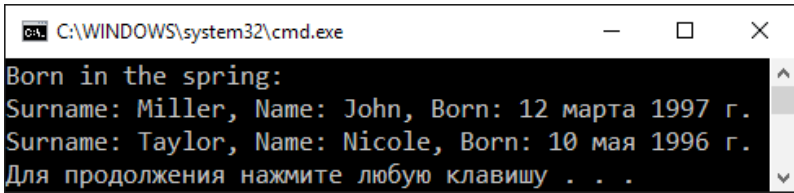



Рисунок 4.2. Использование лямбда выражений в методе коллекции

Откроем сгенерированный CIL-код нашего приложения при помощи утилиты `ildasm.exe`, и убедимся, что на самом деле лямбда выражения транслируется в соответствующий делегат (Рисунок 4.3).

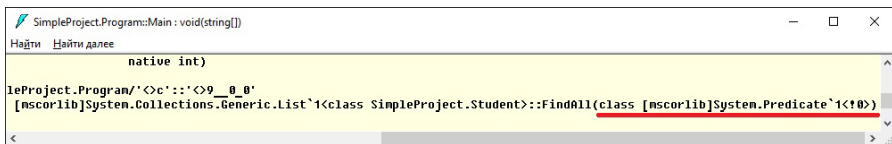


Рисунок 4.3. CIL-код метода FindAll()

В версии 6.0 языка C# появилась возможность использовать лямбда-выражения для определения тела методов и свойств, которое указывается справа от лямбда-выражения. В качестве демонстрации приведем простой пример (Рисунок 4.4).

```
using System;
using static System.Console;

namespace SimpleProject
{
    class Program
    {
        class ExampleCalc
        {
```

```

        public string CurrentDate =>
            $"{\tThe current date {DateTime.Now.
                ToLongDateString()}\n";

        public int AddInt(int x, int y) => x + y;

        public static void AddVoid(int x, int y) =>
            WriteLine($"{x} + {y} = {x + y}");
    }

    static void Main(string[] args)
    {
        ExampleCalc calc = new ExampleCalc();

        WriteLine(calc.CurrentDate);

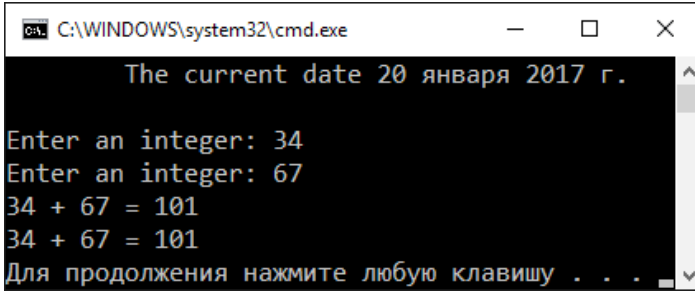
        try
        {
            Write("Enter an integer: ");
            int n1 = int.Parse(ReadLine());

            Write("Enter an integer: ");
            int n2 = int.Parse(ReadLine());

            WriteLine($"{n1} + {n2} =
                        {calc.AddInt(n1, n2)}");
            ExampleCalc.AddVoid(n1, n2);
        }

        catch (Exception ex)
        {
            WriteLine(ex.Message);
        }
    }
}

```



```
cmd C:\WINDOWS\system32\cmd.exe
The current date 20 января 2017 г.
Enter an integer: 34
Enter an integer: 67
34 + 67 = 101
34 + 67 = 101
Для продолжения нажмите любую клавишу . . .
```

Рисунок 4.4. Использование лямбда-выражения для определения тела методов

Еще одно важное применение лямбда выражений — это их использование при написании запросов LINQ, которые будут рассмотрены нами в шестом разделе текущего урока.

5. Extension методы

Представим следующую ситуацию: при написании своего приложения Вы используете библиотеку, разработанную другим программистом, естественно в этой библиотеке содержатся классы с какой-то функциональностью. В определенный момент времени Вы понимаете, что например в одном из этих классов очень необходим дополнительный метод или методы. Если бы эту библиотеку разработали Вы, то нужно было внести изменения в исходный код и перекомпилировать его, но так как исходного кода у Вас нет, то простого решения не существует. Одним из способов решения этой проблемы являются **методы расширения**, которые позволяют добавлять дополнительную функциональность в уже скомпилированные типы без создания производного класса.

Методы расширения должны находиться в статическом классе при этом сами методы тоже должны быть статическими. Отличие методов расширения от обычных статических методов состоит в том, что первому параметру в расширяющих методах в качестве модификатора необходимо указать ключевое слово `this`. Тип данных первого параметра должен совпадать с типом данных к которому будет относиться этот метод.

Рассмотрим следующий пример, в котором создадим метод расширения `NumberWords()` для типа `string`, который, как Вы знаете, является запечатанным классом, что не позволяет использовать его в качестве базового класса.

Данный метод определяет количество слов в строке, для его корректной работы необходимо удалить «лишние» пробелы. При помощи метода `Trim()` класса `String` удаляются начальные и конечные пробелы в строке, повторяющиеся пробелы заменяются одиночными при помощи метода `Replace()` класса `Regex` (регулярные выражения будут рассматриваться в следующем уроке).

```
using static System.Console;

namespace SimpleProject
{
    static class ExampleExtensions
    {
        public static int NumberWords(this string data)
        {
            if (string.IsNullOrEmpty(data))
            {
                return 0;
            }

            data = System.Text.RegularExpressions.Regex.
                Replace(data.Trim(), @"\s+", " ");

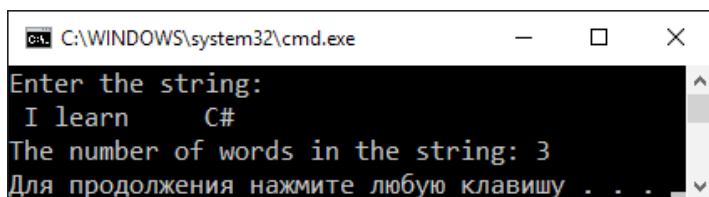
            return data.Split(' ').Length;
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
            WriteLine("Enter the string:");

            string str = ReadLine();
        }
    }
}
```

```
        WriteLine($"The number of words in the string:  
                    {str.NumberWords()}");  
    }  
}
```

Возможный результат работы программы представлен на рисунке 5.1.



```
C:\WINDOWS\system32\cmd.exe
Enter the string:
I learn C#
The number of words in the string: 3
Для продолжения нажмите любую клавишу . . .
```

Рисунок 5.1. Пример расширяющего метода

6. LINQ to Object

Роль LINQ

Большинство разрабатываемых программ в процессе выполнения осуществляют взаимодействие с какими-то хранилищами данных, начиная от элементарных массивов заканчивая базами данных. При этом способы получения информации из этих хранилищ, естественно, различные. Возможность работать однотипно с различными источниками данных появилась благодаря разработке языка интегрированных запросов (*Language Integrated Query* (LINQ)), который появился в версии .NET 3.5.

Как следует из названия, LINQ это язык запросов, то есть Вы при помощи LINQ-запросов указываете **что** Вы хотите получить, а **как** это сделать решает сам язык по отношению к конкретному источнику данных, которым должен быть объект, реализующий интерфейс `IEnumerable`.

В данном уроке мы рассмотрим только одну часть LINQ — LINQ to Object, которая позволяет получать информацию из различных коллекций. Однако существуют еще несколько разновидностей LINQ, которые Вы освоите в процессе дальнейшего изучения языка C#:

- LINQ to DataSet — используется для получения данных из DataSet;

- LINQ to XML — применяется для получения информации из файлов XML;
- LINQ to Sql — используется для получения данных из MS SQL Server;
- LINQ to Entities — используется при работе с технологией Entity Framework;
- Parallel LINQ (PLINQ) — применяется для выполнения параллельных запросов.

Для выполнения LINQ-запросов необходимо подключить пространство имен `System.Linq`.

Исследование операций запросов LINQ

Запрос LINQ — это набор инструкций по извлечению данных из указанного источника, написанных с использованием различных операторов LINQ. В самом простом виде LINQ-запрос содержит три оператора и выглядит следующим образом.

```
результат = from имя переменной in источник данных
              select имя переменной;
```

LINQ-запрос всегда начинается с оператора `from`, при помощи которого объявляется переменная диапазона, представляющая каждый элемент в исходном источнике данных и имеет тип этого элемента, сам источник данных указывается после оператора `in`. Оператор `select` обеспечивает возврат значений, полученных при выполнении всех операторов, расположенных между `from` и `select`. Продемонстрируем это на примере (Рисунок 6.1).


```

using System.Collections.Generic;
using System.Linq;
using static System.Console;

namespace SimpleProject
{
    class Program
    {
        static void Main(string[] args)
        {
            int[] arrayInt = { 5, 34, 67, 12, 94, 42 };

            IEnumerable<int> query = from i in arrayInt
                                    select i;

            WriteLine("The array to change:");

            foreach (int item in query)
            {
                Write($"{item}\t");
            }

            arrayInt[0] = 25;

            WriteLine("\nThe array after the change:");

            foreach (int item in query)
            {
                Write($"{item}\t");
            }
            WriteLine();
        }
    }
}

```

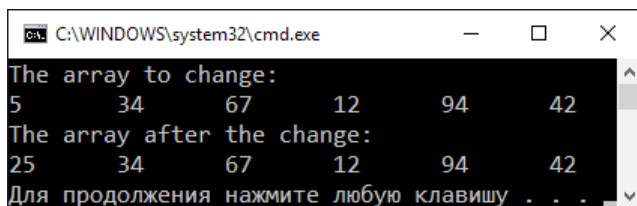
A screenshot of a Windows command prompt window titled "C:\WINDOWS\system32\cmd.exe". The window has a black background with white text. It displays the following text:
The array to change:
5 34 67 12 94 42
The array after the change:
25 34 67 12 94 42
Для продолжения нажмите любую клавишу . . .
The first line shows an array of six numbers: 5, 34, 67, 12, 94, 42. The second line shows the same array after a transformation, where the first element (5) has been changed to 25. The third line is a prompt for the user to press any key to continue.

Рисунок 6.1. Простейший пример LINQ-запроса

Предыдущий запрос просто возвращает все элементы массива целых чисел без каких-либо изменений. Такого рода запросы особой ценности не представляют и могут использоваться только в качестве примера, однако, в этом же примере показана важная особенность запросов LINQ — **отложенное выполнение**. На самом деле выполнение LINQ-запроса происходит не в момент его создания и присваивания его результатов переменной, а при обращении к этой переменной с целью получения этих самых результатов (в данном случае в `foreach`). Благодаря такому подходу каждый результат выполнения одного и того же LINQ-запроса по отношению к одной и той же коллекции всегда будет содержать самые актуальные данные.

Если существует необходимость в немедленном выполнении LINQ-запроса, тогда следует вызвать один из методов приводящий LINQ-запрос к одному из типов коллекций, например `ToList()` или `ToArray()`.

Довольно часто LINQ-запрос применяется для получения определенной выборки из всей коллекции, в таких запросах необходимо использовать оператор `where`, после которого находится логическое условие, применяемое к каждому элементу источника данных,

и только в том случае если заданное условие будет истинным, оператор `where` вернет соответствующий элемент. Для того чтобы продемонстрировать возможности использования оператора `where`, приведем следующий пример, целью которого является получение множества только четных элементов целочисленного массива (Рисунок 6.2).

```
using System.Collections.Generic;
using System.Linq;
using static System.Console;

namespace SimpleProject
{
    class Program
    {
        static void Main(string[] args)
        {
            int[] arrayInt = { 5, 34, 67, 12, 94, 42 };

            IEnumerable<int> query = from i in arrayInt
                                    where i % 2 == 0
                                    select i;

            WriteLine("Only the even elements:");
            foreach (int item in query)
            {
                Write($"{item}\t");
            }

            WriteLine();
        }
    }
}
```

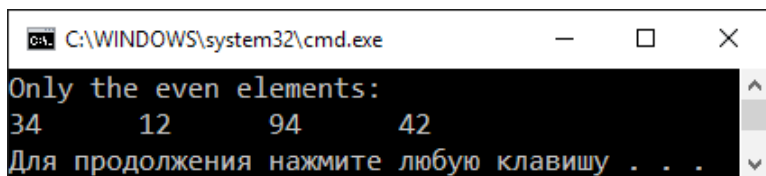


Рисунок 6.2. Использование оператора `where` в запросе LINQ

Чтобы упорядочить полученные результаты по возрастанию или по убыванию в LINQ-запросе применяется оператор `orderby` с указанием поля, по значениям которого будет осуществляться сортировка и направление сортировки. Сортировка по возрастанию применяется по умолчанию и значение `ascending` указывать необязательно, при необходимости сортировать элементы по убыванию направление сортировки — `descending` — указывается обязательно. Добавим в предыдущий пример сортировку результатов по убыванию (Рисунок 6.3).

```
using System.Collections.Generic;
using System.Linq;
using static System.Console;

namespace SimpleProject
{
    class Program
    {
        static void Main(string[] args)
        {
            int[] arrayInt = { 5, 34, 67, 12, 94, 42 };

            IEnumerable<int> query = from i in arrayInt
                                    where i % 2 == 0
                                    orderby i descending
                                    select i;
```

```

        WriteLine("Even elements descending:");
        foreach (int item in query)
        {
            Write($"{item}\t");
        }

        WriteLine();
    }
}

```

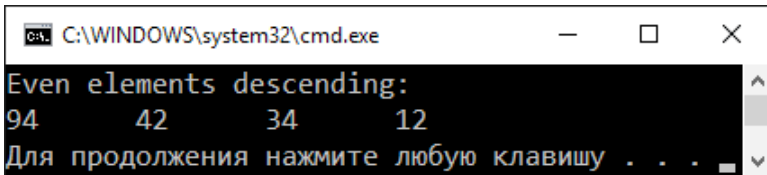


Рисунок 6.3. Сортировка результатов в запросе LINQ

Оператор `group` позволяет получить подмножество данных на основе критерия, указанного после ключевого слова `by`. В следующем примере напишем LINQ-запрос, в котором сформируем группы элементов на основании цифры, на которую оканчиваются эти элементы (Рисунок 6.4).

```

using System.Collections.Generic;
using System.Linq;
using static System.Console;

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

```

```

int[] arrayInt = { 5, 34, 67, 12, 94, 42 };

IEnumerable<IGrouping<int, int>> query =
    from i in arrayInt
    group i by i % 10;

WriteLine("Forming groups of criteria:");
foreach (IGrouping<int, int> key in query)
{
    Write($"Key: {key.Key}\nValue:");
    foreach (int item in key)
    {
        Write($"\\t{item}");
    }
    WriteLine();
}
}
}

```

```

cmd. C:\WINDOWS\system32\cmd.exe
Forming groups of criteria:
Key: 5
Value: 5
Key: 4
Value: 34 94
Key: 7
Value: 67
Key: 2
Value: 12 42
Для продолжения нажмите любую клавишу . . .

```

Рисунок 6.4. Группировка результатов
в LINQ-запросе по критерию

Хочется обратить Ваше внимание на то, что в результате группировки формируются объекты реализующие

интерфейс `IGrouping<T, K>`. Так в предыдущем примере мы получили группы типа `IGrouping<int, int>`, где ключом и значениями является тип `int`. Поскольку полученные при группировке объекты являются списком в списке, то для получения значения каждого элемента группы необходимо использовать вложенный цикл `foreach`.

Ключевое слово `into` применяется для создания идентификатора, в котором необходимо сохранить временные результаты работы операторов `group`, `join` или `select`, в том случае если после них требуется выполнить дополнительные операции LINQ-запроса. В следующем примере необходимо получить только те группы из предыдущего примера, в которых количество элементов больше одного (Рисунок 6.5).

```
using System.Collections.Generic;
using System.Linq;
using static System.Console;

namespace SimpleProject
{
    class Program
    {
        static void Main(string[] args)
        {
            int[] arrayInt = { 5, 34, 67, 12, 94, 42 };

            IEnumerable<IGrouping<int, int>> query =
                from i in arrayInt
                group i by i % 10 into res
                where res.Count() > 1
                select res;
```

```

WriteLine("Groups with the number of elements
          is greater than 1:");

foreach (IGrouping<int, int> key in query)
{
    Write($"Key: {key.Key}\nValue:");
    foreach (int item in key)
    {
        Write($"\\t{item}");
    }
    WriteLine();
}
}
}

```

```

C:\WINDOWS\system32\cmd.exe
Groups with the number of elements is greater than 1:
Key: 4
Value: 34      94
Key: 2
Value: 12      42
Для продолжения нажмите любую клавишу . . .

```

Рисунок 6.5. Использование временного идентификатора запроса

Оператор `let` предназначен для создания новой переменной диапазона в качестве временного хранилища промежуточных данных. В следующем примере данный оператор применяется для создания массива слов для каждой строки из исходного массива строк. Полученный массив слов используется в дополнительном операторе `from`, который допускается, когда в LINQ-запросе необходимо задействовать больше одной коллекции. Оператор

`where` используется для определения количества символов в каждом слове промежуточного массива (Рисунок 6.6).

```
using System.Collections.Generic;
using System.Linq;
using static System.Console;

namespace SimpleProject
{
    class Program
    {
        static void Main(string[] args)
        {
            string[] poem = {"All the world's a stage,",
                             "And all the men and women
                             merely players;", "They have
                             their exits and their
                             entrances,", "And one man in
                             his time plays many parts,",
                             "His acts being seven ages..."
            };

            IEnumerable<string> query = from p in poem
                                       let words = p.Split(' ', ';', ',')
                                       from w in words
                                       where w.Count() > 5
                                       select w;

            WriteLine("Words, in which more
                      than 5 characters:");

            foreach (string item in query)
            {
                WriteLine($"{item}");
            }
        }
    }
}
```

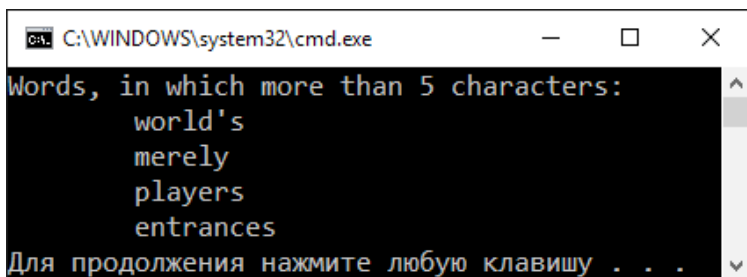


Рисунок 6.6. Использование временной переменной в LINQ-запросе

Оператор `join` позволяет соединить элементы из двух источников данных на основании равенства соответствующих ключей в каждом элементе. Ключи проверяются только на равенство, что обеспечивается оператором `equals`. Следующий пример демонстрирует использование одного из типов соединения — групповое соединение. Данный тип соединения формирует последовательность результатов, связывая элементы на основании некоего условия, вследствие чего формируется коллекция коллекций, которая может служить основой для вложенного запроса. В примере на рисунке 6.7 соединяются элементы из коллекции `List<Student>` с соответствующими элементами коллекции `List<Group>` на основании идентификатора группы.

```
using System.Collections.Generic;
using System.Linq;
using static System.Console;

namespace SimpleProject
{
    class Student
    {
```

```

        public string FirstName { get; set; }
        public string LastName { get; set; }
        public int GroupId { get; set; }
    }

    class Group
    {
        public int Id { get; set; }
        public string Name { get; set; }
    }

    class Program
    {
        static void Main(string[] args)
        {
            List<Group> groups = new List<Group>
                { new Group { Id = 1, Name = "27PPS11" },
                  new Group { Id = 2, Name = "27PPS12" } };
            List<Student> students = new List<Student> {
                new Student { FirstName = "John",
                              LastName = "Miller", GroupId = 2 },
                new Student { FirstName = "Candice",
                              LastName = "Leman", GroupId = 1 },
                new Student { FirstName = "Joey",
                              LastName = "Finch", GroupId = 1 },
                new Student { FirstName = "Nicole",
                              LastName = "Taylor", GroupId = 2 }
            };

            IEnumerable<Student> query = from g in groups
                                          join st in students on g.Id equals
                                          st.GroupId into res
                                          from r in res
                                          select r;

            WriteLine("\tStudents in groups:");
        }
    }

```

```

foreach (Student item in query)
{
    WriteLine($"Surname: { item.LastName},
               Name: { item.FirstName},
               Group: {groups.First(g => g.Id ==
               item.GroupId).Name}");
}
}
}

```

```

C:\WINDOWS\system32\cmd.exe
Students in groups:
Surname: Leman, Name: Candice, Group: 27PPS11
Surname: Finch, Name: Joey, Group: 27PPS11
Surname: Miller, Name: John, Group: 27PPS12
Surname: Taylor, Name: Nicole, Group: 27PPS12
Для продолжения нажмите любую клавишу . . .

```

Рисунок 6.7. Использование группового соединения

При выводе результатов для получения названия группы используется метод расширения `First()`, который возвращает первый элемент коллекции, удовлетворяющий заданному условию (совпадение идентификатора группы).

Рассмотренные в этом разделе способы построения LINQ-выражения с использованием различных LINQ-операторов имеют общее название **синтаксис запроса**.

Возврат результата запроса LINQ. Анонимные типы

Как Вы уже заметили в примерах предыдущего раздела, типом возврата LINQ-запроса обычно является объект, реализующий интерфейс `IEnumerable<T>`, однако

понять какой тип скрывается за `T` не всегда просто, а неправильное определение типа возврата приведет к ошибке на этапе компиляции. Поэтому, чтобы облегчить жизнь программистам, было принято решение использовать в качестве возврата LINQ-запросов **неявно типизированные переменные**.

Суть неявной типизации сводится к следующему: если на момент компиляции тип переменной неизвестен, тогда при объявлении переменной вместо указания типа данных используется ключевое слово `var`, а конкретный тип данных определяется динамически на основании значения, указанного при инициализации этой переменной (Рисунок 6.8).

```
using static System.Console;

namespace SimpleProject
{
    class Program
    {
        static void Main(string[] args)
        {
            var number = 56;
            WriteLine($"Variable {number}
                        has type {number.GetType()}");

            double salary = 6784.54;
            WriteLine($"Variable {salary}
                        has type {salary.GetType()}");
        }
    }
}
```

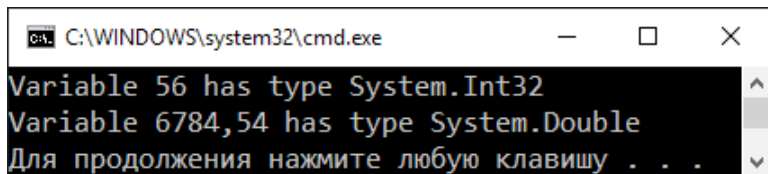


Рисунок 6.8. неявно типизированные переменные

Объявление неявно типизированной переменной без указания начального значения недопустимо, так как компилятор будет не в состоянии определить тип данных этой переменной, по той же причине нельзя присваивать в качестве начального значения `null` (Рисунок 6.9).

```
namespace SimpleProject
{
    class Program
    {
        static void Main(string[] args)
        {
            var number;
            var salary = null;
        }
    }
}
```

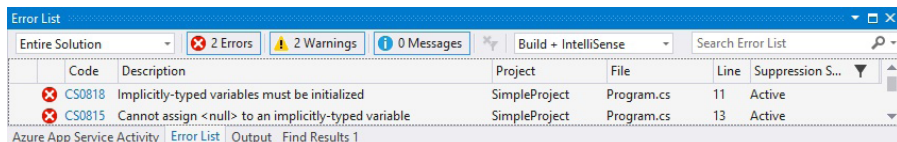


Рисунок 6.9. Ошибки инициализации неявно типизированных переменных

Обычно при создании класса Вы предполагаете, что в дальнейшем он будет активно использоваться

и выполнять определенную функциональность. Поэтому помимо полей в Вашем классе обычно присутствуют свойства и специальные методы, а также события и конструкторы. Однако в тех случаях, когда нет необходимости в выполнении классом специальной функциональности, и при этом Ваш класс будет использоваться только в текущем приложении, существует возможность объявления **анонимного типа**.

При создании анонимного типа применяется ключевое слово `new`, после которого имя типа не указывается (отсюда и название), а при объявлении свойств используется синтаксис инициализации объектов. Уникальное имя этому типу будет сгенерировано компилятором автоматически, однако до момента компиляции этот тип неизвестен, поэтому он объявляется при помощи ключевого слова `var`.

Анонимный тип наследуется напрямую от `object`, поэтому в нем уже реализованы все базовые методы, продемонстрируем это на примере (Рисунок 6.10).

```
using static System.Console;

namespace SimpleProject
{
    class Program
    {
        static void Main(string[] args)
        {
            var worker = new { FirstName = "John",
                               LastName = "Doe",
                               Salary = 7456.32 };
            WriteLine($"String: {worker}");
        }
    }
}
```

```

        WriteLine($"Type: {worker.GetType()}");
        WriteLine($"Hash code: {worker.GetHashCode()}");
    }
}

```

```

C:\WINDOWS\system32\cmd.exe
String: { FirstName = John, LastName = Doe, Salary = 7456,32 }
Type: <>f__AnonymousType0`3[System.String,System.String,System.Double]
Hash code: -2031831612
Для продолжения нажмите любую клавишу . . .

```

Рисунок 6.10. Создание анонимного типа

Анонимные типы не являются полноценными классами, потому что имеют ряд существенных ограничений:

- свойства анонимных типов доступны только для чтения;
- отсутствует возможность создания методов, событий и т.д.;
- анонимные типы не поддерживают наследование, так как являются неявно запечатанными.

Как Вы уже понимаете, широкое использование анонимных типов в обычном коде ограничено и даже не приветствуется. Однако анонимные типы незаменимы, когда необходимо создать новый класс динамически при выполнении LINQ-запросов.

В следующем примере анонимные типы используются в качестве результата внутреннего соединения, которое создается с помощью оператора `join`. При этом типе соединения каждый элемент коллекции `List<Student>` соответствует каждому элементу коллекции `List<Group>`, если для элемента первой

коллекции нет соответствия во второй коллекции, то он не добавляется в результирующий набор (студент Joey Finch) (Рисунок 6.11).

```
using System.Collections.Generic;
using System.Linq;
using static System.Console;

namespace SimpleProject
{
    class Student
    {
        public string FirstName { get; set; }
        public string LastName { get; set; }
        public int GroupId { get; set; }
    }

    class Group
    {
        public int Id { get; set; }
        public string Name { get; set; }
    }

    class Program
    {
        static void Main(string[] args)
        {
            List<Group> groups =
                new List<Group> {
                    new Group { Id = 1,
                        Name = "27PPS11" },
                    new Group { Id = 2,
                        Name = "27PPS12" } };
        }
    }
}
```

```

List<Student> students = new List<Student> {
    new Student { FirstName = "John",
        LastName = "Miller", GroupId = 2 },
    new Student { FirstName = "Candice",
        LastName = "Leman", GroupId = 1 },
    new Student { FirstName = "Joey",
        LastName = "Finch", GroupId = 3 },
    new Student { FirstName = "Nicole",
        LastName = "Taylor", GroupId = 2 }
};

var query = from g in groups
            join st in students on g.Id
            equals st.GroupId
            select new { FirstName =
                st.FirstName, LastName =
                st.LastName, GroupName = g.Name };

WriteLine("\tStudents in groups:");

foreach (var item in query)
{
    WriteLine(item);
}
}
}

```

```

C:\WINDOWS\system32\cmd.exe
Students in groups:
{ FirstName = Candice, LastName = Leman, GroupName = 27PPS11 }
{ FirstName = John, LastName = Miller, GroupName = 27PPS12 }
{ FirstName = Nicole, LastName = Taylor, GroupName = 27PPS12 }
Для продолжения нажмите любую клавишу . . .

```

Рисунок 6.11. Использование анонимных типов
в LINQ-запросе

Применение запросов LINQ к объектам коллекций

При создании LINQ-запросов к объектам коллекций можно создать LINQ-выражение с использованием синтаксиса запроса, рассмотренного ранее (Рисунок 6.12).

```
using System;
using System.Collections.Generic;
using System.Linq;
using static System.Console;

namespace SimpleProject
{
    class Student
    {
        public string FirstName { get; set; }
        public string LastName { get; set; }
        public DateTime BirthDate { get; set; }
        public override string ToString()
        {
            return $"Surname: {LastName}, Name: {FirstName},
                Born: {BirthDate.ToLongDateString()}";
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
            const double daysOfYear = 365.25;

            List<Student> students = new List<Student> {
                new Student {
                    FirstName = "John",
                    LastName = "Miller",
                    BirthDate = new DateTime(1997,3,12)
                },
            },
```

```

        new Student {
            FirstName = "Candice",
            LastName = "Leman",
            BirthDate = new DateTime(1998,7,22)
        },

        new Student {
            FirstName = "Joey",
            LastName = "Finch",
            BirthDate = new DateTime(1996,11,30)
        },

        new Student {
            FirstName = "Nicole",
            LastName = "Taylor",
            BirthDate = new DateTime(1996,1,10)
        }
    };

    WriteLine($"\\tThe current date:
               {DateTime.Now.ToLongDateString()}\\n");

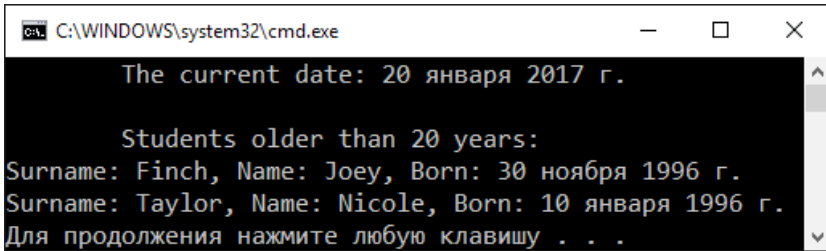
    var query = from s in students
                where (DateTime.Now - s.BirthDate).
                    Days / daysOfYear > 20
                select s;

    WriteLine("\\tStudents older than 20 years:");

    foreach (var item in query)
    {
        WriteLine(item);
    }

}
}
}

```



```

C:\WINDOWS\system32\cmd.exe

The current date: 20 января 2017 г.

Students older than 20 years:
Surname: Finch, Name: Joey, Born: 30 ноября 1996 г.
Surname: Taylor, Name: Nicole, Born: 10 января 1996 г.
Для продолжения нажмите любую клавишу . . .
  
```

Рисунок 6.12. Применение синтаксиса запроса к объекту-коллекции

В данном коде следует обратить Ваше внимание на константу `daysOfYear` — количество дней в году, равную `365.25` (вот почему каждый четвертый год високосный). Ее необходимо использовать в качестве делителя для получения количества лет при делении значения свойства `Days` структуры `TimeSpan`, так как это свойство возвращает количество полных дней в заданном интервале времени.

Однако существует еще один способ создания LINQ-выражения с применением различных расширяющих методов, которые принимают в качестве параметра делегат. Некоторые из этих методов мы уже рассматривали в предыдущих разделах, когда применяли стандартные делегаты. Обычно делегат представлен в виде лямбда-выражения, которое воздействует на каждый элемент коллекции, тем самым обеспечивая выбор информации из коллекции.

Для получения результата необходимые методы выстраиваются в цепочку вызовов, такой способ формирования LINQ-выражения называется **синтаксис метода**. Применим этот способ к предыдущему примеру и получим тот же результат (Рисунок 6.13).

```
using System;
using System.Collections.Generic;
using System.Linq;
using static System.Console;

namespace SimpleProject
{
    // класс Student остался прежним

    class Program
    {

        static void Main(string[] args)
        {
            const double daysOfYear = 365.25;

            // код остался прежним
            WriteLine($"\\tThe current date:
                        {DateTime.Now.ToLongDateString()}\\n");

            var query = students.Where(s =>
                (DateTime.Now - s.BirthDate).
                Days / daysOfYear > 20).Select(s => s);

            WriteLine("\\tStudents older than 20 years:");

            foreach (var item in query)
            {
                WriteLine(item);
            }

        }
    }
}
```

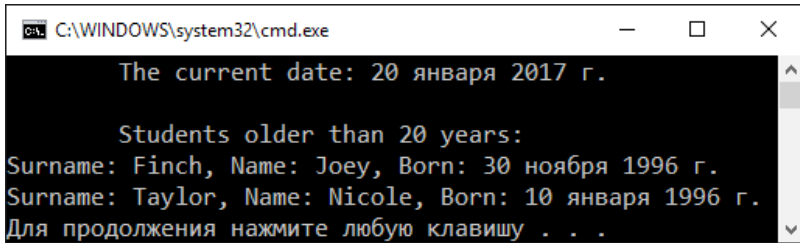


Рисунок 6.13. Применение синтаксиса метода к объекту-коллекции

Метод `Where()` возвращает коллекцию, состоящую из элементов, значение которых удовлетворяет указанному лямбда-выражению. Метод `Select()` преобразует каждый элемент коллекции в требуемый вид.

К сожалению, привести примеры использования всех расширяющих методов в одном уроке довольно проблематично, поэтому для получения более подробной информации обо всех методах расширения, используемых при написании LINQ-выражений, рекомендуем Вам обратиться в MSDN.

В окончании этого раздела представим еще один способ формирования LINQ-выражений, который сочетает в себе способы запроса и метода. Создается такое LINQ-выражение довольно просто: написанный Вами LINQ-запрос необходимо заключить в скобки, а потом вызвать необходимый расширяющий метод. Довольно часто такой способ используется, когда необходимо применить функции агрегирования для нахождения: общего количества элементов (`Count()`), среднего арифметического значения элементов (`Average()`), максимального (`Max()`) и минимального элемента коллекции (`Min()`). В нашем

случае мы определим минимальный возраст студентов и самого младшего из них (Рисунок 6.14).

```
using System;
using System.Collections.Generic;
using System.Linq;
using static System.Console;

namespace SimpleProject
{
    // класс Student остался прежним
    class Program
    {
        static void Main(string[] args)
        {
            const double daysOfYear = 365.25;

            // код остался прежним

            WriteLine($"The current date:
                        {DateTime.Now.ToLongDateString()}\n");

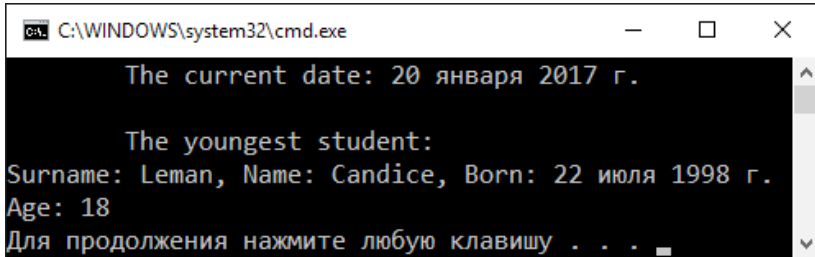
            WriteLine($"The youngest student:");
            var student = from s in students
                           where s.BirthDate ==
                               (from b in students
                                select b.BirthDate).Max()
                           select s;

            foreach (var item in student)
            {
                WriteLine(item);
            }

            var minAge = (from s in students
                           select s).Min(s => (DateTime.Now -
                                                  s.BirthDate).Days / daysOfYear);
        }
    }
}
```



```
        WriteLine($"Age: {(int)minAge}");  
    }  
}
```



The screenshot shows a Windows command prompt window titled "C:\WINDOWS\system32\cmd.exe". The output of a LINQ query is displayed in Russian. The first line is "The current date: 20 января 2017 г.". The second line is "The youngest student:". The third line is "Surname: Leman, Name: Candice, Born: 22 июля 1998 г.". The fourth line is "Age: 18". The fifth line is "Для продолжения нажмите любую клавишу . . .".

Рисунок 6.14. Применение смешанного запроса к объекту-коллекции

Домашнее задание

Разработать приложение «Тамагочи». Жизненный цикл персонажа — 1-2 минуты. Персонаж случайным образом выдаёт просьбы (но подряд одна и та же просьба не выдаётся). Просьбы могут быть следующие: Покормить, Погулять, Уложить спать, Полечить, Поиграть. Если просьбы не удовлетворяются трижды, персонаж «заболевает» и просит его полечить. В случае отказа — «умирает». Персонаж отображается в консольном окне при помощи **псевдографики**.

Диалог с персонажем осуществляется посредством вызова метода `Show()` класса `MessageBox` из пространства имен `System.Windows.Forms`. За получением подробной информации по работе с этим методом обратитесь к Вашему преподавателю или в MSDN.

Для решения этой задачи Вам понадобится класс `Timer` из пространства имен `System.Timers`, событие которого `Elapsed`, типа делегата `ElapsedEventHandler`, происходит через определенный интервал времени, который задан в свойстве `Interval`. Методы `Start()` и `Stop()` запускают и останавливают таймер, соответственно.

Вы также можете захотеть делать паузы в работе приложения, в этом случае можно вызвать метод `Sleep()` класса `Thread` из пространства имен `System.Threading`, передав в него необходимое количество миллисекунд.

Урок №9

Делегаты, события. LINQ

© Юрий Задерей.

© Компьютерная Академия «Шаг»

www.itstep.org

Все права на охраняемые авторским правом фото-, аудио- и видеопроизведения, фрагменты которых использованы в материале, принадлежат их законным владельцам. Фрагменты произведений используются в иллюстративных целях в объёме, оправданном поставленной задачей, в рамках учебного процесса и в учебных целях, в соответствии со ст. 1274 ч. 4 ГК РФ и ст. 21 и 23 Закона Украины «Про авторське право і суміжні права». Объём и способ цитируемых произведений соответствует принятым нормам, не наносит ущерба нормальному использованию объектов авторского права и не ущемляет законные интересы автора и правообладателей. Цитируемые фрагменты произведений на момент использования не могут быть заменены альтернативными, не охраняемыми авторским правом аналогами, и как таковые соответствуют критериям добросовестного использования и честного использования.

Все права защищены. Полное или частичное копирование материалов запрещено. Согласование использования произведений или их фрагментов производится с авторами и правообладателями. Согласованное использование материалов возможно только при указании источника.

Ответственность за несанкционированное копирование и коммерческое использование материалов определяется действующим законодательством Украины.