Microsoft .Net Framework and C# Programming Language

Lesson 9

Delegates. Events. LINQ

Contents

1. Delegates

The concept of a delegate

Before giving the definition of a delegate, I would like to remind you about the pointers to the functions used in C++. As you remember, a function pointer can be assigned with an address of a function whose return type and parameters are the same as the return type and parameters of the pointer itself. But there is also a certain structure in C++ called an array of pointers to a function that allows storing several addresses of the same functions when the above conditions are fulfilled. So delegates in C# are analogous to arrays of pointers to a function in C++, but at a higher level.

A delegate is a type-safe reference type that allows you to store references to methods of a given signature and use them to call these methods.

Base classes for delegates

System.Delegate

The base class for all delegate types is the `System.Delegate` class. It provides the basic functionality when working with delegates. Let's consider the main methods and properties of this class:

- `object Clone()` – creates an incomplete copy of the calling delegate.
- `Combine(Delegate, Delegate)` — concatenates the call lists of the specified delegates.

- `CreateDelegate(Type, MethodInfo)` — creates a delegate of the specified type.
- `object DynamicInvoke(params object[] args)` — dynamically calls the method represented by the current delegate.
- `Delegate[] GetInvocationList()` — returns the list of delegate calls.
- `int GetHashCode()` — returns the hash code of the current delegate.
- `Delegate Remove(Delegate, Delegate)` — removes the last occurrence of the call list of the delegate from the call list of the other delegate.
- `bool operator == (Delegate, Delegate)` — the overloaded operator 'strict equality', returns `true` if the delegates are equal, otherwise — `false`.
- `bool operator != (Delegate, Delegate)` — the overloaded operator 'unequal', returns `true` if the delegates are unequal, otherwise `false`.
- `MethodInfo Method { get; }` — a property that returns the method represented by the current delegate.
- `object Target { get; }` — a property that returns an instance of the class whose method calls the current delegate.

System.MulticastDelegate

In fact, all delegates are inherited from the `System.MulticastDelegate` class, which in turn inherits from `System.Delegate`. This class provides multicast delegates, that is, the ability to store references to an arbitrary number of methods. Multicasting is provided by an internal list, which

stores references to methods that correspond to the specified signature of the delegate.

The syntax of the delegate declaration

As you already understood from the above, delegates have basic classes, but direct inheritance from these classes is prohibited. An attempt to create an inheritance class from one of them will result in an error at the compilation stage (Figure 1.1).

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

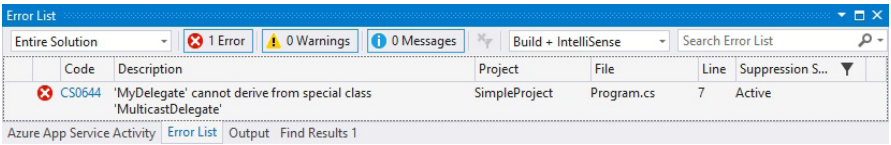

Figure 1.1. Error: the class cannot be derived from MulticastDelegate

To create delegates, use the `delegate` keyword and the general syntax of the delegate declaration is as follows:

```
[access modifier] delegate data _ type DelegateName
    (parameters);
```

Где:

- access modifier is any of the existing access modifiers in C# (optional part);
- `delegate` is a keyword for declaring a delegate;
- data type and parameters define the signature of methods whose references can be stored in this delegate (parameters may be missing).

We give examples of delegate declaration:

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

In this case, a public delegate named `IntDelegate` is declared, allowing you to store references to methods that take a real value as a parameter and return an integer value.

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

This example demonstrates the declaration of a closed `VoidDelegate` delegate, which can store references to methods that take an integer value and return nothing.

Where should I post the declaration of the delegates? In fact, when declaring a delegate, the compiler generates a class derived from `System.MulticastDelegate`, which means that the declared delegate automatically becomes a class. Therefore, delegates can be written in the form of a field of another class (nested class) or in the `namespace` block as a separate class, thus access to the delegate from the calling code will be different. The decision on the place of the delegate declaration should be taken independently, proceeding from the logic of your application.

Objectives and tasks of delegates

Using the same delegate, you can call different methods, and it does not matter whether these are instance or static methods, the main thing is that the signature of methods coincides with the signature of the delegate, and the methods are determined not at the compilation stage, but at the execution stage. That is, at the time of the creation of

the program, you do not know which method the user will call when executing your program, but you can provide him/her with this capability in the form of methods of a certain signature.

Due to this feature, delegates are widely used in various C# language constructs. Delegates:

- are the basis for the events (section 2 of the current lesson);
- are the basis for anonymous methods and lambda expressions (sections 3 and 4);
- are used to define callback methods;
- can be called in both synchronous and asynchronous mode, that is, in another stream simultaneously with some code (it will be considered in subsequent courses).

As an example of the use of delegates, let us cite the `Calculator` class, which contains methods that allow performing elementary arithmetic operations. These methods take two parameters of `double` type and return a value of `double` type. In addition to the class, declare a delegate whose data type and parameters are the same as the signature of methods in the Calculator class. The possible output of the program is shown in Figure 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 = ' ';
            // defining an arithmetic operator
            foreach (char item in expression)
            {
```

```
            if (item == '+' || item == '-' || item ==
                        '*' || item == '/')
            {
                sign = item;
                break;
            }
        }
        try
        {
           // getting operand values
            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; // group
                         // transformation of methods
                    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);
            }
        }
    }
}
```

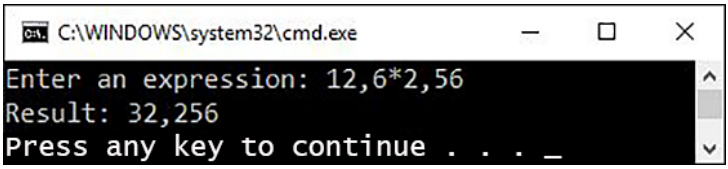

Figure 1.2. Example of using a delegate

It should be noted that when creating a delegate type object, the **name of the method** whose signature matches the signature of the delegate is passed to the constructor, regardless of whether this method is instantiated or static.

Also in the previous example, an easier way to initialize a delegate is shown, which only specifies the name of the method with the required signature, without explicitly invoking the delegate constructor. This possibility is called the **group transformation of methods**.

Regardless of the way the delegate is initialized, attempting to specify a method name with an invalid signature as a parameter will result in an error at the compilation stage (Figure 1.3).

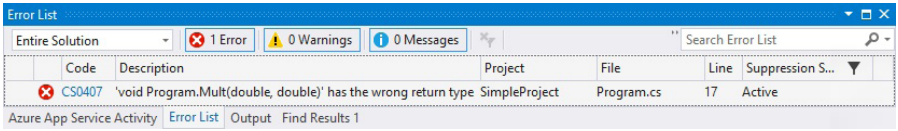

Figure 1.3. Error: invalid return type

```
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
        }
    }
}
```

Calling multiple methods through a delegate (multicasting)

In the previous section, using a delegate, one specific method was called, which does not fully reveal all the possibilities of using delegates. After all, by its nature, a delegate can contain many references to methods of a certain signature, thus in the delegate you can create a list of methods that will be called automatically when the delegate is called, this delegates option is called multicasting.

This ability of the delegate is provided due to the presence of the + and — overloaded operators for adding and deleting the specified method references from the list of calls, and, as a rule, these operators are applied in the abbreviated form: += and –=. Using these operators causes the call of

the `Combine(Delegate, Delegate)` and `Remove(Delegate, Delegate)` methods of the `System.Delegate` class, you can easily verify this by looking at the CIL code of your application using the ildasm.exe utility (Figure 1.4).

```
using System;

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

    // the Calculator class remains unchanged

    class Program
    {
        static void Main(string[] args)
        {
            Calculator calc = new Calculator();
            CalcDelegate delAll = null;
            CalcDelegate delDiv = calc.Div;

            delAll += delDiv; // adds to the list of calls
            delAll -= delDiv; // deletes from the list
                              // of calls

        }
    }
}
```

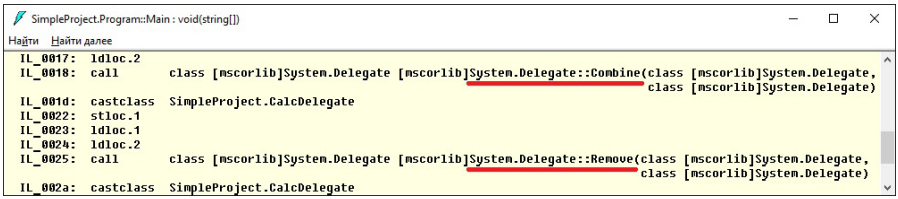

Figure 1.4. Overloading operators + and — in delegates

The use of multicasting is traditionally and more clearly demonstrated by the example of the delegate returning the `void` type. However, we will break the pattern and demonstrate multicasting, using the methods of the class from the previous section, only we will make the necessary changes for this.

If you form a chain of calls from the methods that return a type that is different from `void` (in our case `double`), then calling these methods through a delegate call, you will get the result that returns the last method from this list of calls (you can verify this yourself). In order to get the result of all the methods of the list, it is necessary to use the `GetInvocationList()` method, which returns an array of all methods, the references to which are contained in the current delegate (Figure 1.5).

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

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

    // the Calculator class remains unchanged

    class Program
    {
        static void Main(string[] args)
        {
            Calculator calc = new Calculator();
            CalcDelegate delAll = calc.Add; // group
                                // transformation of methods
            delAll += Calculator.Sub;
            delAll += calc.Mult;
            delAll += calc.Div;
```

```
            foreach (CalcDelegate item in delAll.
                    GetInvocationList()) // an array
                                         // of delegates
            {
                try
                {
                    // call
                    WriteLine($"Result: {item(5.7, 3.2)}");
                }

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

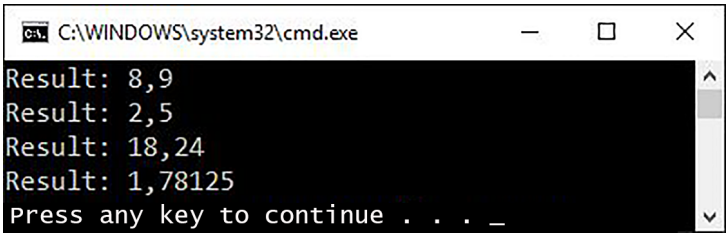

Figure 1.5. Calling multiple methods through a delegate

Creating generic delegates

In the lesson 8, the possibility of creating generic delegates was mentioned, in this section we will consider this issue in more detail.

Generic delegates allow you to safely call any methods that correspond to the generic form of the delegate that is

specified when you declare it. The general form of the generic delegate is as follows:

```
[access modigier] delegate retun _ type
            DelegateName<parameter _ types> (parameters);
```

Parameter types mean any data types that correspond to the types of method parameters when they are called.

An example of using generic delegates will continue the topic of calculations, but it will be much simpler than the previous one. We created a generic delegate and a class with three methods that allow us to get the sum of characters, integers and real numbers. We draw your attention to the explicit cast of the result in the method of adding two characters to the `char` type, which is necessary because of implicit casting `char` to `int` (Figure 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')}");
        }
    }
}
```

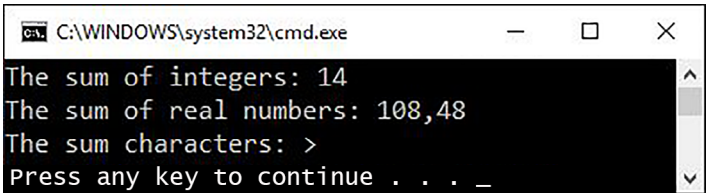

Figure 1.6. Using the generic delegate

As you noticed, in this example, instead of creating three delegates to call each method, it was sufficient to create one generic delegate.

There are a number of standard generic delegates in the System namespace, we'll look at some of them.

The `Action<T>` generic delegate provides a call of methods that do not return a value and can take up to 16 parameters. This delegate is required when calling the various methods of the standard classes of the System namespace. One such class is the `List<T>` collection, the `ForEach()` method of which provides a call of the method, specified as a parameter, for each list element. The signature of the called method corresponds to the `Action<T>` delegate. In the following example, using the `ForEach()` method, we call the `FullName()` method for each student from the list, thus obtaining its name and surname (Figure 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);
        }
    }
}
```

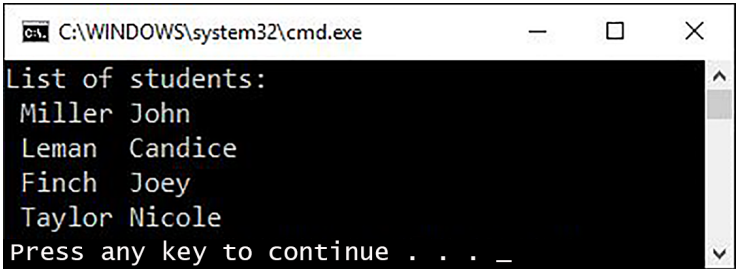

Figure 1.7. Using the Action<T> generic delegate

The Func<TResult> generic delegate provides a call of the methods that can take up to 16 parameters and return a value whose type is specified when the delegate is declared. The Func<TResult> delegate is also quite often used, for example most methods of the Enumerable class accept this delegate as a parameter. We will still consider these methods in the sixth section of the current lesson, and now we will consider the Select<TSource, TResult>() generic method, by which the resulting sequence of elements is generated by performing the conversion method for each element of the original sequence. The conversion method is just set using the Func<TResult> delegate. In the current example, we use the FullName() method to form a sequence of lines — the surname and name of the student, to work correctly the method needs to connect the System.Linq namespace (Figure 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);
            }
        }
    }
}
```

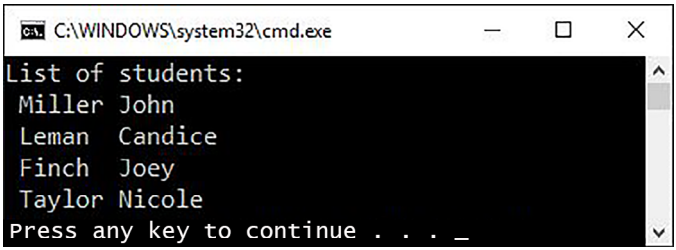

Figure 1.8. Using the Func<TResult> generic delegate

As you noticed, in this example, the `Select()` method returns a sequence that implements the `IEnumerable<string>` generic interface, also when calling this method, the types of the parameter (`group.Select<Student,string>(FullName)`) are not specified, they can be omitted, since they are defined automatically.

The following generic delegate, `Predicate<T>`, provides a call of the methods that take one parameter and return a Boolean value as the result — the result of checking the transferred parameter to the specified criteria. This delegate is used in the methods that perform actions on a specific condition, for example, methods of searching for the `Array` and `List<T>` collection. In the following example, using the `FindAll()` method, we get a list of all the students born in the spring, we define the search criteria using the `OnlySpring()` method (Figure 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);
            }
        }
    }
}
```

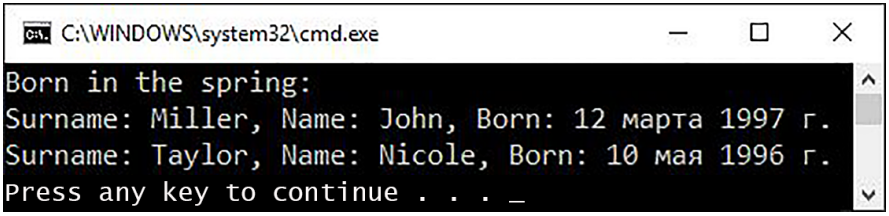

Figure 1.9. Using the Predicate<T> generic delegate

Another generic delegate, `Comparison<T>`, provides an invocation of the methods that take two parameters of the same type and return an integer value as the result: negative if the first parameter is less than the second, positive if the first parameter is greater than the second and zero if the parameter values are equal. In particular, this delegate can be used when sorting collection elements, using the overloaded method `Sort(Comparison<T>)`. Sort our collection

of students by date of birth, set the sorting parameters using the `SortBirthDate()` method (Figure 1.10).

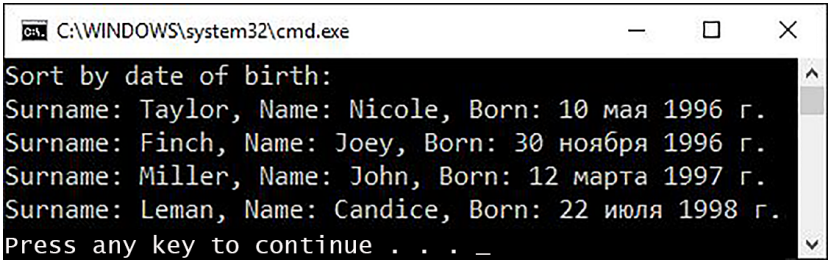

Figure 1.10. Using the Comparison<T> generic delegate

```
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);
            }
        }
    }
}
```

2. Events

The concept of an event

In order to better understand the events, we will describe a certain life situation. Each person fur sure has one event to happen once a year — Birthday. People usually accept congratulations on this day, and quite often a **certain** number of people are invited to celebrate this event at a festive table. Here the key word is 'certain, because you still celebrate in a 'narrow circle' and do not exactly invite someone you do not know at all on your holiday.

Similarly, when the program is running, certain events occur (text entry, pressing a button, etc.) and there are a certain number of objects that must somehow react to this event, therefore they register their method for handling this event. Just as you will not invite an unfamiliar person to your table, events in the program will not notify disinterested objects of their occurrence. When an event occurs in the program, all registered object method handlers are invoked to execute, the event as if says to them: "join the club!"

The syntax of the event declaration

When an event is declared, the `event` keyword is used and the general form of recording looks like this:

```
[access modifier] event DelegateName EventName;
```

As you can see, when the event is declared, the delegate associated with this event is indicated. Therefore, before declaring

an event, you must create a delegate that will contain references to the methods that are called when the event occurs.

In the following example, we will simulate an event that has already happened to you in the STEP Academy, an exam. In this example, the `Teacher` class calls the `examEvent` using the `Exam()` method. Thus, it notifies all the students, who expressed their interest in the exam through a subscription to the `examEvent` (operation +=). After that, the appropriate method is invoked for each element of the `Student` class, as a reaction to the `examEvent` (Figure 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");
        }
    }
}
```

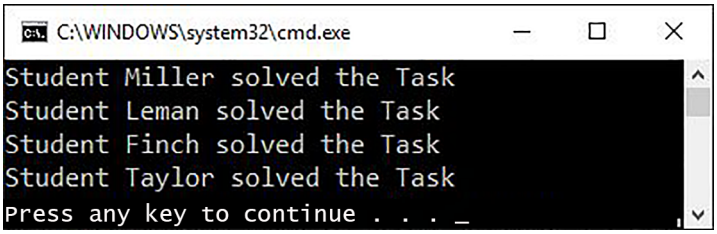

Figure 2.1. Example of using an event

The `Teacher` class encapsulates the whole mechanism of working with the event, this class is called the dispatcher class, the main purpose of which is to provide convenience when working with the event. This class contains the Exam() method, the so-called dispatch method, in which the existence of handler methods subscribed to this event is checked, and only if they exist, these methods are invoked. IntelliSense prompts us to remove this check, thereby simplifying the call of the `ExamDelegate` (Figure 2.2).

However, this should not be done, because a situation is possible when no method subscribes to your event and then an exceptional situation is generated when it is called (we suggest you check it yourself).

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

ExamDelegate Teacher.examEvent

Delegate invocation can be simplified.

Figure 2.2. Delegate call can be simplified

Necessity and features of the application of events

By this point, you may be asking yourself the question: Why to use events, if there are delegates? In this section you will receive an answer to your question.

In fact, the use of delegates in pure form involves a number of difficulties that are associated with ensuring encapsulation when using member variables of the delegate type. Let's demonstrate this in the previous example, making appropriate adjustments (Figure 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;
            }
            // direct invocation
            teacher.examEvent.Invoke("Overall rating 2!");

            teacher.examEvent = null; // new value

            teacher.Exam("Task"); // leads to nothing
        }
    }
}
```

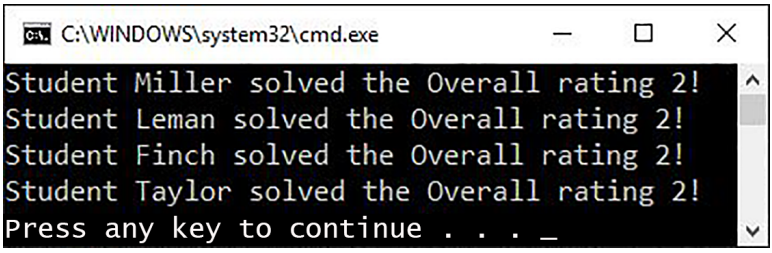

Figure 2.3. Problems of using delegates in their pure form

In the `Teacher` class, the examEvent field is now of the `ExamDelegate` type and has the `public` access modifier, so it is possible to access this field from the invoking code. In this case, the invoking code can both assign a new value to this field, and refer to the list of calls of the delegate directly (shown in the code). After assigning `null` to the examEvent field, the call list does not contain any references to methods and any access to this list will lead to nothing.

This problem can be solved by specifying the `examEvent` field of the `private` access modifier and creating additional methods for generating the call list. Therefore, the creators of the C# language, knowing how much time it will take to write the same code, created a special `event`.

In fact, the `event` keyword serves as the wrapper for the delegate with the `private` access modifier and additional

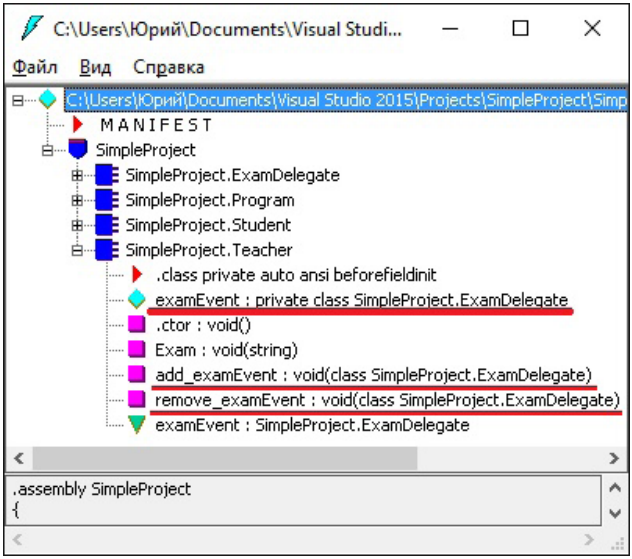

Figure 2.4. Teacher class after compilation

methods that provide interaction with this delegate and begin with add and remove to add and remove event handlers, respectively. This can be easily verified by using the ildasm.exe utility to look at the structure of any class containing `event` (Figure 2.4).

When creating events, you can do without creating a special delegate, and use the `EventHandler<T>` generic delegate, with T as the type, the child of the `EventArgs` class. In this case, event handlers should not return a value and must take two parameters: the first parameter is a reference to the object that generated this event, the second is an `EventArgs` object containing the necessary additional information about the event. We demonstrate the use of the `EventHandler` delegate in our example, to get an identical result, we created an additional class, `ExamEventArgs`, the child of `EventArgs`, which contains the Task property (Figure 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);
        }
    }
}
```

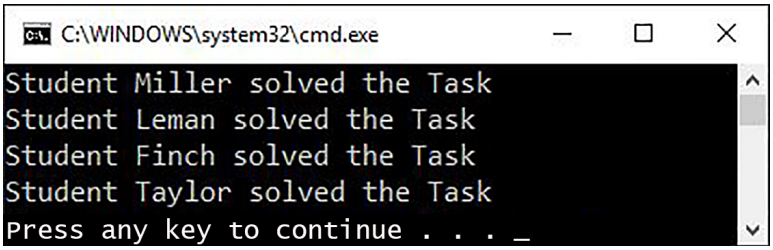

Figure 2.5. Using the EventHandler<T> generic delegate

Applying an event for a multicast delegate

The events were initially intended for the use of the multicast delegate, which was demonstrated in the previous examples. In fact, when performing an event subscription, only the += and –= operations are used to add and remove, respectively,

events do not support the = operation. The following example demonstrates this feature of events (Figure 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");
        }
    }
}
```

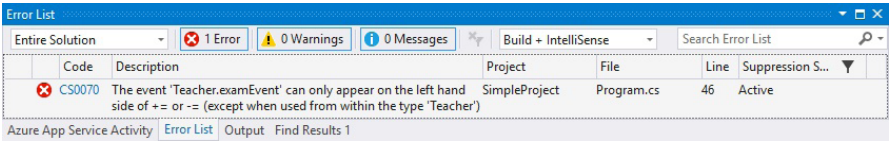

Figure 2.6. Error: only += and -= are used with the event

You probably noticed that after writing +=, IntelliSense prompts you to press Tab (Figure 2.7).

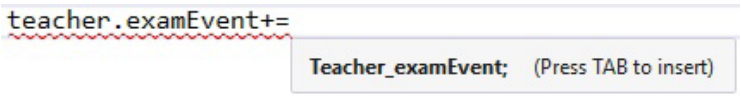

Figure 2.7. IntelliSense hint

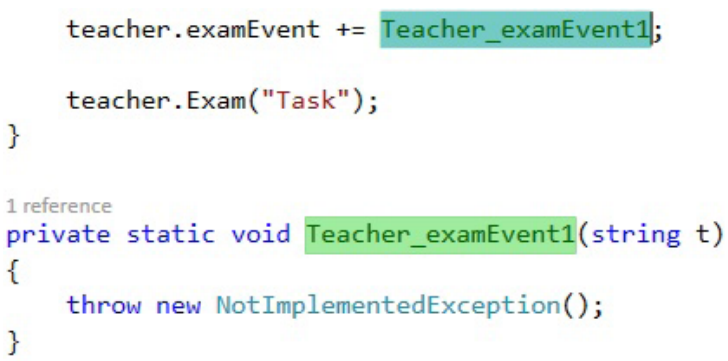

Figure 2.8. Automatic creation of the call method pattern

And if you give credence to its recommendations, then after clicking on the key, a pattern of the calling the method of the corresponding signature will automatically be created in the code, and the name of this method will be written after the += construction, (Figure 2.8).

Using event-based access tools

Usually the use of the standard += and –= operations fully satisfy the needs for managing the list of method calls. However, in this case the list is formed as the methods are added, and the call of these methods is carried out in a direct sequence from the beginning to the end of the list, which does not always correspond to the task. Therefore, if you need to change the order of calling methods, use the expanded `event` form, which contains methods for adding and removing the `add` and `remove` event handlers, the general form looks like this:

```
[access modifier]event DelegateName EventName
{
    add
    {
        //code of adding an event to the chain of events
    }
    remove
    {
        //event deletion code in the chain of events
    }
}
```

Thus, we write our own implementation of the methods, which, when the `event` form is shortened, is created automatically (Figure 2.4).

For an example of using event-based means of access, let's use the familiar example with passing the exam by students, but now the methods will be called randomly. To achieve this, a unique value, which acts as the key for the `SortedList<int, ExamDelegate>` collection when you add each method to this collection, is generated for each event handler in the add method. For this purpose, the _sortedEvents field of the `SortedList<int, ExamDelegate>` type was created in the Teacher class to sort the methods automatically when adding them to this collection. The removal of the event handler from the collection is done in the `remove` method based on the index assigned to this handler (Figure 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");
        }
    }
}
```

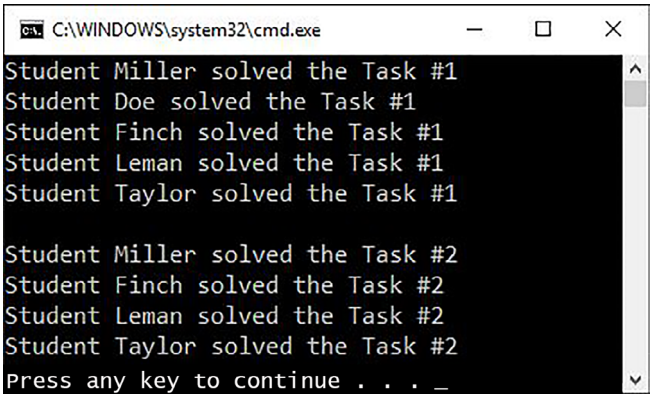

Figure 2.9. Using event-based access tools

3. Anonymous methods

In all the previous examples, the methods of classes were specified as the methods called through delegates or events. However, if these methods are called only with the help of delegates and are not called directly anywhere, then it makes sense to use a special block of code called an **anonymous method**. The general form is presented below.

```
delegate [(parameters)] {
    // runnable code
};
```

As you can see, when creating an anonymous method, the `delegate` keyword is used followed by the parameters that are optional, if you do not plan to use them in the runnable code. The runnable code is recorded in curly brackets and finally, after the closed curly bracket, there must be a semicolon, otherwise an error will be generated at the compilation stage. The following example demonstrates the use of anonymous methods (Figure 3.1).

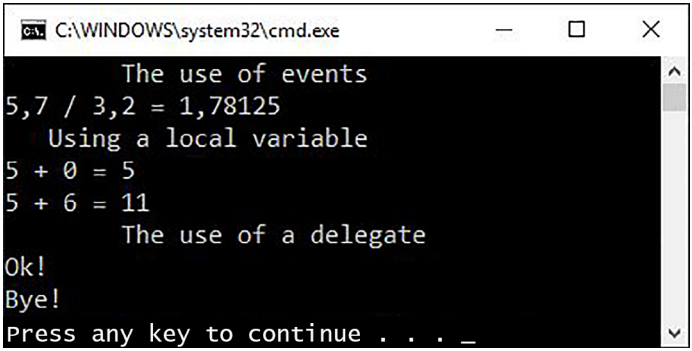

Figure 3.1. Using anonymous methods

```
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 Dispacher
    {
        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");
            Dispacher dispacher = new Dispacher();
```

```
            // anonymous method
            dispacher.eventDouble +=
                      delegate (double a, double b)
            {
                if (b != 0)
                {
                    return a / b;
                }

                throw new DivideByZeroException();
            };

            double n1 = 5.7, n2 = 3.2;
            WriteLine($"{n1} / {n2} =
               {dispacher.OnEventDouble(n1, n2)}"); // call
            WriteLine("Using a local variable");
            int number = 5;
            dispacher.eventVoid += delegate (int n) // anonymous
                                                    // method
            {
                WriteLine($"{number} + {n} = { number + n}");
            };

            dispacher.OnEventVoid(); // call
            dispacher.OnEventVoid(6);
            WriteLine("\tThe use of a delegate");
            AnonimDelegateVoid voidDel =
                  new AnonimDelegateVoid(delegate
                  { WriteLine("Ok!"); });
            // anonymous method
            voidDel += delegate { WriteLine("Bye!"); };
            voidDel(); // call
        }
    }
}
```

This example demonstrates the possibility of using the anonymous methods both with events and with delegates. We

draw your attention to the possibility of using local variables of method where these anonymous methods are defined. As you noticed, anonymous methods can be used even when declaring delegates.

The use of anonymous methods does not affect the code execution speed, their main advantage is the simplification of the code, the absence of the need to create additional methods.

4. Lambda expressions

While anonymous methods appeared in the version of C# 2.0, then already the C# 3.0 version presented a construction that allows to write the anonymous methods themselves in a more abbreviated form — lambda expression.

The lambda expression consists of two parts separated by a lambda operator (=>). The left side of the lambda expression must contain a list of parameters, and the right side contains the runnable code (the body of the lambda expression) that affects the parameters specified on the left.

```
(parameters) => { // code that uses parameters  }
```

For a better understanding of lambda expressions, we will demonstrate a comparative analysis between them and anonymous methods.

The anonymous method record in general form looks like this:

```
delegate (parameters) { // runnable code};
```

Here is a general form of the lambda expression:

```
(parameters) => { // runnable code}
```

As you can see, the lambda expression consists of the same parts as the anonymous method but looks more compact.

Implementation of the method body as an expression

Depending on the number of operators in the body of lambda expressions, they are divided into single and block.

As the name implies, single lambda expressions consist of one operator, and block ones — of the set of operators that must be enclosed in curly brackets.

The use of any lambda expression can be divided into three stages:

- Declaring a delegate of the necessary signature.
- Creating an instance of this delegate and initializing it by a compatible lambda expression.
- Accessing to the delegate instance, in which the lambda expression is calculated.

To demonstrate this, we rewrite an example of the previous section, but using lambda expressions and small changes (Figure 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 Dispacher
    {
        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");
            Dispacher dispacher = new Dispacher();

            // explicit typing
            dispacher.eventDouble += (double a, double b) =>
            {
                if (b != 0)
                {
                    return a / b;
                }
                throw new DivideByZeroException();
            };

            double n1 = 5.7, n2 = 3.2;
            WriteLine($"{n1} / {n2} =
                {dispacher.OnEventDouble(n1, n2)}"); // call

            WriteLine("\tSingle lambda expression");

            int number1 = 5, number2 = 6;
```

```
            dispacher.eventInt += n => number1 + n;
            // implicit typing

            WriteLine($"{number1} + {number2} =
               {dispacher.OnEventInt(number2)}"); // call

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

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

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

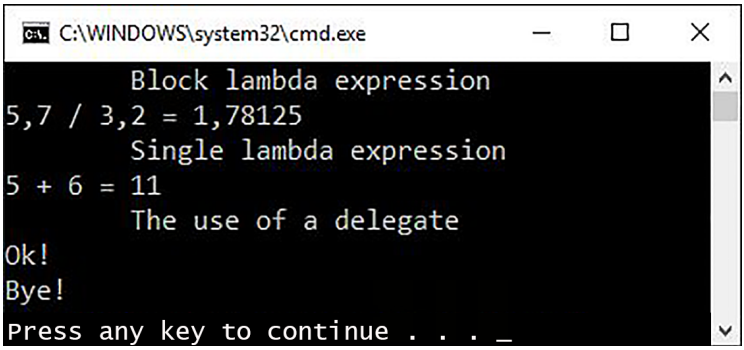

Figure 4.1. Using lambda expressions

We draw your attention to the explicit and implicit typing of parameters of lambda expressions. With implicit typing, the parameter type of the lambda expressions is determined based on the signature of the delegate, if one parameter is used, then the brackets of the parameter list can be omitted.

The value of lambda expressions is that they can be applied anywhere where it is possible to use delegates. For example, in some methods of the `List` collection, you need to use the `Predicate<T>` delegate, that is, the use of the method of a specific signature is implied, but now there is no need to create an anonymous method, because you can use a lambda expression. Let's change the familiar example from the first section, now the lambda expression is applied in the `FindAll()` method (Figure 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);
            }
        }
    }
}
```

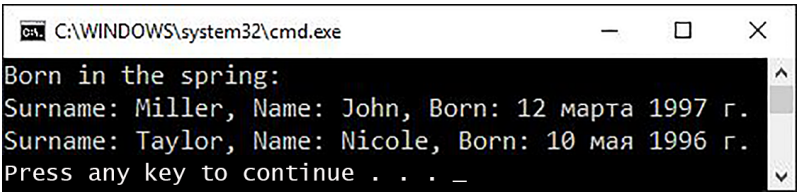

Figure 4.2. Using lambda expressions in the collection method

Open the generated CIL-code of our application with the ildasm.exe utility, and make sure that in fact the lambda expression is translated into the corresponding delegate (Figure 4.3).

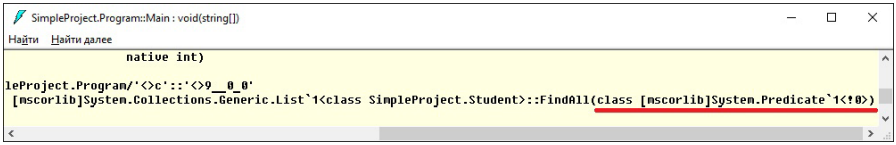

Figure 4.3. The CIL code of the FindAll() method

In the version 6.0 of the C# language, it became possible to use lambda expressions to define the body of methods and properties, which is indicated to the right of the lambda expression. As a demonstration, let's give a simple example (Figure 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);
            }
        }
    }
}
```

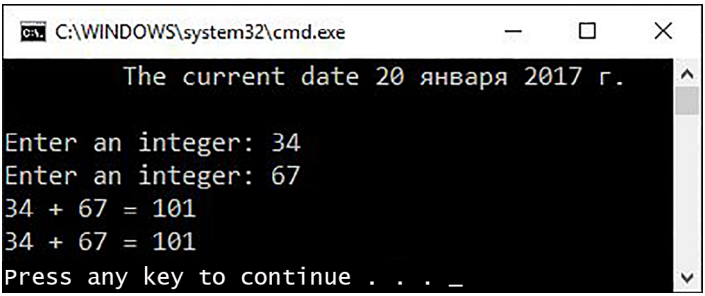

Figure 4.4. Using lambda expressions to define s body of methods

Another important application of lambda expressions is their use in writing LINQ queries, which we will cover in the sixth section of the current lesson.

5. Extension methods

Imagine the following situation: when you write your application, you use a library developed by another programmer, naturally this library contains classes with some functionality. At some point in time, you understand that for example, in one of these classes, an additional method or methods are very necessary. If it was you who developed this library, you would have to make changes to the source code and recompile it, but since you do not have the source code, there is no simple solution. One way to solve this problem is the extension methods, which allow you to add additional functionality to precompiled types without creating a derived class.

Extension methods must be in a static class, and the methods themselves must also be static. The difference between the extension methods and the usual static methods is that the `this` keyword should be specified as the modifier for the first parameter in the extension methods. The data type of the first parameter must match the data type to which this method will apply.

Consider the following example, in which we will create the `NumberWords()` extension method for a `string` type, which, as you know, is a sealed class, which does not allow you to use it as a base class. This method determines the number of words in a string; in order to correctly work with it, it is necessary to remove "extra" spaces. Using the `Trim()` method of the `String` class, the leading and trailing spaces of

the string are removed, repeated spaces are replaced by single ones using the `Replace()` method of the `Regex` class (regular expressions will be discussed in the next lesson).

```
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()}");
        }
    }
}
```

The possible output of the program is shown in Figure 5.1.

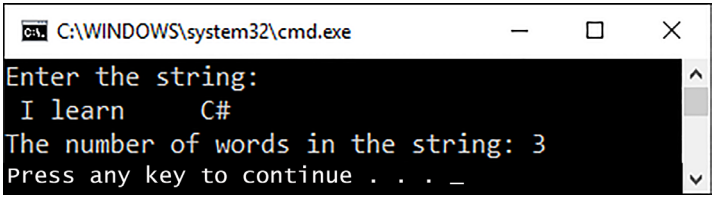

Figure 5.1. Example of an extension method

6. LINQ to Object

The role of LINQ

Most of the developed programs perform interaction with some data storages, from elementary arrays to databases. At the same time, the ways of obtaining information from these storages are naturally different. The ability to work in the same way with different data sources has emerged thanks to the development of the Language Integrated Query (LINQ), which appeared in the .NET 3.5 version.

As the name suggests, LINQ is the query language, that is, using LINQ queries you specify what you **want** to retrieve, and the language itself resolves **how** to do this with respect to the particular data source that the object that implements the `IEnumerable` interface should be.

In this lesson, we will cover only one part of LINQ — LINQ to Object, which allows you to obtain information from various collections. However, there are several more varieties of LINQ that you will learn in the process of further learning the C# language:

- LINQ to DataSet is used to retrieve data from the DataSet;
- LINQ to XML is used to retrieve information from XML files;
- LINQ to Sql is used to retrieve data from MS SQL Server;
- LINQ to Entities is used when working with the Entity Framework technology;

- Parallel LINQ(PLINQ) is used to perform parallel queries.

To execute LINQ queries, you must connect the System. Linq namespace.

Investigation of LINQ query operations

A LINQ query is a set of instructions for retrieving data from a specified source written using different LINQ statements. In the simplest form, the LINQ query contains three statements and looks like this.

```
result = from variable name in data source
            select variable name;
```

The LINQ query always starts with the `from` operator, which declares a range variable that represents each element in the source data source and has the type of that element, and the data source itself is specified after the `in` operator. The `select` operator returns the values got when all operators between `from` and `select` are executed. Let's illustrate this with an example (Figure 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();
        }
    }
}
```

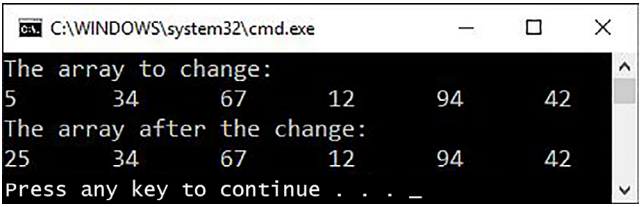

Figure 6.1. The simplest example of a LINQ query

The previous query simply returns all the elements of the array of integers without any changes. These kinds of queries are not of special value and can be used only as an example, however, the same example shows an important feature of LINQ queries — deferred **execution**. In fact, the execution of the LINQ-queries does not occur at the moment of its creation

and the assignment of its results to a variable, but when accessing this variable in order to obtain these results (in this case, `foreach`). Due to this approach, each result of executing the same LINQ-query in relation to the same collection will always contain the most current data.

If there is a need to immediately execute a LINQ query, then you should call one of the methods that casts the LINQ request to one of the collection types, for example, `ToList()` or `ToArray()`.

Quite often the LINQ-query is used to get a certain sample from the entire collection, in such queries it is necessary to use the `where` operator followed by a logical condition applied to each element of the data source, and only if the specified condition is true, the `where` operator returns the corresponding element. In order to demonstrate the possibility of using the `where` clause, we give the following example, the purpose of which is to get a set of only even elements of an integer array (Figure 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();
        }
    }
}
```

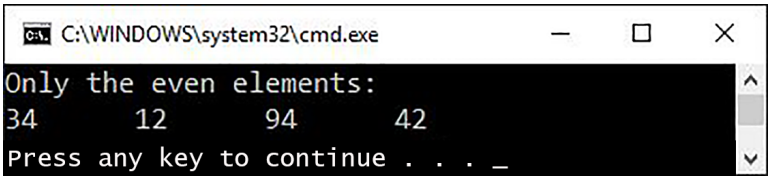

Figure 6.2. Using the where clause in the LINQ query

To order to sort the results in ascending or descending order in the LINQ query, the `orderby` operator is used, specifying the field, by the values of which the sorting will occur, and the sorting direction. Sorting in ascending order is applied by default and the `ascending` value is optional; if you want to sort the items in `descending` order, descending value is required. Add the sorting of the results in descending order to the previous example (Figure 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();
        }
    }
}
```

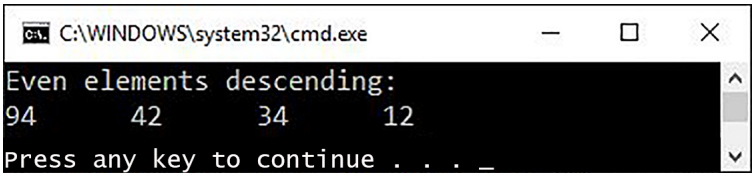

Figure 6.3. Сортировка результатов в запросе LINQ

The `group` operator allows you to obtain a subset of data based on the criterion specified after the `by` keyword. In the following example, we write a LINQ query, in which we will form groups of elements based on the digit which these elements end with (Figure 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();
            }
        }
    }
}
```

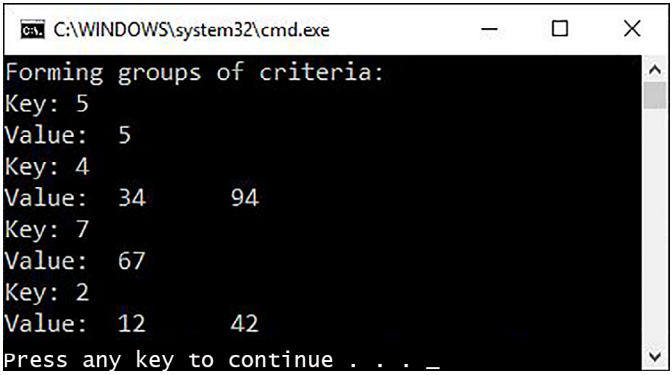

Figure 6.4. Grouping results in a LINQ query by criterion

I want to draw your attention to the fact that as a result of the grouping, objects realizing the `IGrouping<T, K>` interface are formed. So in the previous example we got groups of `IGrouping<int, int>` type, where the key and values are the `int` type. Because the grouped objects are a list in the list, you must use the nested `foreach` loop to get the value of each group element.

The `into` keyword is used to create an identifier in which it is necessary to save the temporary results of the `group`, `join` or `select` operators, if after them you need to perform additional LINQ-query operations. In the following example, you only need to obtain the groups from the previous example, in which the number of elements is greater than one (Figure 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();
            }
        }
    }
}
```

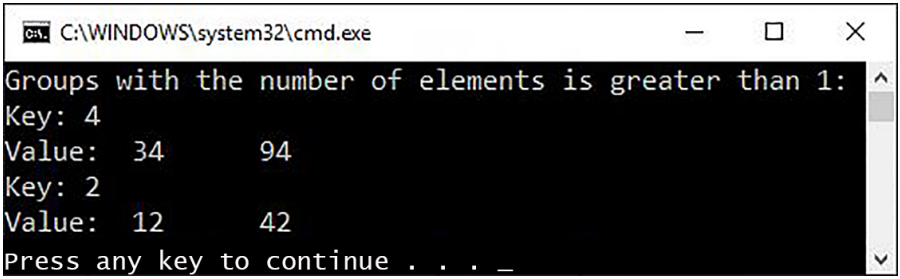

Figure 6.5. Using the temporary request identifier

The `let` operator is intended to create a new range variable as a temporary storage of intermediate data. In the following example, this operator is used to create an array of words for each string from the original array of strings. The received array of words is used in the additional `from` clause, which is allowed when more than one collection is needed in the LINQ-query. The `where` clause is used to determine the number of characters in each word in the intermediate array (Figure 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($"\t{item}");
            }

        }
    }
}
```

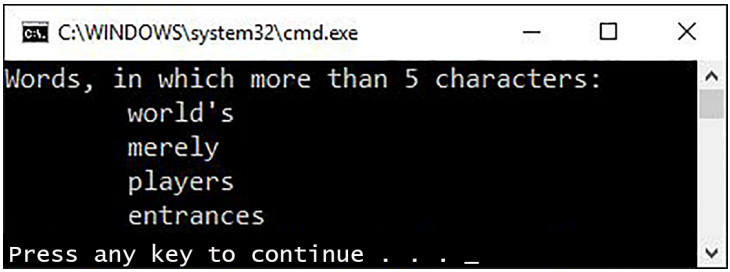

Figure 6.6. Using a temporary variable in a LINQ query

The `join` operator allows you to join elements from two data sources based on the equality of the corresponding keys in each element. The keys are checked only for equality, which is provided by the `equals` operator. The following example demonstrates the use of one of the connection types, a group connection. This type of connection forms a sequence of results, linking the elements based on a certain condition, resulting in a collection of collections that can form the basis for the nested query. In the example in Figure 6.7, the elements from the `List<Student>` collection are joined to the corresponding elements of the `List<Group>` collection based on the group identifier.

```
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}");
                }
            }
        }
}
```

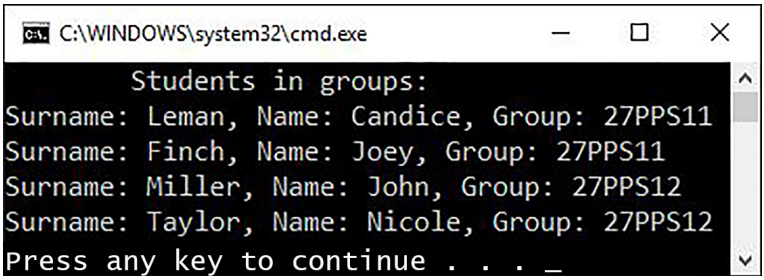

Figure 6.7. Using a group connection

When displaying the results, the `First()` extension method is used to get the name of the group, this method returns the first element of the collection that satisfies the specified condition (match the group identifier).

The ways of constructing a LINQ expression using different LINQ operators discussed in this section have a common name for the **query syntax**.

Returning the result of the LINQ query. Anonymous types

As you already noticed in the examples of the previous section, the LINQ query return type is usually an object that implements the `IEnumerable<T>` interface, but it is not always easy to understand what type hides behind T, and incorrectly defining the return type will result in an error at the compilation stage. Therefore, to make life easier for programmers,

it was decided to use **implicitly typed variables** as a return of LINQ queries.

The essence of implicit typing is the following: if the variable type is unknown at the time of compilation, then when declaring a variable, instead of specifying the data type, the `var` keyword is used, and the specific data type was dynamically defined based on the value specified when this variable was initialized (Figure 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()}");

        }
    }
}
```

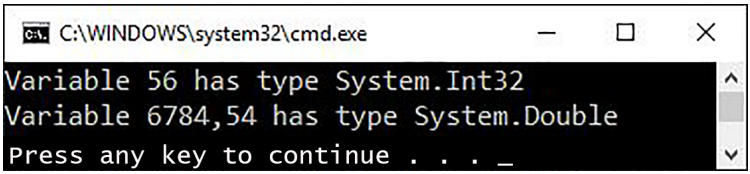

Figure 6.8. Implicitly typed variables

Declaring an implicitly typed variable without specifying the initial value is invalid, since the compiler will not be able to define the data type of this variable, for the same reason, you cannot assign `null` as the initial value (Figure 6.9).

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

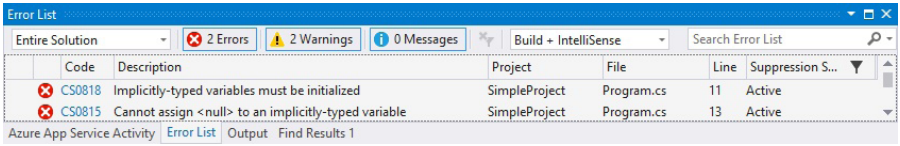

Figure 6.9. Initialization errors for implicitly typed variables

Usually when creating a class, you assume that in the future it will be actively used and perform certain functionality. Therefore, in addition to the fields in your class, there are usually properties and special methods, as well as events and constructors. However, in cases where there is no need for the class to perform special functionality, and your class will be used only in the current application, there is the possibility of declaring an **anonymous type**.

When an anonymous type is created, the `new` keyword is used, after which the type name is not specified (hence

the name), and the object initialization syntax is used when declaring properties. A unique name for this type will be generated automatically by the compiler, however, until the compilation time, this type is unknown, so it is declared using the `var` keyword.

Anonymous type is inherited directly from `object`, therefore all base methods are implemented in it, we will demonstrate this by example (Figure 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()}");

        }
    }
}
```

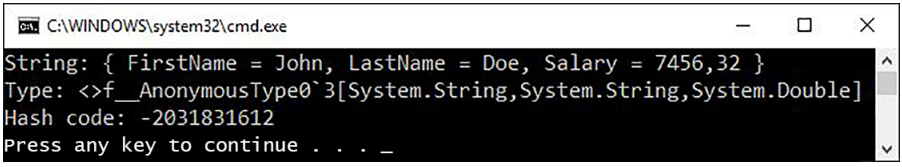

Figure 6.10. Creating an anonymous type

Anonymous types are not full-fledged classes, because they have a number of significant limitations:

- the properties of anonymous types are read-only;
- there is no possibility to create methods, events, etc.;
- anonymous types do not support inheritance, since they are implicitly sealed.

As you already know, widespread use of anonymous types in ordinary code is limited and not even welcomed. However, anonymous types are indispensable when you need to create a new class dynamically when executing LINQ queries.

In the following example, anonymous types are used as the result of an internal `join`, which is created using the join operator. In this connection type, each element of the `List<Student>` collection corresponds to each element of the `List<Group>` collection, if the first collection does not match the second collection, it is not added to the result set (student Joey Finch) (Figure 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);
            }
        }
    }
}
```

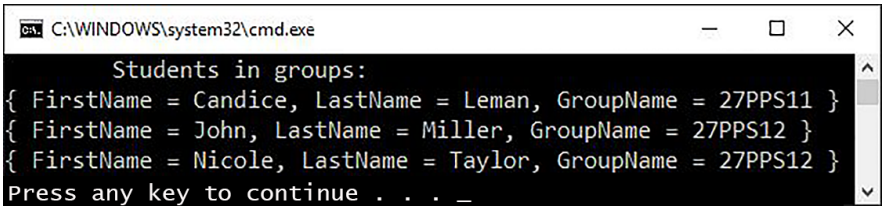

Figure 6.11. Using anonymous types in a LINQ query

Applying LINQ Queries to Collection Objects

When creating LINQ queries against collection objects, you can create a LINQ expression using the query syntax discussed earlier (Figure 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);
            }
        }
    }
}
```

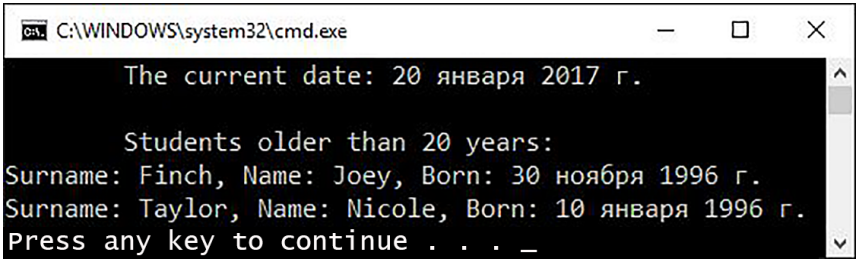

Figure 6.12. Applying the query syntax to a collection object

In this code, you should pay attention to the constant `daysOfYear` — the number of days in the year equal to 365.25 (that's why every fourth year is a leap year). It must be used as a divisor to get the number of years when dividing the value of the Days property of a `TimeSpan` structure, since this property returns the number of full days in a given time interval.

However, there is another way to create a LINQ expression using different extensibility methods that accept a delegate as a parameter. Some of these methods have already

been considered in the previous sections, when standard delegates were applied. Usually the delegate is represented as a lambda expression, which affects every element of the collection, thereby ensuring the selection of information from the collection.

To get the result, the necessary methods are arrayed in a chain of calls, this way of forming a LINQ expression is called the **method syntax**. Let's apply this method to the previous example and get the same result (Figure 6.13).

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

namespace SimpleProject
{
    // the Student class remains the same

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

            // the code remains the same
            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);
            }
        }
    }
}
```

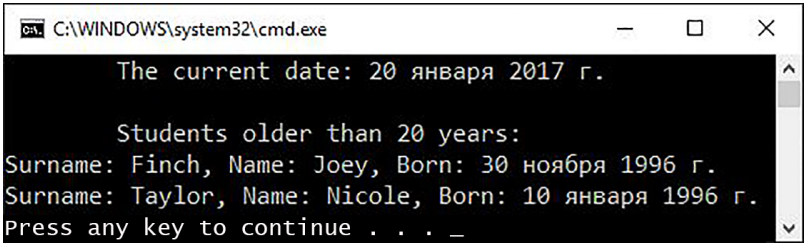

Figure 6.13. Applying the method syntax to a collection object

The `Where()` method returns a collection of elements whose value satisfies the specified lambda expression. The `Select()` method converts each element of the collection to the required form.

Unfortunately, it is quite problematic to give examples of the use of all the extension methods in one lesson, so for more information about all the extension methods used in writing LINQ expressions, we recommend that you contact MSDN.

At the end of this section, we present yet another way of generating LINQ expressions, which combines the methods of the query and the method. This LINQ expression is created quite simply: the LINQ query you wrote should be enclosed in parentheses, and then call the necessary extension method. Quite often this method is used when you need to apply the aggregation functions to find: the total number of elements

(`Count()`), the average of the elements (`Average()`), the maximum (`Max()`) and the minimum element of the collection (`Min()`). In our case, we define the minimum age of students and the youngest of them (Figure 6.14).

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

namespace SimpleProject
{
    // the Student remains the same
    class Program
    {
        static void Main(string[] args)
        {
            const double daysOfYear = 365.25;

            // the code remains the same
            WriteLine($"\tThe current date:
                      {DateTime.Now.ToLongDateString()}\n");

            WriteLine($"\tThe 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}");
        }
    }
}
```

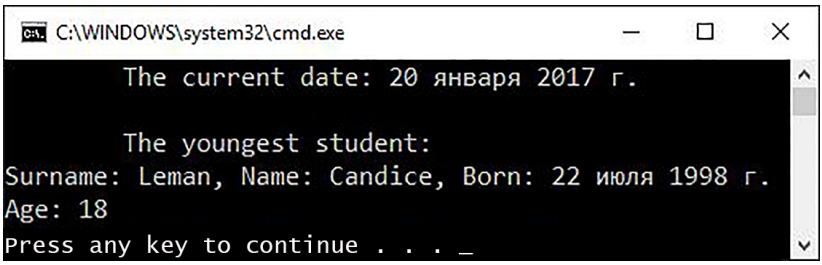

Figure 6.14. Applying a mixed query to a collection object

Homework assignment

1. Develop the application 'Tamagotchi' (Gigapets). The life cycle of the character is 1-2 minutes. The character randomly issues queries (but the same query is not given in a row). Queries can be the following: Feed, Walk, Sleep, Treat, Play. If the queries are not met three times, the character "falls ill" and asks to treat. In case of refusal, it "dies". The character is displayed in the console window using **pseudo-graphics**.

Dialogue with the character is performed by calling the `Show()` method of the `MessageBox` class from the `System.Windows.Forms` namespace. For detailed information on how to use this method, contact your teacher or MSDN.

To solve this problem, you will need a `Timer` class from the `System.Timers` namespace, whose `Elapsed` event of the `ElapsedEventHandler` delegate type occurs after a certain amount of time that is specified in the `Interval` property. The `Start()` and `Stop()` methods start and stop the timer, respectively.

You may also want to pause the application, and in this case you can call the `Sleep()` method of the `Thread` class from the `System.Threading namespace`, passing the required number of milliseconds to it.

Lesson 9
Delegates. Events. LINQ