

**top**

КОМПЬЮТЕРНАЯ  
АКАДЕМИЯ

# ПЛАТФОРМА MICROSOFT.NET И ЯЗЫК ПРОГРАММИРОВАНИЯ

# C#

# Урок №6

## Интерфейсы

### Содержание

1. Понятие интерфейса.....	3
2. Синтаксис объявления интерфейсов.....	4
3. Примеры создания интерфейсов.....	6
4. Интерфейсные ссылки.....	7
5. Интерфейсные свойства и индексаторы.....	15
6. Наследование интерфейсов .....	20
7. Проблемы сокрытия имен при наследовании интерфейсов.....	22
8. Анализ стандартных интерфейсов .....	28
IEnumerable .....	30
IComparable.....	33
IComparer.....	36
ICloneable.....	40
9. Домашнее задание.....	48

# 1. Понятие интерфейса

---

Интерфейс можно сравнить с неким обязательством, которое берет на себя класс или структура в случае реализации данного интерфейса. Интерфейс содержит сигнатуры свойств, методов или событий, при этом все они должны быть обязательно реализованы в классе или структуре.

В самом интерфейсе нельзя прописывать реализацию его членов, также нельзя создать экземпляр интерфейса, но можно создать ссылку на интерфейс. Синтаксис интерфейса можно сравнить с абстрактным классом, у которого все методы являются абстрактными. Однако у интерфейса не может быть конструкторов, он не может содержать поля или перегрузки операторов. Все методы интерфейса по умолчанию являются `public` (модификатор доступа нельзя изменить при реализации метода в классе-наследнике) и их нельзя объявить как `virtual` или `static`.

Интерфейс является альтернативой абстрактному классу, которая позволяет осуществить множественное наследование в языке C# — класс или структура могут реализовывать любое количество интерфейсов.

Также следует обратить Ваше внимание на различие между базовыми классами и интерфейсами в описании процесса наследования, говорят, что класс наследуется от базового класса, но класс реализует интерфейс.

## 2. Синтаксис объявления интерфейсов

Объявление интерфейса осуществляется с помощью служебного слова `interface` и имеет следующий вид:

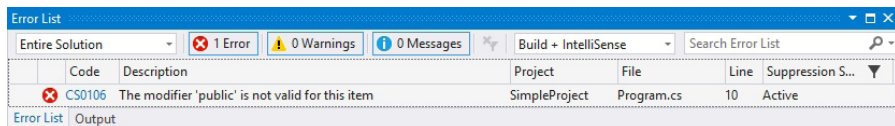
```
[модификатор доступа] interface имя_интерфейса
{
    //члены интерфейса
}
```

Название интерфейса обычно начинается с заглавной буквы I, что является общепринятой практикой, и свидетельствует о хорошем стиле программирования. Например, основной работой исследователя является исследования и изобретения, тогда интерфейс исследователя можно описать следующим образом:

```
public interface IResearcher
{
    void Investigate();
    void Invent();
}
```

При создании интерфейса следует помнить, что нельзя указывать модификаторы доступа у его членов. Попытка явно указать любой модификатор доступа (даже `public`), приведет к ошибке на этапе компиляции (Рисунок 2.1).

```
public interface IResearcher
{
    public void Investigate();
    void Invent();
}
```



**Рисунок 2.1.** Ошибка: модификатор доступа не допустим для этого элемента

### 3. Примеры создания интерфейсов

В качестве примера, рассмотрим интерфейс `IWorker`, который описывает некоего работника.

```
public interface IWorker
{
    event EventHandler WorkEnded;
    bool IsWorking { get; }
    string Work();
}
```

Соответственно приведенному интерфейсу каждый класс, который реализует данный интерфейс, должен реализовать метод `Work()`, описывающий выполнение некоторой специфической работы, свойство `IsWorking` указывающее занят ли работник и событие `WorkEnded` (будут изучены в одном из последующих уроков), оповещающее о завершении выполнения работы.

Следующий интерфейс `IManager` обязует каждый класс, который его реализует, иметь свойство `ListOfWorkers`, которое позволяет записывать и получать список работников, и реализацию методов `Organize()`, `MakeBudget()` и `Control()`.

```
interface IManager
{
    List<IWorker> ListOfWorkers { get; set; }
    void Organize();
    void MakeBudget();
    void Control();
}
```

## 4. Интерфейсные ссылки

В C# можно создать ссылку на интерфейс, и присвоить ей экземпляр любого класса, который реализует данный интерфейс. Через интерфейсные ссылки можно вызывать только методы данного интерфейса, при этом выполнится версия метода, которая реализована в конкретном классе. Если понадобится вызвать метод, который не является частью данного интерфейса, тогда необходимо привести интерфейсную ссылку к соответствующему типу.

Для приведения к типу интерфейса можно использовать явное приведение, однако в таком случае мы не застрахованы от попытки приведения к типу, который не поддерживает необходимый интерфейс. В этом случае будет сгенерирована исключительная ситуация, перехватить которую мы можем, используя инструкцию `try-catch` (более подробно этот механизм будет рассмотрен в последующем уроке).

Однако более надежный способ приведения к определенному интерфейсу заключается в использовании операторов `is` или `as`, которые были изучены Вами в уроке №4. Использование этих операторов с интерфейсами, аналогично их использованию для приведения ссылки к определенному классу.

В следующем примере мы объединим полученные знания для создания модели магазина, в котором есть директор, продавец, кассир и кладовщик. Все они являются людьми (класс `Human`) и сотрудниками данного магазина (интерфейс `IEmployee`), но директор руководит

сотрудниками (интерфейс `IManager`), которые выполняют свою работу по-разному (интерфейс `IWorker`).

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

namespace SimpleProject
{
    abstract class Human
    {
        public string FirstName { get; set; }
        public string LastName { get; set; }
        public DateTime BirthDate { get; set; }
        public override string ToString()
        {
            return $"\\nФамилия: {LastName}
                    Имя: {FirstName} Дата рождения:
                    {BirthDate.ToLongDateString()}";
        }
    }
    abstract class Employee : Human
    {
        public string Position { get; set; }
        public double Salary { get; set; }

        public override string ToString()
        {
            return base.ToString() + $"\\nДолжность:
                    {Position} Заработная плата: {Salary} $";
        }
    }
    interface IWorker
    {
        bool IsWorking { get; }
        string Work();
    }
}
```



```

interface IManager
{
    List<IWorker> ListOfWorkers { get; set; }
    void Organize();
    void MakeBudget();
    void Control();
}

class Director : Employee, IManager
{
    public List<IWorker> ListOfWorkers { get; set; }

    public void Control()
    {
        WriteLine("Контролирую работу!");
    }

    public void MakeBudget()
    {
        WriteLine("Формирую бюджет!");
    }

    public void Organize()
    {
        WriteLine("Организирую работу!");
    }
}

class Seller : Employee, IWorker
{
    bool isWorking = true;

    public bool IsWorking
    {
        get
        {
            return isWorking;
        }
    }
}

```

```
        public string Work()
        {
            return "Продаю товар!";
        }
    }

    class Cashier : Employee, IWorker
    {
        bool isWorking = true;
        public bool IsWorking
        {
            get
            {
                return isWorking;
            }
        }

        public string Work()
        {
            return «Принимаю оплату за товар!»;
        }
    }

    class Storekeeper : Employee, IWorker
    {
        bool isWorking = true;
        public bool IsWorking
        {
            get
            {
                return isWorking;
            }
        }

        public string Work()
        {
            return "Учитываю товар!";
        }
    }
}
```

```

    }
}

class Program
{
    static void Main(string[] args)
    {
        Director director = new Director { LastName =
            "Doe", FirstName = "John", BirthDate =
            new DateTime(1998, 10, 12), Position =
            "Директор", Salary = 12000 };

        IWorker seller = new Seller { LastName =
            "Beam", FirstName = "Jim", BirthDate =
            new DateTime(1956, 5, 23), Position =
            "Продавец", Salary = 3780 };

        if (seller is Employee)
            WriteLine($"Заработная плата продавца:
                {(seller as Employee).Salary}");
            // приведение интерфейсной ссылки
            // к классу Employee

        director.ListOfWorkers = new List<IWorker> {
            seller, new Cashier { LastName = "Smith",
            FirstName = "Nicole",
            BirthDate = new DateTime(1956, 5, 23),
            Position = "Кассир", Salary = 3780 },
            new Storekeeper { LastName = "Ross",
            FirstName = "Bob", BirthDate =
            new DateTime(1956, 5, 23),
            Position = "Кладовщик", Salary = 4500 }
        };

        WriteLine(director);
        if (director is IManager) // использование
                                    // оператора is
    }
}

```

```

    {
        director.Control();
    }

    foreach (IWorker item in director.
        ListOfWorkers)
    {
        WriteLine(item);

        if (item.IsWorking)
        {
            WriteLine(item.Work());
        }
    }
}
}
}

```

Результат работы программы представлен на рисунке 4.1.

```

C:\WINDOWS\system32\cmd.exe
Заработная плата продавца: 3780
Фамилия: Doe Имя: John Дата рождения: 12 октября 1998 г.
Должность: Директор Заработная плата: 12000 $
Контролирую работу!
Фамилия: Bean Имя: Jim Дата рождения: 23 мая 1956 г.
Должность: Продавец Заработная плата: 3780 $
Продаю товар!
Фамилия: Smith Имя: Nicole Дата рождения: 23 мая 1956 г.
Должность: Кассир Заработная плата: 3780 $
Принимаю оплату за товар!
Фамилия: Ross Имя: Bob Дата рождения: 23 мая 1956 г.
Должность: Кладовщик Заработная плата: 4500 $
Учитываю товар!
Для продолжения нажмите любую клавишу . . .

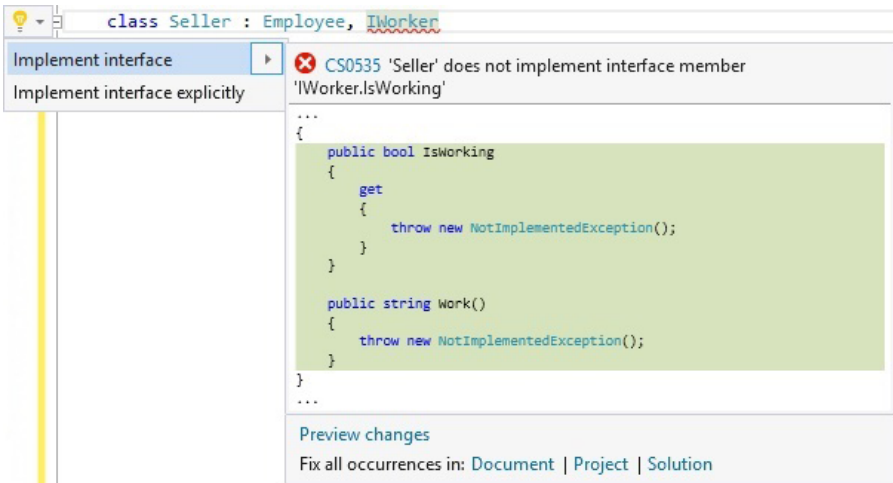
```

**Рисунок 4.1.** Использование интерфейсных ссылок

Как Вы могли заметить в предыдущем примере, синтаксис наследования интерфейсов аналогичен наследованию класса. После имени класса наследника ставится двоеточие и через запятую перечисляются интерфейсы, которые должен реализовать данный класс (все методы каждого из интерфейсов). Причем если класс наследуется еще и от базового класса, тогда этот класс должен обязательно располагаться сразу после двоеточия, а за ним через запятую все интерфейсы.

Процесс реализации классом или структурой определенного интерфейса значительно упрощен благодаря еще одной специальной возможности Visual Studio 2015, которая позволяет получить шаблоны всех необходимых членов данного интерфейса.

Рассмотрим данный процесс на примере класса `Seller`. После одиночного нажатия левой клавиши мыши на имени интерфейса `IWorker` в строке наследования, в левой



**Рисунок 4.2.** Неявная реализация интерфейса `IWorker`

части этой строки появляется кнопка с изображением лампочки. При нажатии на которую, появляется список возможных действий. Нажав на один из пунктов этого списка, Вы можете создать неявную (*Implement interface*) или явную (*Implement interface explicitly*) реализацию данного интерфейса. Пример неявной реализации интерфейса представлен на рисунке 4.2.

Внешний вид сгенерированного средой разработки кода (представлен в окне предварительного просмотра) Вам уже знаком, подобное Вы могли видеть в уроке №4 при реализации абстрактного класса. Как и тогда, необходимую реализацию членов интерфейса Вы должны написать самостоятельно, заменив в каждом из членов строку генерации исключительной ситуации (будет рассмотрена в последующем уроке).

До текущего момента мы работали с неявными реализациями интерфейсов, с явными реализациями Вы ознакомитесь в одном из разделов данного урока.

## 5. Интерфейсные свойства и индексаторы

Свойства в интерфейсах являются аналогом полей в классах и могут быть представлены (как и обычные свойства) в трех вариантах: для чтения и записи, только для чтения и только для записи.

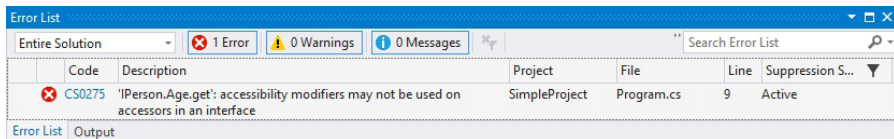
```
interface IPerson
{
    string LastName { get; set; }

    int Age { get; }

    string Gender { set; }
}
```

Как Вы могли заметить, синтаксис свойств в интерфейсе похож на синтаксис объявления автоматических свойств, однако между ними существует большая разница. Свойства в интерфейсе не являются автоматически реализованными, а только представляют спецификацию свойства, которое будут реализовано в классе или структуре. При этом запрещается указывать модификаторы доступа для аксессоров `get` и `set` даже `public` (Рисунок 5.1).

```
interface IPerson
{
    int Age { public get; }
}
```



**Рисунок 5.1.** Ошибка: нельзя использовать модификаторы доступа в аксессорах интерфейса

Как Вы уже знаете из предыдущего урока, в языке C# существуют индексаторы, которые являются специальным видом свойств, поэтому синтаксис их объявления в интерфейсе, подобен синтаксису объявления свойств и имеет следующий вид.

```
тип_элемента this [тип_данных index]
{
    get;
    set;
}
```

Также как и свойства, индексаторы могут иметь один аксессор `get` или `set` только для чтения или только для записи, соответственно. Приведем пример использования индексаторов в интерфейсе (Рисунок 5.2).

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

namespace SimpleProject
{
    interface IIndexer
    {
        string this[int index]
        {
            get;
            set;
        }
    }
}
```



```

    }
    string this[string index]
    {
        get;
    }
}

enum Numbers { one, two, three, four, five };

class IndexerClass : IIndexer
{
    string[] _names = new string[5];

    public string this[int index]
    {
        get
        {
            return _names[index];
        }
        set
        {
            _names[index] = value;
        }
    }

    public string this[string index]
    {
        get
        {
            if (Enum.IsDefined(typeof(Numbers),
                                index))
                return _names[(int)Enum.
                               Parse(typeof(Numbers), index)];
            else
                return «»;
        }
    }
}

```

```

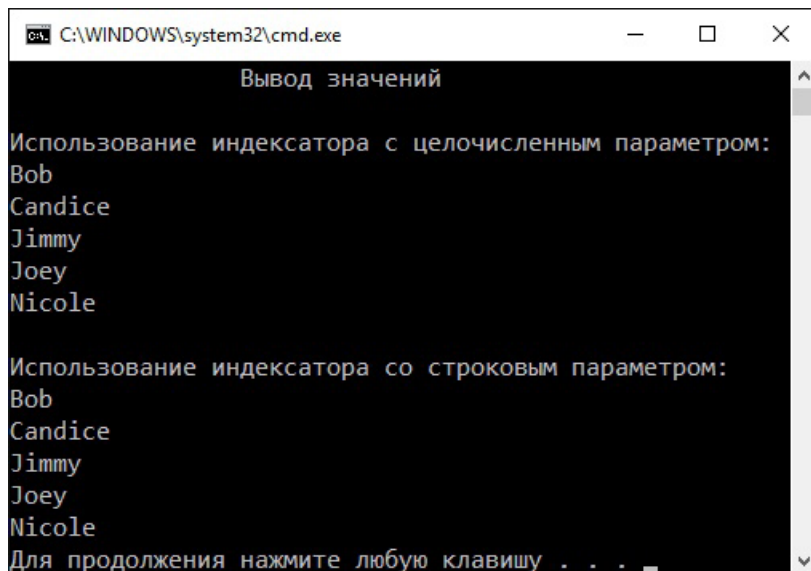
public IndexerClass()
{
    // запись значений, используя индексатор
    // с целочисленным параметром
    this[0] = "Bob";
    this[1] = "Candice";
    this[2] = "Jimmy";
    this[3] = "Joey";
    this[4] = "Nicole";
}
}
class Program
{
    static void Main(string[] args)
    {
        IndexerClass indexerClass =
            new IndexerClass();

        WriteLine("\t\tВывод значений\n");
        WriteLine(«Использование индексатора
                    с целочисленным параметром:»);
        for (int i = 0; i < 5; i++)
        {
            WriteLine(indexerClass[i]);
        }

        WriteLine(«\nИспользование индексатора
                    со строковым параметром:»);
        foreach (string item in Enum.
                    GetNames(typeof(Numbers)))
        {
            WriteLine(indexerClass[item]);
        }
    }
}

```

Результат работы программы.



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

Вывод значений

Использование индекса с целочисленным параметром:
Bob
Candice
Jimmy
Joey
Nicole

Использование индекса со строковым параметром:
Bob
Candice
Jimmy
Joey
Nicole
Для продолжения нажмите любую клавишу . . .
```

Рисунок 5.2. Использование интерфейсных индексов

## 6. Наследование интерфейсов

Также как и класс, интерфейс может наследоваться от других интерфейсов. При этом Вы можете создать иерархию интерфейсов, где производные интерфейсы будут расширять свою функциональность за счет других, ранее созданных интерфейсов.

Если класс реализует интерфейс, который в свою очередь наследует другой интерфейс, то в этом классе должны быть реализованы методы всех интерфейсов по иерархии.

```
interface IA
{
    string A1(int n);
}

interface IB
{
    int B1(int n);
    void B2();
}

interface IC : IA, IB
{
    void C1(int n);
}

class InherInterface : IC
{
    public string A1(int n)
```

```
{  
    throw new NotImplementedException();  
}  
  
public int B1(int n)  
{  
    throw new NotImplementedException();  
}  
  
public void B2()  
{  
    throw new NotImplementedException();  
}  
  
public void C1(int n)  
{  
    throw new NotImplementedException();  
}  
}
```

## 7. Проблемы сокрытия имен при наследовании интерфейсов

Иногда нужно объявить класс, который реализует несколько интерфейсов. И может возникнуть такая ситуация, когда у этих интерфейсов есть методы с одинаковым названием и сигнатурой. Например:

```
interface IA
{
    void Show();
}

interface IB
{
    void Show();
}

interface IC
{
    void Show();
}
```

Реализация этих интерфейсов классом может выглядеть следующим образом:

```
using static System.Console;
namespace SimpleProject
{
    public class ImplicitRealization : IA, IB, IC
    {
        public void Show()
```

```

    {
        WriteLine("Hello, implicit realization!");
    }
}

```

И хотя данный код не приводит к ошибке, возникает вопрос: метод, какого интерфейса будет вызван, когда вызовется метод экземпляра класса `ImplicitRealization`? Хуже того, вызов метода `Show()` при помощи ссылок на различные интерфейсы всегда будет приводить к выполнению одних и тех же действий, что, в большинстве случаев, не будет соответствовать ожидаемому результату (Рисунок 7.1).

```

namespace SimpleProject
{
    class Program
    {
        static void Main(string[] args)
        {
            ImplicitRealization er = new
ImplicitRealization();
            er.Show();

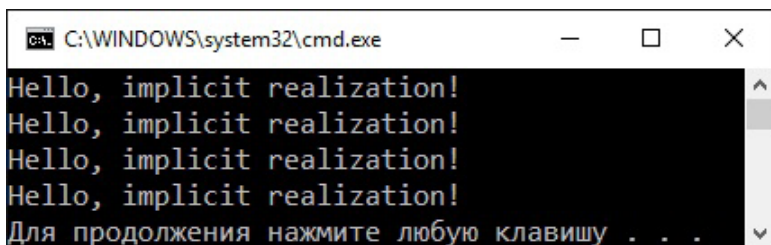
            IA iA = new ImplicitRealization();
            iA.Show();

            IB iB = new ImplicitRealization();
            iB.Show();

            IC iC = new ImplicitRealization();
            iC.Show();
        }
    }
}

```

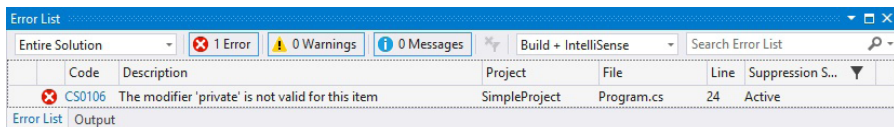
Результат работы программы.



**Рисунок 7.1.** Вызов методов класса без явной реализации

Поэтому в такой ситуации следует осуществить явную реализацию интерфейсов в классе, для облегчения этого процесса предлагаем Вам обратиться к рисунку 4.2, только сейчас необходимо выбрать пункт «Implement interface explicitly». При этом обращаем Ваше внимание на отсутствие модификатора доступа у данных реализаций, так как по умолчанию у них модификатор `private`, указание модификатора доступа явно, приведет к ошибке на этапе компиляции (Рисунок 7.2).

```
class ExplicitRealization : IA
{
    private void IA.Show()
    {
        throw new NotImplementedException();
    }
}
```



**Рисунок 7.2.** Ошибка: модификатор доступа не допустим для этого элемента



Так как у явных реализаций интерфейсов модификатор доступа неявно `private`, то вызвать их через экземпляр класса невозможно. Однако выход из сложившейся ситуации существует, для этого необходимо явным образом привести экземпляр класса к типу того интерфейса, метод которого требуется вызвать в данном случае. При явной реализации интерфейсов допускается метод одного из интерфейсов реализовать неявно, именно он будет вызываться при вызове этого метода через экземпляр класса.

Можно также осуществить вызов метода через ссылку на интерфейс, при этом однозначно вызывается метод соответствующего интерфейса. Покажем все вышеперечисленное на примере (Рисунок 7.3).

```
using static System.Console;

namespace SimpleProject
{
    interface IA
    {
        void Show();
    }

    interface IB
    {
        void Show();
    }

    interface IC
    {
        void Show();
    }

    class ExplicitRealization : IA, IB, IC
    {
```

```

void IA.Show()
{
    WriteLine("Interface IA");
}

void IB.Show()
{
    WriteLine("Interface IB");
}

public void Show()
{
    WriteLine("Interface IC");
}
}

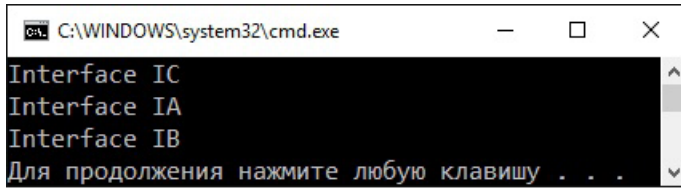
class Program
{
    static void Main(string[] args)
    {
        ExplicitRealization er =
            new ExplicitRealization();
        er.Show(); // вызов метода интерфейса IC неявно

        ((IA)er).Show(); // вызов метода интерфейса IA
                        // явно

        IB iB = new ExplicitRealization();
        iB.Show(); // вызов метода интерфейса IB через
                // ссылку
    }
}

```

Применение явной реализации интерфейсов полезно еще и в ситуации, когда Вы хотите скрыть реализацию



**Рисунок 7.3.** Явная реализация интерфейсов

какого-то метода на уровне объекта. Вызвать такой метод можно будет только через явное приведение к правильному интерфейсу или через ссылку на этот интерфейс.

## 8. Анализ стандартных интерфейсов

В языке C# существует большое количество стандартных интерфейсов, некоторые из которых мы рассмотрим в данном разделе, а с остальными Вы продолжите знакомиться по мере изучения языка.

Для демонстрации примеров создадим три класса.

Класс `StudentCard`, который описывает студенческий билет.

```
class StudentCard
{
    public int Number { get; set; }
    public string Series { get; set; }

    public override string ToString()
    {
        return $"Студенческий билет: {Series} {Number}";
    }
}
```

Класс `Student`, описывающий студента.

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

И класс `Auditory`, в котором находится массив типа `Student` — студенты, присутствующие в аудитории, и метод `Sort()`, вызывающий одноименный метод у класса `Array`.

```

class Auditory : IEnumerable
{
    Student[] students =
    {
        new Student {
            FirstName = "John",
            LastName = "Miller",
            BirthDate = new DateTime(1997, 3, 12),
            StudentCard = new StudentCard { Number=189356,
                                             Series="AB" }
        },
        new Student {
            FirstName = "Candice",
            LastName = "Leman",
            BirthDate = new DateTime(1998, 7, 22),
            StudentCard = new StudentCard { Number=345185,
                                             Series="XA" }
        },
        new Student {
            FirstName = "Joey",
            LastName = "Finch",
            BirthDate = new DateTime(1996, 11, 30),
            StudentCard = new StudentCard { Number=258322,
                                             Series="AA" }
        },
        new Student {
            FirstName = "Nicole",
            LastName = "Taylor",
            BirthDate = new DateTime(1996, 5, 10),
            StudentCard = new StudentCard { Number=513484,
                                             Series="AA" }
        }
    };
    public void Sort()
    {
        Array.Sort(students);
    }
}

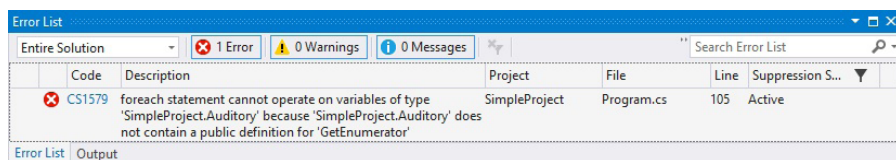
```

Первое, что мы хотим сделать в нашей программе — это вывести список всех студентов в аудитории с использованием `foreach`, что на самом деле приведет к ошибке на этапе компиляции (Рисунок 8.1).

```
class Program
{
    static void Main(string[] args)
    {
        Auditory auditory = new Auditory();

        WriteLine("\n+++++++ список студентов +++++\n");

        foreach (Student student in auditory)
        {
            WriteLine(student);
        }
    }
}
```



**Рисунок 8.1.** Ошибка: оператор `foreach` не может работать с типом `Auditory`

Чтобы устранить эту ошибку необходимо реализовать интерфейс `IEnumerable` в классе `Auditory`.

## `IEnumerable`

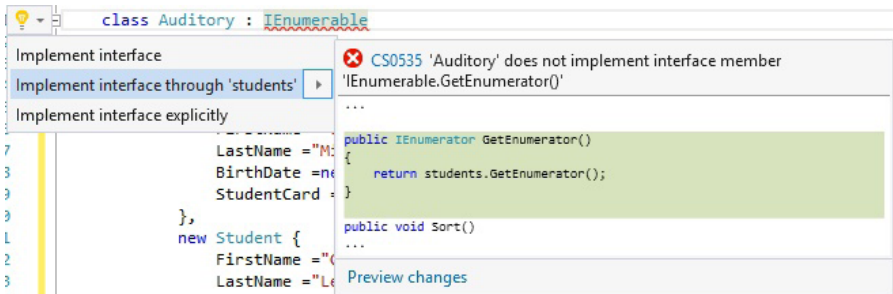
Интерфейс `IEnumerable` позволяет, при помощи перечислителя, осуществить перебор всех элементов не обобщенной коллекции.

Для реализации данного интерфейса, в классе `Auditory` необходимо реализовать метод `GetEnumerator()`, который возвращает интерфейс `IEnumerator`.

В интерфейсе `IEnumerator` находятся три члена:

- свойство `Current`, которое позволяет получить текущий элемент коллекции, возвращает `object`;
- метод `MoveNext()` — перемещает перечислитель по элементам коллекции, возвращает `false`, если достигли конца коллекции, иначе `true`;
- метод `Reset()` — устанавливает перечислитель в начало коллекции.

Получается, что нам необходимо реализовать в классе `Auditory` еще и все члены интерфейса `IEnumerator`. Однако мы воспользуемся тем фактом, что данный интерфейс уже реализован в классе `Array` — базовом для всех массивов. Такое решение является одним из трех вариантов, которые предлагает нам среда разработки (Рисунок 8.2).



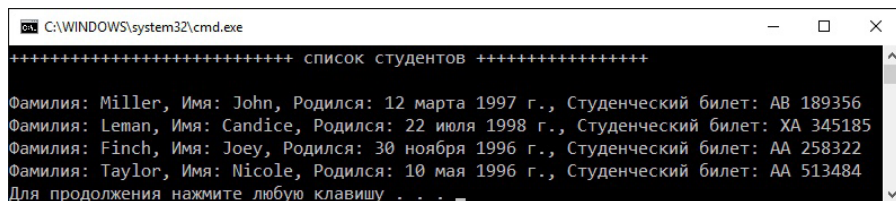
**Рисунок 8.2.** Вариант реализации интерфейса `IEnumerator`

Мы возьмем именно эту реализацию, однако интерфейс `IEnumerator` реализуем явным образом для того

чтобы метод `GetEnumerator()` нельзя было вызвать через экземпляр класса `Auditory`, но `foreach` при этом работал корректно.

```
class Auditory : IEnumerable
{
    // остальной код остался прежним
    IEnumerator IEnumerable.GetEnumerator()
    {
        return students.GetEnumerator();
    }
}
```

А вывод списка студентов с использованием оператора `foreach` будет выглядеть следующим образом (Рисунок 8.3).



The screenshot shows a Windows command prompt window with the title bar "C:\WINDOWS\system32\cmd.exe". The output of a program is displayed, starting with a separator line of asterisks and the text "список студентов". It lists four students with their family names, first names, birth dates, and student IDs. The list ends with a separator line and a prompt for the user to press any key to continue.

```

+++++++ список студентов ++++++
Фамилия: Miller, Имя: John, Родился: 12 марта 1997 г., Студенческий билет: АВ 189356
Фамилия: Leman, Имя: Candice, Родился: 22 июля 1998 г., Студенческий билет: ХА 345185
Фамилия: Finch, Имя: Joey, Родился: 30 ноября 1996 г., Студенческий билет: АА 258322
Фамилия: Taylor, Имя: Nicole, Родился: 10 мая 1996 г., Студенческий билет: АА 513484
Для продолжения нажмите любую клавишу . . .
```

**Рисунок 8.3.** Вывод списка студентов с использованием оператора `foreach`

Нашим следующим действием в программе будет попытка вызвать метод `Sort()` класса `Auditory`, что приведет к ошибке на этапе выполнения (Рисунок 8.4).

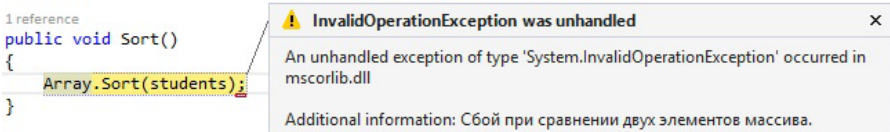
```
class Program
{
    static void Main(string[] args)
    {
        // остальной код остался прежним
        WriteLine("\n+++++++ список студентов +++++\n");
        auditory.Sort();
    }
}
```



```

foreach (Student student in auditory)
{
    WriteLine(student);
}
}

```



**Рисунок 8.4.** Ошибка: невозможно сравнить два элемента массива

Данная ошибка произошла, потому что методу `Sort()` класса `Array` «не известно», как сравнивать между собой экземпляры класса `Student` для их сортировки. Чтобы метод `Sort()` работал корректно, необходимо реализовать интерфейс `IComparable` в классе `Student`.

## IComparable

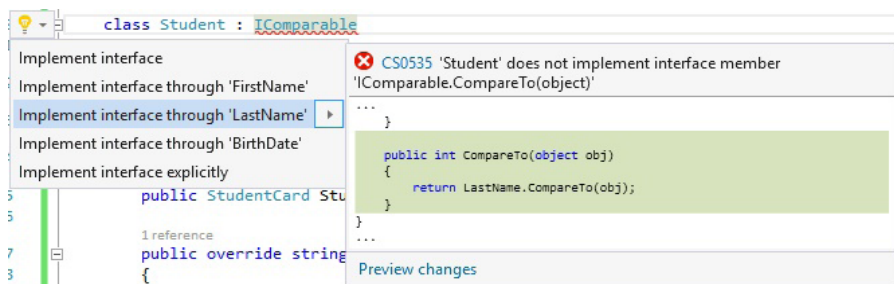
В интерфейсе `IComparable` содержится единственный метод `CompareTo(object)`, реализация которого в классе определяет, как необходимо сравнивать экземпляры этого класса между собой.

Метод `CompareTo(object)` принимает параметр типа `object`, а возвращает целочисленное значение:

- отрицательное, если текущий экземпляр класса предшествует передаваемому параметру при сортировке;
- нулевое, если текущий экземпляр класса находится в той же позиции что и передаваемый параметр;

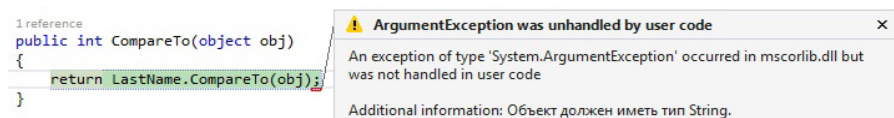
- положительное, если текущий экземпляр класса располагается за передаваемым параметром.

При реализации классом `Student` интерфейса `IComparable` среда разработки предлагает нам все возможные варианты с учетом свойств текущего класса. В данном случае мы хотим осуществить сортировку студентов по фамилии (Рисунок 8.5).



**Рисунок 8.5.** Варианты реализации интерфейса `IComparable`

Однако если мы будем использовать данную реализацию в том виде, который предлагает Visual Studio, мы получим ошибку на этапе выполнения (Рисунок 8.6).



**Рисунок 8.6.** Ошибка на этапе выполнения

Причина данной ошибки — невозможность сравнить между собой свойство `LastName` типа `string` с параметром типа `object`. Вывод: не надо всегда доверять Visual Studio.

Мы напишем собственную реализацию метода `CompareTo(object)`, используя операторы `is` и `as` для проверки и приведения передаваемого параметра.

```

class Student : IComparable
{
    // остальной код остался прежним
    public int CompareTo(object obj)
    {
        if (obj is Student)
        {
            return LastName.CompareTo((obj as Student).
LastName);
        }
        throw new NotImplementedException();
    }
}

```

Хотелось бы обратить Ваше внимание на наличие в методе `CompareTo(object)` строки `throw new NotImplementedException()`. Дело в том, что по требованиям языка C# в случае если метод возвращает значение, то значение должна возвращать каждая часть этого метода. То есть в нашем случае, нужно было написать оператор `else`, а в нем оператор `return`, но какое значение необходимо бы было вернуть в случае несоответствия переданного объекта классу `Student`? Ответить сложно, а вот генерирование исключительной ситуации в этом случае является корректным и не приводит к ошибке на этапе компиляции.

Результаты сортировки списка студентов представлены на рисунке 8.7.

```

C:\WINDOWS\system32\cmd.exe
+++++++ сортировка по фамилии ++++++
Фамилия: Finch, Имя: Joey, Родился: 30 ноября 1996 г., Студенческий билет: AA 258322
Фамилия: Leman, Имя: Candice, Родился: 22 июля 1998 г., Студенческий билет: XA 345185
Фамилия: Miller, Имя: John, Родился: 12 марта 1997 г., Студенческий билет: AB 189356
Фамилия: Taylor, Имя: Nicole, Родился: 10 мая 1996 г., Студенческий билет: AA 513484
Для продолжения нажмите любую клавишу . . .

```

**Рисунок 8.7.** Сортировка студентов по фамилии

Метод `Sort()` класса `Auditory` позволяет отсортировать студентов по фамилии, но как поступить, если нам необходимо осуществить сортировку еще по каким-либо критериям.

Вы скажете, что нужно написать перегруженный метод, и будете правы. Но существует один нюанс: в нашем классе `Student` уже реализован метод `CompareTo(object)` и именно его будет использовать метод `Sort()` класса `Array` при сортировке.

Однако у метода `Sort()` класса `Array` существуют 17 перегруженных версий, и одна из них принимает в качестве второго аргумента ссылку на интерфейс `IComparer`.

## IComparer

Интерфейс `IComparer` содержит метод `Compare(object, object)`, реализация которого в классе позволяет сравнивать между собой переданные объекты.

Метод `Compare(object, object)` принимает два параметра типа `object`, а возвращает целочисленное значение:

- отрицательное, если первый параметр меньше второго параметра;
- нулевое, если первый параметр равен второму параметру;
- положительное, если первый параметр больше второго параметра.

Для того чтобы вызвать перегруженный метод `Sort()` класса `Array`, необходимо создать класс `DateComparer` реализующий интерфейс `IComparer`, а в нем реализовать метод `Compare(object, object)`. В данном случае мы

сравниваем объекты по дате рождения, пользуясь тем, что в структуре `DateTime` существует статический метод `Compare(DateTime t1, DateTime t2)`.

```
class DateComparer : IComparer
{
    public int Compare(object x, object y)
    {
        if (x is Student && y is Student)
        {
            return DateTime.Compare((x as Student).
                BirthDate, (y as Student).BirthDate);
        }
        throw new NotImplementedException();
    }
}
```

Теперь можно написать и перегруженный метод `Sort(IComparer comparer)` в классе `Auditory`.

```
class Auditory : IEnumerable
{
    // остальной код остался прежним

    public void Sort(IComparer comparer)
    {
        Array.Sort(students, comparer);
    }
}
```

Вызов этого метода приводит к следующему результату (Рисунок 8.8).

```
class Program
{
    static void Main(string[] args)
    {
        // остальной код остался прежним
    }
}
```

```

WriteLine(«\n+++++++ сортировка по дате
           рождения ++++++\n»);

auditory.Sort(new DateComparer());

foreach (Student student in auditory)
{
    WriteLine(student);
}
}

```

```

C:\WINDOWS\system32\cmd.exe
+++++++ сортировка по дате рождения ++++++
Фамилия: Taylor, Имя: Nicole, Родился: 10 мая 1996 г., Студенческий билет: AA 513484
Фамилия: Finch, Имя: Joey, Родился: 30 ноября 1996 г., Студенческий билет: AA 258322
Фамилия: Miller, Имя: John, Родился: 12 марта 1997 г., Студенческий билет: AB 189356
Фамилия: Leman, Имя: Candice, Родился: 22 июля 1998 г., Студенческий билет: XA 345185
Для продолжения нажмите любую клавишу . . .

```

**Рисунок 8.8.** Сортировка студентов по дате рождения

Следующее, что мы сделаем в рамках данного урока — создадим копию объекта `Student`. Как Вы знаете, простое присваивание одной ссылке другой не приведет к желаемому результату, так как мы получим две ссылки на один и тот же объект, каждая из них будет изменять этот объект. Продемонстрируем это на простом примере (Рисунок 8.9).

```

using static System.Console;

namespace SimpleProject
{
    class Child
    {

```

```

public string Name { get; set; }
public int Age { get; set; }

public override string ToString()
{
    return $"Имя: {Name}, Возраст: {Age}";
}

}

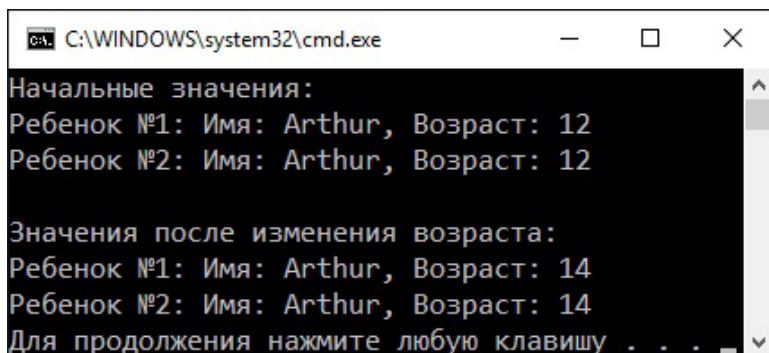
class Program
{
    static void Main(string[] args)
    {
        Child child1 = new Child { Name = "Arthur",
                                   Age = 12 };

        WriteLine("Начальные значения:");
        Child child2 = child1;
        WriteLine($"Ребенок №1: {child1}");
        WriteLine($"Ребенок №2: {child2}");

        child2.Age = 14; // изменяем возраст

        WriteLine(«\nЗначения после изменения
                   возраста:»);
        WriteLine($»Ребенок №1: {child1}»);
        WriteLine($»Ребенок №2: {child2}»);
    }
}

```



```

C:\WINDOWS\system32\cmd.exe
Начальные значения:
Ребенок №1: Имя: Arthur, Возраст: 12
Ребенок №2: Имя: Arthur, Возраст: 12

Значения после изменения возраста:
Ребенок №1: Имя: Arthur, Возраст: 14
Ребенок №2: Имя: Arthur, Возраст: 14
Для продолжения нажмите любую клавишу . . .

```

Рисунок 8.9. Копирование ссылок

Для того чтобы объект мог создавать копию самого себя ему необходимо реализовать интерфейс `ICloneable`.

## ICloneable

Интерфейс `ICloneable` содержит метод `Clone()`, реализация которого в классе позволяет получить копию текущего объекта, возвращает `object`.

Реализация метода `Clone()` в классе может быть различной, различают два понятия поверхностное и глубокое копирование. В случае если в классе отсутствуют ссылочные поля (свойства), то можно воспользоваться методом `MemberwiseClone()` класса `Object`, который создает «поверхностную» копию вызывающего объекта (урок №4). Приведем пример (Рисунок 8.10).

```

using System;
using static System.Console;

namespace SimpleProject
{
    class Child : ICloneable

```



```

{
    public string Name { get; set; }
    public int Age { get; set; }

    public override string ToString()
    {
        return $"Имя: {Name}, Возраст: {Age}";
    }

    public object Clone()
    {
        // поверхностная копия объекта, если нет
        // ссылочных полей
        return this.MemberwiseClone();
    }
}

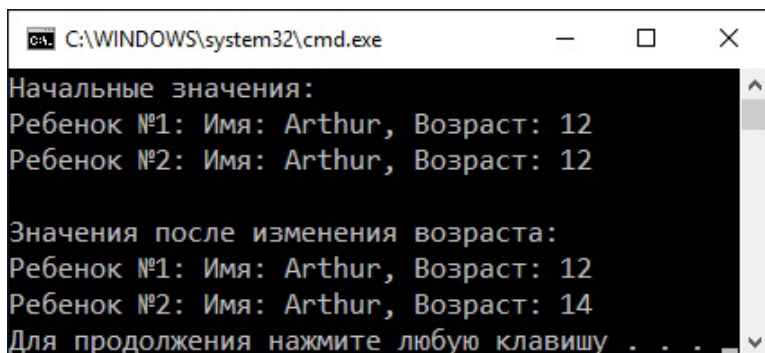
class Program
{
    static void Main(string[] args)
    {
        Child child1 = new Child { Name = "Arthur",
                                   Age = 12 };

        WriteLine("Начальные значения:");
        Child child2 = (Child)child1.Clone();
        WriteLine($"Ребенок №1: {child1}");
        WriteLine($"Ребенок №2: {child2}");

        child2.Age = 14; // изменяем возраст

        WriteLine(@"\nЗначения после изменения
                    возраста:»);
        WriteLine($"»Ребенок №1: {child1}»);
        WriteLine($"»Ребенок №2: {child2}»);
    }
}

```



```

C:\WINDOWS\system32\cmd.exe
Начальные значения:
Ребенок №1: Имя: Arthur, Возраст: 12
Ребенок №2: Имя: Arthur, Возраст: 12

Значения после изменения возраста:
Ребенок №1: Имя: Arthur, Возраст: 12
Ребенок №2: Имя: Arthur, Возраст: 14
Для продолжения нажмите любую клавишу . . .

```

**Рисунок 8.10.** Применение метода `MemberwiseClone()`

Данный способ не подходит для класса `Student`, потому что он содержит ссылочное свойство `StudentCard`. В этом случае метод `Clone()` должен вернуть экземпляр класса `Student`, созданного на основе текущего экземпляра (глубокое копирование). При этом можно создать поверхностную копию объекта `Student`, а потом изменить его ссылочное свойство `StudentCard`. Этот способ реализации метода `Clone()` представлен ниже.

```

class Student : IComparable, ICloneable
{
    // остальной код остался прежним

    public object Clone() // глубокое копирование
    {
        Student temp = (Student)this.MemberwiseClone();
        // поверхностная копия
        temp.StudentCard = new StudentCard {
            Series = this.StudentCard.Series,
            Number = this.StudentCard.Number };
        return temp;
    }
}

```

Результаты работы с методом `Clone()` представлены на рисунке 8.11, как часть итогового кода применения стандартных интерфейсов.

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

namespace SimpleProject
{
    class DateComparer : IComparer
    {
        public int Compare(object x, object y)
        {
            if (x is Student && y is Student)
            {
                return DateTime.Compare((x as Student).
                    BirthDate, (y as Student).BirthDate);
            }
            throw new NotImplementedException();
        }
    }

    class StudentCard
    {
        public int Number { get; set; }
        public string Series { get; set; }

        public override string ToString()
        {
            return $"Студенческий билет: {Series} {Number}";
        }
    }

    class Student : IComparable, ICloneable
    {
        public string FirstName { get; set; }
        public string LastName { get; set; }
    }
}
```

```

public DateTime BirthDate { get; set; }
public StudentCard StudentCard { get; set; }

public override string ToString()
{
    return $"Фамилия: {LastName}, Имя: {FirstName},
        Родился: {BirthDate.ToLongDateString()},
        {StudentCard}";
}

public int CompareTo(object obj)
{
    if (obj is Student)
    {
        return LastName.CompareTo((obj as Student).
            LastName);
    }
    throw new NotImplementedException();
}

public object Clone() // глубокое копирование
{
    Student temp = (Student)this.MemberwiseClone();
    // поверхностная копия
    temp.StudentCard = new StudentCard { Series =
        this.StudentCard.Series, Number =
        this.StudentCard.Number };
    return temp;
}
}

class Auditory : IEnumerable
{
    Student[] students =
    {
        new Student {
            FirstName = "John",

```

```

        LastName = "Miller",
        BirthDate = new DateTime(1997, 3, 12),
        StudentCard = new StudentCard {
            Number=189356, Series="AB" }
    },
    new Student {
        FirstName = "Candice",
        LastName = "Leman",
        BirthDate = new DateTime(1998, 7, 22),
        StudentCard = new StudentCard {
            Number=345185, Series="XA" }
    },
    new Student {
        FirstName = "Joey",
        LastName = "Finch",
        BirthDate = new DateTime(1996, 11, 30),
        StudentCard = new StudentCard {
            Number=258322, Series="AA" }
    },
    new Student {
        FirstName = "Nicole",
        LastName = "Taylor",
        BirthDate = new DateTime(1996, 5, 10),
        StudentCard = new StudentCard {
            Number=513484, Series="AA" }
    }
};

public void Sort()
{
    Array.Sort(students);
}

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

```

```

public void Sort(IComparer comparer)
{
    Array.Sort(students, comparer);
}

class Program
{
    static void Main(string[] args)
    {
        Auditory auditory = new Auditory();

        WriteLine("+++++++ список
                    студентов ++++++\n");

        foreach (Student student in auditory)
        {
            WriteLine(student);
        }
        WriteLine("«\n+++++++ сортировка по
                    фамилии ++++++\n»");
        auditory.Sort();

        foreach (Student student in auditory)
        {
            WriteLine(student);
        }

        WriteLine("«\n+++++++ сортировка по дате
                    рождения ++++++\n»");
        auditory.Sort(new DateComparer());

        foreach (Student student in auditory)
        {
            WriteLine(student);
        }
        WriteLine("\n+++++++ копирование
                    ++++++\n");
    }
}

```

```

Student student1 = new Student { FirstName =
    "Greg", LastName = "Carter", BirthDate =
    new DateTime(1999, 12, 5), StudentCard =
    new StudentCard { Number = 784523,
        Series = "MM" } };
Student student2 = (Student)student1.Clone();
WriteLine(student1);
WriteLine(student2);
WriteLine("\n+++++ изменение
            +++++\n");
student2.StudentCard.Number = 817423;
student2.StudentCard.Series = "KK";
WriteLine(student1);
WriteLine(student2);
    }
}
}

```

```

CA\WINDOWS\system32\cmd.exe
+++++ СПИСОК СТУДЕНТОВ +++++
Фамилия: Miller, Имя: John, Родился: 12 марта 1997 г., Студенческий билет: AB 189356
Фамилия: Leman, Имя: Candice, Родился: 22 июля 1998 г., Студенческий билет: XA 345185
Фамилия: Finch, Имя: Joey, Родился: 30 ноября 1996 г., Студенческий билет: AA 258322
Фамилия: Taylor, Имя: Nicole, Родился: 10 мая 1996 г., Студенческий билет: AA 513484

+++++ сортировка по фамилии +++++
Фамилия: Finch, Имя: Joey, Родился: 30 ноября 1996 г., Студенческий билет: AA 258322
Фамилия: Leman, Имя: Candice, Родился: 22 июля 1998 г., Студенческий билет: XA 345185
Фамилия: Miller, Имя: John, Родился: 12 марта 1997 г., Студенческий билет: AB 189356
Фамилия: Taylor, Имя: Nicole, Родился: 10 мая 1996 г., Студенческий билет: AA 513484

+++++ сортировка по дате рождения +++++
Фамилия: Taylor, Имя: Nicole, Родился: 10 мая 1996 г., Студенческий билет: AA 513484
Фамилия: Finch, Имя: Joey, Родился: 30 ноября 1996 г., Студенческий билет: AA 258322
Фамилия: Miller, Имя: John, Родился: 12 марта 1997 г., Студенческий билет: AB 189356
Фамилия: Leman, Имя: Candice, Родился: 22 июля 1998 г., Студенческий билет: XA 345185

+++++ копирование +++++
Фамилия: Carter, Имя: Greg, Родился: 5 декабря 1999 г., Студенческий билет: MM 784523
Фамилия: Carter, Имя: Greg, Родился: 5 декабря 1999 г., Студенческий билет: MM 784523

+++++ изменение +++++
Фамилия: Carter, Имя: Greg, Родился: 5 декабря 1999 г., Студенческий билет: MM 784523
Фамилия: Carter, Имя: Greg, Родился: 5 декабря 1999 г., Студенческий билет: KK 817423
Для продолжения нажмите любую клавишу . . .

```

Рисунок 8.11. Применение стандартных интерфейсов

## 9. Домашнее задание

1. Разработать абстрактный класс Геометрическая-Фигура с полями ПлощадьФигуры и ПериметрФигуры.

Разработать классы-наследники: Треугольник, Квадрат, Ромб, Прямоугольник, Параллелограмм, Трапеция, Круг, Эллипс и реализовать свойства, которые однозначно определяют объекты данных классов.

Реализовать интерфейс ПростойНУгольник, который имеет свойства: Высота, Основание, УголМеждуОснованиемИСмежнойСтороной, КоличествоСторон, ДлиныСторон, Площадь, Периметр.

Реализовать класс СоставнаяФигура который может состоять из любого количества ПростыхНУгольников. Для данного класса определить метод нахождения площади фигуры.

Предусмотреть варианты невозможности задания фигуры (введены отрицательные длины сторон или при создании объекта треугольника существует пара сторон, сумма длин которых меньше длины третьей стороны и т.п.)

2. Написать приложение, которое будет отображать в консоли простейшие геометрические фигуры заданные пользователем. Пользователь выбирает фигуру и задает ее расположение на экране, а также размер и цвет с помощью меню. Все заданные пользователем фигуры отображаются одновременно на экране. Фигуры (прямоугольник, ромб, треугольник, трапеция, многоугольник) рисуются звездочками или другими символами. Для реализации



программы необходимо разработать иерархию классов (продумать возможность абстрагирования).

Для хранения всех, заданных пользователем фигур, создать класс «Коллекция геометрических фигур» с методом «Отобразить все фигуры». Чтобы отобразить все фигуры указанным методом требуется использовать оператор `foreach`, для чего в классе «Коллекция геометрических фигур» необходимо реализовать соответствующие интерфейсы.