

# Microsoft .Net Framework and C# Programming Language



# Lesson 6

## Interfaces

### Contents

1. The concept of interface .....	3
2. The syntax of interface declaration .....	4
3. Examples of creating the interfaces .....	6
4. Interface references .....	7
5. Interface properties and indexers .....	15
6. Interface inheritance .....	20
7. The problems of names hiding at interface inheritance .....	22
8. Analysis of standard interfaces .....	28
IEnumerable .....	30
IComparable .....	33
IComparer .....	35
ICloneable .....	39
9. Home task .....	48

# 1. The concept of interface

---

The interface can be compared with a certain commitment undertaken by a class or a structure in case of the implementation of this interface. The interface contains signatures of properties, methods or events, and all of them should be necessarily implemented in a class or a structure.

In the interface itself, you cannot specify the implementation of its members, and you also cannot create an instance of the interface, but you can create a reference to the interface. The syntax of interface can be compared to an abstract class, all the methods of which are abstract. However, the interface cannot have constructors, it also cannot contain fields or operator overloading. All the methods of the interface are `public` by default (the access modifier cannot be changed during the implementation of a method in a derived class), and they cannot be declared `virtual` or `static`.

The interface is an alternative to an abstract class that allows implementing multiple inheritance in C#: the class or struct can implement any number of interfaces.

You should pay attention to a distinction between the base classes and interfaces in the description of the inheritance process; it is suggested that the class is inherited from the base class, but the class implements the interface.

## 2. The syntax of interface declaration

Interface declaration is carried out using the `interface` keyword, as the following example shows:

```
[access modifier] interface interface_name
{
    //interface members
}
```

Interface name usually begins with a capital I, which is a common practice that indicates a good programming style. For example, the main work of a researcher is the research and invention, then the researcher interface can be described in a following way:

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

When creating an interface, it is useful to remember that you cannot specify access modifiers to its members. An attempt to explicitly specify any access modifier (even `public`) will cause an error at compile time (Figure 2.1).

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

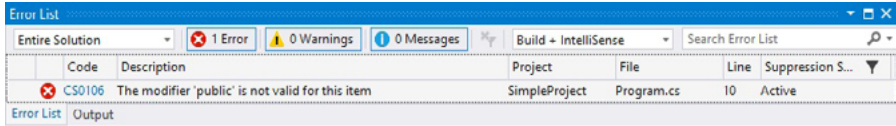


Figure 2.1. Error: The modifier is not valid for this element

### 3. Examples of creating the interfaces

As an example, let's consider the `IWorker` interface that describes a certain worker.

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

According to the interface presented, each class that implements this interface should implement the `Work()` method describing the performance of some particular job, the `IsWorking` property indicating whether the worker is busy and the `WorkEnded` event (will be considered in one of the subsequent lessons) announcing the completion of the work.

The following `IManager` interface obliges each class that implements it to have the `ListOfWorkers` property that allows writing and retrieving a list of workers, and implementing `Organize()`, `MakeBudget()` and `Control()` methods.

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

## 4. Interface references

---

In C# you can create a reference to an interface and assign it an instance of any class that implements this interface. Using the interface references only the methods of this interface can be called, at that the version of the method implemented in a particular class will be executed. If it will be necessary to call a method that is not part of the interface, then it is necessary to cast an interface reference to the appropriate type.

Explicit conversion can be used to perform casting to interface type, but in this case you are not immune from the attempt to cast a type that does not support the necessary interface. In this case, an exception will be generated, which we can intercept using the `try-catch` statement (this mechanism will be discussed in more detail in the subsequent lesson).

However, a more reliable way to cast to a particular interface is to use the `is` or `as` operators, which you have been studied in lesson 4. The use of these operators with the interfaces is similar to their use for casting a link to a particular class.

In the following example, we will combine the acquired knowledge to create a model of a store that has a director, a seller, a cashier and a storekeeper. All of them are human beings (`Human` class) and the employees of this store (`IEmployee` interface), but the director manages employees (`IManager` interface) who perform their work in different ways (`IWorker` interface).

```

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 $"
Last name: {LastName}
First name: {FirstName}
Birth date:
{BirthDate.ToLongDateString()}";
        }
    }

    abstract class Employee : Human
    {
        public string Position { get; set; }
        public double Salary { get; set; }

        public override string ToString()
        {
            return base.ToString() + $"
Position:
{Position} Salary: {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("I control the performance!");
    }

    public void MakeBudget()
    {
        WriteLine("I set the budget!");
    }

    public void Organize()
    {
        WriteLine("I organize the work!");
    }
}

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

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

```

```
        public string Work()
        {
            return "I sell commodities!";
        }
    }

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

        public string Work()
        {
            return "I accept payments for the commodities!";
        }
    }

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

        public string Work()
        {
            return "I keep tabs on commodities!";
        }
    }
}
```

```

    }
}

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

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

        if (seller is Employee)
            WriteLine($"Seller's salary:
                {(seller as Employee).Salary}");
            //casting interface reference
            //to the Employee class

        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 = "Storekeeper", Salary = 4500 }
        };

        WriteLine(director);
        if (director is IManager) //using the is
            //operator
    }
}

```

```

    {
        director.Control();
    }

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

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

```

```

C:\WINDOWS\system32\cmd.exe
Seller's salary: 3780
Last name: Doe First name: John Birth date: October 12, 1998
Position: Director Salary: $ 12000
I control the performance!
Last name: Beam First name: Jim Birth date: May 23, 1956
Position: Seller Salary: $ 3780
I sell commodities!
Last name: Smith First name: Nicole Birth date: May 23, 1956
Position: Cashier Salary: $ 3780
I accept payments for the commodities!
Last name: Ross Name: Bob Birth date: May 23, 1956
Position: Storekeeper Salary: $ 4500
I keep tabs on commodities!
Press any key to continue . . . _

```

Figure 4.1. Using the interface references

As you might have noticed in the previous example, interface inheritance syntax is similar to that of class inheritance. The colon follows the name of the child class, and the interfaces

that this class should implement are listed separated by commas (all the methods of each interface). Moreover, if the class is inherited from a base class, then it should always be listed right after the colon, followed by the rest interfaces separated by a comma.

The process of implementing a certain interface by the class or struct has been significantly simplified thanks to yet another feature of Visual Studio 2015 that allows you to get the templates of the required members of the interface.

Consider this process on the example of the `Seller` class. After a single left click on the name of `IWorker` interface in the inheritance line, a button with a lightbulb icon appears on the left side of this line that displays a list of possible actions when pressed. After pressing one of the items on this list, you can create implicit (*Implement interface*) or explicit (*Implement interface explicitly*) implementation of this interface. An example of implicit interface implementation is shown in Figure 4.2.

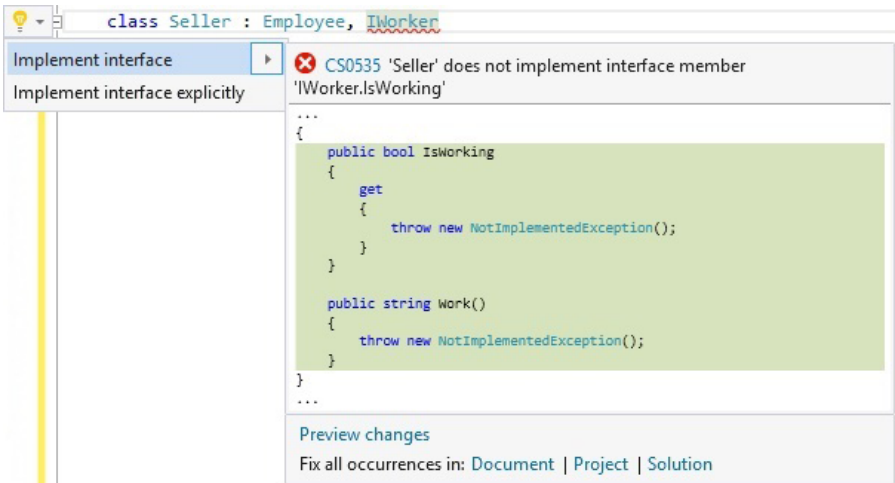


Figure 4.2. Implicit implementation of the `IWorker` interface

The appearance of the code generated by the framework (presented in a preview window) is already known to you, you might see the similar code in lesson 4 in the implementation of an abstract class. Just as then, the necessary implementation of the interface members should be written by your own, replacing in each members the line of exception generation (it will be discussed in the subsequent lesson).

Until now, we have worked with implicit interface implementations; you will learn about explicit implementations in one of the following sections of this lesson.

## 5. Interface properties and indexers

The properties in interfaces are similar to the fields in classes and can be represented (just as the usual properties) in three options: read-write, read-only and write-only.

```
interface IPerson
{
    string LastName { get; set; }
    int Age { get; }
    string Gender { set; }
}
```

As you may have noticed, the syntax of properties in an interface is similar to the syntax for declaring automatic properties, but there is a big difference between them. The properties in an interface are not automatically implemented, but only represent the specification of a property that will be implemented in the class or structure. At that it is prohibited to set access modifiers for the `get` and `set` accessors, even `public` (Figure 5.1).

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

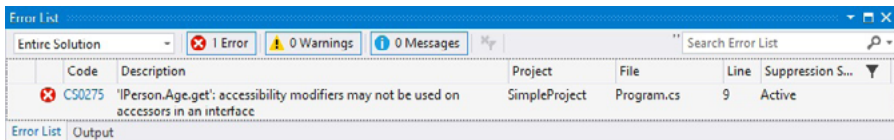


Figure 5.1. Error: accessibility modifiers may not be used on accessors in an interface

As you already know from the previous lesson, in C# there are indexers being a special kind of properties, so the syntax of their declaration in an interface is similar to the syntax of property declaration, and is looking the following way.

```
item_type this [data_type index]
{
    get;
    set;
}
```

Just as the properties, the indexers may have only one accessor, `set` or `get` for read-only or write-only, respectively. Here is an example of using indexers in an interface (Figure 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()
{
    //writing values using an indexer
    //with an integer parameter
    this[0] = "Bob";
    this[1] = "Candice";
    this[2] = "Jimmy";
    this[3] = "Joey";
    this[4] = "Nicole";
}
}

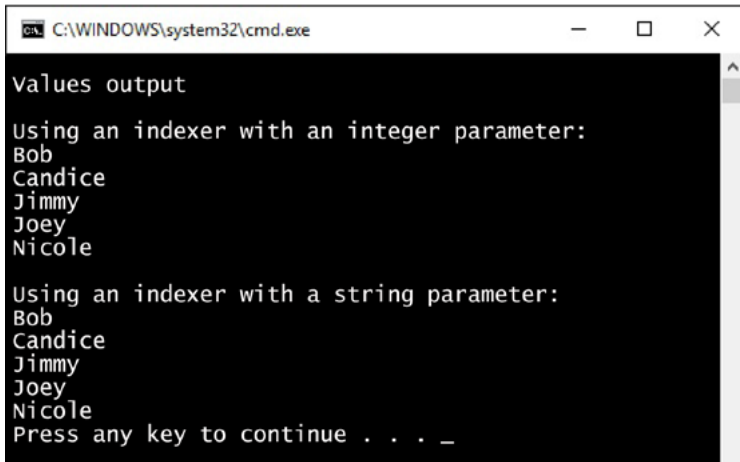
```

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

        WriteLine("\t\tValue output\n");
        WriteLine("Using an indexer with
                    an integer parameter:");
        for (int i = 0; i < 5; i++)
        {
            WriteLine(indexerClass[i]);
        }

        WriteLine("\nUsing an indexer with
                    a string parameter:");
        foreach (string item in Enum.
                    GetNames(typeof(Numbers)))
        {
            WriteLine(indexerClass[item]);
        }
    }
}
```

Program outcome.

A screenshot of a Windows command prompt window titled 'C:\WINDOWS\system32\cmd.exe'. The window has a black background with white text. The output shows 'Values output' followed by two sections. The first section, 'Using an indexer with an integer parameter:', lists the names 'Bob', 'Candice', 'Jimmy', 'Joey', and 'Nicole'. The second section, 'Using an indexer with a string parameter:', also lists the same names. At the bottom, it says 'Press any key to continue . . . \_' with a cursor under the underscore.

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

Values output

Using an indexer with an integer parameter:
Bob
Candice
Jimmy
Joey
Nicole

Using an indexer with a string parameter:
Bob
Candice
Jimmy
Joey
Nicole
Press any key to continue . . . _
```

Figure 5.2. The use of interface indexers

## 6. Interface inheritance

Just as the class, the interface can be inherited from other interfaces. In this case, you can create an interface hierarchy, where the child interfaces will expand their functionality at the cost of other previously created interfaces.

If a class implements an interface that in turn inherits another interface, then all the methods of all interfaces in the hierarchy should be implemented in this class.

```
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. The problems of names hiding at interface inheritance

Sometimes you need to declare a class that implements multiple interfaces. And a situation may arise where these interfaces have methods with the same name and signature. For example:

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

interface IB
{
    void Show();
}

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

The implementation of these interfaces by a class might look like this:

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

```

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

```

Although the code does not cause an error, the question arises: the method of which interface will be called, when an instance method of the `ImplicitRealization` class is called? Worse yet, the call of the `Show()` method using the references to different interfaces will always lead to execution of the same actions, which, in most cases, will not match the expected result (Figure 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();
        }
    }
}

```

Program outcome.

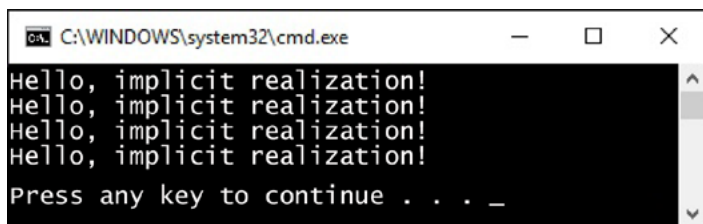


Figure 7.1. Calling class methods without explicit implementation

Therefore, this situation requires to perform an explicit interface implementation in a class; in order to facilitate this process, we suggest you to refer to Figure 4.2, but this time it is necessary to choose «Implement interface explicitly». At the same time, we draw your attention to the absence of access modifiers in these implementations, since their default modifier is `private`, specifying a modifier explicitly will cause an error at compile time (Figure 7.2).

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

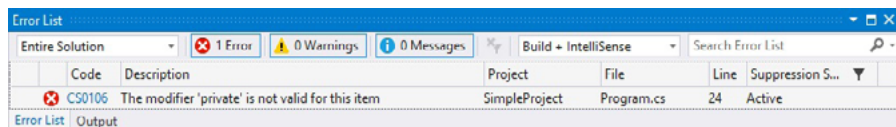


Figure 7.2. Error: The modifier is not valid for this element

Since explicit interface implementations have an implicit `private` access modifier, they cannot be called through the class



instance. However, there is a solution to this problem. For this it is necessary to explicitly cast a class instance to the type of interface, the method of which should be called in this case. At explicit interface implementation, it is possible to implement one of the interface method implicitly; it will be called when calling this method is called through a class instance.

It is also possible to call the method using an interface reference, unambiguously calling a corresponding interface method. We will demonstrate all of the above on an example (Figure 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(); //implicit call of the
                  //IC interface method

        ((IA)er).Show(); //explicit call of the IC
                        //interface method

        IB iB = new ExplicitRealization();
        iB.Show(); //calling the IB interface method
                  //using a reference
    }
}

```

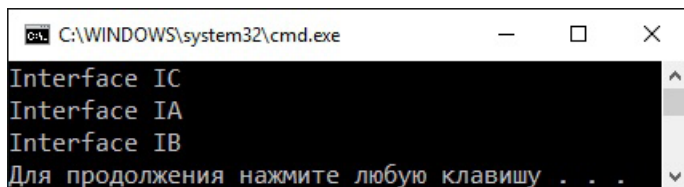


Figure 7.3. Explicit interface implementation

The use of explicit interface implementation is also useful when you want to hide the implementation of a method at the object level. This method will be possible to be called only using an explicit cast to a proper interface, or through a reference to this interface.

## 8. Analysis of standard interfaces

C# has a large number of standard interfaces, some of which will be considered in this section, while the rest of them will be learned by you progressively as you learn the language.

To demonstrate the examples, let's create three classes.

The `StudentCard` class that describes a student card.

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

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

The `Student` class describing a student.

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

And the `Auditory` class that contains an array of the `Student` type (the students present in the auditory) and the `Sort()` method calling the homonymous method in the `Array` class.

```

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);
    }
}

```

The first thing we want to do in our program is to display the list of all the students in the auditory using the `foreach` statement that actually causes an error at compile time (Figure 8.1).

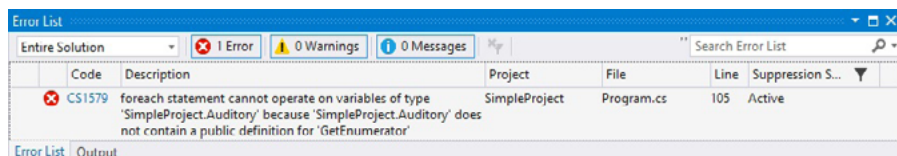


Figure 8.1 Error: `foreach` statement cannot operate on variables of type `Auditory`

To eliminate this error, you need to implement the `IEnumerable` interface in the `Auditory` class.

## `IEnumerable`

The `IEnumerable` interface allows to iterate over the elements of nongeneric collection using the enumerator.

In order to implement this interface, the `GetEnumerator()` method returning `IEnumerator` interface should be implemented in the `Auditory` class.

The `IEnumerator` interface contains three members:

- `Current` property that allows getting a current element of the collection; returns `object`;
- `MoveNext()` method iterates over a collection using the enumerator; returns `false` when reaching the end of the collection, otherwise returns `true`;
- `Reset()` method sets the enumerator to the beginning of the collection.

It turns out that we need to implement all the members of the `IEnumerator` interface in the `Auditory` class. However,

we will take advantage of the fact that this interface is already implemented in the `Array` class, which is the basis for all arrays. This decision is one of the three options that the framework suggests us (Figure 8.2).

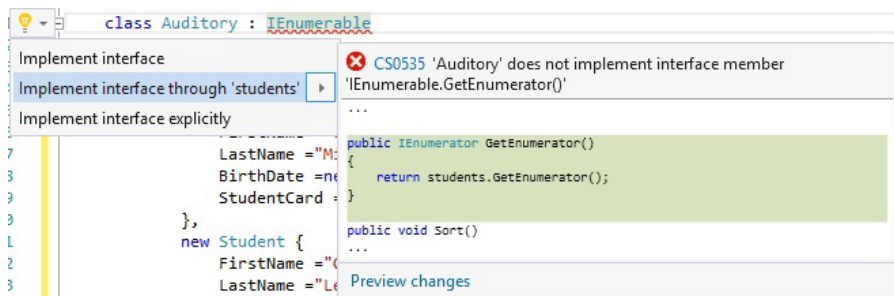


Figure 8.2. The option of the `Enumerable` interface implementation

We'll choose this particular implementation, but we will implement the `IEnumerator` interface explicitly so the `GetEnumerator()` method could not be called via an instance of the `Auditory` class, while the `foreach` statement is working correctly.

```
class Auditory : IEnumerator
{
    //rest of the code remains unchanged
    IEnumerator IEnumerator.GetEnumerator()
    {
        return students.GetEnumerator();
    }
}
```

Displaying the list of students using the `foreach` statement will look the following way (Figure 8.3).

```

C:\WINDOWS\system32\cmd.exe
+++++ list of students +++++
Last name: Miller, First name: John, Birth date: March 12, 1997, Student card: AB 189356
Last name: Leman, First name: Candice, Birth date: July 22, 1998, Student card: XA 345185
Last name: Finch, First name: Joey, Birth date: November 30, 1996, Student card: AA 258322
Last name: Taylor, First name: Nicole, Birth date: May 10, 1996, Student card: AA 513 484
Press any key to continue . . . _

```

Figure 8.3. Displaying the list of students using the foreach statement

Our next step in the program is an attempt to call the `Sort()` method of the `Auditory` class `Auditory`, causing a runtime error (Figure 8.4).

```

class Program
{
    static void Main(string[] args)
    {
        //rest of the code remained unchanged
        WriteLine("\n+++++ sort by last name +++++\n");
        auditory.Sort();
        foreach (Student student in auditory)
        {
            WriteLine(student);
        }
    }
}

```

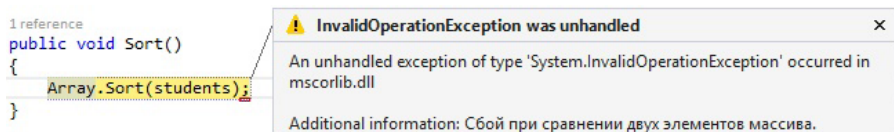


Figure 8.4. Error: cannot compare two array element

This error occurred because the `Sort()` method of the `Array` class "does not know" how to compare the instances of the `Student` class to sort them. In order to make



the `Sort()` method work properly, it is necessary to implement the `Comparable` interface in the `Student` class.

## Comparable

The `Comparable` interface contains a single `CompareTo(object)` method, which implementation in the class determines how it is necessary to compare the instances of this class.

The `CompareTo(object)` method receives a parameter of `object` type, returning an integer value:

- negative if the current class instance precedes a passed parameter in sorting;
- zero if the current class instance is in the same position as a passed parameter;
- positive if the current class instance follows a passed parameter.

When implementing the `Comparable` interface by the `Student` class, the framework offers us all the possible options considering the properties of the current class. In this case, we want to perform sorting of the students by name (Figure 8.5).

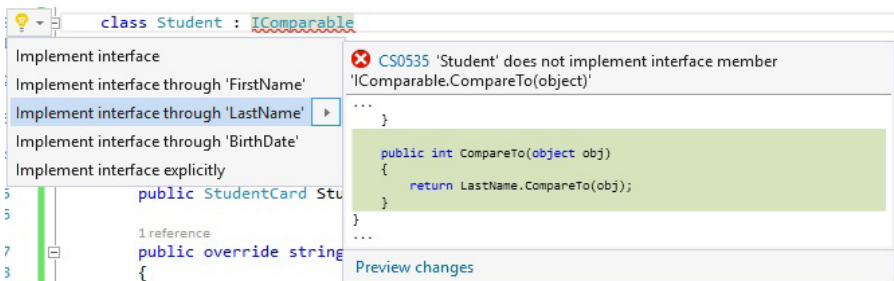


Figure 8.5. Possible implementations of the `Comparable` interface

However, if we use this implementation as the Visual Studio offers us, we will get an error at runtime (Figure 8.6).

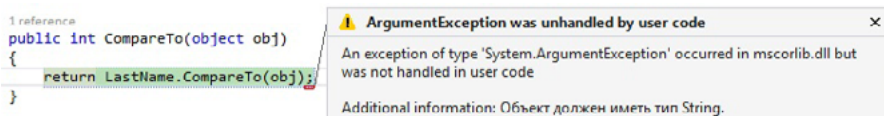


Figure 8.6 Runtime error

The reason for this error is the impossibility to compare the `LastName` property of `string` type with the parameter of `object` type. Summing up: do not always Visual Studio every time.

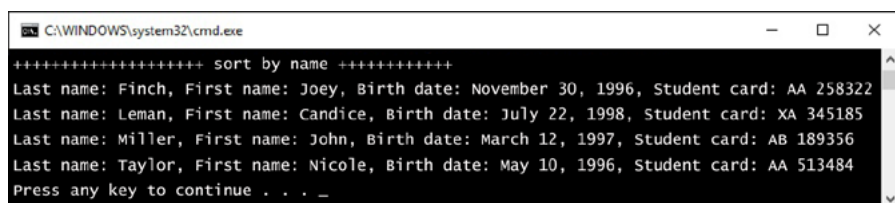
We will write our own implementation of the `CompareTo(object)` method, using the `is` and `as` operators to check and cast the passed parameter.

```
class Student : IComparable
{
    //rest of the code remained unchanged
    public int CompareTo(object obj)
    {
        if (obj is Student)
        {
            return LastName.CompareTo((obj as Student).
                LastName);
        }
        throw new NotImplementedException();
    }
}
```

We would like to draw your attention to the presence of "`throw new NotImplementedException();`" line in the `CompareTo(object)` method. The fact is that according to the C# language requirements, if the method returns

a value, then each part of this method should return this value. In other words, in our case it was necessary to write the `else` statement with the return statement inside, but what value would be necessary to `return` in case the transferred object does not correspond the `Student` class? It is difficult to find an answer, but in this case, exception generation is correct and does not cause an error at compile time.

The results of student list sorting are shown in Figure 8.7.



```

C:\WINDOWS\system32\cmd.exe
+++++++ sort by name ++++++++
Last name: Finch, First name: Joey, Birth date: November 30, 1996, Student card: AA 258322
Last name: Leman, First name: Candice, Birth date: July 22, 1998, Student card: XA 345185
Last name: Miller, First name: John, Birth date: March 12, 1997, Student card: AB 189356
Last name: Taylor, First name: Nicole, Birth date: May 10, 1996, Student card: AA 513484
Press any key to continue . . . _

```

Figure 8.7. Sorting students by name

The `Sort()` method of the `Auditory` class allows sorting students by name, but what if we need to sort students by any other criteria?

You would say that it is necessary to write an overloaded method, and you are right. But there is one issue: the `CompareTo(object)` method is already implemented in our `Student` class, and the `Sort()` method of the `Array` class will use it in sorting.

However, the `Sort()` method of the `Array` class has 17 overloaded versions, and one of them takes a reference to the `IComparer` interface as its second argument.

## IComparer

The `IComparer` interface contains `Compare(object, object)` method, which implementation in a class allows comparing transferred objects.

The `Compare(object, object)` method takes two parameters of object type and returns an integer value:

- negative if the first argument is less than the second parameter;
- zero if the first parameter is equal to the second parameter;
- positive if the first parameter is greater than the second parameter.

In order to call the overloaded `Sort()` method of the `Array` class, it is necessary to create the `DateComparer` class that implements the `IComparer` interface, in which we should implement the `Compare(object, object)` method. In this case we compare the objects by birth date, using the fact that there is a `Compare(DateTime t1, DateTime t2)` static method in the `DateTime` struct.

```
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();
    }
}
```

Now we can write an overloaded `Sort(IComparer comparer)` method in the `Auditory` class.

```

class Auditory : IEnumerable
{
    //rest of the code remains unchanged

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

```

The call of this method leads to the following outcome (Рисунок 8.8).

```

class Program
{
    static void Main(string[] args)
    {
        //rest of the code remains unchanged
        WriteLine("\n+++++++ sort by birth date
                ++++++\n");

        auditory.Sort(new DateComparer());

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

```

```

C:\WINDOWS\system32\cmd.exe
+++++++ sorting by birth date ++++++
Last name: Taylor, First name: Nicole, Birth date: May 10, 1996, Student card: AA 513484
Last name: Finch, First name: Joey, Birth date: November 30, 1996, Student card: AA 258322
Last name: Miller, First name: John, Birth date: March 12, 1997, Student card: AB 189356
Last name: Leman, First name: Candice, Birth date: July 22, 1998, Student card: XA 345185
Press any key to continue . . . _

```

Figure 8.8. Sorting students by birth date

The next thing we will do as part of this lesson is creating a copy of the `Student` object. As you know, a simple assignment of one reference to another will not lead to a desired result, because we will get two references to the same object, and each of them will change this object. Let us demonstrate this with a simple example (Figure 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 $"Name: {Name}, Age: {Age}";
        }
    }

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

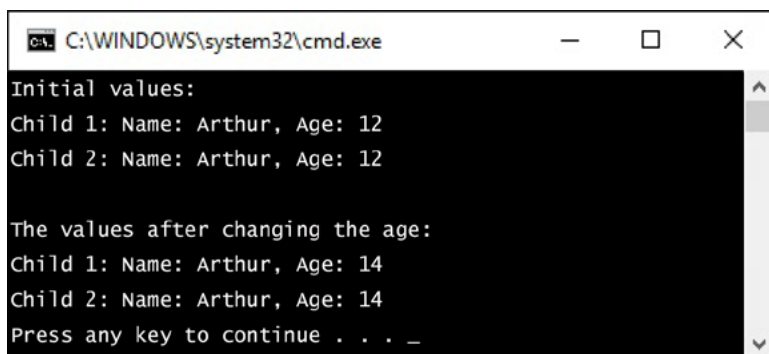
            WriteLine("Initial values:");
            Child child2 = child1;
            WriteLine($"Child 1: {child1}");
            WriteLine($"Child 2: {child2}");

            child2.Age = 14; //changing the age
        }
    }
}
```

```

        WriteLine("\nThe values after changing
                    the age:");
        WriteLine($"Child 1: {child1}");
        WriteLine($"Child 2: {child2}");
    }
}
}

```



```

C:\WINDOWS\system32\cmd.exe
Initial values:
Child 1: Name: Arthur, Age: 12
Child 2: Name: Arthur, Age: 12

The values after changing the age:
Child 1: Name: Arthur, Age: 14
Child 2: Name: Arthur, Age: 14
Press any key to continue . . . _

```

Figure 8.9. Copying references

To make the object be able to create a copy of itself, it is necessary to implement the `ICloneable` interface in it.

## ICloneable

The `ICloneable` interface contains the `Clone()` method, the implementation of which in a class allows obtaining a copy of the current object; returns the `object`.

The implementation of the `Clone()` method in a class can be different, since there are two concepts of shallow and deep copy. In case the class has no reference fields (properties), the `MemberwiseClone()` method of the `Object` class can be used to create a shallow copy of the caller object (Lesson 4).

Here is an example (Figure 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 $"Name: {Name}, Age: {Age}";
        }

        public object Clone()
        {
            //shallow copy of an object if there
            //is no reference fields
            return this.MemberwiseClone();
        }
    }
    class Program
    {
        static void Main(string[] args)
        {
            Child child1 = new Child { Name = "Arthur",
                                         Age = 12 };

            WriteLine("Initial values:");
            Child child2 = (Child)child1.Clone();
            WriteLine($"Child 1: {child1}");
            WriteLine($"Child 2: {child2}");

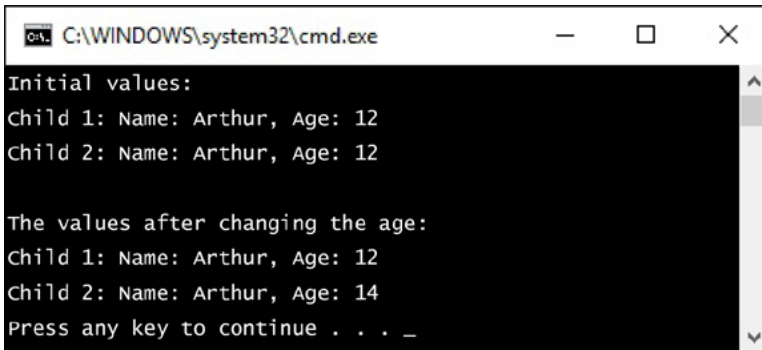
            child2.Age = 14; //changing the age
        }
    }
}
```



```

        WriteLine("\nThe values after changing
                    the age:");
        WriteLine($"Child 1: {child1}");
        WriteLine($"Child 2: {child2}");
    }
}

```



```

C:\WINDOWS\system32\cmd.exe
Initial values:
Child 1: Name: Arthur, Age: 12
Child 2: Name: Arthur, Age: 12

The values after changing the age:
Child 1: Name: Arthur, Age: 12
Child 2: Name: Arthur, Age: 14
Press any key to continue . . . _

```

Figure 8.10. Applying the `MemberwiseClone()` method

This method is not suitable for the `Student` class, because it contains the `StudentCard` reference property. In this case, the `Clone()` method must return an instance of the `Student` class created on the basis of a current instance (deep copy). You can create a shallow copy of the `Student` object, and then change its `StudentCard` reference property. This way of implementing the `Clone()` method is shown below.

```

class Student : IComparable, ICloneable
{
    //rest of the code remains unchanged

    public object Clone() //deep copy
    {

```

```

        Student temp = (Student)this.MemberwiseClone();
                                //shallow copy
        temp.StudentCard = new StudentCard {
            Series = this.StudentCard.Series,
            Number = this.StudentCard.Number };
        return temp;
    }
}

```

The outcome of working with the `Clone()` method is shown in Figure 8.11 as part of the final code that demonstrates the use of standard interfaces.

```

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 $"Student card: {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 $"Last name: {LastName},
                    First name : {FirstName},
                    Birth date: {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() //deep copy
        {
            Student temp = (Student)this.MemberwiseClone();
                                //shallow copy
        }
    }

```

```

        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("+++++ list of
                    students +++++\n");

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

        WriteLine("\n+++++ sort by last name
                    +++++\n");
        auditory.Sort();
    }
}

```

```

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

        WriteLine("\n+++++++ sort by birth date
                ++++++\n");
        auditory.Sort(new DateComparer());

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

        WriteLine("\n+++++++ copying
                ++++++\n");

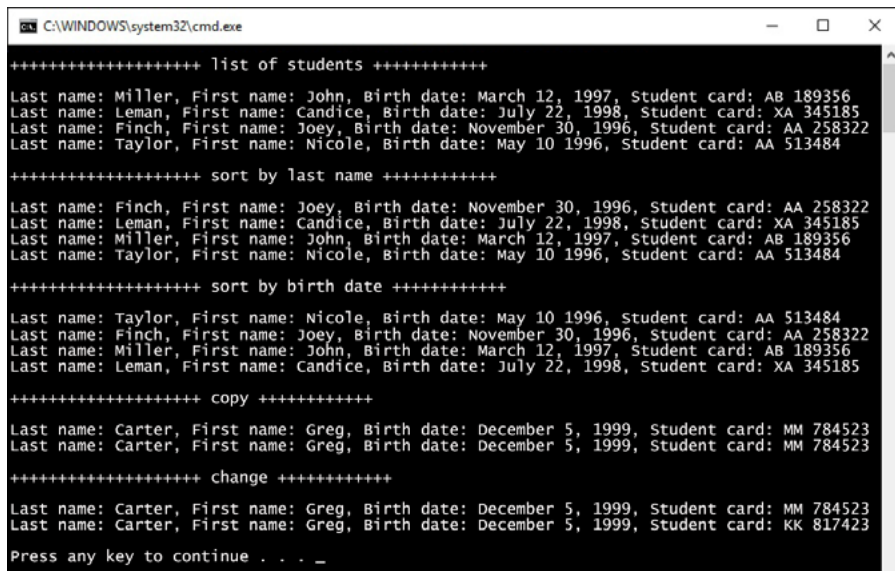
        Student student1 = new Student { FirstName =
            "Greg", LastName = "Carter", BirthDate =
            new DateTime(1999, 12, 5), StudentCard =
            new StudentCard { Number = 784523,
                Series = "MM" } };

        Student student2 = (Student)student1.Clone();
        WriteLine(student1);
        WriteLine(student2);
        WriteLine("\n+++++++ changing
                ++++++\n");

        student2.StudentCard.Number = 817423;
        student2.StudentCard.Series = "KK";

        WriteLine(student1);
        WriteLine(student2);
    }
}

```



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

+++++++ list of students ++++++++
Last name: Miller, First name: John, Birth date: March 12, 1997, Student card: AB 189356
Last name: Leman, First name: Candice, Birth date: July 22, 1998, Student card: XA 345185
Last name: Finch, First name: Joey, Birth date: November 30, 1996, Student card: AA 258322
Last name: Taylor, First name: Nicole, Birth date: May 10 1996, Student card: AA 513484

+++++++ sort by last name ++++++++
Last name: Finch, First name: Joey, Birth date: November 30, 1996, Student card: AA 258322
Last name: Leman, First name: Candice, Birth date: July 22, 1998, Student card: XA 345185
Last name: Miller, First name: John, Birth date: March 12, 1997, Student card: AB 189356
Last name: Taylor, First name: Nicole, Birth date: May 10 1996, Student card: AA 513484

+++++++ sort by birth date ++++++++
Last name: Taylor, First name: Nicole, Birth date: May 10 1996, Student card: AA 513484
Last name: Finch, First name: Joey, Birth date: November 30, 1996, Student card: AA 258322
Last name: Miller, First name: John, Birth date: March 12, 1997, Student card: AB 189356
Last name: Leman, First name: Candice, Birth date: July 22, 1998, Student card: XA 345185

+++++++ copy ++++++++
Last name: Carter, First name: Greg, Birth date: December 5, 1999, Student card: MM 784523
Last name: Carter, First name: Greg, Birth date: December 5, 1999, Student card: MM 784523

+++++++ change ++++++++
Last name: Carter, First name: Greg, Birth date: December 5, 1999, Student card: MM 784523
Last name: Carter, First name: Greg, Birth date: December 5, 1999, Student card: KK 817423

Press any key to continue . . . _
```

Рисунок 8.11. The use of standard interfaces

## 9. Home task

1. Develop a `GeometricalFigure` abstract class with the `FigureArea` and `FigurePerimeter` fields.

Develop the child classes: `Triangle`, `Square`, `Rhombus`, `Rectangle`, `Parallelogram`, `Trapezium`, `Circle`, `Ellipse`, and then implement the properties that unambiguously identify the objects of these classes.

Implement the `SimpleNGon` interface that has the following properties: `Height`, `Base`, `AngleBetweenBaseAndAdjacentSide`, `NumberOfSides`, `SidesLength`, `Area`, `Perimeter`.

Implement the `CompoundShape` class that can consist of any number of `SimpleNGons`. For this class define a method of finding a figure area.

Provide options for the impossibility to set a figure (negative side lengths entered, or when creating a triangle object there is a pair of sides, the length sum of which is less than the length of a third side, etc.)

2. Write an application that displays in the console the simplest geometrical figures specified by the user. The user chooses a figure and defines its location on the screen, the size and color from the menu. All the figures defined by the user are displayed simultaneously on the screen. The figures (rectangle, rhombus, triangle, trapezoid, polygon) are drawn with asterisks or other symbols. To implement the program, it is necessary to develop a class hierarchy (think about the possibility of abstraction).

In order to store all the user-defined figures, create a class called "Collection of geometric figures" with the "Display all"



method. To display all the figures using the specified method, it is necessary to use the foreach statement, for which purpose the corresponding interfaces should be implemented in the "Collection of geometrical figures" class.