

# Microsoft .Net Framework and C# Programming Language



# Lesson 4

## Inheritance, Interfaces

### Contents

<b>1. Inheritance in C#</b>	<b>4</b>
Analysis of Inheritance Mechanisms in C#	4
Access Modifiers Inheritance	5
Features of Using the Constructors in Inheritance	7
The Base Keyword	11
Hiding Names in Inheritance	13
<b>2. Using the Sealed Keyword</b>	<b>16</b>
<b>3. Using References to a Base Class</b>	<b>18</b>
<b>4. Polymorphism in C#. Virtual Methods.</b>	<b>24</b>
What is a Virtual Method?	24
Virtual Method Overriding	25
The Need to Use Virtual Methods	37

<b>5. Abstract Class .....</b>	<b>44</b>
<b>6. Analysis of the Object Base Class .....</b>	<b>53</b>
<b>Home Task. ....</b>	<b>62</b>
Task 1. ....	62
Task 2. ....	62

# 1. Inheritance in C#

## Analysis of Inheritance Mechanisms in C#

Inheritance allows reusing the existing classes, enhancing their functionality at the same time. There are two kinds of inheritance — type inheritance, in which a new type gets all the properties and methods of a parent, and interface inheritance, in which a new type gets a method signature without their implementation from a parent. All the classes that haven't got a specified base class are inherited from a `System.Object` class (a short form of the `object` name). The C# language does not support multiple inheritance. This means that in C# a class may be a descendant of only one class, but it can implement multiple interfaces. In other words, a class cannot be inherited from more than one base class and multiple interfaces.

The syntax for inheritance in C# looks this way:

```
class InheritedClass : BaseClass, Interface1, ...,
    InterfaceN
{
    // fields, properties, events and methods
    // of the class
}
```

Figure 1.1 shows an example of a class diagram (we will discuss how to build a class diagram with the tools of Visual Studio 2015 in the sixth section of this lesson).

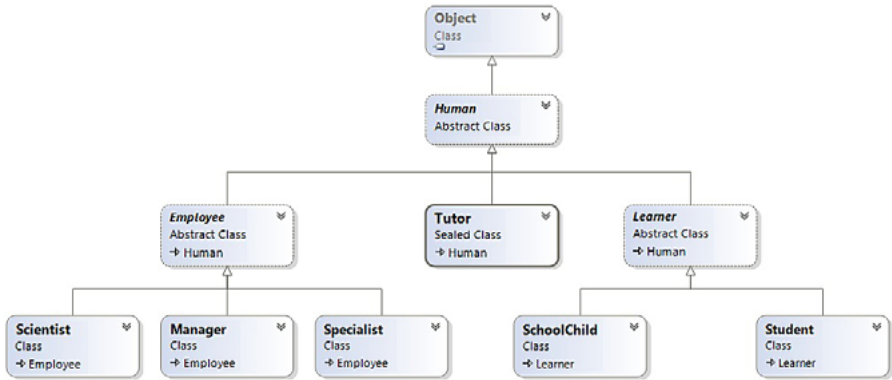


Figure 1.1. Example of a class inheritance hierarchy

The diagram shows that the `Employee`, `Learner`, `Tutor` classes inherit the `Human` class, which in its turn, is inherited from the `System.Object` class. The `Human`, `Learner` and `Employee` classes are abstract, while the `Tutor` is closed. The elements and their relationships depicted in the diagram will be discussed in more detail during this lesson.

## Access Modifiers Inheritance

You have already learned the C# access modifiers in the third lesson, but only `protected` and `protected internal` modifiers are directly related to inheritance. Let's consider their use in the following example.

Assume we have declared a `Human` class, which is a base for `Employee` class. The `firstName` and `lastName` fields of the `Human` class are declared with `protected` modifier, i.e. they are available in all the descendant classes, so we can use these fields in the `Employee` class.

```

using static System.Console;
namespace SimpleProject
{
    public class Human
    {
        int _id;
        protected string firstName;
        protected string lastName;
    }

    public class Employee : Human
    {
        double _salary;

        public Employee(string fName, string lName,
                        double salary)
        {
            firstName = fName;
            lastName = lName;
            _salary = salary;
            //_id = 34; Error
        }

        public void Print()
        {
            WriteLine($"Last name: {lastName}
                       \nFirst name: {firstName}
                       \nSalary: $ {_salary}");
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
            Employee employee = new Employee("John",
                                              "Doe", 2563.57);
        }
    }
}

```

```

        employee.Print();
    }
}

```

Execution of the code above does not cause an error (Figure 1.2).

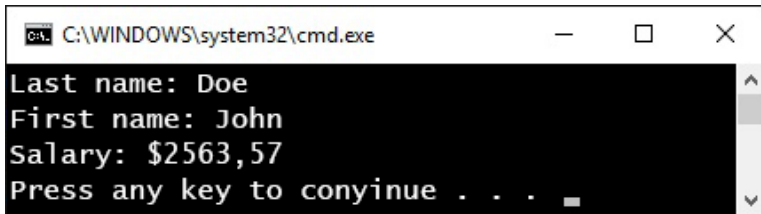


Figure 1.2. Example of using the protected modifier

An attempt to address the `Employee` class to the `_id` field leads to an error (Figure 1.3), because in the `Human` class this field is implicitly declared with `private` modifier, so it is possible to access this field only within the `Human` class.

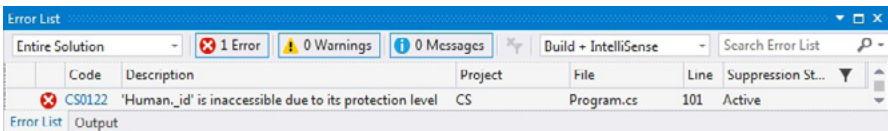


Figure 1.3. Error: address to the closed field of the class

`protected internal` modifier operates similarly to the `protected` modifier, but at the level of current assembly.

## Features of Using the Constructors in Inheritance

When creating a successor class, not one, but a whole chain of constructors is actually called. Firstly, a class constructor is selected, an instance of which is created. This constructor

attempts to address a constructor of its immediate base class that, in its turn, attempts to call a constructor of its base class. This happens until you reach the `System.Object` class, which does not have a base class. As a result, we have a sequential call of constructors of all the classes of the hierarchy, starting with the `System.Object` class, ending with the class, the instance of which we want to create. In this process, each constructor initializes the fields of its own class.

Several constructors can be defined for each class. If we want to call a base class constructor for a descendant class, then it is necessary to use the `base()` keyword.

```
public InheritedClass() : base()  
{  
    // Fields, properties, events and methods  
    // of the class  
}
```

Let's improve the `Human` class by adding the `_birthDate` field and two constructors: constructor that takes first name, last name and birth date as the parameters, and the `Show()` method for data output.

Let's create three constructors in the `Employee` class: the first constructor takes last and first names as the parameters, the second constructor takes last name, first name and salary, and the third takes first name, last name, birth date and salary. Before executing their codes, all these constructors call the corresponding `Human` base class constructors using the `base()` keyword for initialization of certain fields. To avoid code redundancy, we have somewhat changed the `Print()` method by adding a call of the `Show()` method of the `Human` class. So far, it looks a bit clumsy, but we will eliminate this defect in the subsequent sections.



Changes in the classes described above are shown in the following code:

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

namespace SimpleProject
{
    public class Human
    {
        string _firstName;
        string _lastName;
        DateTime _birthDate;

        public Human(string fName, string lName)
        {
            _firstName = fName;
            _lastName = lName;
        }

        public Human(string fName, string lName,
            DateTime date)
        {
            _firstName = fName;
            _lastName = lName;
            _birthDate = date;
        }

        public void Show()
        {
            WriteLine($"Last name: {_lastName}\nFirst name: {_firstName}\nBirth date: {_birthDate.ToShortDateString()}");
        }
    }

    public class Employee : Human
    {
        double _salary;
        public Employee(string fName, string lName) :
            base(fName, lName) { }
    }
}
```

```

    public Employee(string fName, string lName,
                    double salary)
        : base(fName, lName)
    {
        _salary = salary;
    }

    public Employee(string fName, string lName,
                    DateTime date, double salary)
        : base(fName, lName, date)
    {
        _salary = salary;
    }

    public void Print()
    {
        Show();
        WriteLine($"{Salary: {_salary}}");
    }
}

class Program
{
    static void Main(string[] args)
    {
        Employee employee = new Employee("John",
                                          "Doe");

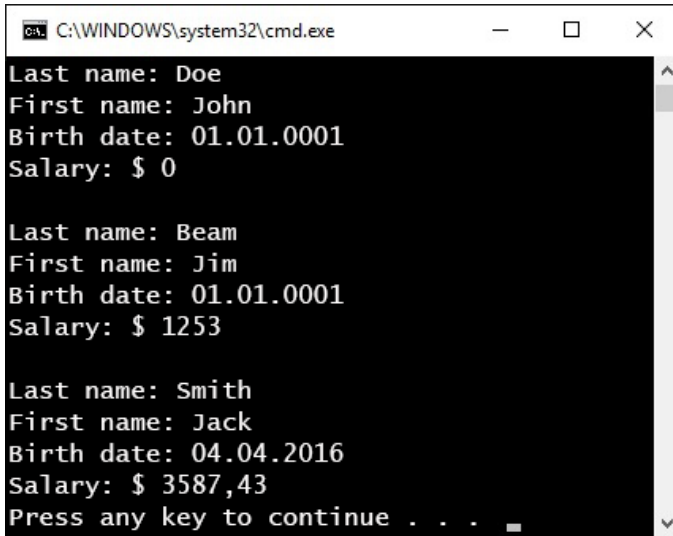
        employee.Print();

        employee = new Employee("Jim", "Beam", 1253);
        employee.Print();

        employee = new Employee("Jack", "Smith",
                                DateTime.Now, 3587.43);
        employee.Print();
    }
}

```

Approximate program outcome is shown in the Figure 1.4.



```
C:\WINDOWS\system32\cmd.exe
Last name: Doe
First name: John
Birth date: 01.01.0001
Salary: $ 0

Last name: Beam
First name: Jim
Birth date: 01.01.0001
Salary: $ 1253

Last name: Smith
First name: Jack
Birth date: 04.04.2016
Salary: $ 3587,43
Press any key to continue . . .
```

Figure 1.4. Using the base keyword in inheritance

## The Base Keyword

The `base` keyword is used not only when creating a descendant class constructor, which should call the class constructor, but also for accessing the members of the base class from the derived, and for calling the base class method when it is overridden the descendant class (overriding methods will be discussed in the later sections). Access to the base class is permitted only in the constructor or methods of the descendant class.

Let's return to the previous example, where we have called the `Show()` method of the `Human` base class in the `Employee` class, the same can be done by using the `base` keyword, the results of the program will stay the same (Figure 1.4).

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

namespace SimpleProject
{
    public class Human
    {
        // Fields and constructors remain the same

        public void Show()
        {
            WriteLine($"Last name: {_lastName}\nFirst name: {_firstName}\nBirth date: {_birthDate.ToShortDateString()}");
        }
    }

    public class Employee : Human
    {
        // Fields and constructors remain the same

        public void Print()
        {
            base.Show();
            WriteLine($"Salary: $ {_salary}");
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
            // Operations remain the same
        }
    }
}
```

The use of the `base` keyword does not change the functionality of the program, but specifies that the `base` class members are used in this case.

## Hiding Names in Inheritance

Let's continue to work with our example. Assume the methods with the same name `Print()` are defined in the `Employee` derived class and `Human` base class. Will this generate an error at the runtime?

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

namespace SimpleProject
{
    public class Human
    {
        // Fields and constructors remain the same
        public void Print()
        {
            WriteLine($" \ nLast name: {_lastName} \
                        nFirst name: {_firstName} \
                        nBirth date: {_birthDate.
                        ToShortDateString()}");
        }
    }

    public class Employee : Human
    {
        // Fields and constructors remain the same
        public void Print()
        {
            WriteLine($"Salary: $ {_salary}");
        }
    }
}
```

```

class Program
{
    static void Main(string[] args)
    {
        Employee employee = new Employee("Jack",
                                           "Smith", DateTime.Now, 3587.43);
        employee.Print();
    }
}

```

Program outcome will not lead to an error (Figure 1.5).



Figure 1.5. Program outcome

However, a warning will appear in the Error List window (Figure 1.6).

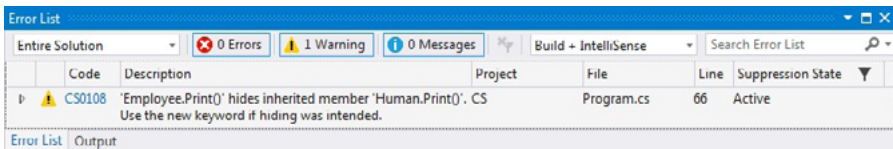


Figure 1.6. Warning: the need to use the new keyword

The essence of this warning is that the `Print()` method of the `Employee` class hides the `Print()` method of the `Human` class, and if it was aimed by the programmer, then it is recommended to use the new keyword in the `Employee` class when declaring the `Print()` method.

The following code demonstrates the only change introduced to the `Employee` class — the `new` keyword when

declaring the `Print()` method. Execution of the previous code with this change will lead to the same result (Figure 1.5), but with no warning.

```
public class Employee : Human
{
    // Fields and constructors remain the same

    public new void Print()
    {
        WriteLine($"Salary: $ {_salary}");
    }
}
```

And finally, the `new` keyword can be used for hiding any member of the base class. Assume that for some reason we want to hide an arbitrary field of the `Human` class in the `Employee` class, then the code will look the following way:

```
public class Human
{
    protected string middleName;

    // Rest of the code remains unchanged
}

public class Employee : Human
{
    new string middleName;

    // Rest of the code remains unchanged
}
```

## 2. Using the Sealed Keyword

Sometimes there are situations when it is necessary to prohibit inheriting a certain class or overriding a certain method. To do this, the `sealed` keyword is used when defining a class or a method. Let's declare a `Tutor` sealed class:

```
public sealed class Tutor : Human { }
```

Thus, we prohibit using the `Tutor` class as a base class, while the `Tutor` class itself can be inherited from another class. Attempt to inherit from the `Tutor` class will cause an error at the compile time (Figure 2.1).

```
sealed class Tutor : Human { }  
class Curator : Tutor { } // Error
```

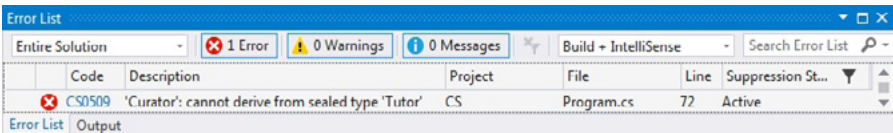


Figure 2.1. Error: cannot inherit from the sealed class

How to determine whether you can inherit from a specific class of .NET libraries? You can view detailed information on the types of different .NET libraries using the `Object Browser`, which can be opened by selecting the `VIEW` main menu, and then selecting `Object Browser` in the drop down list. To go to the desired type, enter its name in the search bar, and then choose the result from the list, which is suitable for you. Right at the bottom you will see information



on the selected type and on the right top there is a list of methods of this type (if any).

For example, we will get information on the `String` class (Figure 2.2). As you can see, this class is sealed.

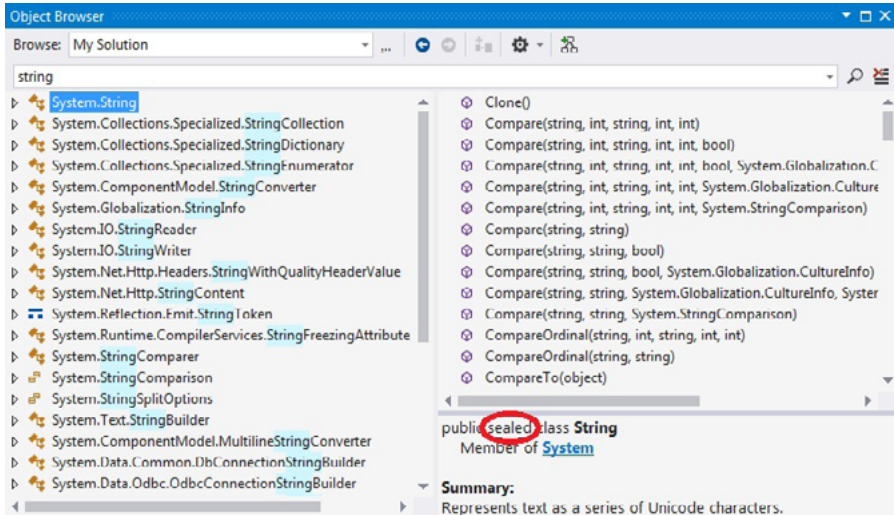


Figure 2.2. Information about the String class in the Object Browser window

The use of the `sealed` keyword when working with methods will be discussed in the section 4 of this lesson.

## 3. Using References to a Base Class

As you know, you cannot assign a variable of one type to a variable of another type in C#. But there is one exception — a reference to an object of any derived class of the base class can be assigned to a reference variable of this base class. It is important to understand the following: when a reference to a derived class is assigned to a reference variable of the base class, then you get access only to those parts of the object that are defined in the base class.

Let us examine this with an example, taking the `Employee` class described earlier as a base class, and create three descendant classes on its basis: `Manager`, `Scientist` and `Specialist`. Declare the required fields and methods in the classes created.

In the `Main()` method we declare a reference to the `Employee` class and assign it an instance of the `Manager` class, also declare an array of the `Employee` type and initialize it with the instances of the `Manager`, `Scientist` and `Specialist` classes. After that, using a `foreach` loop, we will go through an array and call the `Print()` method for each element of the array. All these actions should be carried out properly, since we are working with the reference to the base class.

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

namespace SimpleProject
```

```

{
    public class Human
    {
        // Class implementation remains unchanged
    }

    public class Employee : Human
    {
        // Class implementation remains unchanged
    }

    class Manager : Employee
    {
        string _fieldActivity;

        public Manager(string fName, string lName,
            DateTime date, double salary, string
            activity) : base(fName, lName,
            date, salary)
        {
            _fieldActivity = activity;
        }

        public void ShowManager()
        {
            WriteLine($"Manager. Field of activity:
                {_fieldActivity}");
        }
    }

    class Scientist : Employee
    {
        string _scientificDirection;
        public Scientist(string fName, string lName,
            DateTime date, double salary, string
            direction) : base(fName, lName, date,
            salary)
        {
            _scientificDirection = direction;
        }
    }
}

```

```

    public void ShowScientist()
    {
        WriteLine($"Scientists. Scientific direction:
                    {_scientificDirection}");
    }
}

class Specialist : Employee
{
    string _qualification;
    public Specialist(string fName, string lName,
        DateTime date, double salary, string
        qualification) : base(fName, lName,
        date, salary)
    {
        _qualification = qualification;
    }
    public void ShowSpecialist()
    {
        WriteLine($"Specialist. Qualifications:
                    {_qualification}");
    }
}

class Program
{
    static void Main(string[] args)
    {
        Employee manager = new Manager("John",
            "Doe", new DateTime(1995,7,23),
            3500, "food");

        Employee[] employees = {
            manager,
            new Scientist("Jim", "Beam",
                new DateTime(1956,3,15), 4253, "history"),
            new Specialist("Jack", "Smith",
                new DateTime(1996,11,5), 2587.43, "physics")
        };
    }
}

```

```

foreach (Employee item in employees)
{
    item.Print();
    //item.ShowScientist(); Error

    try
    {
        ((Specialist)item).
            ShowSpecialist(); // the method 1
    }
    catch
    {
    }

    Scientist scientist = item as
        Scientist; // the method 2

    if (scientist != null)
    {
        scientist.ShowScientist();
    }

    if (item is Manager) // the method 3
    {
        (item as Manager).ShowManager();
    }
}
}
}
}

```

But an attempt to call any of the methods of the derived classes will lead to an error at the compile time (Figure 3.1), because the base class does not know anything about the fields and methods in the derived class.

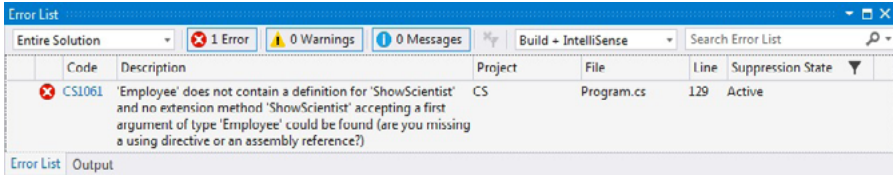


Figure 3.1. Error: there is no definition in the class for the corresponding method

There are three ways to solve this problem (see comments in the code).

*The first way* is implementation of an explicit cast to the required type. Using this method, you should remember that our array, actually, consists of the elements of different classes, and the incorrect typecasting will generate an exception (error at the runtime). Therefore, the `try-catch` instructions should be used together with an explicit cast (this will be discussed in the subsequent lesson).

You should also pay attention to the syntax of the explicit cast to the class:

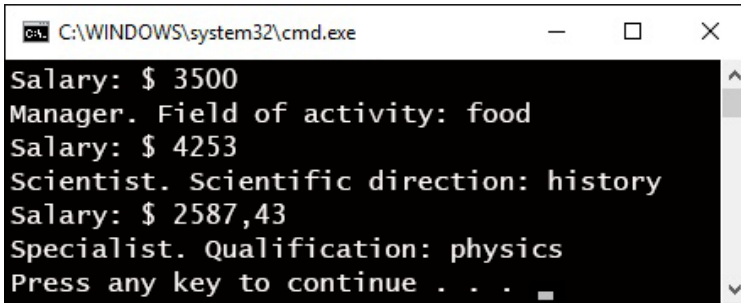
```
((Specialist)item).ShowSpecialist();
```

You can see that the explicit cast itself is enclosed in brackets, i.e. we first perform casting, and then we call the necessary method in the class, to which we have casted.

*The second way* is to use the `as` keyword. Using this operator, we try to cast one type to another, and assign the result to a reference. After this, the reference value is to be compared with `null`. In other words, to check whether the reference contains any value, and if positive, we call the method of the corresponding class.

*The third method* is the use of the `is` keyword. This operator allows determining the compatibility of types, and the result of check is a `bool` data type. If the casting is possible, it returns `true`, otherwise `false`. After receiving a positive result, we can safely perform the casting and call the desired method.

The outcome of the code described above is shown in the Figure 3.2.



```
C:\WINDOWS\system32\cmd.exe
Salary: $ 3500
Manager. Field of activity: food
Salary: $ 4253
Scientist. Scientific direction: history
Salary: $ 2587,43
Specialist. Qualification: physics
Press any key to continue . . .
```

Figure 3.2. The use of the references to the base class

## 4. Polymorphism in C#. Virtual Methods

---

Polymorphism allows derived classes to create own implementation of the methods defined by the base class through a process called method overriding. The `virtual` and `override` keywords are used to implement this process.

### What is a Virtual Method?

Sometimes you need to change the implementation of the methods, which are inherited from the base class. In order to provide this opportunity, you need to use virtual methods that are declared using the `virtual` keyword. By declaring the class method as virtual, you hereby allow derived classes to override this method if necessary.

There are some features related to the virtual methods:

- field members and static methods cannot be declared as virtual.
- the use of virtual methods allows implementing the mechanism of the late binding. In the late binding, the called method is defined at the runtime (rather than at the compile time), depending on the type of the object for which the virtual method is called.
- only a table of the virtual methods is built at the compile time, and the specific address of the method to be called is determined at the runtime.
- When you call the class member method, the following rules are true:



- a method that corresponds a type of the object referred to is called for virtual method;
- a method that corresponds a type of the reference itself is called for non-virtual method.

## Virtual Method Overriding

The `override` keyword is used in a derived class to `override` a virtual base class method.

Consider the features of using virtual methods on the example of the `Human` and `Employee` classes. The code for creating classes shown below is already known to you, but in this case, we are interested in implementing the `Print()` method in different situations.

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

namespace SimpleProject
{
    public class Human
    {
        string _firstName;
        string _lastName;
        DateTime _birthDate;

        public Human(string fName, string lName,
                     DateTime date)
        {
            _firstName = fName;
            _lastName = lName;
            _birthDate = date;
        }
    }
}
```

```

    public void Print()
    {
        WriteLine($"\\nLast name: {_lastName} \\
                    nFirst name: {_firstName} \\
                    nBirth date: {_birthDate.
                    ToShortDateString()}");
    }
}

public class Employee : Human
{
    double _salary;

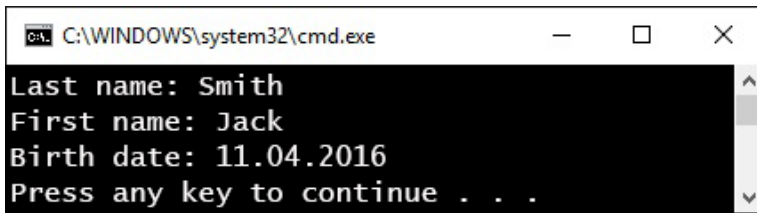
    public Employee(string fName, string lName,
                    DateTime date, double salary)
        : base(fName, lName, date)
    {
        _salary = salary;
    }

    public void Print()
    {
        WriteLine($"Salary: $ {_salary}");
    }
}

class Program
{
    static void Main(string[] args)
    {
        Human employee = new Employee("Jack",
                                        "Smith", DateTime.Now, 3587.43);
        employee.Print();
    }
}

```

For the first example, we have declared the `Print()` method in both classes, and in the `Main()` method, we assigned an instance of the `Employee` derived class to the reference to the `Human` base class, which allows us to create the inheritance mechanism. Ignoring the warnings of the compiler, we run the program and see the result shown in the Figure 4.1.



```
C:\WINDOWS\system32\cmd.exe
Last name: Smith
First name: Jack
Birth date: 11.04.2016
Press any key to continue . . .
```

Figure 4.1. Calling a method without overriding

The result may seem somewhat unexpected, because we have created an instance of the `Employee` class, but it resulted in calling the `Print()` method of the `Human` class. Actually, this situation occurred due to the working principle of the C# compiler. When converting your source code into the CIL code, the compiler attempts to substitute their implementation at the compile time for faster method execution, taking into account the reference to the class, by means of which the method is called.

In this case, when assessing the method, the compiler acts in a following way: a reference that called the `Print()` method is a reference to the `Human` class, while the method itself is declared without the `virtual` keyword, so the implementation of this method is substituted at the compile time. The fact that an instance of `Employee` class is assigned to this reference at the runtime has no effect on calling the `Print()` method.

To demonstrate the second example, we add the `virtual` keyword when declaring the `Print()` method of the `Human` class, and when declaring the `Print()` method of the `Employee` class, we add the `override` keyword.

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

namespace SimpleProject
{
    public class Human
    {
        // Rest of the code remains unchanged
        public virtual void Print()
        {
            WriteLine($"Last name: {_lastName}\nFirst name: {_firstName}\nBirth date: {_birthDate.ToShortDateString()}");
        }
    }

    public class Employee : Human
    {
        // Rest of the code remains unchanged
        public override void Print()
        {
            WriteLine($"Salary: $ {_salary}");
        }
    }

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

```

```
Human employee = new Employee("Jack",  
    "Smith", DateTime.Now, 3587.43);  
employee.Print();  
}  
}  
}
```

What will happen in this situation? The compiler has determined that the reference that called the `Print()` method is a reference to the `Human` class, while the method itself is declared with the `virtual` keyword. It tells the compiler that this method can be overridden in the derived classes, so its implementation should be substituted at the runtime. At the runtime, an instance of the `Employee` class is assigned to the reference to the `Human` class, while the `Print()` method in this class is overridden (`override` keyword), calling the implementation of the method of this particular class. The result is shown in the Figure 4.2.



Figure 4.2. Calling the overridden method

As an experiment, let's make changes to our code: we leave the `virtual` keyword when declaring the `Print()` method of the `Human` class, but remove the `override` keyword when declaring the `Print()` method of the `Employee` class.

```

using System;
using static System.Console;

namespace SimpleProject
{
    public class Human
    {
        // Rest of the code remains unchanged

        public virtual void Print()
        {
            WriteLine($"\\nLast name: {_lastName} \\
                nFirst name: {_firstName} \\
                nBirth date: {_birthDate.
                ToShortDateString()}");
        }
    }

    public class Employee : Human
    {
        // Rest of the code remains unchanged

        public void Print()
        {
            WriteLine($"Salary: $ {_salary}");
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
            Human employee = new Employee("Jack",
                "Smith", DateTime.Now, 3587.43);
            employee.Print();
        }
    }
}

```

Actually, the result of the code execution will not differ from the original result (Figure 4.1). That's because the compiler, though not substituting the implementation of the `Print()` method at the compile time, but at the runtime it turned out, that the `Print()` method is not overridden in the `Employee` class, thus the appropriate method of the `Human` class will be executed.

The base class can have several virtual methods, overriding which in the derived classes, the programmer should remember their names and have to write a certain amount of code. A special function ("feature" in the programmers' jargon) was provided in Visual Studio 2015 to facilitate this process. It works as follows. In the descendant class, you should write the `override` keyword and press the spacebar. This will result in the list of methods that can be overridden. This list will always contain the `System.Object` class methods, but we will talk about them in the section 6. Brief information is displayed next to any method when selecting it from the list (Figure 4.3).

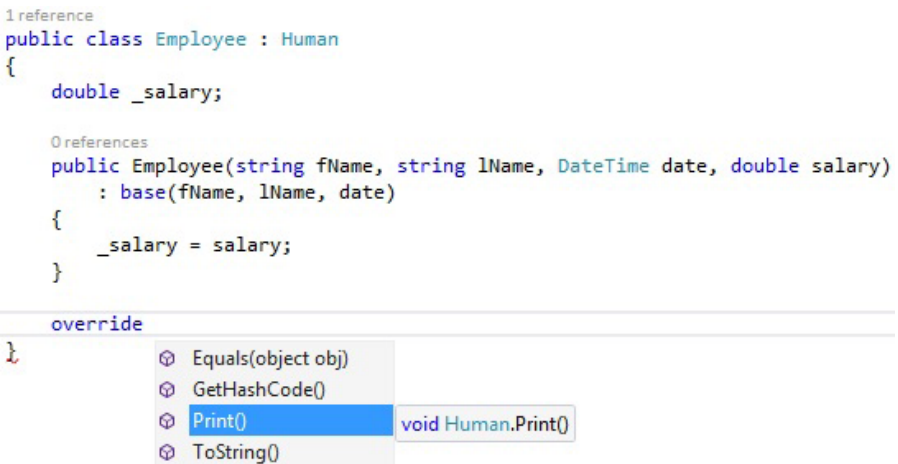


Figure 4.3. List of methods for overriding

After pressing «Enter» or double clicking the selected item with the left mouse button, the framework will generate the override of the method you choose (Figure 4.4).

```

1 reference
public abstract class Employee : Human
{
    double _salary;

    0 references
    public Employee(string fName, string lName, DateTime date, double salary)
        : base(fName, lName, date)
    {
        _salary = salary;
    }

    2 references
    public override void Print()
    {
        base.Print();
    }
}

```

Figure 4.4. The result of overridden method generation

As you may have noticed, the framework automatically substitutes the call of the base class to the method override using the `base` keyword. This behavior is not mandatory, but is used in most cases. At that, the call of the base class method is possible both before and after the implementation in the derived class. Depending on the order of calling the base class method, the result will be corresponding. To demonstrate this, we introduce changes to our example.

```

using System;
using static System.Console;

namespace SimpleProject
{

```



```

public class Human
{
    // Rest of the code remains unchanged

    public virtual void Print()
    {
        WriteLine($"\\nLast name: {_lastName} \\nFirst name: {_firstName} \\nBirth date: {_birthDate.ToShortDateString()}");
    }
}

public class Employee : Human
{
    // Rest of the code remains unchanged

    public override void Print()
    {
        base.Print();
        WriteLine($"Salary: $ {_salary}");
    }
}

class Manager : Employee
{
    string _fieldActivity;

    public Manager(string fName, string lName,
        DateTime date, double salary, string
        activity) : base(fName, lName, date,
        salary)
    {
        _fieldActivity = activity;
    }

    public override void Print()
    {
        Write($"\\nManaged. Field of activity: {_fieldActivity}");
    }
}

```

```

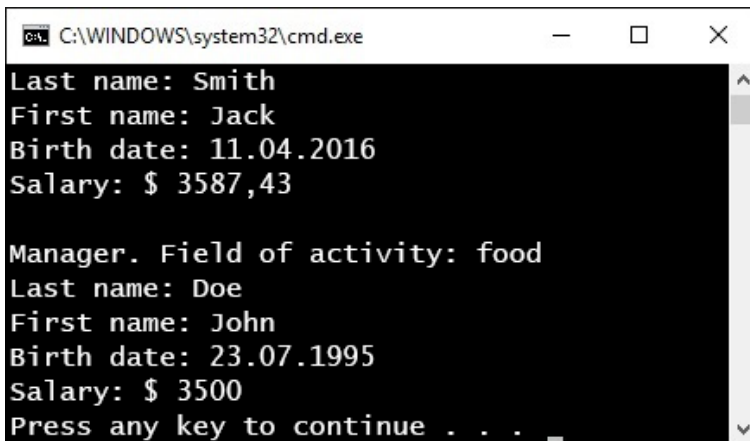
        base.Print();
    }
}

class Program
{
    static void Main(string[] args)
    {
        Human employee = new Employee("Jack",
                                       "Smith", DateTime.Now, 3587.43);
        employee.Print();

        Human manager = new Manager("John",
                                     "Doe", new DateTime(1995, 7, 23), 3500,
                                     "food");
        manager.Print();
    }
}

```

In the `Print()` method of the `Employee` class, we called the `Print()` method of the `Human` base class before its own



```

C:\WINDOWS\system32\cmd.exe
Last name: Smith
First name: Jack
Birth date: 11.04.2016
Salary: $ 3587,43

Manager. Field of activity: food
Last name: Doe
First name: John
Birth date: 23.07.1995
Salary: $ 3500
Press any key to continue . . .

```

Figure 4.5. Calling the base class method in method overriding

implementation, and in the `Manager` class, we did it after the implementation. As you can see, the sequence of the commands being executed affected the results (Figure 4.5).

I would like to draw your attention to a certain feature of the code. Since the `Human` class is a base class for this inheritance chain, and its `Print()` method is declared virtual, then any of the derived classes (`Employee`, `Manager`, etc.) can override this method in accordance with their needs. The result of creating the instances of these classes can be assigned to the reference to the `Human` class, as displayed in the above code.

Inheritance chain can be interrupted in two ways. In the first case, we use the `new` keyword, thus hiding the implementation of the base class method (see Section 1). Let's see where it leads: in the method of `Employee` class we replace the `override` keyword with the `new` keyword.

```
// Rest of the code remains unchanged

public class Employee : Human
{
    // Rest of the code remains unchanged

    public new void Print()
    {
        base.Print();
        WriteLine($"Salary: $ {_salary}");
    }
}

class Manager : Employee
{
    // Rest of the code remains unchanged
```

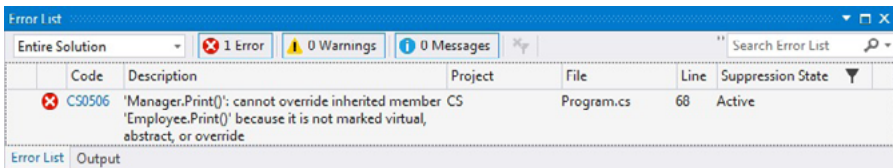
```

public override void Print()
{
    Write($"{\nManager. Field of activity:
        {_fieldActivity}");
    base.Print();
}

// Rest of the code remains unchanged

```

And we get an error at the compile time (Figure 4.6), which occurred due to the impossibility to override the `Print()` method in the `Manager` class, since the corresponding method of the `Employee` base class does not permit this.



Error List						
Entire Solution ▾ 1 Error 0 Warnings 0 Messages						
	Code	Description	Project	File	Line	Suppression State
	CS0506	'Manager.Print()': cannot override inherited member CS 'Employee.Print()' because it is not marked virtual, abstract, or override	CS	Program.cs	68	Active

Figure 4.6. Error: cannot override the method

You can also close further overriding of the base class method. To do this, when declaring an overridden method in the derived class, in addition to the `override` keyword, you should also specify the `sealed` keyword. Let's introduce the appropriate changes in the `Print()` method of the `Employee` class.

```

// Rest of the code remains unchanged
public class Employee : Human
{
    // Rest of the code remains unchanged

    public sealed override void Print()
    {

```

```

        base.Print();
        WriteLine($"Salary: $ {_salary}");
    }
}

class Manager : Employee
{
    // Rest of the code remains unchanged

    public override void Print()
    {
        Write($"\\nManager. Field of activity:
            {_fieldActivity}");
        base.Print();
    }
}

// Rest of the code remains unchanged

```

The changes will also result in an error at the compile time (Figure 4.7), which is associated with the inability to override the `Print()` method in the `Manager` class, because it is declared as sealed in the `Employee` base class.

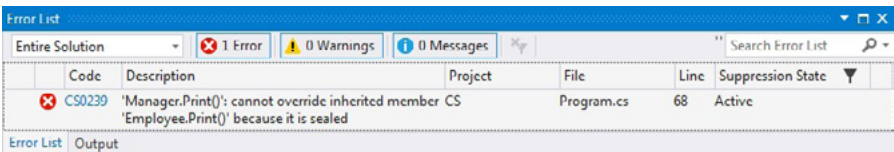


Figure 4.7. Error: cannot override the method

## The Need to Use Virtual Methods

Virtual methods are needed when the derived class requires changing some of the methods defined in the base class. Virtual methods allow the base class to define methods, the implementation of which is common for all derived

classes with the ability to override these methods if necessary. This allows maintaining dynamic polymorphism. Definition of own methods in the derived classes becomes more flexible, still leaving in force the requirement of the consistent interface.

At the end of this section, we will give an extended example based on the code of the section 3, but will change it in accordance with the knowledge acquired. Although the implementation of the [Human](#), [Employee](#) and [Manager](#) classes was already cited in this section, we will show all the code completely, removing thus any extra questions.

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

namespace SimpleProject
{
    public class Human
    {
        string _firstName;
        string _lastName;
        DateTime _birthDate;

        public Human(string fName, string lName,
                     DateTime date)
        {
            _firstName = fName;
            _lastName = lName;
            _birthDate = date;
        }

        public virtual void Print()
        {
```

```

        WriteLine($"\\nLast name: {_lastName} \\nFirst name: {_firstName} \\nBirth date: {_birthDate.ToLongDateString()}");
    }
}

public class Employee : Human
{
    double _salary;

    public Employee(string fName, string lName,
        DateTime date, double salary) :
        base(fName, lName, date)
    {
        _salary = salary;
    }

    public override void Print()
    {
        base.Print();
        WriteLine($"Salary: $ {_salary}");
    }
}

class Manager : Employee
{
    string _fieldActivity;

    public Manager(string fName, string lName,
        DateTime date, double salary,
        string activity) : base(fName,
        lName, date, salary)
    {
        _fieldActivity = activity;
    }
}

```

```

    public override void Print()
    {
        Write($"{nManager. Field of activity:
            {_fieldActivity}");
        base.Print();
    }
}

class Scientist : Employee
{
    string _scientificDirection;

    public Scientist(string fName, string lName,
        DateTime date, double salary, string
        direction) : base(fName, lName, date,
        salary)
    {
        _scientificDirection = direction;
    }

    public override void Print()
    {
        Write($"{nScientist. Scientific direction:
            {_scientificDirection}");
        base.Print();
    }
}

class Specialist : Employee
{
    string _qualification;

    public Specialist(string fName, string lName,
        DateTime date, double salary,
        string qualification) :
        base(fName, lName, date, salary)
    {

```



```

        _qualification = qualification;
    }

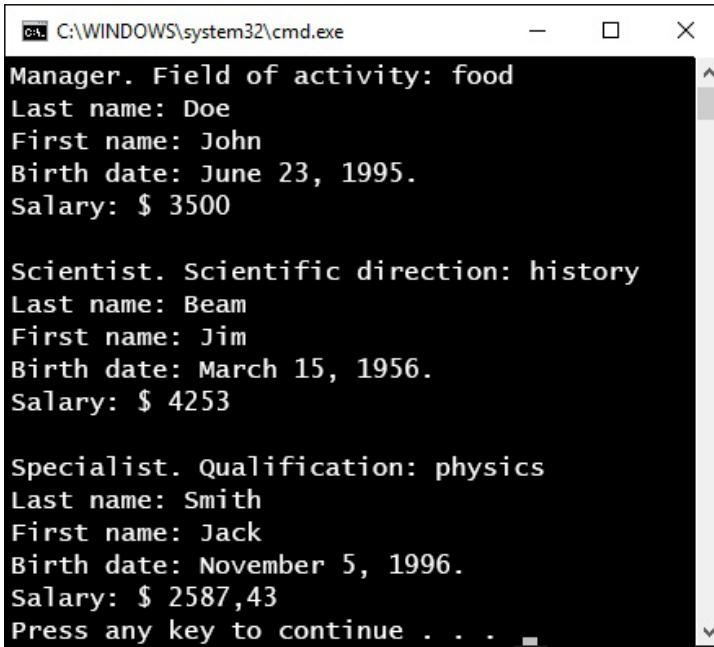
    public override void Print()
    {
        Write($"{_qualification}");
        base.Print();
    }
}

class Program
{
    static void Main(string[] args)
    {
        Human[] people = {
            new Manager("John", "Doe",
                new DateTime(1995, 7, 23), 3500,
                "food"),
            new Scientist("Jim", "Beam",
                new DateTime(1956, 3, 15), 4253, "history"),
            new Specialist("Jack", "Smith",
                new DateTime(1996, 11, 5), 2587.43, "physics")
        };

        foreach (Human item in people)
        {
            item.Print(); // polymorphism
        }
    }
}

```

The result of the program (Figure 4.8).



```
C:\WINDOWS\system32\cmd.exe
Manager. Field of activity: food
Last name: Doe
First name: John
Birth date: June 23, 1995.
Salary: $ 3500

Scientist. Scientific direction: history
Last name: Beam
First name: Jim
Birth date: March 15, 1956.
Salary: $ 4253

Specialist. Qualification: physics
Last name: Smith
First name: Jack
Birth date: November 5, 1996.
Salary: $ 2587,43
Press any key to continue . . .
```

Figure 4.8. The use of polymorphism

What distinguishes this code from the previous version? If you do not consider overriding of the methods in the derived classes, then the main difference is in the `Main()` method, namely, in the `foreach` loop, where the `Print()` method is called for each element of the `Human` type array. We call this method without worrying about whether its implementation exists in each particular class. Due to the fact that we overrode the methods in the derived classes, for each element namely its implementation will be called.

If you expand the inheritance hierarchy by creating new classes, and add them to the array, the operation of the `foreach` loop will remain unchanged. And even if you do not override the `Print()` method in the derived class for some reason,

this still won't lead to an error, and the `Human` base class method will be called. We suggest you to check this yourself by commenting the `Print()` method in any derived class. Everything described above is a polymorphism.

## 5. Abstract Class

---

*An **abstract class*** is a class, in which one of the methods is abstract, i.e. a method which has no implementation. Instances of this class cannot be created, but they can be inherited by other classes. The `abstract` keyword is used for declaring an `abstract` class.

It is viable to declare a class abstract if it defines a kind of umbrella term, while the creation and use of the instances of this class makes no sense. Consider the classic example — a `Figure` class. If we create an instance of this class, it is not clear how to draw this figure, how many sides and what area does it have, etc. The `Figure` class may serve as a basis for creating a variety of other classes (`Rectangle`, `Circle`, etc.), being a kind of abstraction that has common functional (constructors, fields, etc.) and methods that should be overridden in derived classes.

The `Human` and `Employee` classes are also present in the examples we discussed earlier. Both classes are some general concepts, since people differ from each other at least anatomically, and employee may be both a cleaner and a president. Therefore, we will further declare them as abstract.

As an example of working with the abstract classes, let's create a `Learner` class inherited from the `Human` class, and declare both classes as abstract.

Essentially, abstract class still remains a class, so it may have everything that is present in a usual class (fields, constructors, methods, etc.), but it differs from a usual class with an abstract method. So we will declare an abstract

`Think()` method in the `Human` class. There is no implementation in the abstract method, so when we declare such a method in the class, we "demand" from the derived classes to override it (using the `override` keyword) on a mandatory basis. If the derived class "does not want" to `override` the abstract method for some reason, then this class should be declared abstract, otherwise the compiler will issue an error (Figure 5.1).

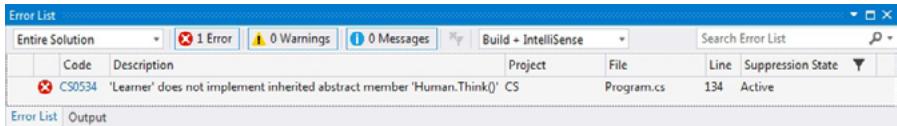


Figure 5.1. Error: the class has no implementation for the `Think()` method

In this case, the decision to declare the `Think()` method abstract is quite justified, because one way or another, all the people think differently, so it would be logical if all the successors of the `Human` class will provide their own implementation of this method.

Let's declare an `_institution` field in the `Learner` class and a `Study()` abstract method. The resulting class will be the base for the `Student` and `SchoolChild` derived classes.

Before showing you the final code, we would like to acquaint you with one more specific possibility of Visual Studio 2015 that allows deciding the methods necessary for overriding in derived classes when inheriting from an abstract class.

After you specify when declaring a class that it is a descendant of an abstract class, the compiler will issue an error immediately (Figure 5.2), indicating that you should override the abstract methods and create a constructor.

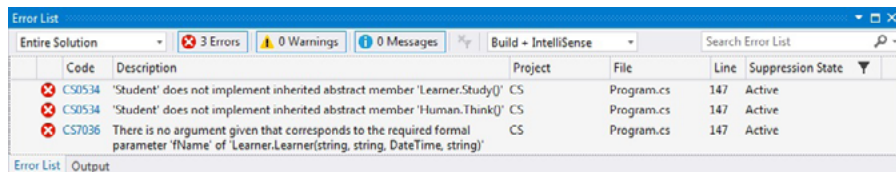


Figure 5.2. Error: no implementation for abstract methods in the class

After a single click on the name of an abstract class in the inheritance string with the left mouse button, the button with an image of light bulb appears at the left of the string; the click on this button will display a list of necessary actions (Figure 5.3 and 5.4).

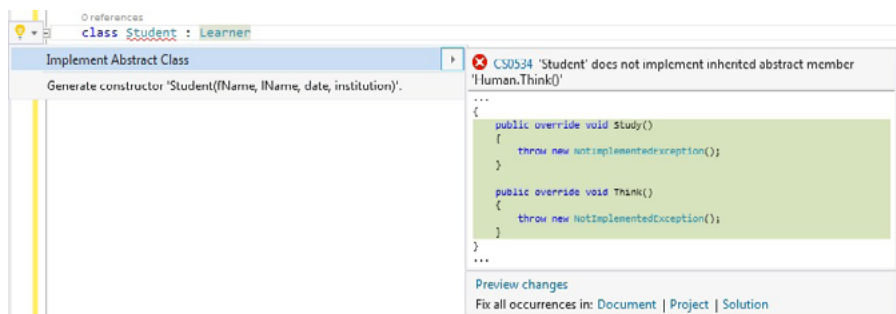


Figure 5.3. The implementation of an abstract class

When moving through items of the list, at the right you will see a preview window for the implementation of the selected item.



Figure 5.4. Constructor creation item

To generate the required code automatically, click on the corresponding list item. The outcome is shown in the Figure 5.5.

```

1 reference
class Student : Learner
{
    0 references
    public Student(string fName, string lName, DateTime date, string institution) : base(fName, lName, date, institution)
    {
    }

    1 reference
    public override void Study()
    {
        throw new NotImplementedException();
    }

    1 reference
    public override void Think()
    {
        throw new NotImplementedException();
    }
}

```

Figure 5.5. Outcome of automatic code generation

When selecting an item, it is better to start with the creation of a constructor. In this case, the item for implementation of abstract class methods will become available. If you do the opposite, you will have to automatically create the class constructor in other way.

As an alternative to the above method, you can call a context menu with a right-click either on the base class, or the derived class and select Quick Actions... (Figure 5.6).

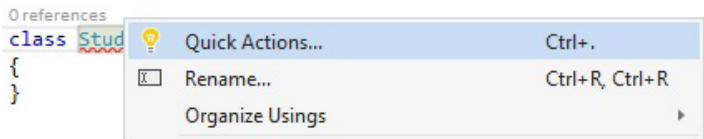


Figure 5.6. Context menu option for code generation

After that, you will see the items familiar to you (Figure 5.3 and 5.4), the selection of which will lead to generation of the code shown in the Figure 5.5.

As you may notice, the framework creates non-empty templates for overriding all the necessary abstract methods. A string called "stub" will be inserted as the implementation of these methods. Actually, this is a generation of an exception (which will be discussed in a subsequent lesson). Your task is to write your own implementation of the method instead of this string.

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

namespace SimpleProject
{
    public abstract class Human
    {
        string _firstName;
        string _lastName;
        DateTime _birthDate;
        public Human(string fName, string lName,
                      DateTime date)
        {
            _firstName = fName;
            _lastName = lName;
            _birthDate = date;
        }

        public abstract void Think();

        public virtual void Print()
        {
            WriteLine($"Last name: {_lastName}\nFirst name: {_firstName}\nBirth date: {_birthDate.ToLongDateString()}");
        }
    }
}
```



```

abstract class Learner : Human
{
    string _institution;

    public Learner(string fName, string lName,
        DateTime date, string institution) :
        base(fName, lName, date)
    {
        _institution = institution;
    }

    public abstract void Study();

    public override void Print()
    {
        base.Print();
        WriteLine($"Institution: {_institution}.");
    }
}

class Student : Learner
{
    string _groupName;

    public Student(string fName, string lName,
        DateTime date, string institution,
        string groupName) : base(fName, lName,
        date, institution)
    {
        _groupName = groupName;
    }

    public override void Think()
    {
        WriteLine("I think as a student.");
    }
}

```

```

    public override void Study()
    {
        WriteLine("I study subjects at university.");
    }

    public override void Print()
    {
        base.Print();
        WriteLine($"I study in the {_groupName}
                    group.");
    }
}

class SchoolChild : Learner
{
    string _className;

    public SchoolChild(string fName, string
                        lName, DateTime date,
                        string institution, string className) :
        base(fName, lName, date, institution)
    {
        _className = className;
    }

    public override void Think()
    {
        WriteLine("I think as a schoolchild.");
    }

    public override void Study()
    {
        WriteLine("I study subjects at school.");
    }

    public override void Print()
    {

```

```

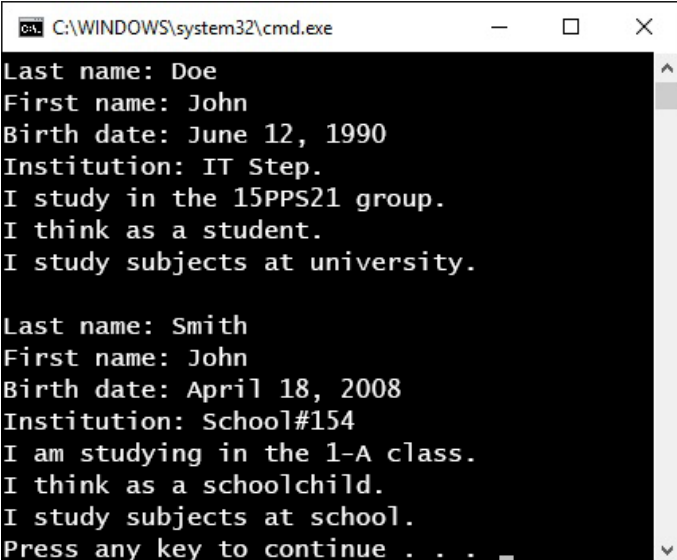
        base.Print();
        WriteLine($"I study in the {_className}
                    class.");
    }
}

class Program
{
    static void Main(string[] args)
    {
        Learner[] learners =
        {
            new Student("John", "Doe",
                new DateTime(1990, 6, 12), "IT Step",
                "15PPS21"),
            new SchoolChild("Jack", "Smith",
                new DateTime(2008, 4, 18),
                "School#154", "1-A")
        };

        foreach (Learner item in learners)
        {
            item.Print();
            item.Think();
            item.Study();
        }
    }
}

```

Outcome (Figure 5.7).



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

Last name: Doe
First name: John
Birth date: June 12, 1990
Institution: IT Step.
I study in the 15PPS21 group.
I think as a student.
I study subjects at university.

Last name: Smith
First name: John
Birth date: April 18, 2008
Institution: School#154
I am studying in the 1-A class.
I think as a schoolchild.
I study subjects at school.
Press any key to continue . . .
```

Figure 5.7. Abstract class inheritance chain

## 6. Analysis of the Object Base Class

---

Since in C# each type is a successor of the `Object` class, then its methods are available for any class, and the virtual methods can be overridden if necessary. Consider the methods of the `Object` class:

- `Equals` is a virtual method that receives an `object` type as a parameter. It returns `true` if the object that calls the method and the object being passed as a parameter are identical, otherwise returns `false`.
- `Equals` is a static method that gets two `object` types as a parameter. Returns `true` if the objects are identical; otherwise returns `false`.
- `Finalize` allows performing operations on cleaning up the resources allocated for a current object before being removed by the garbage collector.
- `GetHashCode` is a virtual method that returns a hash code of a caller object.
- `GetType` returns the `type` class object for a current instance.
- `MemberwiseClone` creates a "superficial" copy of a caller object, to which all the members of a class are copied, but it does not copy the objects, which these members refer to.
- `ReferenceEquals` is a static method that returns `true` if two objects (which are passed as parameters) refer to the same object. If they refer to different objects, it returns `false`.

- `ToString` is a virtual method that returns a string representation of a current object.

We will consider the features of the `Object` class methods gradually, with studying the C# language. In this lesson, we will work with the `GetType()` and `ToString()` methods.

During this lesson, in order to output information on the classes, we used the `Print()` method written by ourselves, but there is a common and much easier way — overriding of the `ToString()` method of the `Object` class. This method returns a representation of an object as a string, it is overridden in all standard data types (`int`, `double`, and etc.) and is called for them automatically when attempting to output variables of these types to the console (`Write()` and the `WriteLine()` methods). You've already seen it, but did not pay attention. Look at the previous code and make sure that the `ToString()` method is not called explicitly anywhere, and a string is output.

Let's introduce changes to the code written previously — in our classes replace the `Print()` method with the overridden `ToString()` method. Due to this, the output of information on classes in the `Main()` method will be simplified.

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

namespace SimpleProject
{
    public abstract class Human
    {
        string _firstName;
        string _lastName;
        DateTime _birthDate;
    }
}
```

```

public Human(string fName, string lName,
              DateTime date)
{
    _firstName = fName;
    _lastName = lName;
    _birthDate = date;
}

public abstract void Think();
public override string ToString()
{
    return $" \nLast name: {_lastName} \nFirst: {_firstName} \nBirth date: {_birthDate.ToLongDateString()} ";
}
}

abstract class Learner : Human
{
    string _institution;

    public Learner(string fName, string lName,
                  DateTime date, string institution) :
        base(fName, lName, date)
    {
        _institution = institution;
    }

    public abstract void Study();
    public override string ToString()
    {
        return base.ToString() + $" \nInstitution: {_institution}.";
    }
}

class Student : Learner
{

```

```

    string _groupName;

    public Student(string fName, string lName,
        DateTime date, string institution,
        string groupName) : base(fName, lName,
            date, institution)
    {
        _groupName = groupName;
    }

    public override void Think()
    {
        WriteLine("I think as a student.");
    }

    public override void Study()
    {
        WriteLine("I study subjects
            at university.");
    }

    public override string ToString()
    {
        return base.ToString() + $"
I study in the
    {_groupName} group.";
    }
}

class SchoolChild : Learner
{
    string _className;
    public SchoolChild(string fName,
        string lName, DateTime date,
        string institution, string className) :
        base(fName, lName, date, institution)
    {
        _className = className;
    }
}

```



```

public override void Think()
{
    WriteLine("I think as a schoolchild.");
}

public override void Study()
{
    WriteLine("I study subjects at school.");
}

public override string ToString()
{
    return base.ToString() + $" \nI study in the
        {_className} class.";
}
}

class Program
{
    static void Main(string[] args)
    {
        Learner[] learners =
        {
            new Student("John", "Doe",
                new DateTime(1990, 6, 12), "IT Step",
                "15PPS21"),
            new SchoolChild("Jack", "Smith",
                new DateTime(2008, 4, 18),
                "School#154", "1-A")
        };
        foreach (Learner item in learners)
        {
            WriteLine(item);
            item.Think();
            item.Study();
        }
    }
}

```

The outcome is similar to the previous one (Figure 6.1).

```
cmd: C:\WINDOWS\system32\cmd.exe
Last name: Doe
First name: John
Birth date: June 12, 1990
Institution: IT Step.
I am studying in the 15PPS21 group.
I think as a student.
I study subjects at university.

Last name: Smith
First name: John
Birth date: April 18, 2008
Institution: School#154
I am studying in the 1-A class.
I think as a schoolchild.
I study subjects at school.
Press any key to continue . . .
```

Figure 6.1. Overriding of the ToString() method in classes

As described above, when calling the `GetType()` method, we obtain the `Type` class object, which in its turn allows getting detailed information on the object that called it. Basically, the `Type` class is used for type reflection, which will be discussed in the subsequent courses. Now we will demonstrate only some of the methods of this class. You can see full details on it in MSDN. Let's take the previous code and introduce changes only in the `Main()` method of the `Program` class, the rest of the code will not be shown in order to save space.

```
// Rest of the code remains unchanged
class Program
{
    static void Main(string[] args)
    {
```

```

Student student = new Student("John",
    "Doe", new DateTime(1990, 6, 12),
    "IT Step", "15PPS21");
WriteLine($"Full name of the type -
    {student.GetType().FullName}.");
WriteLine($"Name of the current element -
    {student.GetType().Name}.");
WriteLine($"Base class of the current
    element - {student.GetType().
    BaseType}.");
WriteLine($"Whether the current element
    is an abstract object-
    {student.GetType().IsAbstract}.");
WriteLine($"Whether the object is a class -
    {student.GetType().IsClass}.");
WriteLine($"Can you access the object
    from the code outside the current
    assembly - {student.GetType().
    IsVisible}.");
    }
}

```

Outcome (Figure 6.2).

```

C:\WINDOWS\system32\cmd.exe
Full name of the type - SimpleProject.Student.
Name of the current element - student.
Base class of the current element - SimpleProject.Learner.
whether the current element is an abstract object - False.
whether the object is a class - True.
Can you access the object from the code outside the current assembly - False.
Press any key to continue . . .

```

Figure 6.2. Some properties of the Type class

The last thing we will consider in this lesson is an opportunity to build a graphical representation of the relationship between the classes in our application — the class diagram represented in the section 1 (Figure 1.1).

To create a class diagram, select Add New Item... in the Project main menu option. A corresponding window will appear, and here you select Class Diagram option (Figure 6.3).

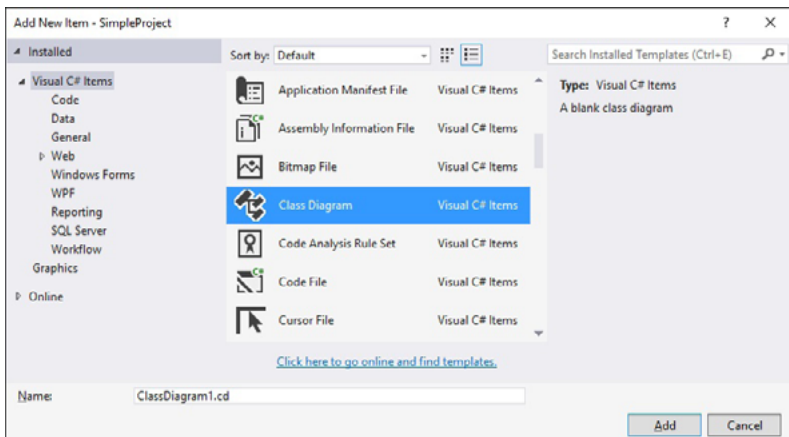


Figure 6.3. Add New Item window

After clicking the Add button, the file with the .cd extension will be added to our project and a diagram creation field will appear (Figure 6.4).

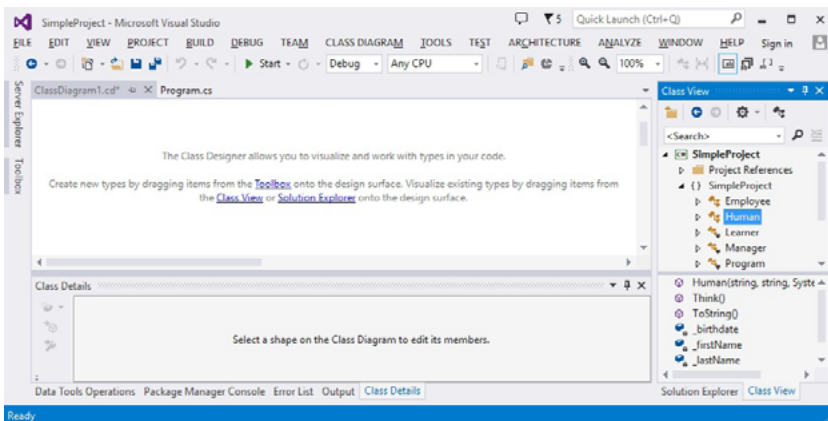


Figure 6.4. Class Diagram file

You just need to drag the necessary classes from the Class View window or the Solution Explorer to this field, and the framework will build relationships between classes automatically. This process is displayed in the Figure 6.5.

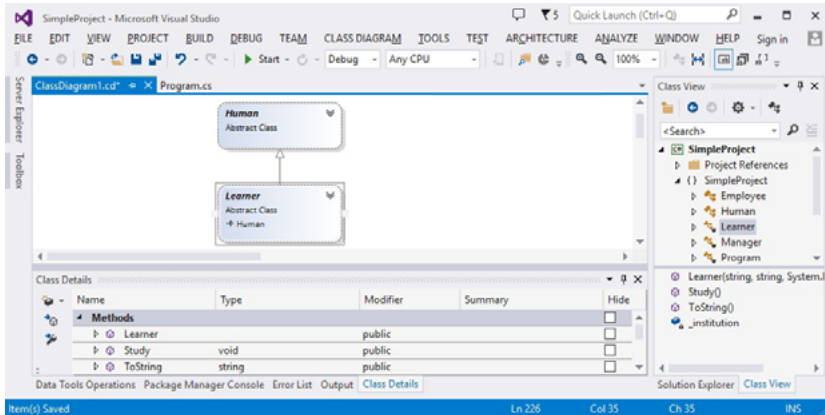


Figure 6.5. The process of building a class diagram

# Home Task

---

## Task 1.

Develop a "Figure" abstract class with the "Area" and "Perimeter" methods. Develop descendant classes: triangle, square, rhombus, rectangle, parallelogram, trapezium, circle, ellipse. Implement constructors that will decisively identify objects of these classes.

Implement a "Composite figure" class, which may consist of any number of "Figures". Define a method of finding the area of figures for this class. Create a class relation diagram.

## Task 2.

Develop a class architecture of commodity hierarchy in the development of commodity traffic management system for a distribution company. Prescribe class members. Create a class relation diagram.

Provide different types of commodities, including:

- household chemicals;
- food.

Provide classes of commodity traffic control (received, sold, written off, transferred).

