

Платформа Microsoft .NET и язык программирования C#



Урок №4

Наследование и полиморфизм

Содержание

1.	Наследование в C#.....	4
	Анализ механизма наследования в C#	4
	Модификаторы доступа при наследовании	5
	Особенности использования	
	конструкторов при наследовании.....	8
	Ключевое слово base	12
	Сокрытие имен при наследовании	13
2.	Использование ключевого слова sealed.....	17
3.	Использование ссылок на базовый класс	19
4.	Полиморфизм в C#. Виртуальные методы	25
	Что такое виртуальный метод?	25
	Переопределение виртуальных методов	26
	Необходимость использования	
	виртуальных методов.....	39

5.	Абстрактный класс	45
6.	Анализ базового класса Object	54
7.	Домашнее задание	63
	Задание 1.	63
	Задание 2.	63

1. Наследование в C#

Анализ механизма наследования в C#

Наследование позволяет повторно использовать уже имеющиеся классы, но при этом расширять их функциональные возможности. Существует два вида наследования — наследование типа, при котором новый тип получает все свойства и методы родителя и наследование интерфейса, при котором новый тип получает от родителя сигнатуру методов, без их реализации. Все классы, для которых не указан базовый класс, наследуются от класса `System.Object` (краткая форма названия `object`). Язык C# не поддерживает множественного наследования. Это означает, что класс в C# может быть наследником только одного класса, но при этом может реализовывать несколько интерфейсов. Другими словами, класс может наследоваться не более чем от одного базового класса и нескольких интерфейсов.

Синтаксис наследования на языке C# выглядит следующим образом:

```
class НаследуемыйКласс : БазовыйКласс, Интерфейс1, .  
    . . .  
        ИнтерфейсN  
{  
    //поля, свойства, события и методы класса  
}
```

На рисунке 1.1 изображен пример диаграммы классов (как построить диаграмму классов средствами Visual Studio 2015, будет рассказано в шестом разделе данного урока).

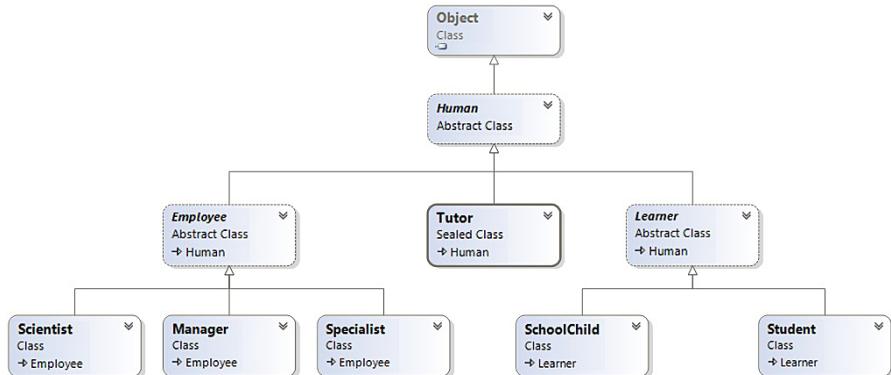


Рисунок 1.1. Пример иерархии наследования классов

Из диаграммы можно видеть, что классы `Employee`, `Learner`, `Tutor` наследуют класс `Human`, который в свою очередь наследуется от `System.Object`. Классы `Human`, `Learner` и `Employee` являются абстрактными классами, а `Tutor` — закрытым. Подробно элементы и их взаимосвязи, изображенные на диаграмме, будут рассмотрены на протяжении данного урока.

Модификаторы доступа при наследовании

В уроке №3 Вы уже изучили модификаторы доступа языка C#, но только модификаторы `protected` и `protected internal` имеют прямое отношение к наследованию. Рассмотрим их применение на следующем примере.

Допустим, у нас объявлен класс `Human`, который является базовым для класса `Employee`. Поля `firstName`

и lastName класса Human объявлены с модификатором **protected**, то есть они доступны во всех классах-наследниках, поэтому мы можем использовать эти поля в классе Employee.

```
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($"Фамилия: {lastName}\nИмя:
                      {firstName}\nЗаработная плата:
                      {_salary} $");
        }
    }
}

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

```

    {
        Employee employee = new Employee("John",
                                         "Doe", 2563.57);
        employee.Print();
    }
}
}

```

Выполнение кода, приведенного выше, не вызывает ошибку (Рисунок 1.2).

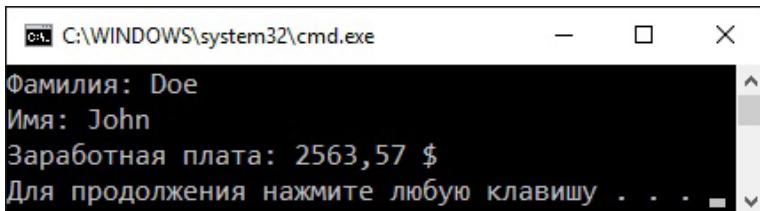


Рисунок 1.2. Пример применения модификатора `protected`

Попытка обратиться в классе `Employee` к полю `_id` приводит к возникновению ошибки (Рисунок 1.3), так как это поле в классе `Human` неявно объявлено с модификатором `private`, поэтому получить доступ к этому полю можно только внутри класса `Human`.

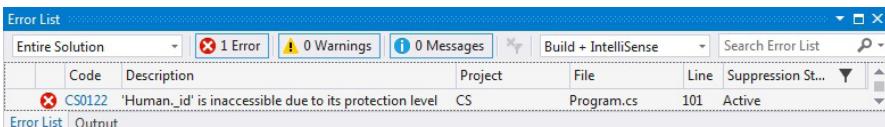


Рисунок 1.3. Ошибка: обращение
к закрытому полю класса

Модификатор `protected internal` работает аналогично модификатору `protected` только на уровне текущей сборки.

Особенности использования конструкторов при наследовании

При создании класса — наследника на самом деле вызывается не один конструктор, а целая цепочка конструкторов. Сначала выбирается конструктор класса, экземпляр которого создается. Этот конструктор пытается обратиться к конструктору своего непосредственного базового класса, тот в свою очередь пытается вызвать конструктор своего базового класса. Так происходит, пока не доходим до класса `System.Object`, который не имеет базового класса. В результате имеем последовательный вызов конструкторов всех классов иерархии, начиная с `System.Object` заканчивая классом, экземпляр которого хотим создать. В этом процессе каждый конструктор инициализирует поля собственного класса.

Для каждого класса можно определить несколько конструкторов. Если мы для класса-наследника хотим вызвать конструктор базового класса, то необходимо использовать ключевое слово `base()`.

```
public НаследуемыйКласс () : base ()  
{  
    // поля, свойства, события и методы класса  
}
```

Усовершенствуем класс `Human`, для этого добавим поле `_birthDate` и два конструктора: конструктор, принимающий в качестве параметров имя и фамилию и конструктор, принимающий в качестве параметров имя, фамилию и дату рождения, а также метод `Show()` для вывода данных.

В классе `Employee` создадим три конструктора: первый конструктор в качестве параметров принимает имя

и фамилию, второй конструктор — имя, фамилию и зарплатную плату, а третий — имя, фамилию, день рождения и зарплатную плату. Все эти конструкторы до выполнения собственного кода, вызывают соответствующие конструкторы базового класса `Human`, используя ключевое слово `base()` для инициализации определенных полей. Чтобы не допустить избыточности кода, мы несколько изменили метод `Print()` — добавили в нем вызов метода `Show()` класса `Human`. Пока что это выглядит несколько топорно, но мы в следующих разделах устраним этот недочет.

Описанные выше изменения в классах представлены в следующем коде:

```
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($"\\nФамилия: {_lastName} \\
                  Имя: {_firstName}\\nДата \\
                  рождения: {_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($"Заработная плата: {_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();
    }
}
}

```

Приблизительный результат работы программы представлен на рисунке 1.4.

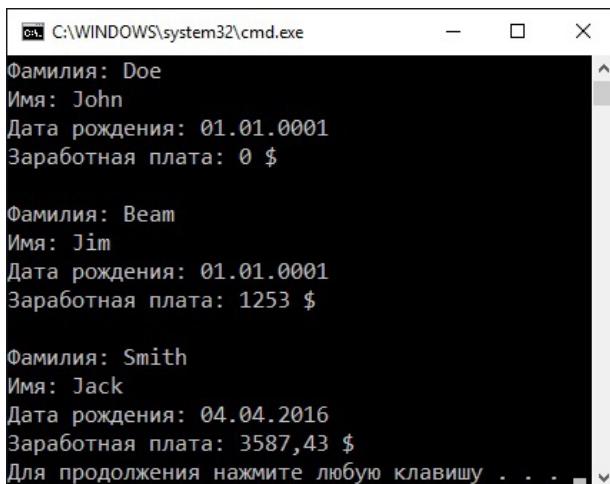


Рисунок 1.4. Использование ключевого слова
при base наследовании

Ключевое слово `base`

Ключевое слово `base` используется не только при создании конструктора класса наследника, который должен вызвать конструктор класса, но и для доступа к членам базового класса из производного и для вызова метода базового класса при его переопределении в классе наследнике (о переопределении методов будет рассказано в последующих разделах). Доступ к базовому классу разрешен только в конструкторе или методах класса наследника.

Вернемся к предыдущему примеру, в нем мы вызывали в классе `Employee` метод `Show()` базового класса `Human`, те же действия можно выполнить с использованием ключевого слова `base`, результаты работы программы будут аналогичными (Рисунок 1.4).

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

namespace SimpleProject
{
    public class Human
    {
        // поля и конструкторы остаются прежними

        public void Show()
        {
            WriteLine($"\\nФамилия: {_lastName}\\nИмя: {_firstName}\\nДата рождения:
{_birthDate.ToShortDateString()}");
        }
    }

    public class Employee : Human
    {
        // поля и конструкторы остаются прежними
```

```

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

class Program
{
    static void Main(string[] args)
    {
        // операции остаются прежними
    }
}
}

```

Использование ключевого слова `base` не изменяет функциональность программы, а как бы конкретизирует, что в данной ситуации используются члены базового класса.

Сокрытие имен при наследовании

Продолжим работу с нашим примером, допустим у нас в базовом классе `Human` и в производном классе `Employee` определены методы с одинаковым именем `Print()`. Будет ли сгенерирована ошибка при выполнении такого кода?

```

using System;
using static System.Console;

namespace SimpleProject
{
    public class Human
    {
        // поля и конструкторы остаются прежними
    }
}

```

```
public void Print()
{
    WriteLine($"\\nФамилия: {_lastName} \\
    Имя: {_firstName}\\nДата рождения:
    {_birthDate.ToShortDateString() }");
}

public class Employee : Human
{
    // поля и конструкторы остаются прежними

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

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

Результат выполнения программы ошибки не вызовет (Рисунок 1.5).

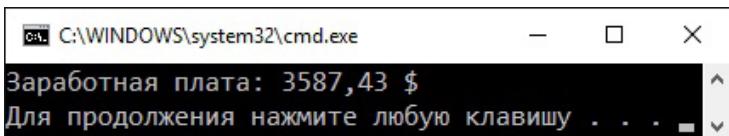


Рисунок 1.5. Результат работы программы

Однако в окне Error List появится Warning (предупреждение) (Рисунок 1.6).

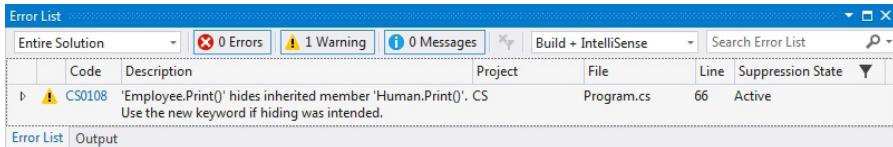


Рисунок 1.6. Предупреждение: необходимость использования ключевого слова new

Смысл этого предупреждения сводиться к следующему: метод Print() класса `Employee` скрывает метод Print() класса `Human`, и если этого и добивался программист, тогда рекомендуется использовать ключевое слово `new` у класса `Employee` при объявлении метода Print().

Следующий код демонстрирует только изменение, которое было внесено в класс `Employee` — ключевое слово `new` при объявлении метода Print(). Выполнение предыдущего кода с этим изменением даст такой же результат (Рисунок 1.5), однако предупреждения уже не будет.

```
public class Employee : Human
{
    // поля и конструкторы остаются прежними

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

И напоследок, ключевое слово `new` можно применять для скрытия любого члена базового класса. Допустим, по какой-то причине мы захотели в классе `Employee`

скрыть произвольное поле класса `Human`, тогда код будет выглядеть следующим образом:

```
public class Human
{
    protected string middleName;

    // остальной код остался прежним
}

public class Employee : Human
{
    new string middleName;

    // остальной код остался прежним
}
```

2. Использование ключевого слова sealed

Иногда возникают ситуации, когда необходимо запретить наследовать некоторый класс или переопределять некоторый метод. Для этого при определении класса или метода применяется ключевое слово `sealed` (запечатанный). Объявим запечатанный класс `Tutor`:

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

Тем самым мы запрещаем использовать класс `Tutor` в качестве базового класса, при этом сам класс `Tutor` может быть наследован от другого класса. Попытка наследоваться от класса `Tutor` приводит к ошибке на этапе компиляции (Рисунок 2.1).

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

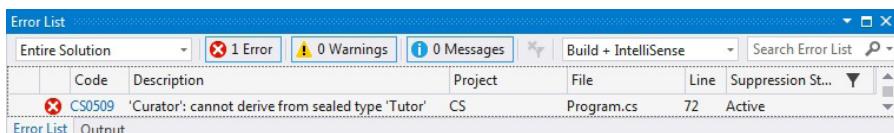
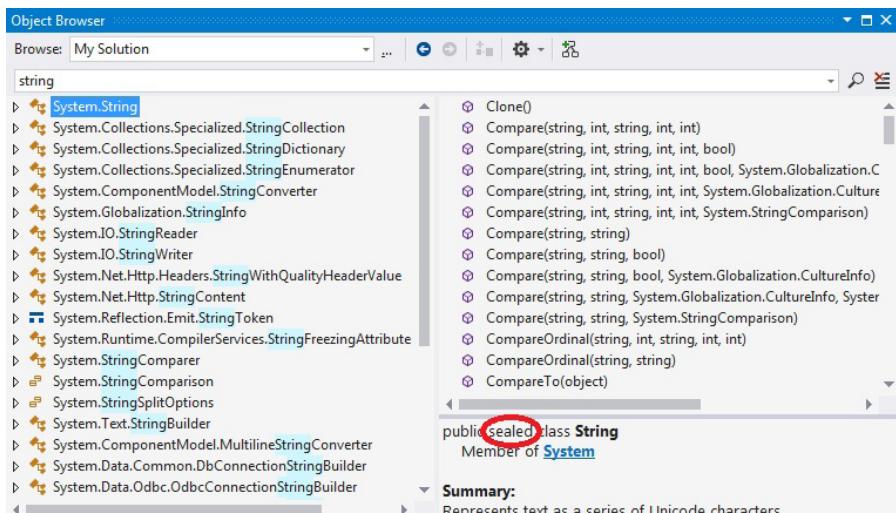


Рисунок 2.1 Ошибка: невозможность наследоваться от запечатанного класса

Как определить, можно ли наследоваться от определенного класса из библиотек .NET? Посмотреть подробную информацию о типах из различных библиотек

.NET можно при помощи окна Object Browser, открыть которое можно, если выбрать пункт главного меню VIEW, а затем пункт Object Browser в выпадающем списке. Для перехода к интересующему Вас типу необходимо в строке поиска ввести его имя, а после этого выбрать из полученного списка тот результат, который Вам подходит. Справа в нижней части окна Вы увидите информацию о выбранном Вами типе, а справа вверху список методов данного типа (если они существуют).

Для примера мы получим информацию о классе `String` (Рисунок 2.2), как Вы можете заметить, этот класс является запечатанным.



**Рисунок 2.2. Информация о классе String
в окне Object Browser**

Использование ключевого слова `sealed` при работе с методами, будет рассмотрено в разделе 4 данного урока.

3. Использование ссылок на базовый класс

Как известно, в C# нельзя присвоить переменной одного типа значение переменной другого типа. Но существует одно исключение — ссылочной переменной базового класса можно присвоить ссылку на объект любого класса-наследника этого базового класса. Важно понимать следующее — когда ссылка на производный класс присваивается ссылочной переменной базового класса, Вы получаете доступ только к тем частям объекта, которые определены в базовом классе.

Разберем это на примере, взяв в качестве базового класса, описанный нами ранее, класс `Employee` и создадим на его основе три класса-наследника `Manager`, `Scientist` и `Specialist`. В созданных классах объявим необходимые поля и методы.

В методе `Main()` объявим ссылку на класс `Employee` и присвоим ей экземпляр класса `Manager`, также объявим массив типа `Employee` и проинициализируем его экземплярами классов `Manager`, `Scientist` и `Specialist`. После этого при помощи цикла `foreach` пройдем по массиву и у каждого элемента массива вызовем метод `Print()`. Все эти действия выполняться корректно, так как мы работаем со ссылкой на базовый класс.

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

namespace SimpleProject
```

```
{  
    public class Human  
    {  
        // реализация класса остается прежней  
    }  
  
    public class Employee : Human  
    {  
        // реализация класса остается прежней  
    }  
  
    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($"Менеджер. Сфера деятельности:  
                      {_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($"Ученый. Научное направление:
              {_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($"Специалист. Квалификация: {_qualification}");
    }
}

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

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

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

    try
    {
        ((Specialist)item).
            ShowSpecialist(); // Способ №1
    }
    catch
    {
    }
}

Scientist scientist = item as
    Scientist; // Способ №2

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

if (item is Manager) // Способ №3
{
    (item as Manager).ShowManager();
}
}
```

Но попытка вызвать любой из методов классов-наследников приведет к ошибке на этапе компиляции (Рисунок 3.1), потому что в базовом классе ничего не известно о полях и методах в классах-наследниках.

3. Использование ссылок на базовый класс

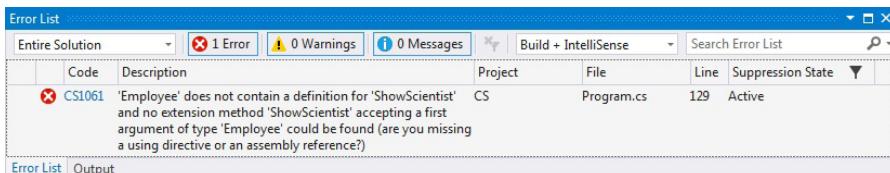


Рисунок 3.1. Ошибка: в классе нет определения для соответствующего метода

Для решения этой проблемы существуют три способа (в коде выделены комментариями).

Первый способ — осуществление явного приведения к необходимому типу. Применяя этот способ, Вы должны помнить о том, что в нашем массиве, по сути, находятся элементы различных классов, и некорректное приведение типа приведет к генерации исключительной ситуации (ошибке на этапе выполнения). Поэтому совместно с операцией явного приведения необходимо использовать инструкцию `try-catch` (более подробно об этом будет рассмотрено в одном из последующих уроков).

Следует также обратить Ваше внимание на синтаксис явного приведения к классу:

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

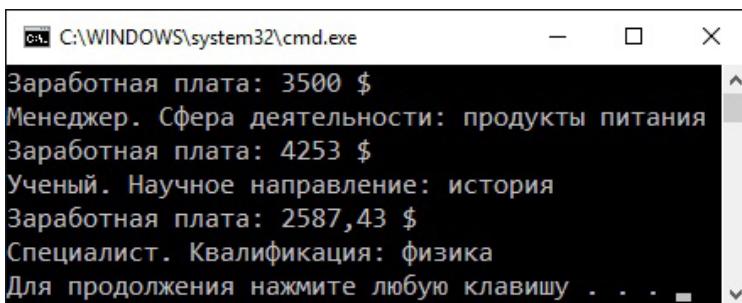
Вы видите, что само явное приведение находится в скобках, то есть вначале мы осуществляем приведение, а уже потом у того класса к которому привели, вызываем необходимый метод.

Второй способ заключается в использовании ключевого слова `as`. При помощи этого оператора мы пытаемся привести один тип к другому и присваиваем результат ссылке. После этого значение ссылки необходимо сравнить

с `null`, то есть проверить содержится ли в ссылке какое-нибудь значение и только при положительном результате вызывать метод соответствующего класса.

Третий способ — использование ключевого слова `is`. Оператор `is` позволяет определить совместимость типов, при этом результатом проверки будет тип данных `bool`. В случае если приведение типов возможно, то возвращается `true`, иначе `false`. После получения положительного результата мы можем смело осуществлять приведение типов и вызывать нужный нам метод.

Результат работы, описанного выше, кода представлен на рисунке 3.2.



```
C:\WINDOWS\system32\cmd.exe
Заработка плата: 3500 $
Менеджер. Сфера деятельности: продукты питания
Заработка плата: 4253 $
Ученый. Научное направление: история
Заработка плата: 2587,43 $
Специалист. Квалификация: физика
Для продолжения нажмите любую клавишу . . .
```

Рисунок 3.2. Использование ссылок на базовый класс

4. Полиморфизм в C#. Виртуальные методы

Полиморфизм позволяет производным классам создавать собственную реализацию методов, определенных базовым классом благодаря процессу, который называется переопределение метода. Для реализации данного процесса используются ключевые слова `virtual` и `override`.

Что такое виртуальный метод?

Иногда необходимо изменить реализацию методов, которые наследуются от базового класса. Для того чтобы предоставить такую возможность необходимо использовать виртуальные методы, которые объявляются с использованием ключевого слова `virtual`. Объявив метод класса как виртуальный, Вы тем самым позволяете классам-наследникам переопределять данный метод в случае необходимости.

Существуют некоторые особенности, касающиеся виртуальных методов:

- поля-члены и статические методы не могут быть объявлены как виртуальные.
- применение виртуальных методов позволяет реализовывать механизм позднего связывания. При позднем связывании определение вызываемого метода происходит на этапе выполнения (а не при компиляции) в зависимости от типа объекта, для которого вызывается виртуальный метод.

- на этапе компиляции строится только таблица виртуальных методов, а конкретный адрес метода, который будет вызван, определяется на этапе выполнения.
- При вызове метода-члена класса действуют следующие правила:
- для виртуального метода вызывается метод, соответствующий типу объекта, на который имеется ссылка;
- для не виртуального метода вызывается метод, соответствующий типу самой ссылки.

Переопределение виртуальных методов

Для переопределения виртуального метода базового класса, в классе-наследнике используется ключевое слово `override`.

Рассмотрим особенности использования виртуальных методов на примере работы с классами `Human` и `Employee`. Код создания классов, приведенный ниже, Вам уже знаком, но в данной ситуации нас интересует выполнение метода `Print()` в различных ситуациях.

```
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($"\\nФамилия: {_lastName} \\
              Имя: {_firstName}\\nДата \\
              рождения: {_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($"Заработная плата: {_salary} $");
    }
}

class Program
{
```

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

Для первого примера мы объявили метод `Print()` в обоих классах, а в методе `Main()` мы ссылке на базовый класс `Human` присвоили экземпляр производного класса `Employee`, что позволяет нам сделать механизм наследования. Не обращая внимания на предупреждения компилятора, запускаем программу на выполнение и видим результат, показанный на рисунке 4.1.

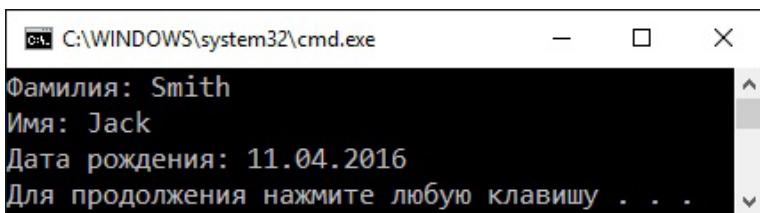


Рисунок 4.1. Вызов метода без переопределения

Результат работы программы может показаться несколько неожиданным, ведь мы создали экземпляр класса `Employee`, а вызвался метод `Print()` класса `Human`. На самом деле эта ситуация происходит благодаря принципу работы компилятора языка C#. При преобразовании Вашего исходного кода в CIL код, компилятор, для более быстрого выполнения методов, пытается подставить их реализацию еще на этапе компиляции при этом учитывает

ссылку на класс, при помощи которой и вызывается данный метод.

В данном случае, при оценке метода, компилятор действовал таким образом: ссылка, вызвавшая метод `Print()`, является ссылкой на класс `Human`, при этом сам метод объявлен без ключевого слова `virtual`, поэтому реализация этого метода подставляется на этапе компиляции. То, что этой ссылке на этапе выполнения присваивается экземпляр класса `Employee`, уже никак не влияет на вызов самого метода `Print()`.

Для демонстрации второго примера добавим при объявлении метода `Print()` класса `Human` ключевое слово `virtual`, а при объявлении метода `Print()` класса `Employee` добавим ключевое слово `override` — переопределение метода.

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

namespace SimpleProject
{
    public class Human
    {
        // остальной код остался прежним
        public virtual void Print()
        {
            WriteLine($"\\nФамилия: {_lastName} \\
        имя: {_firstName}\\nДата рождения: {_birthDate.ToString()}");
        }
    }

    public class Employee : Human
    {
```

```
// остальной код остался прежним

public override void Print()
{
    WriteLine($"Заработка плата: {_salary} $");
}

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

Что же произойдет в данной ситуации? Компилятор определил, что ссылка, вызвавшая метод `Print()`, является ссылкой на класс `Human` и при этом сам метод объявлен с ключевым словом `virtual`. Это говорит компилятору о том, что такой метод может быть переопределен в классах-наследниках, поэтому его реализацию необходимо подставить на этапе выполнения. На этапе выполнения ссылке на класс `Human` присваивается экземпляр класса `Employee`, при этом метод `Print()` в этом классе переопределен (ключевое слово `override`), поэтому вызывается реализация метода именно этого класса. Результат на рисунке 4.2.



Рисунок 4.2. Вызов переопределенного метода

В качестве эксперимента внесем изменения в наш код: оставим ключевое слово `virtual` при объявлении метода `Print()` класса `Human`, но удалим ключевое слово `override` при объявлении метода `Print()` класса `Employee`.

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

namespace SimpleProject
{
    public class Human
    {
        // остальной код остался прежним

        public virtual void Print()
        {
            WriteLine($"\\nФамилия: {_lastName} \\
                      Имя: {_firstName}\\nДата рождения: {_birthDate.ToShortDateString()}");
        }
    }

    public class Employee : Human
    {
        // остальной код остался прежним

        public void Print()
        {
            WriteLine($"Заработка plata: {_salary} $");
        }
    }
}
```

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

На самом деле результат выполнения кода не будет отличаться от первоначального результата (Рисунок 4.1). Потому что компилятор хоть и не подставил реализацию метода `Print()` на этапе компиляции, однако на этапе выполнения выяснилось, что в классе `Employee` метод `Print()` не переопределен, а значит, будет выполнен соответствующий метод класса `Human`.

В базовом классе могут быть несколько виртуальных методов, при переопределении которых в классах-наследниках, программисту необходимо помнить их названия и приходиться писать определенное количество кода. Для облегчения данного процесса в Visual Studio 2015 предусмотрена специальная функция (на жаргоне программистов — «фича» (*англ. feature*)), которая работает следующим образом. В классе наследнике необходимо написать ключевое слово `override` и нажать пробел. Появится список методов, переопределение которых возможно. В этом списке всегда будут присутствовать методы класса `System.Object`, но о них мы поговорим в главе 6. При выборе любого метода из списка рядом с ним выводится краткая информация (Рисунок 4.3).

4. Полиморфизм в C#. Виртуальные методы

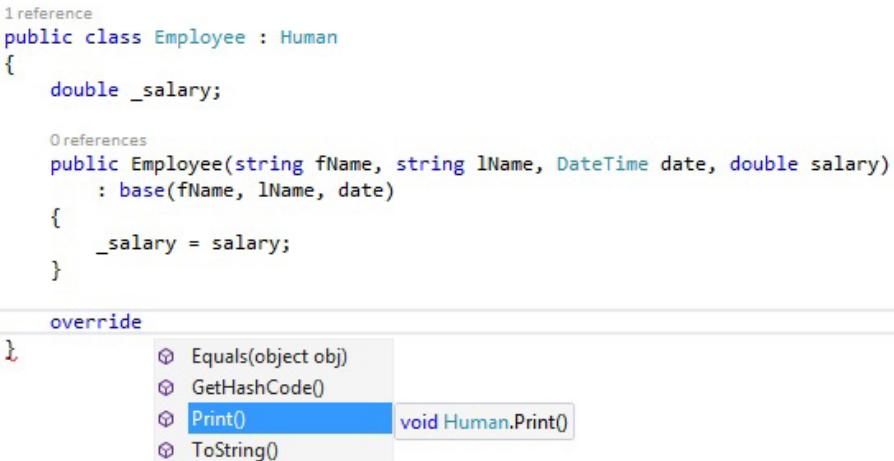


Рисунок 4.3. Список методов для переопределения

После нажатия клавиши «Enter» или двойного клика левой клавишей мыши на выбранном пункте, среда разработки сгенерирует переопределение выбранного Вами метода (Рисунок 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();
    }
}
```

Рисунок 4.4. Результат генерации переопределенного метода

Как Вы могли заметить, среда разработки автоматически подставляет в переопределение метода вызов метода базового класса с применением ключевого слова `base`. Такое поведение является не обязательным, но используется в большинстве случаев. При этом вызов метода базового класса возможен как до реализации в классе-наследнике, так и после нее. Исходя из порядка вызова метода базового класса, полученный результат будет соответствующим. Для того чтобы это продемонстрировать, внесем изменения в наш пример.

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

namespace SimpleProject
{
    public class Human
    {
        // остальной код остался прежним

        public virtual void Print()
        {
            WriteLine($"\\nФамилия: {_lastName}\\nИмя: {_firstName}\\nДата рождения: {_birthDate.ToShortDateString()}");
        }
    }

    public class Employee : Human
    {
        // остальной код остался прежним

        public override void Print()
        {
            base.Print();
        }
    }
}
```

```
        WriteLine($"Заработкая плата: {_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($"{\nМенеджер. Сфера деятельности:
              {_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,
                                   "продукты питания");
        manager.Print();
    }
}
```

В методе `Print()` класса `Employee` мы вызвали метод `Print()` базового класса `Human` до собственной реализации, а в классе `Manager` после. Как Вы можете заметить, последовательность выполняемых действий отразилась на результате (Рисунок 4.5).

The screenshot shows a command-line window titled 'C:\WINDOWS\system32\cmd.exe'. The output displays two sets of information for employees:

```
Фамилия: Smith
Имя: Jack
Дата рождения: 11.04.2016
Заработка плата: 3587,43 $

Менеджер. Сфера деятельности: продукты питания
Фамилия: Doe
Имя: John
Дата рождения: 23.07.1995
Заработка плата: 3500 $
Для продолжения нажмите любую клавишу . . .
```

Рисунок 4.5. Вызов метода базового класса
при переопределении метода

Хочется обратить Ваше внимание на некоторую особенность кода. Из-за того что класс `Human` является базовым классом для этой цепочки наследования и его метод `Print()` объявлен как виртуальный, любой из классов-наследников (`Employee`, `Manager` и т.д.) может переопределить этот метод в соответствии со своими потребностями. Результат же создания экземпляров этих классов можно присвоить ссылке на класс `Human`, что и отображено в приведенном коде.

Цепочку наследования можно прервать двумя способами. В первом случае для этого используется ключевое слово `new`, тем самым мы скрываем реализацию метода

базового класса (см. раздел 1). Посмотрим, к чему это приведет: заменим в методе класса `Employee` ключевое слово `override` на ключевое слово `new`.

```
// осталной код остался прежним

public class Employee : Human
{
    // осталной код остался прежним

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

class Manager : Employee
{
    // осталной код остался прежним

    public override void Print()
    {
        Write($"\nМенеджер. Сфера деятельности:
            {_fieldActivity}");
        base.Print();
    }
}

// осталной код остался прежним
```

И получим ошибку на этапе компиляции (Рисунок 4.6), которая произошла из-за того что невозможно в классе `Manager` переопределить метод `Print()`, так как соответствующий метод базового класса `Employee` этого не допускает.

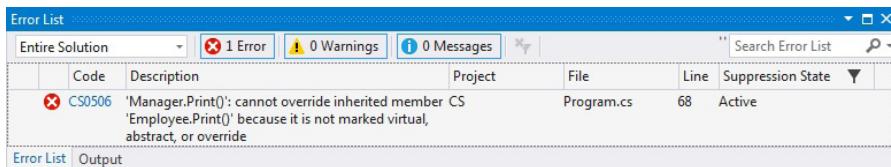


Рисунок 4.6. Ошибка: невозможно переопределить метод

Также можно закрыть дальнейшее переопределение метода базового класса. Для этого при объявлении переопределенного метода в классе-наследнике, помимо ключевого слова `override`, следует также указать ключевое слово `sealed`. Внесем соответствующее изменение в метод `Print()` класса `Employee`.

```
// осталной код остался прежним
public class Employee : Human
{
    // осталной код остался прежним
    public sealed override void Print()
    {
        base.Print();
        WriteLine($"Заработка плата: {_salary}");
    }
}

class Manager : Employee
{
    // осталной код остался прежним
    public override void Print()
    {
        Write($"{\nМенеджер. Сфера деятельности:
{_fieldActivity}}");
        base.Print();
    }
}
// осталной код остался прежним
```

Внесенные изменения тоже приведут к ошибке на этапе компиляции (Рисунок 4.7), которая связана с невозможностью переопределения метода `Print()` в классе `Manager`, из-за того что он объявлен как запечатанный в базовом классе `Employee`.

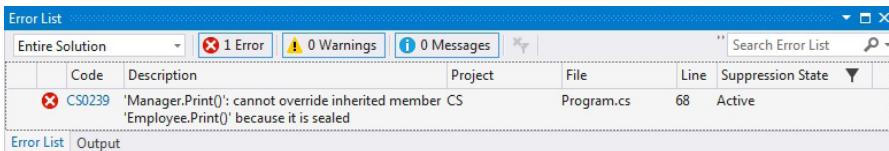


Рисунок 4.7. Ошибка: невозможно переопределить метод

Необходимость использования виртуальных методов

Виртуальные методы необходимы, когда классу-наследнику нужно изменить некоторые методы, определенные в базовом классе. Виртуальные методы позволяют определить базовому классу методы, реализация которых является общей для всех производных классов с возможностью переопределять эти методы в случае необходимости. Это позволяет поддерживать динамический полиморфизм. Определение у классов-наследников собственных методов становится более гибким, по-прежнему оставляя в силе требование согласующегося интерфейса.

В завершении этого раздела приведем расширенный пример, взяв за основу код из раздела 3, но изменим его в соответствии с полученными знаниями. И хотя реализация классов `Human`, `Employee` и `Manager` уже приводилась в текущем разделе, мы покажем весь код полностью, сняв тем самым все лишние вопросы.

```
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($"\\nФамилия: {_lastName} \\
                       Имя: {_firstName}\\nДата рождения: {_birthDate.ToString()}");
        }
    }

    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($"Заработка плата: {_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($"\\nМенеджер. Сфера деятельности:
              {_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;
    }
}
```

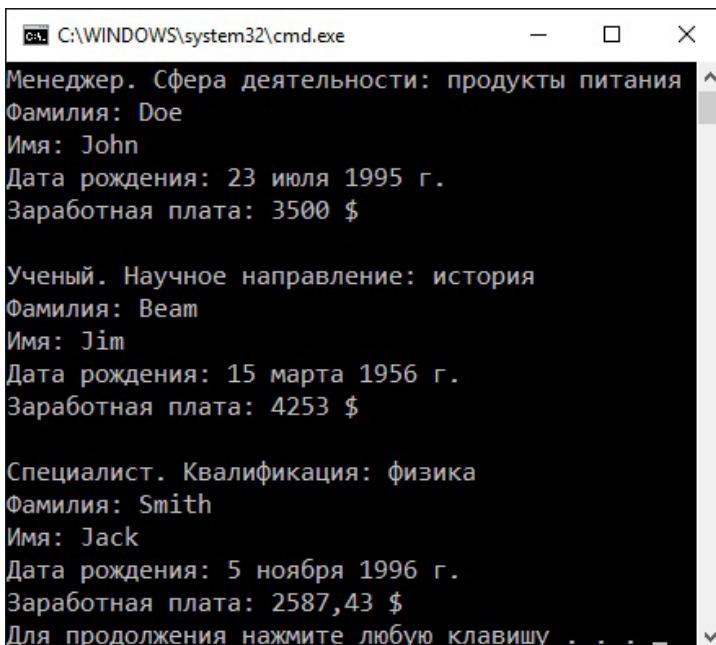
```
public override void Print()
{
    Write($"\\nУченый. Научное направление:
          {_scientificDirection}");
    base.Print();
}
}

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($"\\nСпециалист. Квалификация:
              {_qualification}");
        base.Print();
    }
}

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

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

Результат работы программы (Рисунок 4.8).



The screenshot shows a command prompt window titled 'C:\WINDOWS\system32\cmd.exe'. The output displays three instances of the 'Human' class with different implementations of the 'Print' method:

```
Менеджер. Сфера деятельности: продукты питания
Фамилия: Doe
Имя: John
Дата рождения: 23 июля 1995 г.
Заработка плата: 3500 $

Ученый. Научное направление: история
Фамилия: Beam
Имя: Jim
Дата рождения: 15 марта 1956 г.
Заработка плата: 4253 $

Специалист. Квалификация: физика
Фамилия: Smith
Имя: Jack
Дата рождения: 5 ноября 1996 г.
Заработка плата: 2587,43 $
Для продолжения нажмите любую клавишу . . .
```

Рисунок 4.8. Использование полиморфизма

Чем отличается этот код от предыдущего варианта? Если не брать во внимание переопределение методов в классах-наследниках, то главное отличие находится в методе `Main()`, а точнее в цикле `foreach`, где для каждого элемента массива типа `Human` вызывается метод `Print()`. Мы вызываем этот

метод, абсолютно не беспокоясь о том, существует ли его реализация в каждом конкретном классе. Благодаря тому, что мы переопределили методы в классах-наследниках, для каждого элемента будет вызвана именно его реализация.

Если в будущем Вы расширите иерархию наследования, создав новые классы, и добавите их в массив, работа цикла `foreach` останется без изменений. И даже если по какой-то причине Вы не переопределите метод `Print()` в классе-наследнике, то все равно ошибки не последует, а будет вызван метод базового класса `Human`. Предлагаем Вам проверить это самостоятельно, закомментировав метод `Print()` в любом классе-наследнике. Все вышеописанное является полиморфизмом.

5. Абстрактный класс

Абстрактный класс — это класс, у которого один из методов является абстрактным, то есть метод, у которого нет реализации. Экземпляры такого класса создавать нельзя, но его могут наследовать другие классы. Для объявления абстрактного класса используется ключевое слово `abstract`.

Класс целесообразно объявлять абстрактным, если он определяет некое обобщающее понятие, при этом создание и использование экземпляров такого класса лишено всякого смысла. Возьмем классический пример — класс `Figure` (геометрическая фигура). Если мы создадим экземпляр этого класса, то не понятно как рисовать эту фигуру, сколько у нее сторон, какая площадь и так далее. Класс `Figure` может служить базой для создания множества других классов (`Rectangle`, `Circle` и т.д.), являясь некой абстракцией, имеющей общий функционал (конструкторы, поля и т. д.) и методы которые должны быть переопределены в классах-наследниках.

В рассмотренных нами ранее примерах тоже присутствуют такие классы — `Human` и `Employee`. Оба эти класса являются некоторыми общими понятиями — люди как минимум различаются друг от друга анатомически, а сотрудником может быть как уборщица, так и президент. Поэтому при дальнейшем использовании мы будем объявлять их абстрактными.

В качестве примера работы с абстрактными классами создадим класс `Learner` (учащийся) — наследник класса `Human`, при этом оба класса объявим абстрактными.

Абстрактный класс по своей сути все равно остается классом, поэтому у него может быть все, что есть в обычном классе (поля, конструкторы, методы и т.д.), но у него существует и отличие от обычного класса — абстрактный метод. Вот и мы в классе `Human` объявим абстрактный метод `Think()` (думать). В абстрактном методе отсутствует реализация, объявляя такой метод в классе, мы «требуем» от классов-наследников его переопределения (с использованием ключевого слова `override`) в обязательном порядке. Если по какой-то причине класс-наследник «не желает» переопределять абстрактный метод, тогда этот класс должен быть объявлен абстрактным или компилятор выдаст ошибку (Рисунок 5.1).

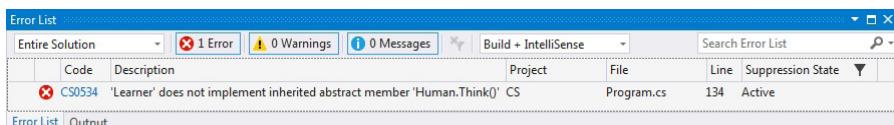


Рисунок 5.1. Ошибка: в классе нет реализации для метода `Think()`

В данном случае решение объявить метод `Think()` абстрактным является вполне обоснованным, ведь так или иначе все люди думают по-разному, поэтому будет логично, если все наследники класса `Human` предоставят собственную реализацию этого метода.

В классе `Learner` объявим поле `_institution` (учебное заведение) и абстрактный метод `Study()` (изучение), полученный класс будет базовым для двух классов-наследников `Student` и `SchoolChild`.

Прежде чем продемонстрировать Вам итоговый код хочется ознакомить Вас с еще одной специфической

возможностью Visual Studio 2015, которая позволяет при наследовании от абстрактного класса определиться с методами, необходимыми для переопределения, в классах-наследниках.

После того как Вы при объявлении класса укажите, что он является наследником абстрактного класса, компилятор тут же выдаст ошибку (Рисунок 5.2), сообщая о том что необходимо переопределить абстрактные методы и создать конструктор.

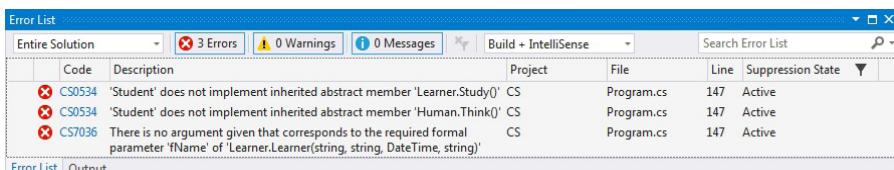


Рисунок 5.2. Ошибка: в классе нет реализации для абстрактных методов

После одиночного нажатия левой клавиши мыши на имени абстрактного класса в строке наследования, в левой части строки появляется кнопка с изображением лампочки. При нажатии на которую, появляется список необходимых действий (Рисунок 5.3 и 5.4).

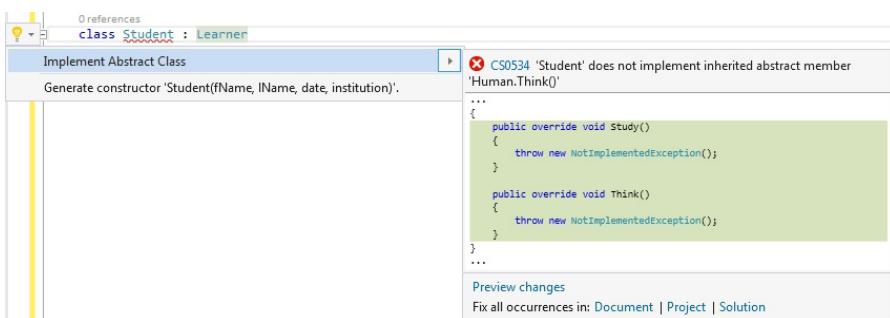


Рисунок 5.3. Пункт реализация абстрактного класса

Перемещаясь по пунктам списка, справа Вы увидите окно предпросмотра реализации выбранного пункта.



Рисунок 5.4. Пункт создание конструктора

Для автоматической генерации требуемого кода необходимо кликнуть на соответствующем пункте списка, полученный результат представлен на рисунке 5.5.

```
1 reference
class Student : Learner
{
    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();
    }
}
```

Рисунок 5.5. Результат автоматической генерации кода

При выборе пункта лучше начинать с создания конструктора, в этом случае пункт реализации методов абстрактного класса будет доступен. Если Вы поступите наоборот, то автоматическое создание конструктора класса Вам нужно будет осуществлять другим способом.

В качестве альтернативы, приведенному выше способу, можно вызвать контекстное меню при клике правой кнопкой мыши, как на базовом классе, так и на классе-наследнике и выбрать пункт **Quick Actions...** (Рисунок 5.6).

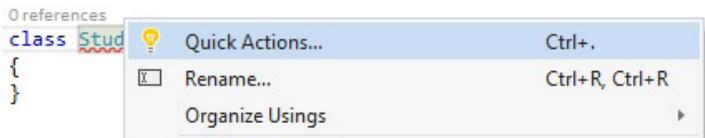


Рисунок 5.6. Пункт контекстного меню для генерации кода

После этого Вы увидите знакомые Вам пункты (Рисунок 5.3 и 5.4), выбор которых приведет к генерации кода приведенного на рисунке 5.5.

Как Вы могли заметить, среда разработки создает шаблоны переопределения всех необходимых абстрактных методов не пустыми. В качестве реализации этих методов будет вставлена строка, так называемая «заглушка». На самом деле это генерация исключительной ситуации (будет рассмотрена в последующем уроке), Ваша задача вместо этой строки написать собственную реализацию метода.

```
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($"\\nФамилия: {_lastName} \\
        Имя: {_firstName}\\nДата рождения:
        {_birthDate.ToString()}");
    }
}

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($"Учебное заведение:
        {_institution}.");
    }
}

class Student : Learner
{
    string _groupName;
    public Student(string fName, string lName,
                  DateTime date, string institution,
                  string groupName) : base(fName, lName,
                  date, institution)
```

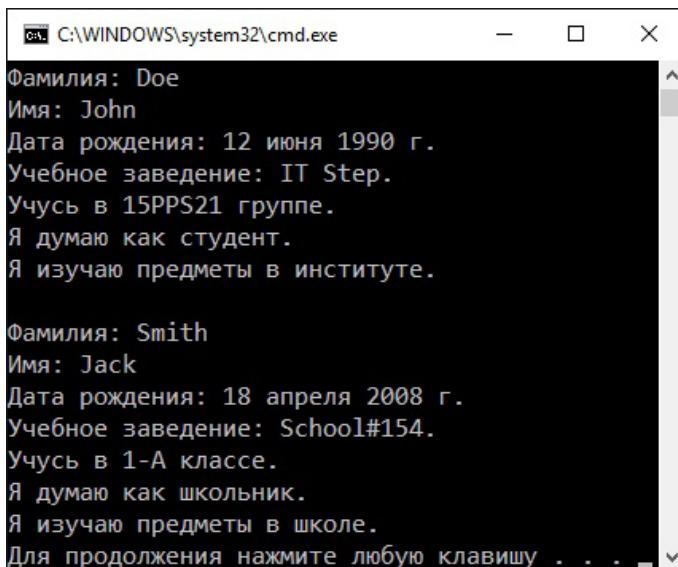
```
{  
    _groupName = groupName;  
}  
  
public override void Think()  
{  
    WriteLine("Я думаю как студент.");  
}  
  
public override void Study()  
{  
    WriteLine("Я изучаю предметы в  
институте.");  
}  
  
public override void Print()  
{  
    base.Print();  
    WriteLine($"Учусь в {_groupName} группе.");  
}  
}  
  
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("Я думаю как школьник.");  
    }  
}
```

```
public override void Study()
{
    WriteLine("Я изучаю предметы в школе.");
}

public override void Print()
{
    base.Print();
    WriteLine($"Учусь в {_className} классе.");
}

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();
        }
    }
}
```

Результат работы программы (Рисунок 5.7).



C:\WINDOWS\system32\cmd.exe

```
Фамилия: Doe
Имя: John
Дата рождения: 12 июня 1990 г.
Учебное заведение: IT Step.
Учуясь в 15PPS21 группе.
Я думаю как студент.
Я изучаю предметы в институте.

Фамилия: Smith
Имя: Jack
Дата рождения: 18 апреля 2008 г.
Учебное заведение: School#154.
Учуясь в 1-А классе.
Я думаю как школьник.
Я изучаю предметы в школе.
Для продолжения нажмите любую клавишу . . .
```

Рисунок 5.7. Цепочка наследования абстрактного класса

6. Анализ базового класса Object

Так как каждый тип в C# является наследником от класса `Object`, то его методы доступны для любого класса, а виртуальные могут быть переопределены в случае необходимости. Рассмотрим методы класса `Object`:

- `Equals` — виртуальный метод, в качестве параметра получает объект типа `object`. Возвращает `true`, если объект, вызывающий метод, и объект, передаваемый в качестве параметра, одинаковые иначе возвращает `false`.
- `Equals` — статический метод, в качестве параметров получает два объекта типа `object`. Возвращает `true`, если объекты одинаковые, иначе возвращает `false`.
- `Finalize` — позволяет выполнить операции по очистке ресурсов, которые занимает текущий объект перед тем, как он будет утилизирован сборщиком мусора.
- `GetHashCode` — виртуальный метод, возвращает хэш-код вызывающего объекта.
- `GetType` — возвращает объект класса `Type` для текущего экземпляра.
- `MemberwiseClone` — создает «поверхностную» копию вызывающего объекта, в которую копируются все члены класса, но при этом не копируются объекты, на которые ссылаются эти члены.
- `ReferenceEquals` — статический метод, возвращает `true`, если два объекта (которые передаются в качестве параметров) ссылаются на один и тот же объект. Если они

ссылаются на разные объекты, то возвращает `false`.

- `ToString` — виртуальный метод, возвращает строковое представление текущего объекта.

Особенности работы с методами класса `Object` мы будем рассматривать постепенно, по мере изучения языка C#, в этом уроке мы поработаем с методами `GetType()` и `ToString()`.

На протяжении этого урока для вывода информации о классах мы использовали метод `Print()`, который сами и написали, но есть способ общепринятый и гораздо проще — переопределение метода `ToString()` класса `Object`. Данный метод возвращает представление объекта в виде строки, он переопределен у всех стандартных типов данных (`int`, `double` и т. д.) и вызывается для них автоматически при попытке вывода переменных этих типов на консоль (методы `Write()` и `WriteLine()`). Вы уже это видели, но не обращали внимания, посмотрите предыдущие коды и убедитесь в том, что метод `ToString()` нигде явно не вызывается, а строка выводиться.

Внесем изменения в написанный ранее код — заменим в наших классах метод `Print()` переопределенным методом `ToString()`. Благодаря этому упроститься вывод информации о классах в методе `Main()`.

```
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 $"\\nФамилия: {_lastName} \\
        Имя: {_firstName}\\nДата рождения:
        {_birthDate.ToString()}";
    }
}

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() + $"\\nУчебное
            заведение: {_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("Я думаю как студент.");
    }

    public override void Study()
    {
        WriteLine("Я изучаю предметы в институте.");
    }

    public override string ToString()
    {
        return base.ToString() + $"\\nУчусь в
            {_groupName} группе.";
    }
}

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("Я думаю как школьник.");
}

public override void Study()
{
    WriteLine("Я изучаю предметы в школе.");
}

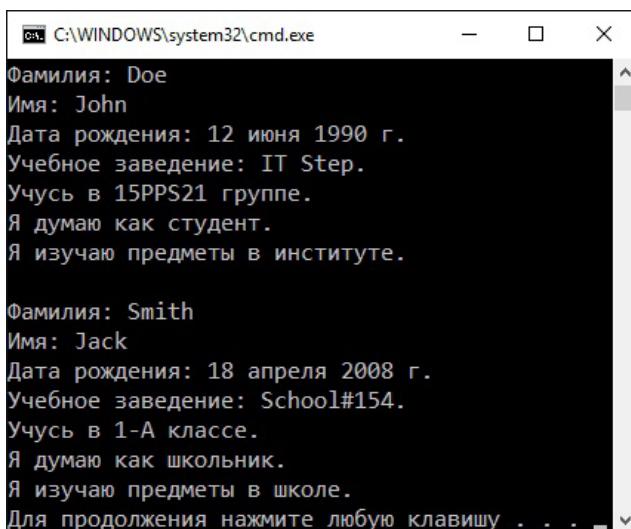
public override string ToString()
{
    return base.ToString() + $"\\nУчусь в
{_className} классе.";
}

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();
    }
}
}
```

Результат работы программы аналогичны предыдущим (Рисунок 6.1).



The screenshot shows a Windows Command Prompt window titled 'cmd C:\WINDOWS\system32\cmd.exe'. It displays two sets of information, each starting with 'Фамилия:' followed by a name. The first set is for 'Doe' and the second for 'Smith'. Each entry includes their first name, date of birth, place of study, current status, and current activities.

Фамилия:	Имя:	Дата рождения:	Учебное заведение:	Учусь в	Я думаю как	Я изучаю
Doe	John	12 июня 1990 г.	IT Step.	15PPS21 группе.	студент.	предметы в институте.
Smith	Jack	18 апреля 2008 г.	School#154.	1-А классе.	школьник.	предметы в школе.

Рисунок 6.1. Переопределение метода `ToString()` в классах

Как было написано выше, при вызове метода `GetType()` мы получим объект класса `Type`, который в свою очередь позволяет получить исчерпывающую информацию о вызвавшем его объекте. В основном класс `Type` применяется при рефлексии типов, которая будет рассмотрена в последующих курсах. Сейчас мы продемонстрируем лишь некоторые методы этого класса, полную информацию

о котором Вы можете посмотреть в MSDN. Возьмем предыдущий код и внесем изменения только в метод Main() класса Program, для экономии места весь остальной код приводиться не будет.

```
// остальной код остался прежним

class Program
{
    static void Main(string[] args)
    {
        Student student = new Student("John",
            "Doe", new DateTime(1990, 6, 12),
            "IT Step", "15PPS21");

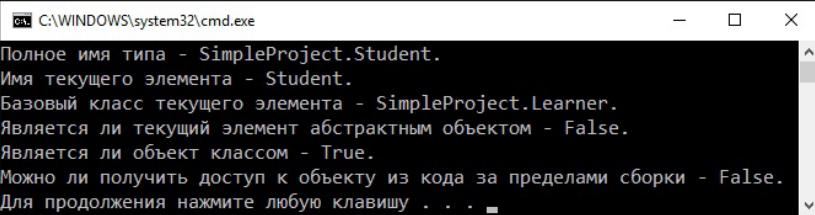
        WriteLine($"Полное имя типа - {student.
            GetType().FullName}.");

        WriteLine($"Имя текущего элемента -
            {student.GetType().Name}.");

        WriteLine($"Базовый класс текущего
            элемента - {student.GetType().
            BaseType}.");

        WriteLine($"Является ли текущий
            элемент абстрактным объектом -
            {student.GetType().IsAbstract}.");
        WriteLine($"Является ли объект классом -
            {student.GetType().IsClass}.");
        WriteLine($"Можно ли получить доступ
            к объекту из кода за пределами
            сборки - {student.GetType().
            IsVisible}.");
    }
}
```

Результат работы программы (Рисунок 6.2).



```
С:\WINDOWS\system32\cmd.exe
Полное имя типа - SimpleProject.Student.
Имя текущего элемента - Student.
Базовый класс текущего элемента - SimpleProject.Learner.
Является ли текущий элемент абстрактным объектом - False.
Является ли объект классом - True.
Можно ли получить доступ к объекту из кода за пределами сборки - False.
Для продолжения нажмите любую клавишу . . .
```

Рисунок 6.2. Некоторые свойства класса Type

Последнее что мы рассмотрим в данном уроке — это возможность построения в нашем приложении графического представления отношений между классами — диаграммы классов, изображение которой было представлено в главе 1 (Рисунок 1.1).

Для того чтобы создать диаграмму классов необходимо в пункте **Project** главного меню выбрать пункт **Add New Item...**, в результате чего появится одноименное окно, в котором необходимо выбрать пункт **Class Diagram** (Рисунок 6.3).

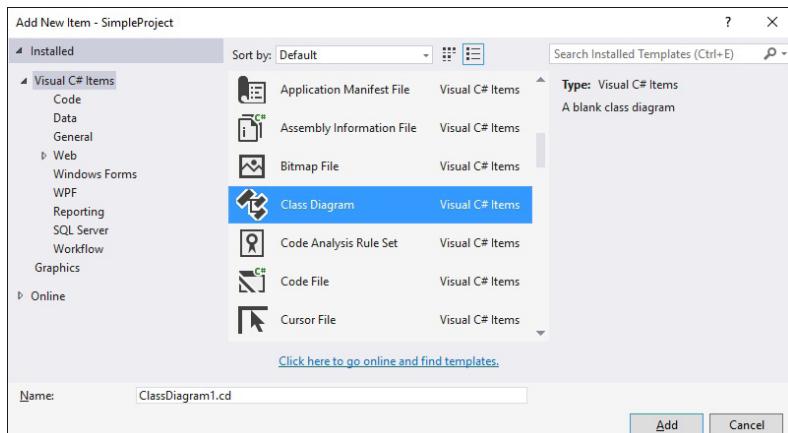


Рисунок 6.3. Окно Add New Item

Урок №4

После нажатия на кнопку Add в наш проект будет добавлен файл с расширением .cd и появиться поле построения диаграммы (Рисунок 6.4).

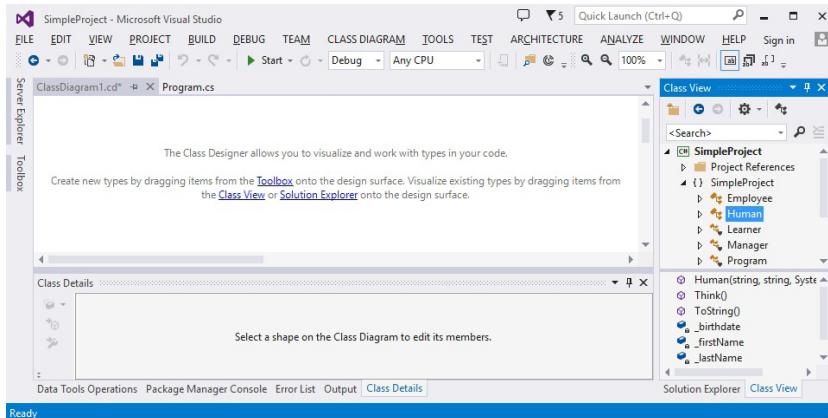


Рисунок 6.4. Файл Class Diagram

Вам нужно просто перетянуть на это поле необходимые классы из окна Class View или Solution Explorer, а среда разработки построит отношения между классами автоматически, данный процесс отображен на рисунке 6.5.

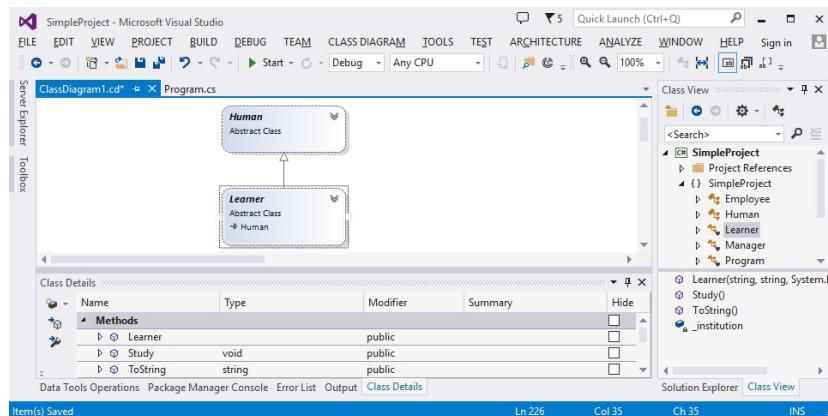


Рисунок 6.5. Процесс построения диаграммы классов

7. Домашнее задание

Задание 1.

Разработать абстрактный класс «Геометрическая Фигура» с методами «Площадь Фигуры» и «Периметр Фигуры». Разработать классы-наследники: Треугольник, Квадрат, Ромб, Прямоугольник, Параллелограмм, Трапеция, Круг, Эллипс. Реализовать конструкторы, которые однозначно определяют объекты данных классов.

Реализовать класс «Составная Фигура», который может состоять из любого количества «Геометрических Фигур». Для данного класса определить метод нахождения площади фигуры. Создать диаграмму взаимоотношений классов.

Задание 2.

Разработать архитектуру классов иерархии товаров при разработке системы управления потоками товаров для дистрибуторской компании. Прописать члены классов. Создать диаграммы взаимоотношений классов.

Должны быть предусмотрены разные типы товаров, в том числе:

- бытовая химия;
- продукты питания.

Предусмотреть классы управления потоком товаров (пришло, реализовано, списано, передано).



Урок №4

Наследование и полиморфизм

© Юрий Задерей.

© Компьютерная Академия «Шаг»
www.itstep.org.

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

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

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