# Основные свойства ОО программирования: наследование, полиморфизм

1. Иерархии классов. Наследование.
2. Виртуальные методы. Механизм позднего связывания.
3. Абстрактные классы. Класс object.
4. Интерфейсы. Работа с объектами через стандартные интерфейсы .NET

## Иерархии классов. Наследование.

В данной теме рассматриваются вопросы, связанные с реализацией в С# таких свойств ООП как наследование и полиморфизм.

**Наследование** – это такое отношение между классами, когда один класс частично или полностью повторяет структуру и поведение другого класса (одиночное наследование) или других (множественное наследование) классов.

**Полиморфизм** – это свойство ООП, при котором методы разных классов (но относящихся к одной иерархии наследования) могут иметь одно и то же имя, но выполнять разные действия

Наследование устанавливает между классами иерархию «общее-частное»:

* Человек (общее) – Студент (частное),
* Транспорт (общее) – Автомобиль (частное),
* Геометрическая\_фигура (общее) – прямоугольник (частное) и т.д.

Класс, который находится выше в иерархии классов, называют базовым классом или суперклассом, класс, который находится ниже в иерархии классов называют производным или подклассом.

Подкласс обычно расширяет или ограничивает существующую структуру (поля) и поведение (методы) своего суперкласса. Помимо наследуемых, каждый подкласс имеет свои собствен­ные уникальные атрибуты, операции и связи.

**Преимущества наследования**

1. Возможность использования повторного кода (это можно сделать и при помощи агрегирования)
2. Возможность реализовать производный класс на основе базового, упрощая реализацию кода для производного класса.
3. Возможность обработки производных классов методами, разработанными при проектировании базового класса.

Класс в С# может иметь произвольное количество потомков и только одного предка. При описании класса имя его предка записывается в заголовке класса после двоеточия.

[ атрибуты ] [ спецификаторы ] **class** имякласса [ : предки ] тело класса

Основное достоинство наследования состоит в том, что, создав базовый класс, который определяет общие атрибуты для множества объектов, его можно использовать для создания любого числа более специализированных производных классов.

**Доступ к элементам класса:**

* public – открытые.
* private – закрытые, не наследуются.
* protected – защищенные, определен доступ для наследников класса, наследуются.

**Конструкторы**

В иерархии классов как базовые, так и производные классы могут иметь собственные конструкторы. При этом конструктор базового класса создает часть объекта, соответствующую базовому классу, а конструктор производного класса — часть объекта, соответствующую производному классу.

Если конструкторы определены и в базовом, и в производном классе, в процессе создания объектов должны выполниться конструкторы обоих классов. В этом случае необходимо использовать ключевое слово **base**, которое имеет два назначения:

* вызвать конструктор базового класса;
* получить доступ к элементу базового класса.

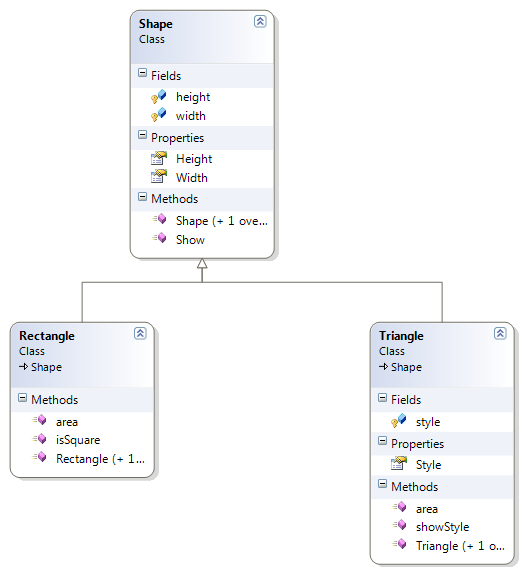
конструктор\_производного\_класса (список\_параметров) : base (список\_аргументов)

{

// тело конструктора

**}**

Рассмотрим наследование классов на примере (Шилдт).



class Shape //базовый класс

{

protected double width; //ширина

protected double height;//высота

public double Width

{

get { return width; }

set { width = value; }

}

public double Height

{

get { return height; }

set { height = value; }

}

public Shape()//конструктор без параметров

{

width = 0;

height = 0;

}

public Shape(double w, double h) //конструктор с параметрами

{

width = w; height = h;

}

public void Show()

{

Console.WriteLine("ширина=" + width + " высота=" + height);

}

}

class Triangle: Shape //производный класс

{

protected string style;

public string Style

{

get { return style; }

set { style = value; }

}

public Triangle()

: base()//коструктор без параметров

{

style = "";

}

public Triangle(double w, double h, string s)

: base(w, h) //конструктор с параметрами

{

style = s;

}

public double area() //вычисление площади

{

return width\*height/2;

}

public void showStyle()

{

Console.WriteLine("Треугольник "+style);

}

}

class Rectangle : Shape

{

public bool isSquare()

{

if (width == height) return true;

else return false;

}

public Rectangle() : base() { }

public Rectangle(double w, double h) : base(w, h) { } //конструктор с параметрами

public double area()

{

return width \* height;

}

}

class Program

{

static void Main(string[] args)

{

Triangle t = new Triangle();

t.Show();

Rectangle r = new Rectangle();

r.Show();

Triangle t1 = new Triangle(10, 15, "прямоугольный");

Triangle t2 = new Triangle(4, 4, "равнобедренный");

Rectangle r1 = new Rectangle(5,6);

t1.Show();

t1.showStyle();

Console.WriteLine("площадь=" + t1.area());

t2.Show();

t2.showStyle();

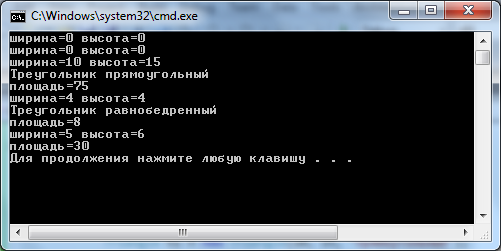
Console.WriteLine("площадь=" + t2.area());

r1.Show();

Console.WriteLine("площадь=" + r1.area());

}

}



Производный класс может определить элемент, имя которого совпадает с именем элемента базового класса. В этом случае элемент базового класса становится скрытым в производном классе. Поскольку с точки зрения формального синтаксиса языка С# эта ситуация не является ошибочной, компилятор выдаст предупреждающее сообщение, которое должно послужить напоминанием о факте сокрытия имени. Для предотвращения этого предупреждения перед членом производного класса необходимо поставить ключевое слово new.

class Shape

{

protected double width;

protected double height;

**public string name = "SHAPE";//новый элемент**

. . . . .

public void Show()

{

Console.WriteLine(name+": ширина=" + width + " высота=" + height);

}

}

class Triangle: Shape

{

protected string style;

**new public string name;//скрывает name базового класса**

public string Style

. . . . .

public Triangle(double w, double h, string s)

: base(w, h)

{

style = s;

**name = "triangle";**

}

public void showStyle()

{

Console.WriteLine(name+" "+style);//обращение к новому элементу

}

}

class Program

{

static void Main(string[] args)

{

Triangle t1 = new Triangle(10, 15, "прямоугольный");

Triangle t2 = new Triangle(4, 4, "равнобедренный");

t1.Show();

t1.showStyle();

Console.WriteLine("площадь=" + t1.area());

t2.Show();

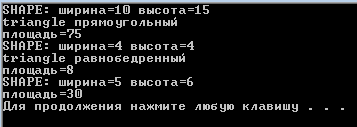
t2.showStyle();

Console.WriteLine("площадь=" + t2.area());

}

}

}



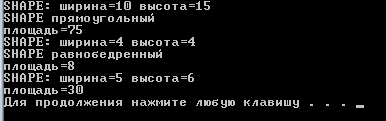
Изменим метод showStyle() класса Triangle:

public void showStyle()

{

Console.WriteLine(**base.name**+" "+style);

}



## Виртуальные методы. Механизм позднего связывания

С# — строго типизированный язык. Поэтому переменная одного типа (в том числе пользовательского) обычно не может ссылаться на объект другого типа.

**НО ссылочной переменной базового класса можно присвоить ссылку на объект любого класса, производного от этого базового класса.**

Triangle t1 = new Triangle(10, 15, "прямоугольный");

**Shape s = t1;**

s.Show();

Но в этом случае при использовании переменной базового класса, мы получим доступ только к тем полям, который имеются в базовом классе.

width

height

style

Triangle t1

Shape s

****

Shape[] arr = new Shape[4];

arr[0] = t1; arr[1] = t2; arr[2] = r1; arr[3] = r;

foreach (Shape s in arr) s.Show();//вызывается Show для класса Shape

**Виртуальным** называется метод, объявляемый с помощью ключевого слова **virtual** в базовом классе и переопределяемый в одном или нескольких производных классах.

При использовании виртуальных методов, тот метод, который нужно вызвать, С# определяет по типу объекта, на который указывает ссылка, причем решение принимается **динамически, во время выполнения программы**. Следовательно, если имеются ссылки на различные объекты, будут выполняться различные версии виртуального метода.

Чтобы объявить метод в базовом классе виртуальным, его объявление необходимо предварить ключевым словом **virtual**. При переопределении виртуального метода в производном классе используется модификатор **override**. При переопределении метода описания методов в базовом и производном классе должны совпадать.

Переопределение виртуального метода формирует базу для одной из самых мощных концепций ООП **динамического (позднее) связывания**. **Позднее связывание** – это механизм вызова переопределенного метода во время выполнения программы, а не в период компиляции. Именно благодаря позднему связыванию в С# реализуется динамический полиморфизм.

class Shape //базовый класс

{

protected double width; //ширина

protected double height;//высота

**. . .**

**virtual public void Show()**

**{**

**Console.WriteLine("Shape: ширина=" + width + " высота=" + height);**

**}**

}

class Triangle : Shape //производный класс

{

protected string style;

. . .

**override public void Show()**

**{**

**Console.WriteLine("Triangle: ширина=" + width + " высота=" + height+" стиль="+style);**

**}**

}

class Rectangle : Shape

{

. . . .

**override public void Show()**

**{**

**Console.WriteLine("Rectngle: ширина=" + width + " высота=" + height);**

**}**

}

class Program

{

static void Main(string[] args)

{

Rectangle r = new Rectangle();

Triangle t1 = new Triangle(10, 15, "прямоугольный");

Triangle t2 = new Triangle(4, 4, "равнобедренный");

Rectangle r1 = new Rectangle(5, 6);

Shape[] arr = new Shape[4];

arr[0] = t1; arr[1] = t2; arr[2] = r1; arr[3] = r;

**foreach (Shape s in arr) s.Show();**

}

}



### Таблица виртуальных функций

Стандартное решение для обеспечения динамического связывания, используемое разработчиками компиляторов заключается в следующем. Для каждого класса, допускающего полиморфное поведение объектов создается специальная таблица, содержащая адреса виртуальных функций, объявленных в этом классе. В определение класса добавляется указатель на таблицу виртуальных функций vtbl (его автоматически добавляет конструктор), т .е. размер памяти, занимаемой полиморфным объектом увеличится на размер этого указателя. Адреса одноименных виртуальных функций классов, находящихся в отношении наследования, помещаются в ячейки таблицы vtbl c одинаковыми индексами. Если какая-то виртуальная функция не переопределяется в производном классе, то в vtbl помещается адрес соответствующей функции базового класса.

Выбор виртуальной функции выполняется следующим образом.

Если в программе есть указатель на объект базового класса, то компилятор будет обращаться к скрытому элементу vptr, который содержит адрес vtbl. Указатель vptr содержит адрес начала таблицы vtbl. Затем имя вызываемой функции преобразуется в индекс таблицы. т. к. адреса виртуальных функций бзового и производного класса, хранятся в одинаковом порядке, то элемент с индексом 0 всегда содержит адрес функции show(), а с индексом 1 – адрес функции input(). т. о. если мы хотим извлечь адрес функции input(), то мы можем получить его как vptr+1.

Процесс вычисления адреса функции осуществляется во время выполнения программы – это и будет позднее связывание.

Таким образом, вызов виртуального метода, в отличие от обычных методов и функций, выполняется через дополнительный этап получения адреса метода из таблицы.

### Динамическая идентификация типов

**Динамическая идентификация типов** (runtime type identification — RTTI) позволяет определить тип объекта во время выполнения программы, что необходимо во многих ситуациях. Например, можно совершенно точно узнать, на объект какого типа в действительности указывает ссылка на базовый класс.

В С# предусмотрено три ключевых слова, которые поддерживают динамическую идентификацию типов: is , as и typeof.

С помощью оператора is можно определить, имеет ли рассматриваемый объект заданный тип. Общая форма его записи имеет следующий вид:

выражение is ТИП

Здесь тип элемента выражение сравнивается с элементом ТИП. ЕСЛИ ТИП элемента выражение совпадает (или совместим) с элементом ТИП, результат выполнения операции принимается равным значению ИСТИНА. В противном случае — значению ЛОЖЬ. Следовательно, если результат истинен, выражение можно привести к типу, заданному элементом ТИП.

class Shape{}

class Triangle: Shape{}

class Rectangle: Shape{}

Shape[] arr = new Shape[5];

. . . . .

int count1 = 0;

foreach (Shape p in arr)

if (p is Triangle) count1++;

Console.WriteLine("В массиве " + count1 + " объектов типа Triangle");

Иногда во время работы программы требуется выполнить операцию приведения типов, не генерируя исключение в случае, если попытка окажется неудачной. Для этого предусмотрен оператор as, формат которого таков:

выражение as тип

foreach (Shape p in arr)

{

Rectangle rt = p as Rectangle;

if (rt != null) count2++;

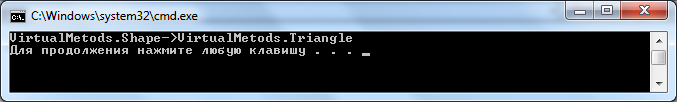
}

Console.WriteLine("В массиве " + count2 + " объектов типа Rectangle");

Операторы is и as проверяют совместимость двух типов. Оператор typeof используется для получения информации о типе. Операция возвращает объект типа Type, который содержит информацию о заданном типе.

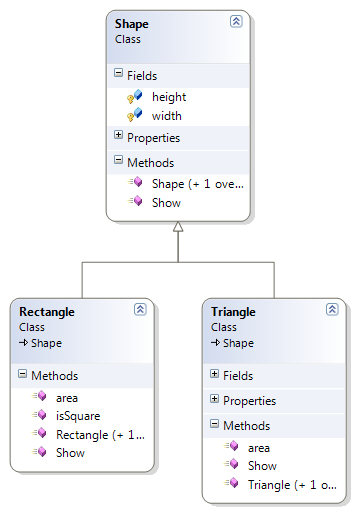
Type t = typeof(Triangle);

Console.WriteLine(t.BaseType + "->" + t.FullName);



## 3. Абстрактные классы. Класс object.

Рассмотрим иерархию классов на базе класса Shape.



Классы Triangle и Rectangle содержат поля width и height (Triangle содержит еще дополнительное поле style), свойства, конструкторы, метод area() для определения площади фигуры, метод Show() для вывода информации о фигуре на экран.

Метод Show() является виртуальным методом. Его он определяется в базовом классе Shape, а затем переопределяется в производном классе.

Метод area() также хорошо было бы вынести в базовый класс (т.к. он имеется во всех производных от Shape классах), но не понятно, по какой формуле нужно будет вычислять площадь для объектов класса Shape.

В таком случае создается базовый класс, определяющий только своего рода "шаблон", который унаследуют все производные классы, причем каждый из них заполнит этот "шаблон" собственной информацией. Такой класс определяет методы, которые производные классы должны реализовать, но сам при этом не обеспечивает реализации одного или нескольких методов. Подобный «шаблон» называется абстрактным методом.

**Абстрактный метод** создается с помощью модификатора типа **abstract.** . Абстрактный метод не содержит тела и, следовательно, не реализуется базовым классом. Поэтому производный класс обязательно должен его переопределить.

Для объявления абстрактного метода используйте следующий формат записи.

abstract тип имя(список\_параметров);

Тело абстрактного метода отсутствует.

**Свойства абстрактных методов**

1. Абстрактный метод автоматически является виртуальным.
2. Абстрактный метод не может быть статическим.
3. Свойства также могут быть абстрактными.

**Класс**, содержащий один или несколько абстрактных методов, также должен быть объявлен как **абстрактный** с помощью спецификатора abstract, который ставится перед объявлением class.

Поскольку абстрактный класс имеет абстрактные (не реализуемые) методы, невозможно создать его экземпляры, или объекты. Если производный класс выводится из абстрактного, он может реализовать все абстрактные методы базового класса.   
Если производный класс реализует не все методы абстрактного класса, то он также является абстрактным. Таким образом, атрибут abstract наследуется до тех пор, пока реализация класса не будет полностью достигнута.

abstract class Shape //базовый класс

{

protected double width; //ширина

protected double height;//высота

. . . . .

virtual public void Show()

{

Console.WriteLine("Shape: ширина=" + width + " высота=" + height);

}

public abstract double area();

}

class Triangle : Shape //производный класс

{

protected string style;

override public double area() //вычисление площади

{

return width \* height / 2;

}

override public void Show()

{

Console.WriteLine("Triangle: ширина=" + width + " высота=" + height + " стиль=" + style);

}

}

class Rectangle : Shape

{

. . . .

override public double area()

{

return width \* height;

}

override public void Show()

{

Console.WriteLine("Rectngle: ширина=" + width + " высота=" + height);

}

}

class Program

{

static void Main(string[] args)

{

Rectangle r = new Rectangle();

Triangle t1 = new Triangle(10, 15, "прямоугольный");

Triangle t2 = new Triangle(4, 4, "равнобедренный");

Rectangle r1 = new Rectangle(5, 6);

Shape[] arr = new Shape[4];

arr[0] = t1; arr[1] = t2; arr[2] = r1; arr[3] = r;

Console.WriteLine("Площади фигур:");

foreach (Shape s in arr)

{

s.Show();

Console.WriteLine("Площадь=" + s.area());

}

}

}

### Класс object

В С# определен специальный класс с именем object, который является неявным базовым классом всех других классов и типов (включая типы значений). Т.е. переменная типа object может указывать на объект любого типа (в том числе и массив). Ссылку типа object можно использовать в качестве ссылки на любой другой тип, включая типы значений.

object[] mas=new object[4];

mas=arr;

foreach (Shape s in mas)

{

s.Show();

Console.WriteLine("Площадь=" + s.area());

}

Класс object содержит следующие методы, которые можно переопределять в других классах:

* public virtual bool Equals(object obj) - сравнивает текущий объект с object obj и возвращает true, если объекты одинаковые и false в противном случае. По умолчанию метод определяет, ссылается ли вызывающий метод объект и object obj на один и тот же элемент. Можно переопределить этот метод так, чтобы он сравнивал содержимое двух объектов.
* public virtual int GetHashCode () возвращает хеш-код, связанный с вызывающим объектом. Этот хеш-код можно использовать с любым алгоритмом, который применяет хеширование как средство доступа к объектам, хранимым в памяти.

При перегрузке оператора "==" необходимо переопределить методы Equals (object) и GetHashCode (), поскольку функции оператора "==" и метода Equals (object), как правило, должны быть идентичными. Переопределив метод Equals (object), рекомендуется переопределить и метод GetHashCode (), чтобы они были совместимы.

* protected Finalize() - выполняет завершающие действия перед процессом сбора мусора, доступен через деструктор.
* public Type GetType() - получает тип объекта во время выполнения программы.
* protected object MemberwiseClone() - выполняет "поверхностное копирование" объекта, т.е. не копирует содержимое объектов.
* public virtual string ToString() - возвращает строку, которая описывает объект. Метод ToString() автоматически вызывается при выводе объекта с помощью метода WriteLine(). Метод ToString() переопределяется во многих классах.

class Rectangle : Shape

{

. . . . .

override public bool Equals(object ob)

{

if (width == ((Rectangle)ob).Width && height == ((Rectangle)ob).Height) return true;

else return false;

}

override public string ToString()

{

return "Rectngle: ширина=" + width + " высота=" + height;

}

override public void Show()

{

Console.WriteLine(this.ToString());

}

override public int GetHashCode()

{

return (int)(height + width)%100;

}

public static bool operator ==(Rectangle a, Rectangle b)

{

return a.Equals(b);

}

public static bool operator !=(Rectangle a, Rectangle b)

{

return !a.Equals(b);

}

}

static void Main(string[] args)

{

Rectangle r = new Rectangle();

Triangle t1 = new Triangle(10, 15, "прямоугольный");

Triangle t2 = new Triangle(4, 4, "равнобедренный");

Rectangle r1 = new Rectangle(5, 6);

Shape[] arr = new Shape[4];

arr[0] = t1; arr[1] = t2; arr[2] = r1; arr[3] = r;

Console.WriteLine("Площади фигур:");

foreach (Shape s in arr)

{

s.Show();

}

Console.WriteLine("r==r1=" + (r == r1));

Console.WriteLine("r!=r1=" + (r != r1));

}

### Запрещение наследования

Для отключения механизма наследования используется ключевое слово sealed. Нельзя одновременно объявлять класс как abstract и sealed. т.к. абстрактный класс предполагает, что у него обязательно должны быть наследники.

sealed class Point

{

double x;

double y;

public Point(int x = 0, int y = 0)

{

this.x = 0;

this.y = 0;

}

public void Show()

{

Console.WriteLine(x + ", " + y);

}

}

class ColorPont : Point

{

int color;

public ColorPont(double x, double y, int color)

: base(x, y)

{

this.color = color;

}

}

При компиляции такой программы возникнет ошибка: error CS0509: 'AbstractClass.ColorPont': cannot derive from sealed type 'AbstractClass.Point'

## 4. Интерфейсы

Интерфейс определяет набор методов, которые будут реализованы классом. Сам интерфейс не реализует методы. Таким образом, **интерфейс** — это логическая конструкция, которая описывает методы, не устанавливая жестко способ их реализации.

Интерфейсы синтаксически подобны абстрактным классам. Однако в интерфейсе ни один метод не может включать тело, т.е. интерфейс не предусматривает реализации. Он определяет, **что должно быть сделано**, но не уточняет, как.

Для реализации интерфейса класс должен обеспечить способы реализации методов, описанных в интерфейсе. Каждый класс может определить собственную реализацию. Таким образом, два класса могут реализовать один и тот же интерфейс различными способами, но все классы поддерживают одинаковый набор методов.

Интерфейсы объявляются с помощью ключевого слова interface:

interface имя

{

тип\_возврата имя\_метода1 {список\_параметров) ;

тип\_возврата имя\_метода2 {список\_параметров) ;

// . . .

тип\_возврата имя\_методаN(список\_параметров) ;

}

В интерфейсе методы **неявно являются открытыми** (public-методами), при этом не разрешается явным образом указывать спецификатор доступа.

Чтобы реализовать интерфейс, нужно указать его имя после имени класса подобно тому, как при создании производного указывается базовый класс. Формат записи класса, который реализует интерфейс, таков:

class имя\_класса : имя\_\_интерфейса

{

// тело класса

}

**Пример 1.**

public interface ISeries

{

int getNext();//возвращает след. число ряда

void reset();//перезапуск

//устанавливает начальное значение

void setStart(int x);

}

class ByTwo : ISeries

{

int start;

int val;

public int getNext()

{

val += 2;

return val;

}

public void reset()

{

val = start;

}

public void setStart(int x)

{

start = x;

val = start;

}

}

static void Main(string[] args)

{

ByTwo ob = new ByTwo();

for (int i = 0; i < 5; i++)

Console.WriteLine("след. число" + ob.getNext());

Console.WriteLine();

ob.reset();

for (int i = 0; i < 5; i++)

Console.WriteLine("след. число" + ob.getNext());

Console.WriteLine();

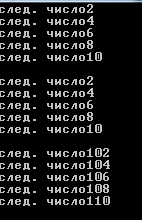
ob.setStart(100);

for (int i = 0; i < 5; i++)

Console.WriteLine("след. число" + ob.getNext());

Console.WriteLine();

}



class Primes:ISeries

{

int start=2;

int val=2;

public int getNext()

{

int i, j;

bool ok;

val++;

for (i = val; i < 10000000; i++)

{

ok = true;

for (j = 2; j < i; j++)

if (i % j == 0) { ok = false; break; }

if (ok) { val = i; break; }//простое число

}

return val;

}

public void reset()

{

val = start;

}

public void setStart(int x)

{

start = x;

val = start;

}

}

static void Main(string[] args)

{

Primes ob = new Primes();

for (int i = 0; i < 5; i++)

Console.WriteLine("след. число" + ob.getNext());

Console.WriteLine();

ob.reset();

for (int i = 0; i < 5; i++)

Console.WriteLine("след. число" + ob.getNext());

Console.WriteLine();

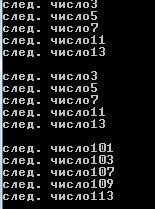
ob.setStart(100);

for (int i = 0; i < 5; i++)

Console.WriteLine("след. число" + ob.getNext());

Console.WriteLine();

}

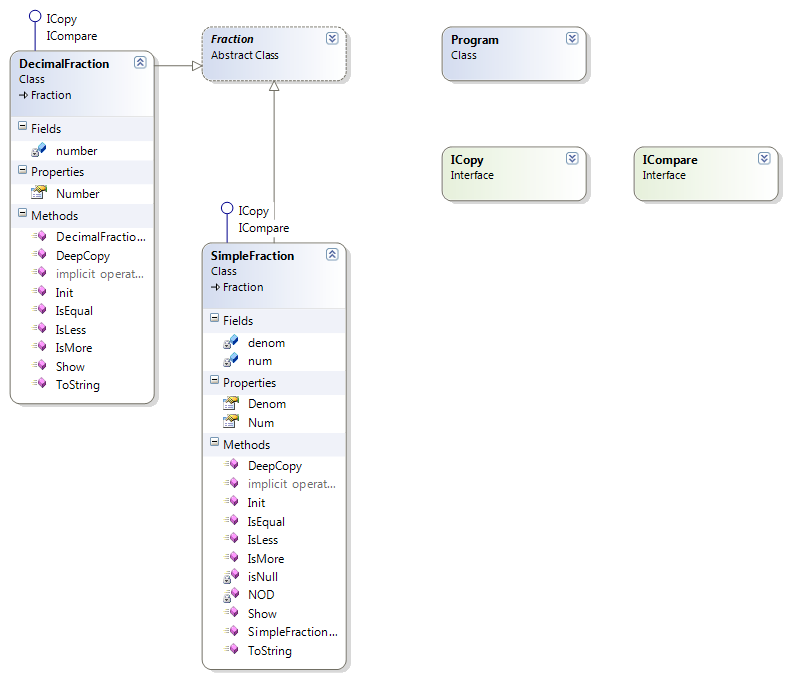


Здесь важно понимать, что, хотя классы Primes и ByTwo генерируют разные ряды чисел, оба они реализуют один и тот же интерфейс ISeries, т.к. каждый класс решает эту задачу так, как "считает" нужным.

**Отличия интерфейса от абстрактного класса:**

* Элементы интерфейса по умолчанию имеют спецификатор доступа publiс и не могут иметь спецификаторов, заданных явным образом.
* Все элементы интерфейса должны быть абстрактными;
* Класс, в списке предков которого задается интерфейс, должен определять все его элементы, потомок абстрактного класса может не переопределять часть абстрактных методов предка (в этом случае производный класс также будет абстрактным);
* Класс может иметь в списке предков несколько интерфейсов, при этом он должен определять все их методы.

**Пример 2**

****

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

namespace Fractions

{

abstract class Fraction

{

protected string name;

protected bool errorFlag=false;

public string Name

{

set { name = value; }

get { return name; }

}

public Fraction()

{

Name = "";

}

public Fraction(string n)

{

Name = n;

}

public abstract void Init();

public abstract void Show();

}

interface ICopy

{

object DeepCopy();

}

interface ICompare

{

bool IsLess(object obj);

bool IsEqual(object obj);

bool IsMore(object obj);

}

class SimpleFraction : Fraction, ICopy, ICompare

{

int num, denom;

public int Num

{

set { num = value; }

get { return num; }

}

public int Denom

{

set { if (value != 0)denom = value; else { Console.WriteLine("Знаменатель не может быть равен 0"); errorFlag = true; } }

get { if (!errorFlag)return denom; else { Console.WriteLine("Ошибка в знаменателе"); return 1; } }

}

public SimpleFraction(string n, int x, int y)

: base(n)

{

int nod = NOD(x, y);

Num = x / nod;

Denom = y / nod;

}

public SimpleFraction()

: base()

{

Num = 1; Denom = 1;

}

public override string ToString() //метод для вывода дроби

{

// Console.Write(name + " = ");

string temp="";

int absNum = Math.Abs(num);

int absDenum = Math.Abs(denom);

if (num == 0) //дробь равна 0

{

//Console.WriteLine(0);

temp="0";

return temp;

}

//отрицательная дробь

if (num \* denom < 0) //Console.Write("-");

temp = temp + "-";

//дробь сокращается

if (absNum % absDenum == 0)

{

// Console.WriteLine(absNum / absDenum);

int number = absNum / absDenum;

temp = temp + number.ToString();

return temp;

}

//числитель меньше знаменателя => целой части нет

if (absNum < absDenum)

{

// Console.WriteLine(absNum + "/" + absDenum);

temp = absNum.ToString() + "/" + absDenum.ToString();

return temp;

}

//числитель больше знаменателя => целая часть есть

// Console.WriteLine((absNum / absDenum) + " " + (absNum % absDenum) + "/" + absDenum);

temp = (absNum / absDenum) + " " + (absNum % absDenum) + "/" + absDenum;

return temp;

}

public override void Show() //метод для вывода дроби

{

Console.WriteLine(this.name+": "+this.ToString());

}

public override void Init()

{

do

{

errorFlag = false;

try

{

Console.WriteLine("Введите числитель");

Num = Convert.ToInt32(Console.ReadLine());

}

catch (Exception e)

{

Console.WriteLine(e.Message);

errorFlag = true;

}

}

while (errorFlag);

do

{ errorFlag = false;

try

{

Console.WriteLine("Введите знаменатель");

Denom = Convert.ToInt32(Console.ReadLine());

}

catch (Exception e)

{

Console.WriteLine(e.Message);

errorFlag = true;

}

}

while (errorFlag);

}

int NOD(int a, int b)

{

int r;

while (b != 0)

{

r = a % b;

a = b; b = r;

}

return a;

}

static bool isNull(double num) //проверка: дробная часть =0

{

double x = Math.Truncate(num);

if (x == num) return true;

else return false;

}

public static implicit operator SimpleFraction(DecimalFraction df)

{

double int\_part = Math.Truncate(df.Number);

double dec\_part = df.Number - int\_part;

double temp = dec\_part;

int power = 1;

while (!isNull(dec\_part))

{

dec\_part \*= 10;

power \*= 10;

}

int chisl = (int)(int\_part \* power + dec\_part);

SimpleFraction sf = new SimpleFraction("приведенная к SimpleFraction дробь", chisl, power);

return sf;

}

public object DeepCopy()

{

SimpleFraction temp = new SimpleFraction(name + "Copy", num, denom);

return temp;

}

public bool IsLess(object obj)

{

SimpleFraction s;

DecimalFraction d = obj as DecimalFraction;

if(d!=null) s=(SimpleFraction)d; else

s=(SimpleFraction)obj;

if (this.denom == s.denom)

return this.num < s.num;

else

{

//приводим к общему знаменателю

int temp1 = this.num \* s.denom;

int temp2 = s.num \* this.denom;

return temp1 < temp2;

}

}

public bool IsEqual(object obj)

{

SimpleFraction s;

DecimalFraction d = obj as DecimalFraction;

if (d != null) s = (SimpleFraction)d;

else

s = (SimpleFraction)obj;

if (this.denom == s.denom)

return this.num == s.num;

else

{

//приводим к общему знаменателю

int temp1 = this.num \* s.denom;

int temp2 = s.num \* this.denom;

return temp1 == temp2;

}

}

public bool IsMore(object obj)

{

SimpleFraction s;

DecimalFraction d = obj as DecimalFraction;

if(d!=null) s=(SimpleFraction)d; else

s = (SimpleFraction)obj;

if (this.denom == s.denom)

return this.num > s.num;

else

{

//приводим к общему знаменателю

int temp1 = this.num \* s.denom;

int temp2 = s.num \* this.denom;

return temp1 > temp2;

}

}

}

class DecimalFraction : Fraction, ICopy, ICompare

{

double number;

public DecimalFraction()

: base()

{

number = 0;

}

public DecimalFraction(string s, double n)

: base(s)

{

number = n;

}

public double Number

{

get

{

return number;

}

set

{

number = value;

}

}

public override string ToString()

{

return number.ToString();

}

public override void Show() //метод для вывода дроби

{

Console.WriteLine(this.name + ": " + this.ToString());

}

public override void Init()

{

do

{

errorFlag = false;

Console.WriteLine("Введите значение");

try

{

number = Convert.ToDouble(Console.ReadLine());

}

catch (Exception e)

{

Console.WriteLine(e.Message);

errorFlag = true;

}

} while (!errorFlag);

}

public static implicit operator DecimalFraction(SimpleFraction sf)

{

double number = sf.Num \* 0.1 / 0.1 / sf.Denom;

DecimalFraction temp = new DecimalFraction("приведенная к DecimalFraction дробь", number);

return temp;

}

public object DeepCopy()

{

DecimalFraction temp = new DecimalFraction(name + "Copy", number);

return temp;

}

public bool IsLess(object obj)

{

DecimalFraction temp;

SimpleFraction s = obj as SimpleFraction;

if (s != null) temp = (DecimalFraction)s;

else

temp = (DecimalFraction)obj;

if (this.Number < temp.Number) return true;

else return false;

}

public bool IsEqual(object obj)

{

DecimalFraction temp;

SimpleFraction s = obj as SimpleFraction;

if (s != null) temp = (DecimalFraction)s;

else

temp = (DecimalFraction)obj;

if (this.Number == temp.Number) return true;

else return false;

}

public bool IsMore(object obj)

{

DecimalFraction temp;

SimpleFraction s = obj as SimpleFraction;

if (s != null) temp = (DecimalFraction)s;

else

temp = (DecimalFraction)obj;

if (this.Number > temp.Number) return true;

else return false;

}

}

class Program

{

static void Main(string[] args)

{

SimpleFraction sf1 = new SimpleFraction("sf1",1,2);

sf1.Show();

SimpleFraction sf2 = new SimpleFraction("sf2",205, 100);

sf2.Show();

DecimalFraction df1 = new DecimalFraction("df1", 5.75);

df1.Show();

Console.WriteLine("Приведение типов");

SimpleFraction s = (SimpleFraction)df1;

Console.WriteLine(s.ToString());

DecimalFraction d = (DecimalFraction)s;

Console.WriteLine(d.ToString());

DecimalFraction df2 =(DecimalFraction) df1.DeepCopy();

df2.Show();

SimpleFraction sf3 = (SimpleFraction)sf1.DeepCopy();

sf3.Show();

Console.WriteLine("проверка Compare");

Console.WriteLine("{0}.IsEqual({1}):{2}",df1,df2, df1.IsEqual(df2));

Console.WriteLine("{0}.IsLess({1}):{2}", df1, df2, df1.IsLess(df2));

Console.WriteLine("{0}.IsMore({1}):{2}", df1, df2, df1.IsMore(df2));

Console.WriteLine("{0}.IsEqual({1}):{2}", sf1, sf2, sf1.IsEqual(sf2));

Console.WriteLine("{0}.IsLess({1}):{2}", sf1, sf2, sf1.IsLess(sf2));

Console.WriteLine("{0}.IsMore({1}):{2}", sf1, sf2, sf1.IsMore(sf2));

Console.WriteLine("{0}.IsEqual({1}):{2}", s, d, s.IsEqual(d));

Console.WriteLine("{0}.IsEqual({1}):{2}", sf1, df2, sf1.IsEqual(df2));

Console.WriteLine("{0}.IsLess({1}):{2}", sf1, df2, sf1.IsLess(df2));

Console.WriteLine("{0}.IsMore({1}):{2}", sf1, df2, sf1.IsMore(df2));

Console.WriteLine("{0}.IsEqual({1}):{2}", df1, sf2, df1.IsEqual(sf2));

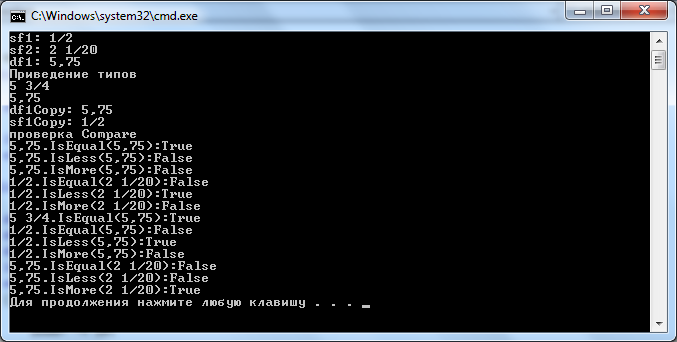
Console.WriteLine("{0}.IsLess({1}):{2}", df1, sf2, df1.IsLess(sf2));

Console.WriteLine("{0}.IsMore({1}):{2}", df1, sf2, df1.IsMore(sf2));

}

}

}

****

## 5. Работа с объектами через стандартные интерфейсы .NET

В библиотеке классов .NET определено множество стандартных интерфейсов, задающих желаемое поведение объектов.

Рассмотрим стандартные интерфейсы на примере:

* **Интерфейс IComparable** определен в пространстве имен System. Он содержит всего один метод CompareTo, возвращающий результат сравнения двух объектов – текущего и переданного ему в качестве параметра:

interface IComparable

{

int CompareTo( object obj )

}

Метод должен возвращать:

* 0, если текущий объект и параметр равны;
* отрицательное число, если текущий объект меньше параметра;
* положительное число, если текущий объект больше параметра.
* **Клонирование** - это создание копии объекта. Копия объекта называется клоном. При присваивании одного объекта ссылочного типа другому копируется ссылка (адрес), а не сам объект. Если необходимо скопировать в другую область памяти поля объекта, можно воспользоваться методом MemberwiseClone(), который любой объект наследует от класса object. При этом объекты, на которые указывают поля объекта, в свою очередь являющиеся ссылками, не копируются. Это называется поверхностным клонированием. Для создания полностью независимых объектов необходимо глубокое клонирование, когда в памяти создается дубликат всего дерева объектов, то есть объектов, на которые ссылаются поля объекта, поля полей и т. д.

Алгоритм глубокого клонирования весьма сложен, поскольку требует рекурсивного обхода всех ссылок объекта и отслеживания циклических зависимостей. Объект, имеющий собственные алгоритмы клонирования, должен объявляться как наследник интерфейса **ICloneable** и переопределять его единственный метод С1оne().

**Пример 3**

//Отсортировать вектор, в который записаны элементы классов SimpleFraction и //DecimalFraction.

abstract class Fraction

{

protected string name;

protected bool errorFlag = false;

public string Name

{

set { name = value; }

get { return name; }

}

public Fraction()

{

Name = "";

}

public Fraction(string n)

{

Name = n;

}

public abstract void Init();

public abstract void Show();

}

class SimpleFraction : Fraction, IComparable, ICloneable

{

int num, denom;

. . . . .

**public object Clone()**

{

SimpleFraction temp = new SimpleFraction(name + "Clone", num, denom);

return temp;

}

**public virtual int CompareTo(object obj)**

{

SimpleFraction s;

DecimalFraction d = obj as DecimalFraction;

if (d != null) s = (SimpleFraction)d;

else

s = (SimpleFraction)obj;

//приводим к общему знаменателю

int temp1 = this.num \* s.denom;

int temp2 = s.num \* this.denom;

if(temp1 < temp2) return -1;

else

if(temp1==temp2) return 0;

else return 1;

}

}

class DecimalFraction : Fraction, IComparable, ICloneable

{

protected double number;

. . . .

public object Clone()

{

DecimalFraction temp = new DecimalFraction(name + "Copy", number);

return temp;

}

public virtual int CompareTo(object obj)

{

DecimalFraction temp;

SimpleFraction s = obj as SimpleFraction;

if (s != null) temp = (DecimalFraction)s;

else

temp = (DecimalFraction)obj;

if (this.Number < temp.Number) return -1;

else

if(this.Number == temp.Number) return 0;

else return 1;

}

}

class Program

{

static void Main(string[] args)

{

SimpleFraction sf1 = new SimpleFraction("sf1", 1, 2);

sf1.Show();

SimpleFraction sf2 = new SimpleFraction("sf2", 205, 100);

sf2.Show();

DecimalFraction df1 = new DecimalFraction("df1", 5.75);

df1.Show();

Console.WriteLine("проверка клонирования");

DecimalFraction df2 = (DecimalFraction)df1.Clone();

df2.Show();

SimpleFraction sf3 = (SimpleFraction)sf1.Clone();

sf3.Show();

Console.WriteLine();

Fraction[] arr = new Fraction[4];

arr[0] = df1; arr[1] = sf2; arr[2] = sf1; arr[3] = df2;

foreach (Fraction f in arr)

f.Show();

Console.WriteLine();

Console.WriteLine("проверка Sort");

Array.Sort(arr);

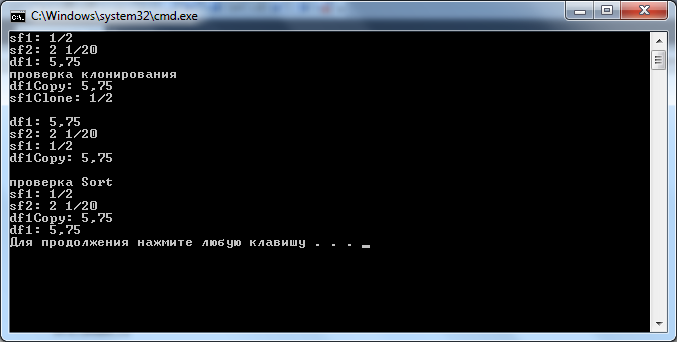
foreach (Fraction f in arr)

f.Show();

Console.WriteLine();

}

}



* Интерфейс **IComparer** определен в пространстве имен System. Collections. Он содержит один метод CompareTo, возвращающий результат сравнения двух объектов, переданных ему в качестве параметров:

interface IComparer

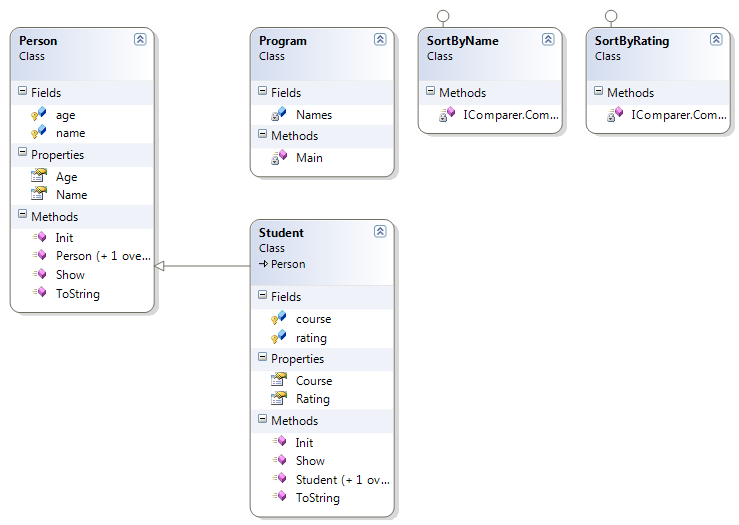
{

int Compare ( object obi, object ob2 )

}

Принцип применения этого интерфейса состоит в там, что для каждого критерия сортировки объектов описывается небольшой вспомогательный класс, реализующий этот интерфейс. Объект этого класса передается в стандартный метод сортировки массива в качестве второго аргумента (существует несколько перегруженных версий этого метода).

**Пример 4**

****

//Отсортировать вектор, в который записаны элементы класса Student по полю //name (age, rating, course).

using System;

using System.Collections;

using System.Linq;

using System.Text;

namespace Stand\_interface

{

class Person: ICloneable

{

protected string name;

protected int age;

public string Name

{

set { name = value; }

get { return name; }

}

public int Age

{

set { if (value > 0)age = value; else age = 0; }

get { return age; }

}

public Person()

{

name = ""; age = 0;

}

public Person(string name, int age)

{

this.name = name; this.age = age;

}

public virtual void Init()

{

string buf;

Console.WriteLine("Введите имя");

Name = Console.ReadLine();

Console.WriteLine("Введите возраст");

buf = Console.ReadLine();

Age = Convert.ToInt32(buf);

}

public override string ToString()

{

return name + ", " + age+"\n ";

}

public virtual void Show()

{

Console.WriteLine(this.ToString());

}

public Person ShallowCopy() //поверхностное копирование

{

return (Person)this.MemberwiseClone();

}

public object Clone()

{

return new Person("Клон" + this.name, this.age);

}

}

class Student : Person

{

protected int course;

protected double rating;

public int Course

{

set { if (value <= 5 && value >= 1) course = value; else course = 1; }

get { return course; }

}

public double Rating

{

set { if (value <= 5 && value >= 0) rating = value; else rating = 0; }

get { return rating; }

}

public Student()

: base()

{

rating = 0.0;

course = 1;

}

public Student(string name, int age, int course, double rating)

: base(name, age)

{

this.Rating = rating;

this.Course = course;

}

public override void Init()

{

base.Init();

Console.WriteLine("Введите курс");

Course = Convert.ToInt32(Console.ReadLine());

Console.WriteLine("введите рейтинг");

Rating = Convert.ToDouble(Console.ReadLine());

}

public override string ToString()

{

return base.ToString() + "курс=" + course + ", рейтинг=" + String.Format("{0:F2}",rating);

}

public override void Show()

{

Console.WriteLine(this.ToString());

}

public Student ShallowCopy() //поверхностное копирование

{

return (Student)this.MemberwiseClone();

}

public object Clone()

{

return new Student("Клон" + this.name, this.age, this.course,this.rating);

}

}

//реализация интерфейса для сортировки

public class SortByName : IComparer

{

int IComparer.Compare(object ob1, object ob2)

{

Student s1 = (Student)ob1;

Student s2 = (Student)ob2;

return String.Compare(s1.Name, s2.Name);

}

}

public class SortByRating : IComparer

{

int IComparer.Compare(object ob1, object ob2)

{

Student s1 = (Student)ob1;

Student s2 = (Student)ob2;

return s1.Rating.CompareTo(s2.Rating);

}

}

class Program

{

static string[] Names = { "Иванов", "Петров", "Сидоров", "Кузнецов", "Соколов", "Галкин", "Палкин", "Воробьев", "Воронов", "Семенов" };

static void Main(string[] args)

{

int size=5;

Random rnd = new Random();

Student[] arr = new Student[size];

for (int i = 0; i < size; i++)

{

Student s = new Student(Names[rnd.Next(0, 10)], rnd.Next(15, 25), rnd.Next(1, 6), rnd.Next(1, 6) + rnd.NextDouble());

arr[i] = s;

}

foreach (Student s in arr)

s.Show();

Console.WriteLine("\nСортировка по имени");

Array.Sort(arr, new SortByName());

foreach (Student s in arr)

s.Show();

Console.WriteLine("\nСортировка по рейтингу");

Array.Sort(arr, new SortByRating());

foreach (Student s in arr)

s.Show();

Console.WriteLine("\nКлонирование");

Person p1=arr[0].ShallowCopy();

p1.Show();

Person p2 = (Person)arr[0].Clone();

p2.Show();

Student s1 = arr[1].ShallowCopy();

s1.Show();

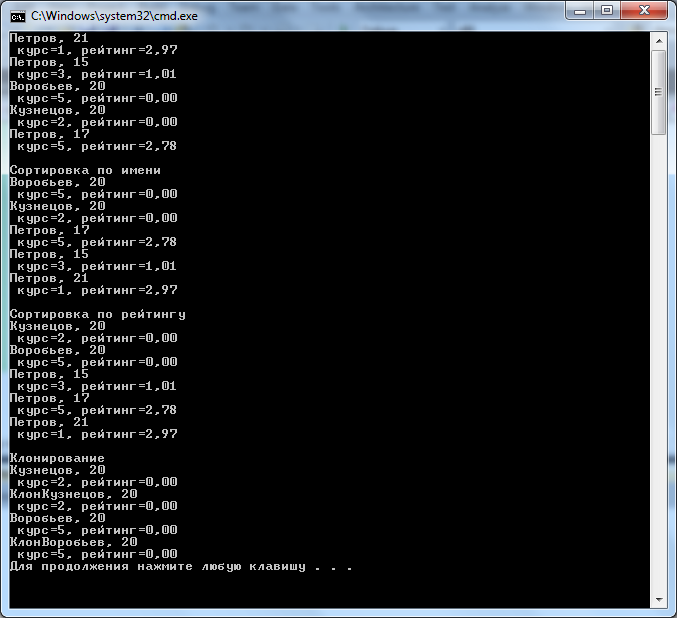
Student s2 = (Student)arr[1].Clone();

s2.Show();

}

}

}



**Перебор объектов (интерфейс IEnumerable) и итераторы**

Оператор foreach является удобным средством перебора элементов объекта.

Массивы и все стандартные коллекции библиотеки .NET позволяют выполнять такой перебор благодаря тому, что в них реализованы интерфейсы IEnumerable и IEnumerator. Для применения оператора foreach к пользовательскому типу данных требуется реализовать в нем эти интерфейсы.

Интерфейс IEnumerable (перечислимый) определяет всего один метод — GetEnuraerator, возвращающий объект типа IEnumerator (перечислитель), который можно использовать для просмотра элементов объекта.

Интерфейс IEnumerator задает три элемента:

* свойство Current, возвращающее текущий элемент объекта;
* метод MoveNext, продвигающий перечислитель на следующий элемент объекта;
* метод Reset, устанавливающий перечислитель в начало просмотра.

Цикл foreach использует эти методы для перебора элементов, из которых состоит объект.

Таким образом, если требуется, чтобы для перебора элементов класса мог применяться цикл foreach, необходимо реализовать четыре метода: GetEnumerator, Current, MoveNext и Reset. Например, если внутренние элементы класса организованы в массив, потребуется описать закрытое поле класса, хранящее текущий индекс в массиве, в методе MoveNext задать изменение этого индекса на 1 с проверкой выхода за границу массива, в методе Current — возврат элемента массива по текущему индексу и т. д.

Это не интересная работа, а выполнять ее приходится часто, поэтому в версию 2.0 были введены средства, облетающие выполнение перебора в объекте — **итераторы**.

Итератор представляет собой блок кода, задающий последовательность перебора элементов объекта. На каждом проходе цикла foreach выполняется один шаг итератора, заканчивающийся выдачей очередного значения. Выдача значения выполняется с помощью ключевого слова **yield**.

**Пример 5**

class FractionsArray : IEnumerable

{

Fraction[] arr;

int size;

int count;

public FractionsArray(int size = 0)

{

this.size = size;

arr = new Fraction[size];

count = 0;

}

public void AddFraction(Fraction f)

{

if (count < size)

{

arr[count] = f;

count++;

}

else return;

}

public IEnumerator GetEnumerator()

{

for (int i = 0; i < count; i++) yield return arr[i];

}

}

class Program

{

static void Main(string[] args)

{

Console.WriteLine("Проверка итератора");

FractionsArray frArr = new FractionsArray(10);

frArr.AddFraction(sf1);

frArr.AddFraction(sf2);

frArr.AddFraction(df1);

frArr.AddFraction(df2);

foreach (Fraction f in frArr) f.Show();

}

}

