# Глава 9
# Интерфейсы и структурные типы

В этой главе рассматриваются специальные виды классов — интерфейсы, структуры и перечисления.

## Синтаксис интерфейса

*Интерфейс* является «крайним случаем» абстрактного класса. В нем задается набор абстрактных методов, свойств и индексаторов, которые должны быть реализованы в производных классах[1]. Иными словами, интерфейс определяет поведение, которое поддерживается реализующими этот интерфейс классами. Основная идея использования интерфейса состоит в том, чтобы к объектам таких классов можно было обращаться одинаковым образом.

Каждый класс может определять элементы интерфейса по-своему. Так достигается полиморфизм: объекты разных классов по-разному реагируют на вызовы одного и того же метода.

Синтаксис интерфейса аналогичен синтаксису класса:

```
[ атрибуты ] [ спецификаторы ] interface имя_интерфейса [ : предки ]
    тело_интерфейса [ ; ]
```

Для интерфейса могут быть указаны *спецификаторы* `new`, `public`, `protected`, `internal` и `private`. Спецификатор `new` применяется для вложенных интерфейсов и имеет такой же смысл, как и соответствующий модификатор метода класса. Остальные спецификаторы управляют видимостью интерфейса. В разных контекстах определения интерфейса допускаются разные спецификаторы. По умолчанию интерфейс доступен только из сборки, в которой он описан (`internal`).

---

[1] Кроме того, в интерфейсе можно описывать события, которые мы рассмотрим в главе 10.

Интерфейс может наследовать свойства нескольких интерфейсов, в этом случае *предки* перечисляются через запятую. *Тело интерфейса* составляют абстрактные методы, шаблоны свойств и индексаторов, а также события.

**ПРИМЕЧАНИЕ**

Методом исключения можно догадаться, что интерфейс не может содержать константы, поля, операции, конструкторы, деструкторы, типы и любые статические элементы.

В качестве примера рассмотрим интерфейс IAction, определяющий базовое поведение персонажей компьютерной игры, встречавшихся в предыдущих главах. Допустим, что любой персонаж должен уметь выводить себя на экран, атаковать и красиво умирать:

```
interface IAction
{
    void Draw();
    int Attack(int a);
    void Die();
    int Power { get; }
}
```

В интерфейсе IAction заданы заголовки трех методов и шаблон свойства Power, доступного только для чтения. Как легко догадаться, если бы требовалось обеспечить еще и возможность установки свойства, в шаблоне следовало указать ключевое слово set, например:

```
int Power { get; set; }
```

В интерфейсе имеет смысл задавать заголовки тех методов и свойств, которые будут по-разному реализованы различными классами разных иерархий.

**ВНИМАНИЕ**

Если некий набор действий имеет смысл только для какой-то конкретной иерархии классов, реализующих эти действия разными способами, уместнее задать этот набор в виде виртуальных методов абстрактного базового класса иерархии. То, что работает в пределах иерархии одинаково, предпочтительно полностью определить в базовом классе (примерами таких действий являются свойства Health, Ammo и Name из иерархии персонажей игры). Интерфейсы же чаще используются для задания общих свойств объектов различных иерархий.

Отличия интерфейса от абстрактного класса:

- элементы интерфейса по умолчанию имеют спецификатор доступа public и не могут иметь спецификаторов, заданных явным образом;
- интерфейс не может содержать полей и обычных методов — все элементы интерфейса должны быть абстрактными;

- класс, в списке предков которого задается интерфейс, должен определять *все* его элементы, в то время как 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.

**ПРИМЕЧАНИЕ**

В библиотеке .NET определено большое количество стандартных интерфейсов, которые описывают поведение объектов разных классов. Например, если требуется сравнивать объекты по принципу больше или меньше, соответствующие классы должны реализовать интерфейс IComparable. Мы рассмотрим наиболее употребительные стандартные интерфейсы в последующих разделах этой главы.

# Реализация интерфейса

В списке предков класса сначала указывается его базовый класс, если он есть, а затем через запятую — интерфейсы, которые реализует этот класс. Таким образом, в C# поддерживается одиночное наследование для классов и множественное — для интерфейсов. Это позволяет придать производному классу свойства нескольких базовых интерфейсов, реализуя их по своему усмотрению.

Например, реализация интерфейса IAction в классе Monster может выглядеть следующим образом:

```
using System;
namespace ConsoleApplication1
{
    interface IAction
    {
        void Draw();
        int Attack( int a );
        void Die();
        int Power { get; }
    }

    class Monster : IAction
    {
        public void Draw()
        {
            Console.WriteLine( "Здесь был " + name );

        }
        public int Attack( int ammo_ )
        {
            ammo -= ammo_;
            if ( ammo > 0 ) Console.WriteLine( "Ба-бах!" );
            else            ammo = 0;
```

```
            return ammo;
        }

        public void Die()
        {
            Console.WriteLine( "Monster " + name + " RIP" );
            health = 0;
        }

        public int Power
        {
            get
            {
                return ammo * health;
            }
        }

        ...
}
```

Естественно, что сигнатуры методов в интерфейсе и реализации должны полностью совпадать. Для реализуемых элементов интерфейса в классе следует указывать спецификатор public. К этим элементам можно обращаться как через объект класса, так и через объект типа соответствующего интерфейса[1]:

```
Monster Vasia = new Monster( 50, 50, "Вася" );  // объект класса Monster
Vasia.Draw();                                   // результат: Здесь был Вася
IAction Actor = new Monster( 10, 10, "Маша" );  // объект типа интерфейса
Actor.Draw();                                   // результат: Здесь был Маша
```

Удобство второго способа проявляется при присваивании объектам типа IAction ссылок на объекты различных классов, поддерживающих этот интерфейс. Например, легко себе представить метод с параметром типа интерфейса. На место этого параметра можно передавать любой объект, реализующий интерфейс:

```
static void Act( IAction A )
{
    A.Draw();
}

static void Main()
{
    Monster Vasia = new Monster( 50, 50, "Вася" );
    Act( Vasia );
    ...
}
```

[1] Этот объект должен содержать ссылку на класс, поддерживающий интерфейс. Естественно, что объекты типа интерфейса, так же как и объекты абстрактных классов, создавать нельзя.

Существует второй способ реализации интерфейса в классе: *явное указание имени интерфейса* перед реализуемым элементом. Спецификаторы доступа при этом не указываются. К таким элементам можно обращаться в программе *только через объект типа интерфейса*, например:

```
class Monster : IAction
{
    int IAction.Power
    {
        get
        {
            return ammo * health;
        }
    }

    void IAction.Draw()
    {
        Console.WriteLine( "Здесь был " + name );
    }
    ...
}
...
IAction Actor = new Monster( 10, 10, "Маша" );
Actor.Draw();                    // обращение через объект типа интерфейса

// Monster Vasia = new Monster( 50, 50, "Вася" );
// Vasia.Draw();                                          ошибка!
```

Таким образом, при явном задании имени реализуемого интерфейса соответствующий метод *не входит в интерфейс класса*. Это позволяет упростить его в том случае, если какие-то элементы интерфейса не требуются конечному пользователю класса.

Кроме того, явное задание имени реализуемого интерфейса перед именем метода позволяет избежать конфликтов при множественном наследовании, если элементы с одинаковыми именами или сигнатурой встречаются более чем в одном интерфейсе[1]. Пусть, например, класс Monster поддерживает два интерфейса: один для управления объектами, а другой для тестирования:

```
interface ITest
{
    void Draw();
}

interface IAction
```

[1] Методы с одинаковыми именами, но с различными сигнатурами к конфликту не приводят, они просто считаются перегруженными.

```
{
    void Draw();
    int Attack( int a );
    void Die();
    int Power { get; }
}

class Monster : IAction, ITest
{
    void ITest.Draw()
    {
        Console.WriteLine( "Testing " + name );
    }

    void IAction.Draw()
    {
        Console.WriteLine( "Здесь был " + name );
    }
    ...
}
```

Оба интерфейса содержат метод Draw с одной и той же сигнатурой. Различать их помогает явное указание имени интерфейса. Обращаться к этим методам можно, используя операцию приведения типа, например:

```
Monster Vasia = new Monster( 50, 50, "Вася" );
((ITest)Vasia).Draw();                    // результат: Здесь был Вася
((IAction)Vasia).Draw();                  // результат:   Testing Вася
```

Впрочем, если от таких методов не требуется разное поведение, можно реализовать метод первым способом (со спецификатором public), компилятор не возражает:

```
class Monster : IAction, ITest
{
    public void Draw()
    {
        Console.WriteLine( "Здесь был " + name );
    }
    ...
}
```

К методу Draw, описанному таким образом, можно обращаться любым способом: через объект класса Monster, через интерфейс IAction или ITest.

Конфликт возникает в том случае, если компилятор не может определить из контекста обращения к элементу, элемент какого именно из реализуемых интерфейсов требуется вызвать. При этом всегда помогает явное задание имени интерфейса.

# Работа с объектами через интерфейсы. Операции is и as

При работе с объектом через объект типа интерфейса бывает необходимо убедиться, что объект поддерживает данный интерфейс. Проверка выполняется с помощью бинарной операции is. Эта операция определяет, совместим ли текущий тип объекта, находящегося слева от ключевого слова is, с типом, заданным справа.

Результат операции равен true, если объект можно преобразовать к заданному типу, и false в противном случае. Операция обычно используется в следующем контексте:

```
if ( объект is тип )
{
    // выполнить преобразование "объекта" к "типу"
    // выполнить действия с преобразованным объектом
}
```

Допустим, мы оформили какие-то действия с объектами в виде метода с параметром типа object. Прежде чем использовать этот параметр внутри метода для обращения к методам, описанным в производных классах, требуется выполнить преобразование к производному классу. Для безопасного преобразования следует проверить, возможно ли оно, например так:

```
static void Act( object A )
{
    if ( A is IAction )
    {
        IAction Actor = (IAction) A;
        Actor.Draw();
    }
}
```

В метод Act можно передавать любые объекты, но на экран будут выведены только те, которые поддерживают интерфейс IAction.

Недостатком использования операции is является то, что преобразование фактически выполняется дважды: при проверке и при собственно преобразовании. Более эффективной является другая операция — as. Она выполняет преобразование к заданному типу, а если это невозможно, формирует результат null, например:

```
static void Act( object A )
{
    IAction Actor = A as IAction;
    if ( Actor != null ) Actor.Draw();
}
```

Обе рассмотренные операции применяются как к интерфейсам, так и к классам.

# Интерфейсы и наследование

Интерфейс может не иметь или иметь сколько угодно интерфейсов-предков, в последнем случае он наследует все элементы всех своих базовых интерфейсов, начиная с самого верхнего уровня. Базовые интерфейсы должны быть доступны в не меньшей степени, чем их потомки. Например, нельзя использовать интерфейс, описанный со спецификатором private или internal, в качестве базового для открытого (public) интерфейса[1].

Как и в обычной иерархии классов, базовые интерфейсы определяют общее поведение, а их потомки конкретизируют и дополняют его. В интерфейсе-потомке можно также указать элементы, переопределяющие унаследованные элементы с такой же сигнатурой. В этом случае перед элементом указывается ключевое слово new, как и в аналогичной ситуации в классах. С помощью этого слова соответствующий элемент базового интерфейса скрывается. Вот пример из документации C#:

```
interface IBase
{
   void F( int i );
}
interface Ileft : IBase
{
   new void F( int i );                // переопределение метода F
}
interface Iright : IBase
{
   void G();
}
interface Iderived : ILeft, IRight {}

class A
{
   void Test( IDerived d ) {
      d.F( 1 );                        // Вызывается ILeft.F
      ((IBase)d).F( 1 );               // Вызывается IBase.F
      ((ILeft)d).F( 1 );               // Вызывается ILeft.F
      ((IRight)d).F( 1 );              // Вызывается IBase.F
   }
}
```

Метод F из интерфейса IBase скрыт интерфейсом ILeft, несмотря на то что в цепочке IDerived — IRight — IBase он не переопределялся.

Класс, реализующий интерфейс, должен определять все его элементы, в том числе унаследованные. Если при этом явно указывается имя интерфейса, оно

[1] Естественно, интерфейс не может быть наследником самого себя.

должно ссылаться на тот интерфейс, в котором был описан соответствующий элемент, например:

```
class A : IRight
{
    IRight.G() { ... }
    IBase.F( int i ) { ... }          // IRight.F( int i ) - нельзя
}
```

Интерфейс, на собственные или унаследованные элементы которого имеется явная ссылка, должен быть указан в списке предков класса, например:

```
class B : A
{
//    IRight.G() { ... }              нельзя!
}
class C : A, IRight
{
    IRight.G() { ... }              // можно
    IBase.F( int i ) { ... }        // можно
}
```

Класс наследует все методы своего предка, в том числе те, которые реализовывали интерфейсы. Он может переопределить эти методы с помощью спецификатора new, но обращаться к ним можно будет только через объект класса. Если использовать для обращения ссылку на интерфейс, вызывается не переопределенная версия:

```
interface IBase
{
   void A();
}

class Base : IBase
{
   public void A() { ... }
}

class Derived: Base
{
   new public void A() { ... }
}

...
Derived d = new Derived ();
d.A();                        // вызывается Derived.A();
IBase id = d;
id.A();                       // вызывается Base.A();
```

Однако если интерфейс реализуется с помощью виртуального метода класса, после его переопределения в потомке любой вариант обращения (через класс или через интерфейс) приведет к одному и тому же результату:

```
interface IBase
{
   void A();
}

class Base : IBase
{
   public virtual void A() { ... }
}

class Derived: Base
{
   public override void A() { ... }
}

...
Derived d = new Derived ();
d.A();                          // вызывается Derived.A();
IBase id = d;
id.A();                         // вызывается Derived.A();
```

Метод интерфейса, реализованный явным указанием имени, объявлять виртуальным запрещается. При необходимости переопределить в потомках его поведение пользуются следующим приемом: из этого метода вызывается другой, защищенный метод, который объявляется виртуальным. В приведенном далее примере метод A интерфейса IBase реализуется посредством защищенного виртуального метода A_, который можно переопределять в потомках класса Base:

```
interface IBase
{
   void A();
}

class Base : IBase
{
   void IBase.A() { A_(); }
   protected virtual void A_() { ... }
}

class Derived: Base
{
   protected override void A_() { ... }
}
```

Существует возможность *повторно реализовать интерфейс*, указав его имя в списке предков класса наряду с классом-предком, уже реализовавшим этот интерфейс. При этом реализация переопределенных методов базового класса во внимание не принимается:

```
interface IBase
{
   void A();
```

```
}

class Base : IBase
{
   void IBase.A() { ... }        // не используется в Derived
}

class Derived : Base, IBase
{
   public void A() { ... }
}
```

Если класс наследует от класса и интерфейса, которые содержат методы с одинаковыми сигнатурами, унаследованный метод класса воспринимается как реализация интерфейса, например:

```
interface Interface1
{
   void F();
}

class Class1
{
   public void F() { ... }
   public void G() { ... }
}

class Class2 : Class1, Interface1
{
   new public void G() { ... }
}
```

Здесь класс `Class2` наследует от класса `Class1` метод `F`. Интерфейс `Interface1` также содержит метод `F`. Компилятор не выдает ошибку, потому что класс `Class2` содержит метод, подходящий для реализации интерфейса.

Вообще при реализации интерфейса учитывается наличие «подходящих» методов в классе независимо от их происхождения. Это могут быть методы, описанные в текущем или базовом классе, реализующие интерфейс явным или неявным образом.

# Стандартные интерфейсы .NET

В библиотеке классов .NET определено множество стандартных интерфейсов, задающих желаемое поведение объектов. Например, интерфейс `IComparable` задает метод сравнения объектов по принципу больше или меньше, что позволяет выполнять их сортировку. Реализация интерфейсов `IEnumerable` и `IEnumerator` дает возможность просматривать содержимое объекта с помощью конструкции `foreach`, а реализация интерфейса `ICloneable` — клонировать объекты.

Стандартные интерфейсы поддерживаются многими стандартными классами библиотеки. Например, работа с массивами с помощью цикла foreach возможна именно потому, что тип Array реализует интерфейсы IEnumerable и IEnumerator. Можно создавать и собственные классы, поддерживающие стандартные интерфейсы, что позволит использовать объекты этих классов стандартными способами.

## Сравнение объектов (интерфейс IComparable)

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.

Реализуем интерфейс IComparable в знакомом нам классе Monster. В качестве критерия сравнения объектов выберем поле health. В листинге 9.1 приведена программа, сортирующая массив монстров по возрастанию величины, характеризующей их здоровье (элементы класса, не используемые в данной программе, не приводятся).

**Листинг 9.1.** Пример реализации интерфейса IComparable

```
using System;
namespace ConsoleApplication1
{
    class Monster : IComparable
    {
        public Monster( int health, int ammo, string name )
        {
            this.health = health;
            this.ammo   = ammo;
            this.name   = name;
        }

        virtual public void Passport()
        {
            Console.WriteLine( "Monster {0} \t health = {1} ammo = {2}",
                               name, health, ammo );
        }

        public int CompareTo( object obj )          // реализация интерфейса
```

*продолжение* ⇨

**Листинг 9.1** (*продолжение*)

```
        {
            Monster temp = (Monster) obj;
            if ( this.health > temp.health ) return  1;
            if ( this.health < temp.health ) return -1;
            return 0;
        }

        string name;
        int health, ammo;
    }

    class Class1
    {   static void Main()
        {
            const int n = 3;
            Monster[] stado = new Monster[n];

            stado[0] = new Monster( 50, 50, "Вася" );
            stado[1] = new Monster( 80, 80, "Петя" );
            stado[2] = new Monster( 40, 10, "Маша" );

            Array.Sort( stado );          // сортировка стала возможной
            foreach ( Monster elem in stado ) elem.Passport();
        }
    }
}
```

Результат работы программы:

```
Monster Маша     health = 40 ammo = 10
Monster Вася     health = 50 ammo = 50
Monster Петя     health = 80 ammo = 80
```

Если несколько объектов имеют одинаковое значение критерия сортировки, их относительный порядок следования после сортировки не изменится.

Во многих алгоритмах требуется выполнять сортировку объектов по различным критериям. В C# для этого используется интерфейс `IComparer`, который рассмотрен в следующем разделе.

## Сортировка по разным критериям (интерфейс IComparer)

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

```
interface IComparer
{
    int Compare ( object ob1, object ob2 )
}
```

Принцип применения этого интерфейса состоит в том, 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 сортировки массива объектов из предыдущего листинга по именам (свойство Name, класс SortByName) и количеству вооружений (свойство Ammo, класс SortByAmmo) приведен в листинге 9.2. Классы параметров сортировки объявлены вложенными, поскольку они требуются только объектам класса Monster.

**Листинг 9.2.** Сортировка по двум критериям

```
using System;
using System.Collections;
namespace ConsoleApplication1
{
    class Monster
    {
        public Monster( int health, int ammo, string name )
        {
            this.health = health;
            this.ammo   = ammo;
            this.name   = name;
        }

        public int Ammo
        {
            get { return ammo; }
            set
            {
                if (value > 0) ammo = value;
                else           ammo = 0;
            }
        }

        public string Name
        {
            get { return name; }
        }

        virtual public void Passport()
        {
            Console.WriteLine( "Monster {0} \t health = {1} ammo = {2}",
                               name, health, ammo );
        }

        public class SortByName : IComparer                              //
        {
            int IComparer.Compare( object ob1, object ob2 )
```

продолжение ⇨

**Листинг 9.2** (*продолжение*)

```
            {
                Monster m1 = (Monster) ob1;
                Monster m2 = (Monster) ob2;
                return String.Compare( m1.Name, m2.Name );
            }
        }

        public class SortByAmmo : IComparer                                //
        {
            int IComparer.Compare( object ob1, object ob2 )
            {
                Monster m1 = (Monster) ob1;
                Monster m2 = (Monster) ob2;
                if ( m1.Ammo > m2.Ammo ) return  1;
                if ( m1.Ammo < m2.Ammo ) return -1;
                return 0;
            }
        }
        string name;
        int health, ammo;
    }

    class Class1
    {   static void Main()
        {
            const int n = 3;
            Monster[] stado = new Monster[n];

            stado[0] = new Monster( 50, 50, "Вася" );
            stado[1] = new Monster( 80, 80, "Петя" );
            stado[2] = new Monster( 40, 10, "Маша" );

            Console.WriteLine( "Сортировка по имени:" );
            Array.Sort( stado, new Monster.SortByName() );
            foreach ( Monster elem in stado ) elem.Passport();

            Console.WriteLine( "Сортировка по вооружению:" );
            Array.Sort( stado, new Monster.SortByAmmo() );
            foreach ( Monster elem in stado ) elem.Passport();
        }
    }
}
```

Результат работы программы:

```
Сортировка по имени:
Monster Вася     health = 50 ammo = 50
Monster Маша     health = 40 ammo = 10
Monster Петя     health = 80 ammo = 80
```

```
Сортировка по вооружению:
Monster Маша      health = 40 ammo = 10
Monster Вася      health = 50 ammo = 50
Monster Петя      health = 80 ammo = 80
```

## Перегрузка операций отношения

Если класс реализует интерфейс IComparable, его экземпляры можно сравнивать между собой по принципу больше или меньше. Логично разрешить использовать для этого операции отношения, перегрузив их. Операции должны перегружаться парами: < и >, <= и >=, == и !=. Перегрузка операций обычно выполняется путем делегирования, то есть обращения к переопределенным методам CompareTo и Equals.

**ПРИМЕЧАНИЕ**

Если класс реализует интерфейс IComparable, требуется переопределить метод Equals и связанный с ним метод GetHashCode. Оба метода унаследованы от базового класса object. Пример перегрузки был приведен в разделе «Класс object» (см. с. 183).

В листинге 9.3 операции отношения перегружены для класса Monster. В качестве критерия сравнения объектов по принципу больше или меньше выступает поле health, а при сравнении на равенство реализуется значимая семантика, то есть попарно сравниваются все поля объектов

**Листинг 9.3.** Перегрузка операций отношения

```
using System;
namespace ConsoleApplication1
{
    class Monster : IComparable
    {
        public Monster( int health, int ammo, string name )
        {
            this.health = health;
            this.ammo   = ammo;
            this.name   = name;
        }

        public override bool Equals( object obj )
        {
            if ( obj == null || GetType() != obj.GetType() ) return false;

            Monster temp = (Monster) obj;
            return health == temp.health &&
                   ammo   == temp.ammo   &&
                   name   == temp.name;
        }

        public override int GetHashCode()
```

продолжение ⇨

**Листинг 9.3** (*продолжение*)

```
        {
            return name.GetHashCode();
        }

        public static bool operator == ( Monster a, Monster b )
        {
            return a.Equals( b );
        }

//    вариант:
//        public static bool operator == ( Monster a, Monster b )
//        {
//            return ( a.CompareTo( b ) == 0 );
//        }

        public static bool operator != ( Monster a, Monster b )
        {
            return ! a.Equals( b );
        }

//    вариант:
//        public static bool operator != ( Monster a, Monster b )
//        {
//            return ( a.CompareTo( b ) != 0 );
//        }

        public static bool operator < ( Monster a, Monster b )
        {
            return ( a.CompareTo( b ) < 0 );
        }

        public static bool operator > ( Monster a, Monster b )
        {
            return ( a.CompareTo( b ) > 0 );
        }

        public static bool operator <= ( Monster a, Monster b )
        {
            return ( a.CompareTo( b ) <= 0 );
        }

        public static bool operator >= ( Monster a, Monster b )
        {
            return ( a.CompareTo( b ) >= 0 );
        }

        public int CompareTo( object obj )
        {
```

```
            Monster temp = (Monster) obj;
            if ( this.health > temp.health ) return  1;
            if ( this.health < temp.health ) return -1;
            return 0;
        }

        string name;
        int health, ammo;
    }

    class Class1
    {   static void Main()
        {
            Monster Вася = new Monster( 70, 80, "Вася" );
            Monster Петя = new Monster( 80, 80, "Петя" );

            if      ( Вася > Петя )  Console.WriteLine( "Вася больше Пети" );
            else if ( Вася == Петя ) Console.WriteLine( "Вася == Петя" );
            else                     Console.WriteLine( "Вася меньше Пети" );
        }
    }
}
```

Результат работы программы не разочаровывает:

```
Вася меньше Пети
```

## Клонирование объектов (интерфейс ICloneable)

*Клонирование* — это создание копии объекта. Копия объекта называется клоном. Как вам известно, при присваивании одного объекта ссылочного типа другому копируется ссылка, а не сам объект (рис. 9.1, *а*). 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 (рис. 9.1, *б*). Это называется *поверхностным клонированием*.

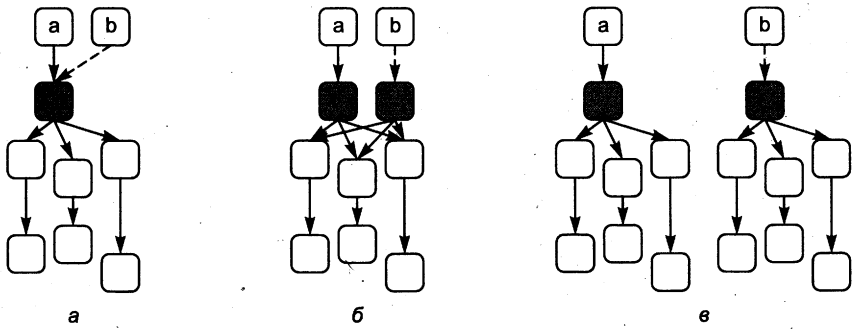


**Рис. 9.1.** Клонирование объектов

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. (рис. 9.1, *в*). 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 `Clone`. В листинге 9.4 приведен пример создания поверхностной копии объекта класса `Monster` с помощью метода `MemberwiseClone`, а также реализован интерфейс `ICloneable`. В демонстрационных целях в имя клона объекта добавлено слово «Клон».

Обратите внимание на то, что метод `MemberwiseClone` можно вызвать только из методов класса. Он не может быть вызван непосредственно, поскольку объявлен в классе `object` как защищенный (`protected`).

**Листинг 9.4.** Клонирование объектов

```
using System;
namespace ConsoleApplication1
{
    class Monster : ICloneable
    {
        public Monster( int health, int ammo, string name )
        {
            this.health = health;
            this.ammo   = ammo;
            this.name   = name;
        }
        public Monster ShallowClone()                    // поверхностная копия
        {
            return (Monster)this.MemberwiseClone();
        }

        public object Clone()                         // пользовательская копия
        {
            return new Monster( this.health, this.ammo, "Клон " + this.name );
        }

        virtual public void Passport()
        {
            Console.WriteLine( "Monster {0} \t health = {1} ammo = {2}",
                               name, health, ammo );
        }

        string name;
        int health, ammo;
    }

    class Class1
    {   static void Main()
```

```
        {
            Monster Вася = new Monster( 70, 80, "Вася" );
            Monster X = Вася;
            Monster Y = Вася.ShallowClone();
            Monster Z = (Monster)Вася.Clone();
            ...
        }
    }
}
```

Объект X ссылается на ту же область памяти, что и объект Вася. Следовательно, если мы внесем изменения в один из этих объектов, это отразится на другом. Объекты Y и Z, созданные путем клонирования, обладают собственными копиями значений полей и независимы от исходного объекта.

## 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. Давайте посмотрим, как это делается.

Интерфейс IEnumerable (*перечислимый*) определяет всего один метод — GetEnumerator, 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, облегчающие 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.

Рассмотрим создание итератора на примере (листинг 9.5). Пусть требуется создать объект, содержащий боевую группу экземпляров типа Monster, неоднократно использованного в примерах этой книги. Для простоты ограничим максимальное количество бойцов в группе десятью.

**Листинг 9.5.** Класс с итератором

```
using System;
using System.Collections;
namespace ConsoleApplication1
{
    class Monster { ... }
    class Daemon  { ... }
    class Stado : IEnumerable                                          // 1
    {
        private Monster[] mas;
        private int n;

        public Stado()
        {
            mas = new Monster[10];
            n = 0;
        }

        public IEnumerator GetEnumerator()
        {
            for ( int i = 0; i < n; ++i ) yield return mas[i];         // 2
        }

        public void Add( Monster m )
        {
            if ( n >= 10 ) return;
            mas[n] = m;
            ++n;
        }
    }

    class Class1
    {   static void Main()
        {
            Stado s = new Stado();
            s.Add( new Monster() );
            s.Add( new Monster("Вася") );
            s.Add( new Daemon() );

            foreach ( Monster m in s ) m.Passport();
        }
    }
}
```

Все, что требуется сделать в версии 2.0 для поддержки перебора, — указать, что класс реализует интерфейс IEnumerable (оператор 1), и описать итератор (оператор 2). Доступ к нему может быть осуществлен через методы MoveNext и Current интерфейса IEnumerator.

За кодом, приведенным в листинге 9.5, стоит большая внутренняя работа компилятора. На каждом шаге цикла foreach для итератора создается «оболочка» — служебный объект, который запоминает текущее состояние итератора и выполняет все необходимое для доступа к просматриваемым элементам объекта. Иными словами, код, составляющий итератор, не выполняется так, как он выглядит — в виде непрерывной последовательности, а разбит на отдельные итерации, между которыми состояние итератора сохраняется.

В листинге 9.6 приведен пример итератора, перебирающего четыре заданных строки.

**Листинг 9.6.** Простейший итератор

```
using System;
using System.Collections;
namespace ConsoleApplication1
{
    class Num : IEnumerable
    {
        public IEnumerator GetEnumerator()
        {
            yield return "one";
            yield return "two";
            yield return "three";
            yield return "oops";
        }
    }

    class Class1
    {   static void Main()
        {
            foreach ( string s in new Num() ) Console.WriteLine( s );
        }
    }
}
```

Результат работы программы:

```
one
two
three
oops
```

Следующий пример демонстрирует перебор значений в заданном диапазоне (от 1 до 5):

```
using System;
using System.Collections;
```

```
namespace ConsoleApplication1
{   class Class1
    {
        public static IEnumerable Count( int from, int to )
        {
            from = 1;
            while ( from <= to ) yield return from++;
        }
        static void Main()
        {
            foreach ( int i in Count( 1, 5 ) ) Console.WriteLine( i );
        }
    }
}
```

Преимущество использования итераторов заключается в том, что для одного и того же класса можно задать различный порядок перебора элементов. В листинге 9.7 описаны две дополнительные стратегии перебора элементов класса Stado, введенного в листинге 9.5, — перебор в обратном порядке и выборка только тех объектов, которые являются экземплярами класса Monster (для этого использован метод получения типа объекта GetType, унаследованный от базового класса object).

**Листинг 9.7.** Реализация нескольких стратегий перебора

```
using System;
using System.Collections;
using MonsterLib;

namespace ConsoleApplication1
{
    class Monster { ... }
    class Daemon  { ... }
    class Stado : IEnumerable
    {
        private Monster[] mas;
        private int n;
        public Stado()
        {
            mas = new Monster[10];
            n = 0;
        }
        public IEnumerator GetEnumerator()
        {
            for ( int i = 0; i < n; ++i ) yield return mas[i];
        }
        public IEnumerable Backwards()                    // в обратном порядке
        {
            for ( int i = n - 1; i >= 0; --i ) yield return mas[i];
        }

        public IEnumerable MonstersOnly()                 // только монстры
```

```
        {
            for ( int i = 0; i < n; ++i )
                if ( mas[i].GetType().Name == "Monster" )
                    yield return mas[i];
        }
        public void Add( Monster m )
        {
            if ( n >= 10 ) return;
            mas[n] = m;
            ++n;
        }
    }

    class Class1
    {   static void Main()
        {
            Stado s = new Stado();
            s.Add( new Monster() );
            s.Add( new Monster("Вася") );
            s.Add( new Daemon() );

            foreach ( Monster i in s )                i.Passport();
            foreach ( Monster i in s.Backwards() )    i.Passport();
            foreach ( Monster i in s.MonstersOnly() ) i.Passport();
        }
    }
}
```

Теперь, когда вы получили представление об итераторах, рассмотрим их более формально.

Блок итератора синтаксически представляет собой обычный блок и может встречаться в теле метода, операции или части get свойства, если соответствующее возвращаемое значение имеет тип IEnumerable или IEnumerator[1].

В теле блока итератора могут встречаться две конструкции:

- yield return формирует значение, выдаваемое на очередной итерации;
- yield break сигнализирует о завершении итерации.

Ключевое слово yield имеет специальное значение для компилятора только в этих конструкциях.

Код блока итератора выполняется не так, как обычные блоки. Компилятор формирует служебный *объект-перечислитель*, при вызове метода MoveNext которого выполняется код блока итератора, выдающий очередное значение с помощью ключевого слова yield. Следующий вызов метода MoveNext объекта-перечислителя возобновляет выполнение блока итератора с момента, на котором он был приостановлен в предыдущий раз.

---

[1] А также тип их параметризованных двойников IEnumerable<T> или IEnumerator<T> из пространства имен System.Collections.Generic, описанного в главе 13.

# Структуры

*Структура* — тип данных, аналогичный классу, но имеющий ряд важных отличий от него:

- структура является *значимым*, а не ссылочным типом данных, то есть экземпляр структуры хранит значения своих элементов, а не ссылки на них, и располагается в стеке, а не в хипе;
- структура не может участвовать в иерархиях наследования, она может только реализовывать интерфейсы;
- в структуре запрещено определять конструктор по умолчанию, поскольку он определен неявно и присваивает всем ее элементам значения по умолчанию (нули соответствующего типа);
- в структуре запрещено определять деструкторы, поскольку это бессмысленно.

**ПРИМЕЧАНИЕ**

Строго говоря, любой значимый тип C# является структурным.

Отличия от классов обусловливают *область применения структур*: типы данных, имеющие небольшое количество полей, с которыми удобнее работать как со значениями, а не как со ссылками. Накладные расходы на динамическое выделение памяти для небольших объектов могут весьма значительно снизить быстродействие программы, поэтому их эффективнее описывать как структуры, а не как классы.

**ПРИМЕЧАНИЕ**

С другой стороны, передача структуры в метод по значению требует и дополнительного времени, и дополнительной памяти.

Синтаксис структуры:

```
[ атрибуты ] [ спецификаторы ] struct имя_структуры [ : интерфейсы ]
    тело_структуры [ ; ]
```

*Спецификаторы* структуры имеют такой же смысл, как и для класса, причем из спецификаторов доступа допускаются только `public`, `internal` и `private` (последний — только для вложенных структур).

*Интерфейсы*, реализуемые структурой, перечисляются через запятую. *Тело структуры* может состоять из констант, полей, методов, свойств, событий, индексаторов, операций, конструкторов и вложенных типов. Правила их описания и использования аналогичны соответствующим элементам классов, за исключением некоторых отличий, вытекающих из упомянутых ранее:

- поскольку структуры не могут участвовать в иерархиях, для их элементов не могут использоваться спецификаторы `protected` и `protected internal`;
- структуры не могут быть абстрактными (`abstract`), к тому же по умолчанию они бесплодны (`sealed`);

- методы структур не могут быть абстрактными и виртуальными;
- переопределяться (то есть описываться со спецификатором override) могут только методы, унаследованные от базового класса object;
- параметр this интерпретируется как значение, поэтому его можно использовать для ссылок, но не для присваивания;
- при описании структуры нельзя задавать значения полей по умолчанию[1] — это будет сделано в конструкторе по умолчанию, создаваемом автоматически (конструктор присваивает значимым полям структуры нули, а ссылочным — значение null).

В листинге 9.8 приведен пример описания структуры, представляющей комплексное число. Для экономии места из всех операций приведено только описание сложения. Обратите внимание на перегруженный метод ToString: он позволяет выводить экземпляры структуры на консоль, поскольку неявно вызывается в методе Console.WriteLine. Использованные в методе спецификаторы формата описаны в приложении.

**Листинг 9.8.** Пример структуры

```
using System;
namespace ConsoleApplication1
{
    struct Complex
    {
        public double re, im;

        public Complex( double re_, double im_ )
        {
            re = re_; im = im_;        // можно использовать this.re, this.im
        }

        public static Complex operator + ( Complex a, Complex b )
        {
            return new Complex( a.re + b.re, a.im + b.im );
        }

        public override string ToString()
        {
            return ( string.Format( "({0,2:0.##};{1,2:0.##})", re, im ) );
        }
    }

    class Class1
    {   static void Main()
        {
            Complex a = new Complex( 1.2345, 5.6 );
```

продолжение ⇨

[1] К статическим полям это ограничение не относится.

**Листинг 9.8** (*продолжение*)

```
            Console.WriteLine( "a = " + a );

            Complex b;
            b.re = 10; b.im = 1;
            Console.WriteLine( "b = " + b );

            Complex c = new Complex();
            Console.WriteLine( "c = " + c );

            c = a + b;
            Console.WriteLine( "c = " + c );
        }
    }
}
```

Результат работы программы:

```
a = (1,23;5,6)
b = (10; 1)
c = ( 0; 0)
c = (11,23;6,6)
```

При выводе экземпляра структуры на консоль выполняется *упаковка*, то есть неявное преобразование в ссылочный тип. Упаковка (это понятие было введено в разделе «Упаковка и распаковка», см. с. 36) применяется и в других случаях, когда структурный тип используется там, где ожидается ссылочный, например, при преобразовании экземпляра структуры к типу реализуемого ею интерфейса. При обратном преобразовании — из ссылочного типа в структурный — выполняется *распаковка*.

*Присваивание структур* имеет, что естественно, значимую семантику, то есть при присваивании создается копия значений полей. То же самое происходит и при передаче структур в качестве параметров по значению. Для экономии ресурсов ничто не мешает передавать структуры в методы по ссылке с помощью ключевых слов `ref` или `out`.

Особенно значительный выигрыш в эффективности можно получить, используя массивы структур вместо массивов классов. Например, для массива из 100 экземпляров класса создается 101 объект, а для массива структур — один объект. Пример работы с массивом структур, описанных в предыдущем листинге:

```
Complex [] mas = new Complex[4];

for ( int i = 0; i < 4; ++i )
{
    mas[i].re = i;
    mas[i].im = 2 * i;
}

foreach ( Complex elem in mas ) Console.WriteLine( elem );
```

Если поместить этот фрагмент вместо тела метода Main в листинге 9.5, получим следующий результат:

```
( 0; 0)
( 1; 2)
( 2; 4)
( 3; 6)
```

# Перечисления

При написании программ часто возникает потребность определить несколько связанных между собой именованных констант, при этом их конкретные значения могут быть не важны. Для этого удобно воспользоваться перечисляемым типом данных, все возможные значения которого задаются списком целочисленных констант, например:

```
enum Menu { Read, Write, Append, Exit }
enum Радуга { Красный, Оранжевый, Желтый, Зеленый, Синий, Фиолетовый }
```

Для каждой константы задается ее символическое имя. По умолчанию константам присваиваются последовательные значения типа int, начиная с 0, но можно задать и собственные значения, например:

```
enum Nums { two = 2, three, four, ten = 10, eleven, fifty = ten + 40 };
```

Константам three и four присваиваются значения 3 и 4, константе eleven — 11. Имена перечисляемых констант внутри каждого перечисления должны быть уникальными, а значения могут совпадать.

Преимущество перечисления перед описанием именованных констант состоит в том, что связанные константы нагляднее; кроме того, компилятор выполняет проверку типов, а интегрированная среда разработки подсказывает возможные значения констант, выводя их список.

Синтаксис перечисления:

```
[ атрибуты ] [ спецификаторы ] enum имя_перечисления [ : базовый_тип ]
    тело_перечисления [ ; ]
```

*Спецификаторы* перечисления имеют такой же смысл, как и для класса, причем допускаются только спецификаторы new, public, protected, internal и private.

*Базовый тип* — это тип элементов, из которых построено перечисление. По умолчанию используется тип int, но можно задать тип и явным образом, выбрав его среди целочисленных типов (кроме char), а именно: byte, sbyte, short, ushort, int, uint, long и ulong. Необходимость в этом возникает, когда значения констант невозможно или неудобно представлять с помощью типа int.

*Тело перечисления* состоит из имен констант, каждой из которых может быть присвоено значение. Если значение не указано, оно вычисляется прибавлением единицы к значению предыдущей константы. Константы по умолчанию имеют спецификатор доступа public.

Перечисления часто используются как вложенные типы, идентифицируя значения из какого-либо ограниченного набора. Пример такого перечисления приведен в листинге 9.9.

**Листинг 9.9.** Пример перечисления

```
using System;
namespace ConsoleApplication1
{
    struct Боец
    {
        public enum Воинское_Звание
        {
            Рядовой, Сержант, Майор, Генерал
        }

        public string Фамилия;
        public Воинское_Звание Звание;
    }
    class Class1
    {   static void Main()
        {
            Боец x;
            x.Фамилия = "Иванов";
            x.Звание = Боец.Воинское_Звание.Сержант;

            Console.WriteLine( x.Звание + " " + x.Фамилия );
        }
    }
}
```

Результат работы программы:

```
Сержант Иванов
```

Перечисления удобно использовать для представления битовых флагов, например:

```
enum Flags : byte
{
    b0, b1, b2, b3 = 0x04, b4 = 0x08, b5 = 0x10, b6 = 0x20, b7 = 0x40
}
```

## Операции с перечислениями

С переменными перечисляемого типа можно выполнять арифметические операции (+, -, ++, --), логические поразрядные операции (^, &, |, ~), сравнивать их с помощью операций отношения (<, <=, >, >=, ==, !=) и получать размер в байтах (`sizeof`).

При использовании переменных перечисляемого типа в целочисленных выражениях и операциях присваивания требуется явное *преобразование типа*. Переменной перечисляемого типа можно присвоить любое значение, представимое с помощью

базового типа, то есть не только одно из значений, входящих в тело перечисления. Присваиваемое значение становится новым элементом перечисления.

Пример:

```
Flags a = Flags.b2 | Flags.b4;
Console.WriteLine( "a = {0}   {0,2:X}", a );
++a;

Console.WriteLine( "a = {0}   {0,2:X}", a );
int x = (int) a;
Console.WriteLine( "x = {0}   {0,2:X}", x );

Flags b = (Flags) 65;
Console.WriteLine( "b = {0}   {0,2:X}", b );
```

Результат работы этого фрагмента программы (`{0,2:X}` обозначает шестнадцатеричный формат вывода):

```
a = 10    0A
a = 11    0B
x = 11     B
b = 65    41
```

Другой пример использования операций с перечислениями приведен в листинге 9.10.

**Листинг 9.10.** Операции с перечислениями

```
using System;
namespace ConsoleApplication1
{
    struct Боец
    {
        public enum Воинское_Звание
        {
            Рядовой, Сержант, Лейтенант, Майор, Полковник, Генерал
        }

        public string Фамилия;
        public Воинское_Звание Звание;
    }

    class Class1
    {   static void Main()
        {
            Боец x;
            x.Фамилия = "Иванов";
            x.Звание  = Боец.Воинское_Звание.Сержант;

            for ( int i = 1976; i < 2006; i += 5 )
            {
                if ( x.Звание < Боец.Воинское_Звание.Генерал ) ++x.Звание;
```

*продолжение* ⇨

**Листинг 9.10** (*продолжение*)

```
                Console.WriteLine( "Год: {0} {1} {2}",
                                   i, x.Звание, x.Фамилия );
            }
        }
    }
}
```

Результат работы программы:

```
Год: 1976 Лейтенант Иванов
Год: 1981 Майор Иванов
Год: 1986 Полковник Иванов
Год: 1991 Генерал Иванов
Год: 1996 Генерал Иванов
Год: 2001 Генерал Иванов
```

# Базовый класс System.Enum

Все перечисления в C# являются потомками базового класса System.Enum, который снабжает их некоторыми полезными методами.

Статический метод GetName позволяет получить символическое имя константы по ее номеру, например:

```
Console.WriteLine( Enum.GetName(typeof( Flags ), 8) );                  // b4
Console.WriteLine( Enum.GetName(typeof( Боец.Воинское_Звание ), 1) );  // Сержант
```

**ПРИМЕЧАНИЕ**

Операция typeof возвращает тип своего аргумента (см. раздел «Рефлексия» в главе 12).

Статические методы GetNames и GetValues формируют, соответственно, массивы имен и значений констант, составляющих перечисление, например:

```
Array names = Enum.GetNames( typeof(Flags) );
Console.WriteLine( "Количество элементов в перечислении: " + names.Length );
foreach ( string elem in names ) Console.Write( "  " + elem );

Array values = Enum.GetValues( typeof(Flags) );
foreach ( Flags elem in values ) Console.Write( "  " + (byte) elem );
```

Статический метод IsDefined возвращает значение true, если константа с заданным символическим именем описана в указанном перечислении, и false в противном случае, например:

```
if ( Enum.IsDefined( typeof( Flags ), "b5" ) )
     Console.WriteLine( "Константа с именем b5 существует" );
else Console.WriteLine( "Константа с именем b5 не существует" );
```

Статический метод GetUnderlyingType возвращает имя базового типа, на котором построено перечисление. Например, для перечисления Flags будет получено System.Byte:

```
Console.WriteLine( Enum.GetUnderlyingType( typeof(Flags) ) );
```

# Рекомендации по программированию

*Интерфейсы* чаще всего используются для задания общих свойств объектов различных иерархий. Основная идея интерфейса состоит в том, что к объектам классов, реализующих интерфейс, можно обращаться одинаковым образом, при этом каждый класс может определять элементы интерфейса по-своему.

Если некий набор действий имеет смысл только для какой-то конкретной иерархии классов, реализующих эти действия разными способами, уместнее задать этот набор в виде виртуальных методов абстрактного базового класса иерархии.

В C# поддерживается одиночное наследование для классов и множественное — для интерфейсов. Это позволяет придать производному классу свойства нескольких базовых интерфейсов. Класс должен определять все методы всех интерфейсов, которые имеются в списке его предков.

В библиотеке .NET определено большое количество стандартных интерфейсов. Реализация стандартных интерфейсов в собственных классах позволяет использовать для объектов этих классов стандартные средства языка и библиотеки.

Например, для обеспечения возможности сортировки объектов стандартными методами следует реализовать в соответствующем классе интерфейсы `IComparable` или `IComparer`. Реализация интерфейсов `IEnumerable` и `IEnumerator` дает возможность просматривать содержимое объекта с помощью конструкции `foreach`, а реализация интерфейса `ICloneable` — клонировать объекты.

Использование *итераторов* упрощает организацию перебора элементов и позволяет задать для одного и того же класса различные стратегии перебора.

Область применения *структур* — типы данных, имеющие небольшое количество полей, с которыми удобнее работать как со значениями, а не как со ссылками. Накладные расходы на динамическое выделение памяти для экземплятров небольших классов могут весьма значительно снизить быстродействие программы, поэтому их эффективнее описывать как структуры.

Преимущество использования *перечислений* для описания связанных между собой значений состоит в том, что это более наглядно и инкапсулировано, чем россыпь именованных констант. Кроме того, компилятор выполняет проверку типов, а интегрированная среда разработки подсказывает возможные значения констант, выводя их список.