Лекция 7. Контейнерные классы

Функциональный тип Делегаты. Делегаты как свойства

Структуры

- Синтаксис структуры:

[атрибуты][спецификаторы] struct имя_структуры
[: интерфейсы] { тело_структуры }

Структуре запрещено:

- определять конструктор по умолчанию, поскольку он определен неявно и присваивает всем своим элементам значения по умолчанию (нули соответствующего типа);
- определять деструктор, поскольку это бессмысленно.

- Спецификаторы структуры имеют такой же смысл, как и для класса. Однако из спецификаторов доступа допускается использовать только **public**, i**nternal** и для вложенных структур еще и **private**.
- Структуры не могут быть абстрактными, поскольку по умолчанию они бесплодны.
- Интерфейсы, реализуемые структурой, перечисляются через запятую.
- Тело структуры может содержать: константы, поля, конструкторы, методы, свойства, индексаторы, операторные методы, вложенные типы и события.
- При описании структуры задавать значение по умолчанию можно только для статических полей. Остальным полям с помощью конструктора по умолчанию будут присвоены нули для полей размерных типов и null для полей ссылочных типов.
- Параметр this интерпретируется как значение, поэтому его можно использовать для ссылок, но не для присваивания.

```
namespace ConsoleApplication1
{    struct SPoint : IComparable
    {   public int x, y;
        public SPoint(int x, int y)
        {           this.x = x; this.y = y;        }
        public double Dlina() //метод
        {           return Math.Sqrt(x * x + y * y);        }
        public override string ToString()
        {           return "(" + x.ToString() + ", " + y.ToString() + ")";        }
        public int CompareTo(object obj)
        {   SPoint b = (SPoint)obj;
            if (this.Dlina() == b.Dlina()) return 0;
            else if (this.Dlina() > b.Dlina()) return 1;
            else return -1;        }
        public static bool operator ==(SPoint a, SPoint b)
        {           return (a.CompareTo(b) == 0);        }
        public static bool operator !=(SPoint a, SPoint b)
        {           return (a.CompareTo(b) != 0);        }    }
    class Program
    {     static void Main()
        {    //создание и заполнение массива структур
            SPoint[] a = new SPoint[4];
            a[0] = new SPoint(-3, 0);
            a[1] = new SPoint(-0, 3);
            a[2] = new SPoint(3, 4);
            a[3] = new SPoint(0, 1);
            //сравнение двух структур
            if (a[0] == a[1])  Console.WriteLine("точки {0} и {1} равноудалены от начала координат\n",
                               a[0].ToString(), a[1].ToString());
            else Console.WriteLine("точки {0} и {1} не равноудалены от начала координат\n",
                           a[0].ToString(), a[1].ToString());
            Array.Sort(a); //сортировка массива структур просмотр массива структур
            foreach (SPoint x in a)
            {   Console.WriteLine("Точка: " + x.ToString());
                Console.WriteLine("удалена от начала координат на расстояние равное: " + x.Dlina());
```

```
file:///C:/Documents and Settings/Шолпан.COM/Мои документы/Visual
точки (-3, 0) и (0, 3) равноудалены от начала координат

Точка: (0, 1)
удалена от начала координат на расстояние равное: 1
```

Что использовать - класс или структуру?

- Если создаваемые типы данных содержат небольшое количество полей, с которыми удобнее работать как со значениями, а не как со ссылками, то расходы на выделение динамической памяти для небольших объектов снизят быстродействие программы, поэтому такие типы данных эффективнее реализовывать через структуры.
- Во всех остальных случаях эффективнее использовать классы. Однако передача структуры в методы по значению потребует и дополнительного времени, и дополнительной памяти для создания копии. В таких случаях эффективнее использовать классы.

Абстрактные структуры данных

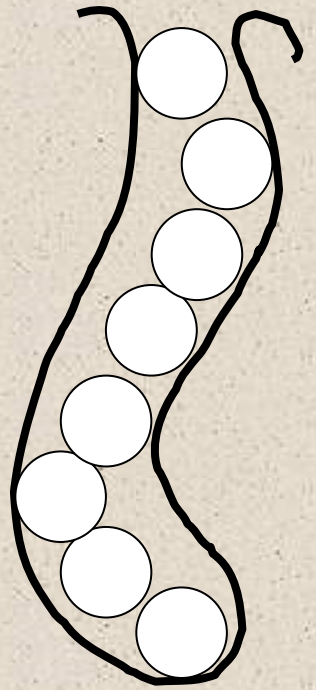
- *Массив* —конечная совокупность однотипных величин. Занимает непрерывную область памяти и предоставляет прямой (произвольный) доступ к элементам по индексу. Память под массив выделяется до начала работы с ним и впоследствии не изменяется.
- В *списке* каждый элемент связан со следующим и, возможно, с предыдущим. Количество элементов в списке может изменяться в процессе работы программы. Каждый элемент списка содержит *ключ*, идентифицирующий этот элемент.
  - *односвязный*, *двусвязный*
  - *кольцевой*
- *Хеш-таблица (ассоциативный массив*, *словарь)* — массив, доступ к элементам которого осуществляется не по номеру, а по ключу (т.е. это таблица, состоящая из пар «ключ-значение»)

Стек

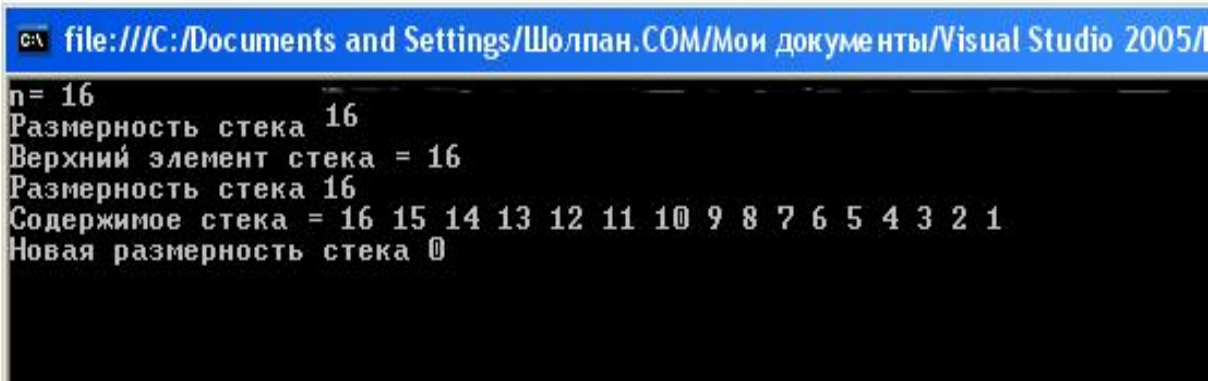
Стек — частный случай однонаправленного списка, добавление элементов в который и выборка из которого выполняются с одного конца, называемого вершиной стека.

Другие операции со стеком не определены.

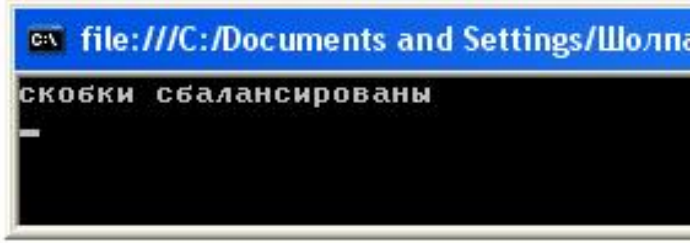
При выборке элемент исключается из стека.



```
using System;
using System.Collections;

namespace ConsoleApplication
{
    class Program
    {
        public static void Main()
        {
            Console.Write("n= ");
            int n = int.Parse(Console.ReadLine());
            Stack intStack = new Stack();
            for (int i = 1; i <= n; i++)
                intStack.Push(i);
            Console.WriteLine("Размерность стека " + intStack.Count);

            Console.WriteLine("Верхний элемент стека = " + intStack.Peek());
            Console.WriteLine("Размерность стека " + intStack.Count);

            Console.Write("Содержимое стека = ");
            while (intStack.Count != 0)
                Console.Write("{0} ", intStack.Pop());
            Console.WriteLine("\nНовая размерность стека " + intStack.Count);
            Console.ReadLine();
        }
    }
}
```

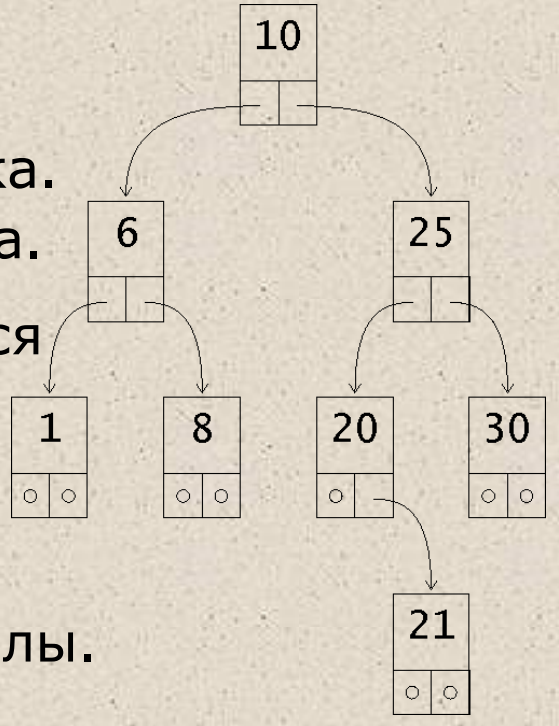


```
using System;
using System.Collections;
using System.IO;

namespace MyProgram
{    class Program
     {        public static void Main()
   {   StreamReader fileIn=new StreamReader("d:\\TextFile1.txt");
    string line=fileIn.ReadToEnd();
    fileIn.Close();
    Stack skobki=new Stack();
    bool flag=true;
 //проверяем баланс скобок
    for ( int i=0; i<line.Length;i++)
    {    //если текущий символ открывающаяся скобка, то помещаем ее в стек
     if (line[i] == '(') skobki.Push(i);
      else if (line[i] == ')') //если текущий символ закрывающаяся скобка, то
     {    //если стек пустой, то для закрывающейся скобки не хватает парной открывающейся
     if (skobki.Count == 0)
      { flag = false; Console.WriteLine("Возможно в позиции " + i + "лишняя ) скобка"); }
       else skobki.Pop(); //иначе извлекаем парную скобку
      }   }
    //если после просмотра строки стек оказался пустым, то скобки сбалансированы
    if (skobki.Count == 0) { if (flag)Console.WriteLine("скобки сбалансированы"); }
    else //иначе баланс скобок нарушен
    {    Console.Write("Возможно лишняя ( скобка в позиции:");
     while (skobki.Count != 0)
     {     Console.Write("{0} ", (int)skobki.Pop());    }
     Console.WriteLine();
    } Console.ReadLine();
   }
     }
}
```



- *Очередь* — частный случай однонаправленного списка, добавление элементов в который выполняется в один конец, а выборка — из другого конца. Другие операции с очередью не определены. При выборке элемент исключается из очереди.

- *Бинарное дерево* — динамическая структура данных, состоящая из узлов, каждый из которых содержит, помимо данных, не более двух ссылок на различные бинарные поддеревья.
- На каждый узел имеется ровно одна ссылка. Начальный узел называется *корнем* дерева.
- Узел, не имеющий поддеревьев, называется *листом*. Исходящие узлы называются *предками*, входящие — *потомками*.
- *Высота дерева* определяется количеством уровней, на которых располагаются его узлы.

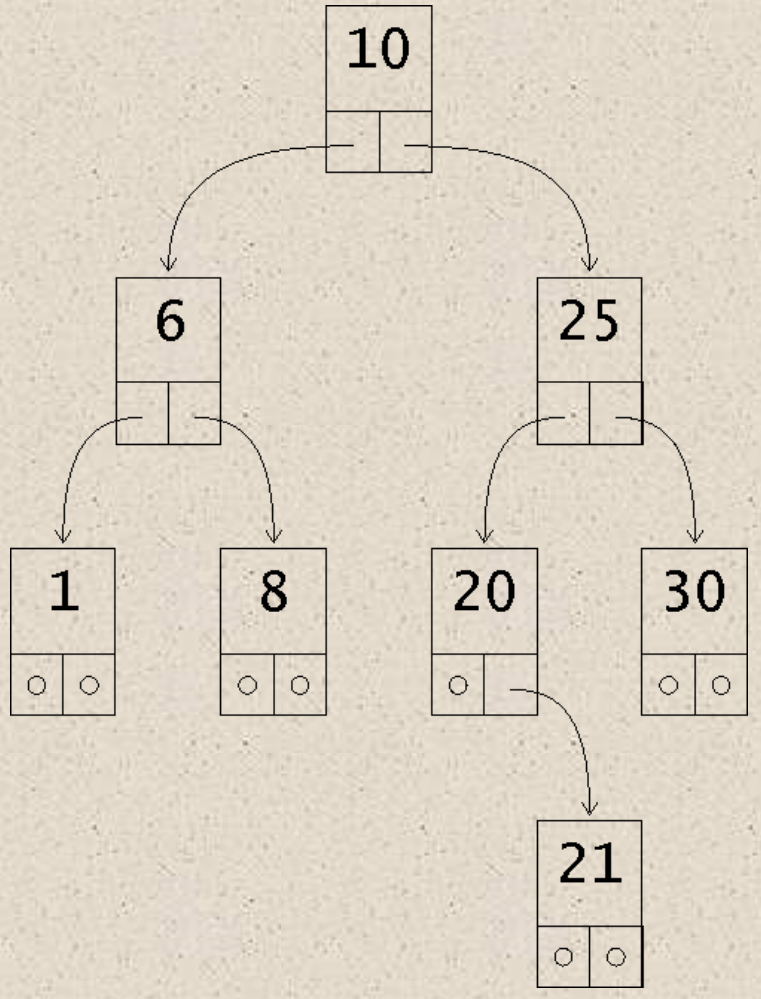


```
using System;
using System.Collections;
namespace MyProgram
{
    class Program
    {
        public static void Main()
        {
            Console.Write("n= ");
            int n = int.Parse(Console.ReadLine());
            Queue intQ = new Queue();
            for (int i = 1; i <= n; i++)
                intQ.Enqueue(i);
            Console.WriteLine("Размерность очереди " + intQ.Count);

            Console.WriteLine("Верхний элемент очереди = " + intQ.Peek());
            Console.WriteLine("Размерность очереди " + intQ.Count);

            Console.Write("Содержимое очереди = ");
            while (intQ.Count != 0)
                Console.Write("{0} ", intQ.Dequeue());
            Console.WriteLine("\nНовая размерность очереди " + intQ.Count);
            Console.ReadLine();
        }
    }
}
```

```
file:///C:/Documents and Settings/Шолпан.COM/Мои документы/Visual Studio 2005/Projec...
n= 20
Размерность очереди 20
Верхний элемент очереди = 1
Размерность очереди 20
Содержимое очереди = 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
Новая размерность очереди 0
```

Дерево поиска

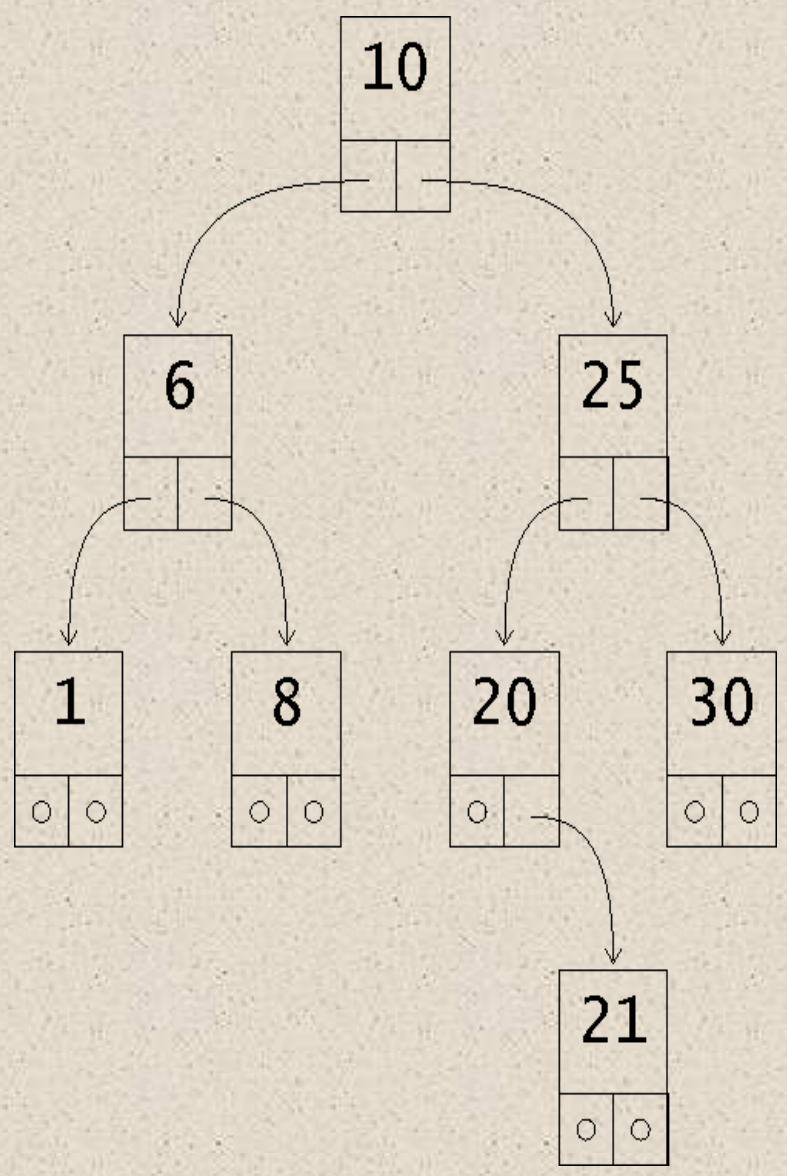
- Если дерево организовано таким образом, что для каждого узла все ключи его левого поддерева меньше ключа этого узла, а все ключи его правого поддерева — больше, оно называется *деревом поиска*. Одинаковые ключи не допускаются. В дереве поиска можно найти элемент по ключу, двигаясь от корня и переходя на левое или правое поддерево в зависимости от значения ключа в каждом узле.



Обход дерева

```
procedure print_tree( дерево );
begin
    print_tree( левое_поддерево )
    посещение корня
    print_tree( правое_поддерево )
end;
```

1 6 8 10 20 21 25 30



- *Граф* — совокупность узлов и ребер, соединяющих различные узлы. Множество реальных практических задач можно описать в терминах графов, что делает их структурой данных, часто используемой при написании программ.
- *Множество* — неупорядоченная совокупность элементов. Для множеств определены операции:
  - проверки принадлежности элемента множеству
  - включения и исключения элемента
  - объединения, пересечения и вычитания множеств.
- Все эти структуры данных называются *абстрактными*, поскольку в них не задается реализация допустимых операций.

Контейнеры

- *Контейнер (коллекция)* - стандартный класс, реализующий абстрактную структуру данных.
- Для каждого типа коллекции определены методы работы с ее элементами, не зависящие от конкретного типа хранимых данных.
- Использование коллекций позволяет сократить сроки разработки программ и повысить их надежность.
- Каждый вид коллекции поддерживает свой набор операций над данными, и быстродействие этих операций может быть разным.
- Выбор вида коллекции зависит от того, что требуется делать с данными в программе и какие требования предъявляются к ее быстродействию.
- В библиотеке .NET определено множество стандартных контейнеров.
- Основные пространства имен, в которых они описаны — System.Collections, System.Collections.Specialized и System.Collections.Generic

Пространство имен System.Collections

ArrayList	Массив, динамически изменяющий свой размер
BitArray	Компактный массив для хранения битовых значений
Hashtable	Хэш-таблица
Queue	Очередь
SortedList	Коллекция, отсортированная по ключам. Доступ к элементам — по ключу или по индексу
Stack	Стек

Пример использования класса **ArrayList**

ConsoleApplication9.Program — Main(string[] args)

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Collections;

namespace ConsoleApplication9
{
    class Program
    {
        static void Main(string[] args)
        {
            ArrayList arr1 = new ArrayList();          // создается массив из 16   элементов
            ArrayList arr2 = new ArrayList(1000);  // создается массив из 1000 элементов
            ArrayList arr3 = new ArrayList();
            arr3.Capacity = 1000;                          // количество элементов задается

            arr1.Add(123); arr1.Add(-2); arr1.Add("Вася");
            int a = (int)arr1[0];
            int b = (int)arr1[1];
            string s = (string)arr1[2];

            Console.WriteLine("a={0}    b={1}    c={2}", a, b, s);
            Console.ReadLine();

        }
    }
}
```

```
file:///C:/Documents and Settings/Шолпан.COM/Мои документы/Visu
a=123    b=-2    c=Вася
```

Основные элементы ArrayList

Capacity	**Количество элементов в массиве**
Count	**Фактическое количество элементов массива**
Item	**Получить или установить значение элемента по заданному индексу**
Add	**Добавление элемента в конец массива**
Clear	**Удаление всех элементов массива**
CopyTo	**Копирование всех элементов в одномерный массив**
Insert	**Вставка элемента в заданную позицию**
Remove	**Удаление первого вхождения элемента**
Reverse	**Изменение порядка следования элементов на обратный**
Sort	**Упорядочивание элементов массива**

Делегаты

Определение делегата

- *Делегат* — это вид класса, предназначенный для хранения ссылок на методы. Делегат, как и любой другой класс, можно передать в качестве параметра, а затем вызвать инкапсулированный в нем метод.
- Делегаты используются для поддержки событий, а также как самостоятельная конструкция языка.
- Описание делегата задает сигнатуру методов, которые могут быть вызваны с его помощью:

[атрибуты] [спецификаторы] delegate тип имя([параметры])

Пример описания делегата:

```
public delegate void D ( int i );
```

- Базовым классом делегата является класс System.Delegate

Использование делегатов

- Делегаты применяются в основном для следующих целей:
  - получения возможности определять вызываемый метод не при компиляции, а динамически во время выполнения программы;
  - обеспечения связи между объектами по типу «источник — наблюдатель»;
  - создания универсальных методов, в которые можно передавать другие методы (поддержки механизма обратных вызовов).

Использование делегата имеет тот же синтаксис, что и вызов метода. Если делегат хранит ссылки на несколько методов, они вызываются последовательно в том порядке, в котором были добавлены в делегат.

ConsoleApplication1.Class1 | Main()

```
using System;
namespace ConsoleApplication1
{
    delegate void Del(ref string s);                // объявление делегата
     class Class1
    {     public static void COO1(ref string s)                    // метод 1
        {  string temp = "";
            for (int i = 0; i < s.Length; ++i)
            {  if (s[i] == 'o' || s[i] == 'O') temp += '0';
                else if (s[i] == 'l') temp += '1';
                else temp += s[i];                }
            s = temp;         }

        public static void Hack(ref string s)                    // метод 2
        {   string temp = "";
            for (int i = 0; i < s.Length; ++i)
                if (i / 2 * 2 == i) temp += char.ToUpper(s[i]);
                else temp += s[i];
                s = temp;         }

        static void Main()
        {   string s = "cool hackers";
            Del d;                                      // экземпляр делегата
            for (int i = 0; i < 2; ++i)
            {   d = new Del(COO1);                  // инициализация методом 1
                if (i == 1) d = new Del(Hack);   // инициализация методом 2

                d(ref s);     // использование делегата для вызова методов
                Console.WriteLine(s);
                Console.ReadLine();
            }
        }
    }
}
```

```
file:///C:/Documents and Settings/Шолпан.COM
c001 hackers
C001 hAcKeRs
_
```

При вызове последовательности методов с помощью делегата необходимо учитывать следующее:

- сигнатура методов должна в точности соответствовать делегату;
- методы могут быть как статическими, так и обычными методами класса;
- каждому методу в списке передается один и тот же набор параметров;
- если параметр передается по ссылке, изменения параметра в одном методе отразятся на его значении при вызове следующего метода;
- если параметр передается с ключевым словом out или метод возвращает значение, результатом выполнения делегата является значение, сформированное последним из методов списка (в связи с этим рекомендуется формировать списки только из делегатов, имеющих возвращаемое значение типа void);
- если в процессе работы метода возникло исключение, не обработанное в том же методе, последующие методы в списке не выполняются, а происходит поиск обработчиков в объемлющих делегат блоках;
- попытка вызвать делегат, в списке которого нет ни одного метода, вызывает генерацию исключения System.NullReferenceException.

Оповещение наблюдателей с помощью делегата

ConsoleApplication1.Class1 | Main()

```
using System;
namespace ConsoleApplication1
{    public delegate void Del(object o);              // объявление делегата
     class Subj                                       // класс-источник
    {    Del dels;                                 // объявление экземпляра делегата
         public void Register(Del d)                  // регистрация делегата
        {  dels += d;          }

        public void OOPS()                            // что-то произошло
        {  Console.WriteLine("ОЙ!");
            if (dels != null) dels(this);      // оповещение наблюдателей
        }     }

    class ObsA                                        // класс-наблюдатель
    {    public void Do(object o)                 // реакция на событие источника
        {             Console.WriteLine("Бедняжка!");          }     }

    class ObsB                                        // класс-наблюдатель
    {
        public static void See(object o)    // реакция на событие источника
        {         Console.WriteLine("Да ну, ерунда!");          }     }

    class Class1
    {   static void Main()
        {   Subj s = new Subj();                 //  объект класса-источника
            ObsA o1 = new ObsA();                 //                    объекты
            ObsA o2 = new ObsA();                 //        класса-наблюдателя
            s.Register(new Del(o1.Do));       //       регистрация методов
            s.Register(new Del(o2.Do));       // наблюдателей в источнике
            s.Register(new Del(ObsB.See));    //   ( экземпляры делегата )
            s.OOPS();                             //     инициирование события
            Console.ReadLine();
        }
    }
}
```

```
file:///C:/Documents an
ОЙ!
Бедняжка!
Бедняжка!
Да ну, ерунда!
```

- В источнике объявляется экземпляр делегата, в этот экземпляр заносятся методы тех объектов, которые хотят получать уведомление об изменении состояния источника. Этот процесс называется регистрацией делегатов. При регистрации имя метода добавляется к списку. При наступлении "часа X" все зарегистрированные методы поочередно вызываются через делегат.
- Для обеспечения обратной связи между наблюдателем и источником делегат объявлен с параметром типа object, через который в вызываемый метод передается ссылка на вызывающий объект. Следовательно, в вызываемом методе можно получать информацию о состоянии вызывающего объекта и посылать ему сообщения.
- Связь "источник — наблюдатель" устанавливается во время выполнения программы для каждого объекта по отдельности. Если наблюдатель больше не хочет получать уведомления от источника, можно удалить соответствующий метод из списка делегата с помощью метода Remove или перегруженной операции вычитания, например:
- public void UnRegister(Del d) // удаление делегата { dels -= d; }

Операции

- Делегаты можно *сравнивать на равенство и неравенство*. Два делегата равны, если они оба не содержат ссылок на методы или если они содержат ссылки на одни и те же методы в одном и том же порядке.
- С делегатами одного типа можно *выполнять операции* простого и сложного *присваивания*.
- Делегат, как и строка string, является неизменяемым типом данных, поэтому при любом изменении создается новый экземпляр, а старый впоследствии удаляется сборщиком мусора.

- Использование делегата имеет тот же синтаксис, что и вызов метода. Если делегат хранит ссылки на несколько методов, они вызываются последовательно в том порядке, в котором были добавлены в делегат.

Операции над делегатами

- С делегатами одного типа можно выполнять операции простого и сложного присваивания, например:

```
Del d1 = new Del( o1.Do ); // o1.Do
Del d2 = new Del( o2.Do ); // o2.Do
Del d3 = d1 + d2; // o1.Do и o2.Do
d3 += d1; // o1.Do, o2.Do и o1.Do
d3 -= d2; // o1.Do и o1.Do
```

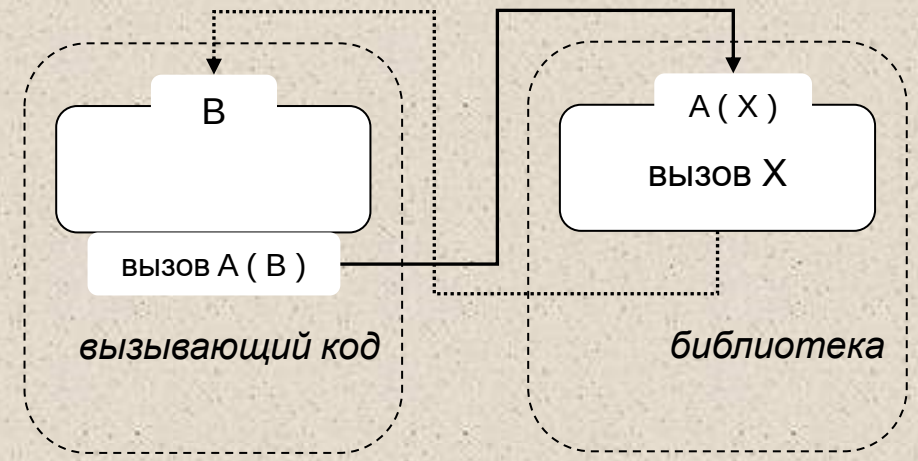
Передача делегата через список параметров

ConsoleApplication1.Class1 | Main()

```
using System;
namespace ConsoleApplication1
{
    public delegate double Fun(double x);           // объявление делегата

    class Class1
    {
        public static void Table(Fun F, double x, double b)
        {
            Console.WriteLine(" ----- X ----- Y -----");
            while (x <= b)
            {
                Console.WriteLine("| {0,8:0.000} | {1,8:0.000} |", x, F(x));
                x += 1;
            }
            Console.WriteLine(" ----------------------");
        }

        public static double Simple(double x)
        {
            return 1;
        }

        static void Main()
        {
            Console.WriteLine(" Таблица функции Sin ");
            Table(new Fun(Math.Sin), -2, 2);

            Console.WriteLine(" Таблица функции Simple ");
            Table(new Fun(Simple), 0, 3);
        }
    }
}
```

```
C:\WINDOWS\system32\cmd.exe
 Таблица функции Sin
 ----- X ----- Y -----
|   -2,000 |   -0,909 |
|   -1,000 |   -0,841 |
|    0,000 |    0,000 |
|    1,000 |    0,841 |
|    2,000 |    0,909 |
 ----------------------
 Таблица функции Simple
 ----- X ----- Y -----
|    0,000 |    1,000 |
|    1,000 |    1,000 |
|    2,000 |    1,000 |
|    3,000 |    1,000 |
 ----------------------
Для продолжения нажмите любую клавишу . . .
```

Обратный вызов (callback)

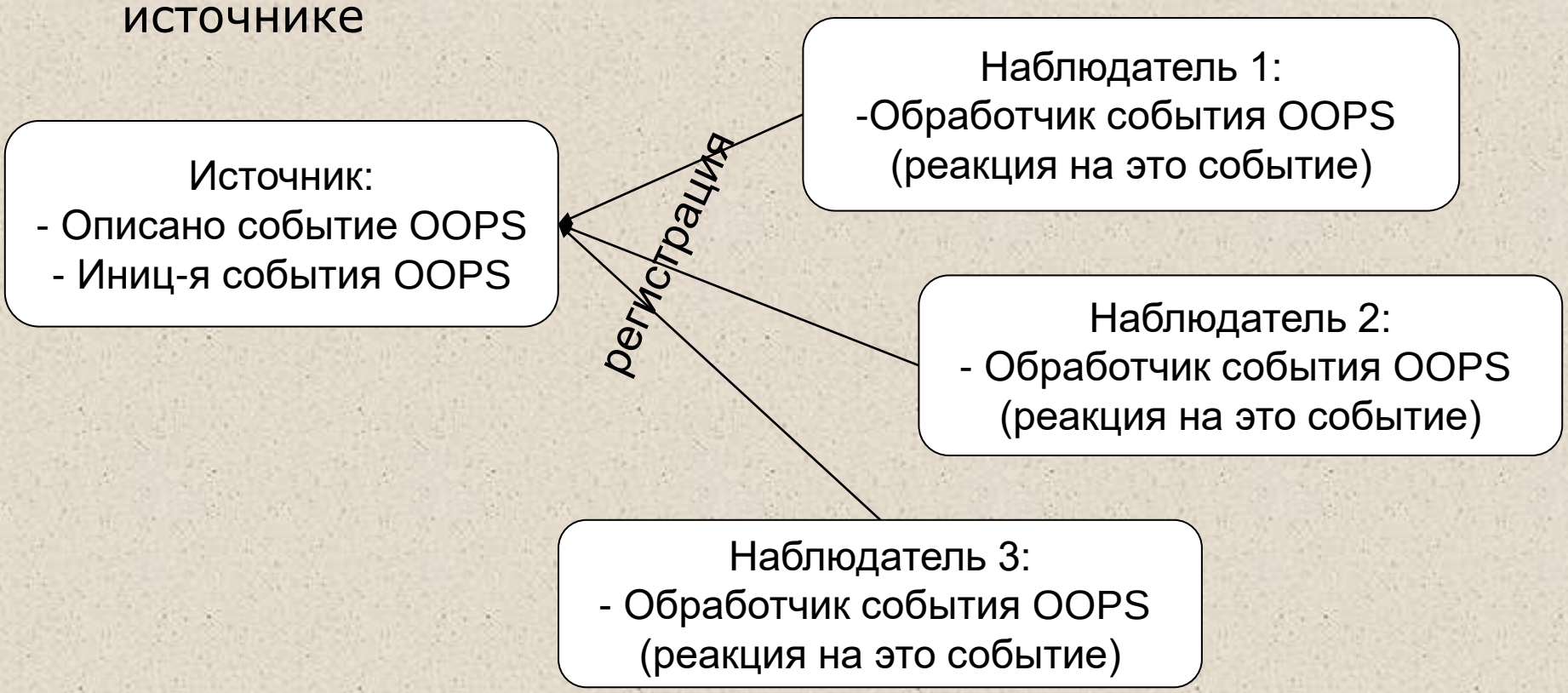


- Обратный вызов (callback) представляет собой вызов функции, передаваемой в другую функцию в качестве параметра. Допустим, в библиотеке описана функция A, параметром которой является имя другой функции. В вызывающем коде описывается функция с требуемой сигнатурой (B) и передается в функцию A. Выполнение функции A приводит к вызову B, то есть управление передается из библиотечной функции обратно в вызывающий код.

События

Определение события

- *Событие* — элемент класса, позволяющий ему посылать другим объектам (наблюдателям) уведомления об изменении своего состояния.
- Чтобы стать наблюдателем, объект должен иметь обработчик события и зарегистрировать его в объекте-источнике



Механизм событий

- События построены на основе делегатов: с помощью делегатов вызываются методы-обработчики событий. Поэтому *создание события* в классе состоит из следующих частей:
  - описание делегата, задающего сигнатуру обработчиков событий;
  - описание события;
  - описание метода (методов), инициирующих событие.
- Синтаксис события:

[атрибуты] [спецификаторы] event тип имя

Пример

```
public delegate void Del( object o );     // объявление делегата
class A
{
    public event Del Oops;                 // объявление события
    ...
}
```

- *Обработка событий* выполняется в классах-получателях сообщения. Для этого в них описываются методы-обработчики событий, сигнатура которых соответствует типу делегата. Каждый объект (не класс!), желающий получать сообщение, должен зарегистрировать в объекте-отправителе этот метод.
- Событие — это удобная абстракция для программиста. На самом деле оно состоит из закрытого статического класса, в котором создается экземпляр делегата, и двух методов, предназначенных для добавления и удаления обработчика из списка этого делегата.
- Внешний код может работать с событиями единственным образом: добавлять обработчики в список или удалять их, поскольку вне класса могут использоваться только операции += и -=. Тип результата этих операций — void, в отличие от операций сложного присваивания для арифметических типов. Иного способа доступа к списку обработчиков нет.

Оповещение наблюдателей с помощью событий

ConsoleApplication1.Class1 | Main()

```
using System;
namespace ConsoleApplication1
{    public delegate void Del();                          // объявление делегата

    class Subj                                             // класс-источник
    {   public event Del Oops;                             // объявление события
        public void CryOops()                          // метод, инициирующий событие
        {    Console.WriteLine( "ОЙ!" );
            if ( Oops != null ) Oops();          }     }

    class ObsA                                             // класс-наблюдатель
    {    public void Do()                             // реакция на событие источника
        {   Console.WriteLine("Бедняжка!" );             }     }

    class ObsB                                             // класс-наблюдатель
    {    public static void See()                     // реакция на событие источника
        {              Console.WriteLine( "Да ну, ерунда!" );          }     }

    class Class1
    {      static void Main()
        {
            Subj s  = new Subj();                  //     объект класса-источника

            ObsA o1 = new ObsA();                  //                       объекты
            ObsA o2 = new ObsA();                  //           класса-наблюдателя

            s.Oops += new Del( o1.Do );            //                    добавление
            s.Oops += new Del( o2.Do );            //                  обработчиков
            s.Oops += new Del( ObsB.See );         //                     к событию

            s.CryOops();                           //       инициирование события
            Console.ReadLine();
        }
    }
}
```

```
file:///C:/Documents and Settings/Шолпан.C
ОЙ!
Бедняжка!
Бедняжка!
Да ну, ерунда!
```

Еще немного про делегаты и события

- Делегат можно вызвать асинхронно (в отдельном потоке), при этом в исходном потоке можно продолжать вычисления.
- Анонимный делегат (без создания класса-наблюдателя):

```
s.Oops += delegate ( object sender, EventArgs e )
                { Console.WriteLine( "Я с вами!" ); };
```

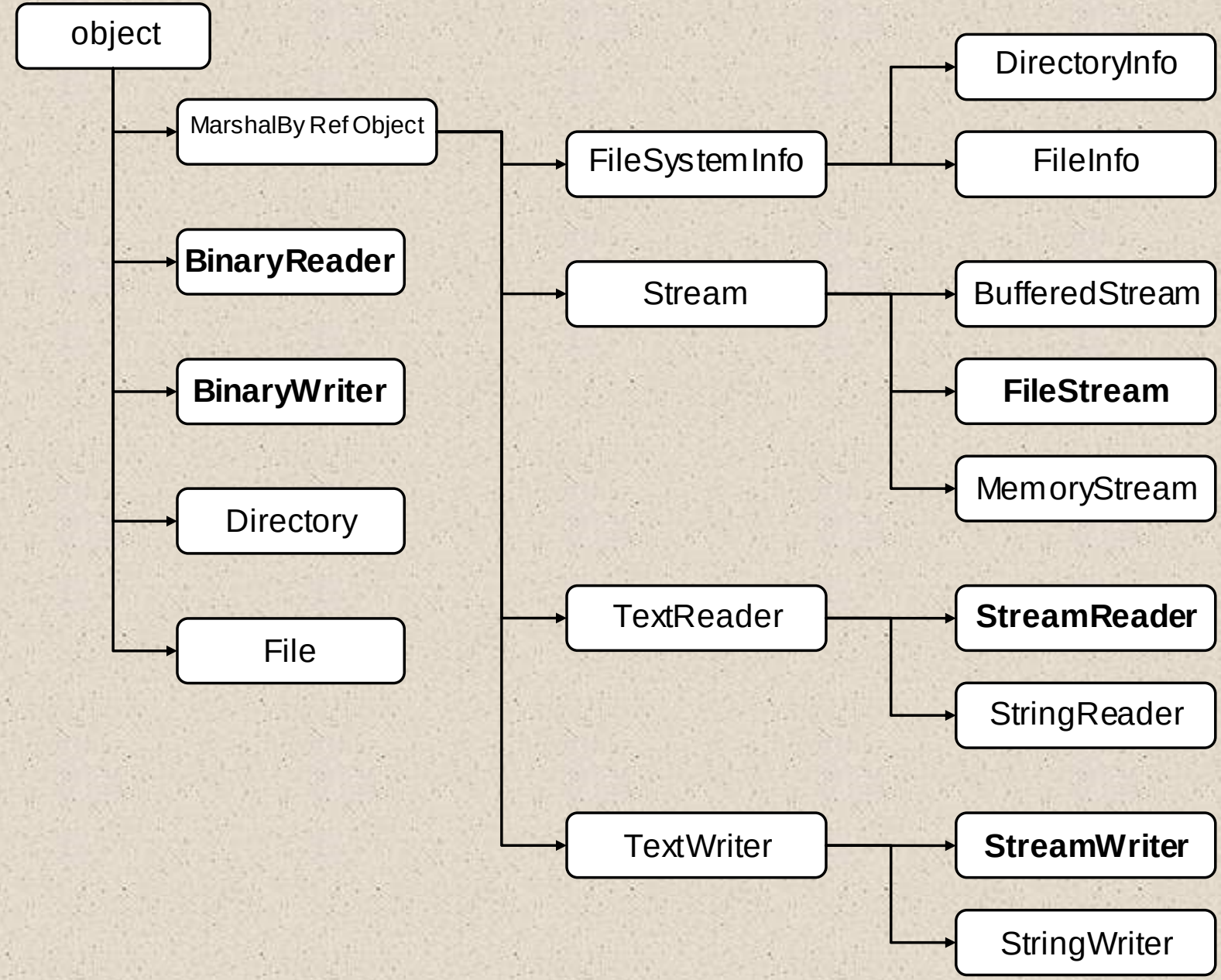
- Делегаты и события обеспечивают гибкое взаимодействие взаимосвязанных объектов, позволяющее поддерживать их согласованное состояние.
- События включены во многие стандартные классы .NET, например, в классы пространства имен Windows.Forms, используемые для разработки Windows-приложений.

Работа с файлами

Общие принципы работы с файлами

- *Чтение (ввод)* — передача данных с внешнего устройства в оперативную память, обратный процесс — *запись (вывод)*.
- Ввод-вывод в C# выполняется с помощью подсистемы ввода-вывода и классов библиотеки .NET. Обмен данными реализуется с помощью потоков.
- *Поток* (stream) — абстрактное понятие, относящееся к любому переносу данных от источника к приемнику. Потоки обеспечивают надежную работу как со стандартными, так и с определенными пользователем типами данных, а также единообразный и понятный синтаксис.
- Поток определяется как последовательность байтов и не зависит от конкретного устройства, с которым производится обмен.
- Обмен с потоком для повышения скорости передачи данных производится, как правило, через *буфер*. Буфер выделяется для каждого открытого файла.

Классы .NET для работы с потоками



Уровни обмена с внешними устройствами

Выполнять обмен с внешними устройствами можно на уровне:

- *двоичного представления данных*
  - (BinaryReader, BinaryWriter);
- *байтов*
  - (FileStream);
- *текста*, то есть символов
  - (StreamWriter, StreamReader).

Доступ к файлам

- *Доступ* к файлам может быть *последовательным*, когда очередной элемент можно прочитать (записать) только после аналогичной операции с предыдущим элементом, и *произвольным*, или *прямым*, при котором выполняется чтение (запись) произвольного элемента по заданному адресу.
- Текстовые файлы позволяют выполнять только последовательный доступ, в двоичных и байтовых потоках можно использовать оба метода.
- Прямой доступ в сочетании с отсутствием преобразований обеспечивает высокую скорость получения нужной информации.

Пример чтения из текстового файла

```
static void Main()   // весь файл -> в одну строку
{
        try
        {
            StreamReader f = new StreamReader( "text.txt" );
            string s = f.ReadToEnd();
            Console.WriteLine(s);
            f.Close();
        }
        catch( FileNotFoundException e )
        {
            Console.WriteLine( e.Message );
            Console.WriteLine( " Проверьте правильность имени файла!" );
            return;
        }
        ...
}
```

Построчное чтение текстового файла

```
StreamReader f = new StreamReader( "text.txt" );
string s;
long i = 0;

while ( ( s = f.ReadLine() ) != null )
    Console.WriteLine( "{0}: {1}", ++i, s );
f.Close();
```

Чтение чисел из текстового файла - 1

```
try {
        List<int> a = new List<int>();
        StreamReader file_in = new StreamReader(@"D:\FILES\1024");
        Regex regex = new Regex("[^0-9-+]+");
        List<string> list = new List<string>(
                        regex.Split(file_in.ReadToEnd().TrimStart(' ')));
        foreach (string temp in list) a.Add(Convert.ToInt32(temp));
        foreach (int temp in a) Console.WriteLine(temp);
        ...
}
catch (FileNotFoundException e)
        {  Console.WriteLine("Нет файла" + e.Message); return;
        }
catch (FormatException e)
        {  Console.WriteLine(e.Message); return;
        }
```

Чтение чисел из текстового файла - 2

```
try {
        StreamReader file_in = new StreamReader(@"D:\FILES\1024");
        char[] delim = new char[] { ' ' };
        List<string> list = new List<string>(
                file_in.ReadToEnd().Split(delim,
                        StringSplitOptions.RemoveEmptyEntries));
        List<int> a = list.ConvertAll<int>(string2int);
        foreach (int temp in a) Console.WriteLine(temp);
        ...
}
catch (FileNotFoundException e)
        {  Console.WriteLine("Нет файла" + e.Message); return;
        }
catch (FormatException e)
        {  Console.WriteLine(e.Messa
        }
```

```
public static int string2int(string s)
        {
            return Convert.ToInt32(s);
        }
```

Перечисления

Определение перечисления

Перечисление – набор связанных констант:

```
enum Menu { Read, Write, Append, Exit };

enum Радуга { Красный, Оранжевый, Желтый, Зеленый, Снний, Фиолетовый };

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

enum Flags : byte

{    b0, b1, b2, b3 = 0x04, b4 = 0x08, b5 = 0x10, b6 = 0x20, b7 = 0x40    };
```

- Имена перечисляемых констант внутри каждого перечисления должны быть уникальными, а значения могут совпадать.
- Преимущества перечисления перед описанием именованных констант:связанные константы нагляднее; компилятор выполняет проверку типов; интегрированная среда разработки подсказывает возможные значения констант.
- Все перечисления являются потомками базового класса System.Enum

Операции с перечислениями

- С переменными перечисляемого типа можно выполнять:
  - арифметические операции (+, –, ++, ––),
  - логические поразрядные операции (^, &, |, ~),
  - сравнения (<, <=, >, >=, ==, !=)
  - 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.

```
Flags a = Flags.b2 | Flags.b4;

++a;

int x = (int) a;

Flags b = (Flags) 65;
```

СПАСИБО ЗА ВНИМАНИЕ