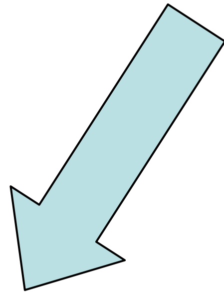
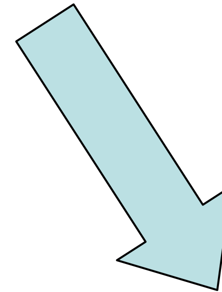
C#: пользовательские типы

типы данных C#





<u>типы значения:</u>

- **перечисления (enum)**
- **структуры (struct)**

<u>ссылочные типы:</u>

- **массивы**
- **классы (class)**

перечисления

- Существует возможность декларировать перечислимые типы (*enums*), объекты которых представляются именованными константами.
- Перечислимые типы являются типами значений, определяемые на основе некоторого целочисленного типа, называемого *базовым* (по умолчанию — **int**).
- Каждая константа представляет собой значение базового типа

перечисления

- Пример:

  public enum Coin : **uint**

  ```
  {
      Penny = 1,
      Nickey = 5,
      Dime = 10,
      Quarter = 25
  }
  ```

- Обращение к элементам перечисления всегда производится через тип перечисления:

  Coin.Penny, Coin.Nickey, …

массивы

- На основе пользовательского или примитивного типа можно строить *массивы* элементов данного типа
- Тип массива является ссылочным (System.Array)
- Количество элементов массива — это свойство конкретного объекта-массива, которое задаётся при построении
- Можно строить массивы массивов

массивы

- в дополнение к массивам массивов можно строить многомерные массивы

```
int[ ] array = new int[3];
string[ ] array1 =
    new string[ ] {"first", "second"};
int[ ][ ] arrayOfArrays =
    new int[ ][ ] {{1, 2, 3}, {4, 5}, {6}};
int[ , ] twoDimensionalArray =
    new int[ , ] {{1, 2}, {3, 4}};
```

массивы

- Любой тип массива наследуется от System.Array, любой объект-массив имеет все свойства и методы этого типа
- Полезные свойства и методы:
  - Length — общее количество элементов в массиве (во всех размерностях)
  - Rank — количество измерений в массиве
  - Copy, CopyTo — копирование элементов массива в другой массив

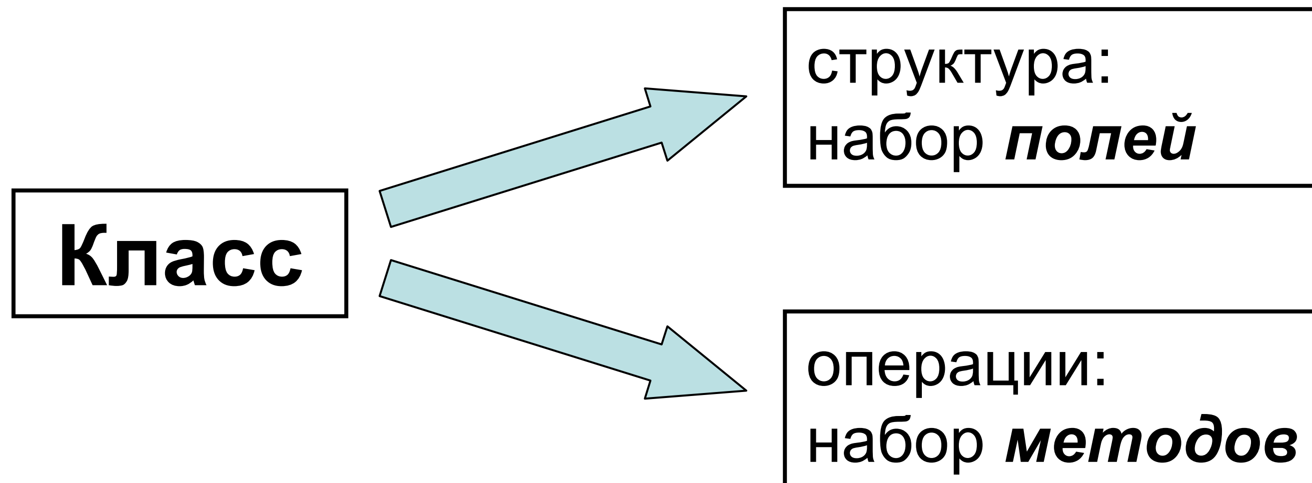
массивы

- изменение длины массива:

```
int[ ] arr = new int[10] {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
...
int[ ] newarr = new int[15];
Array.Copy(arr, newarr, arr.Length);
arr = newarr; // arr={1,2,3,4,5,6,7,8,9,10,0,0,0,0,0}
...
int[ ] newarr = new int[5];
Array.Copy(arr, newarr, newarr.Length);
arr = newarr; // arr={1,2,3,4,5}
```

class

- **Класс** – ссылочный тип, объекты которого могут иметь сложную структуру и быть задействованы в некотором наборе операций



class

- для каждой операции в классе определяется её **сигнатура** и **реализация**

Сигнатура = имя + список типов параметров

- **реализация** – набор инструкций, выполняемых каждый раз, когда эта операция вызывается

class

- **Абстрактный класс** может не определять реализацию для некоторых своих операций – такие операции называются **абстрактными** (**abstract**)

класс и операции должны быть помечены модификатором **abstract**

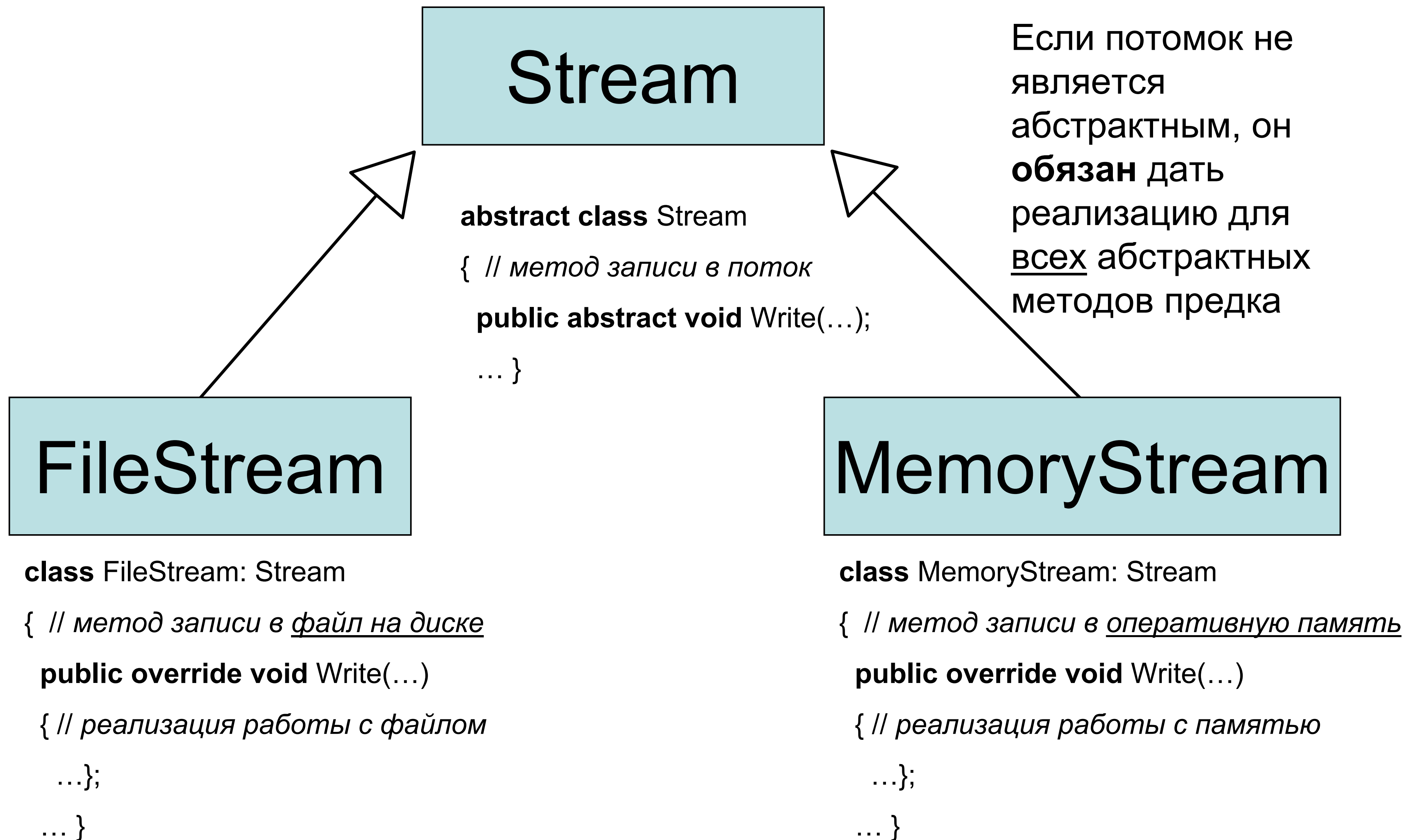
```
abstract class A
{
    public abstract void P();
}
```

абстрактные методы

- **абстрактные** методы являются **виртуальными**, дополнительно помечать их модификатором **virtual** не нужно
- для перегрузки **виртуальных** и **абстрактных** методов в классах-потомках используется модификатор **override**

```
class B: A
{
    public override void P() { … }
}
```

абстрактный класс



статические члены

- Поля и операции могут быть **статическими** (**static**), т.е. относиться не к объекту класса, а к классу в целом; обращение к статическому полю осуществляется через класс (но не через экземпляр)

```
class A
{
   public static int X = 10;
}
```

```
A a = new A();
```

~~a.X~~ — **некорректное** обращение

A.X — **корректное** обращение

члены класса

- константы
- поля
- методы
- свойства
- индексированные свойства
- события
- операторы
- вложенные типы

константы

Константы являются единожды вычисляемыми и неизменными далее значениями

```
public class A
{
  public const int MyConst = 25;
  public const double Phi = 1.61803398875989;
}
```

ПОЛЯ

Поля — задают структуру данных. Перед каждым полем указывается модификатор доступа. По умолчанию — **private**.

```
public class A
{
  int j = 1;
  private int k = 0;
  protected float f = 0.0f;
  public static int i = 1; // могут быть статическими
}
```

методы

Методы определяют допустимые операции над классом

```
public class A {
  public A() {…}          // конструктор
  ~A() {…}                // деструктор
  void P(int x) {…}       // по умолчанию - private
  protected int Calc() {…}
}
```

параметры методов

- Ссылочные типы в методы всегда передаются по ссылке, типы значения — по значению.
- Можно определить параметры значения, передающиеся по ссылке, и выходные параметры.
- Параметры, передающиеся по ссылке, помечаются модификатором **ref**, выходные параметры помечаются модификатором **out**.

параметры методов

- При вызове операций значения параметров помечаются тем же модификатором (**ref** или **out**)

```
public class A {
  public void f(int a, ref int b, out int c)
  { c = b – a; b += a;  }
  public static void Main()
  {
    A a = new A();
    int n = 3;
    int m; // m инициализировать не обязательно
    a.f(1, ref n, out m); // модификаторы повторяются
  }
}
```

свойства

Свойства (**properties**) — «виртуальные» поля. Каждое свойство имеет один или оба *метода доступа* **get** и **set**, которые определяют действия при чтении и модификации этого свойства.

Обращение к свойству — чтение (возможно, только если у него есть метод **get**) или изменение значения свойства (возможно, только если у него есть метод **set**) — происходит также, как к полю.

свойства

оба метода доступа описываются внутри декларации свойства

```
public class A
{
  private int odd = 1;
  public int OddValue
  {
    get { return odd; }  // метод доступа для чтения
    set                         // метод доступа для записи
    {
      if (value % 2 != 1)     // value — устанавливаемое значение
        throw new ArgumentException();
      else odd = value;
    }
  }
}
```

индексированные свойства

Индексированное свойство или **индексер** (**indexer**) — это свойство, зависящее от набора параметров.

Обращение к индексеру объекта производится так, как будто этот объект был бы массивом, индексированным набором индексов соответствующих типов.

индексированные свойства

```
public class MyArrayList {
  int[ ] items = new int[10];
  public int this [int Index] {
    get  {   return items[Index];    }
    set  {   items[Index] = value;  }
  }
}
…
MyArrayList L = new MyArrayList();
L[0] = 2; // запись
L[1] = L[0] + 3;  // чтение и запись
```

делегатный тип

- **Делегатный тип** (**delegate**) — ссылочный тип, аналог указателей на функцию в Си. Определяется так же как и абстрактный метод.
- Пример определения:

public delegate int BinaryOp(**int** x, **int** y);

BinaryOp — новый делегатный тип

делегатный тип

```
public class A {
  private int x = 0;
  public A(int x) { this.x = x; }
  public static int Op1(int a, int b)  { return a + b; }
  public int Op2(int a2, int b2)  { return x + a2 + b2; }
  void Test () {
    A a = new A(24);
    BinaryOp op1 = A.Op1; // op1 — указатель на статический метод
    BinaryOp op2 = a.Op2; // op2 — указатель на обычный метод
    int z =
        op1(1, 2)  // вызов A.Op1(1, 2)
     + op2(3, 4); // вызов a.Op2(3, 4)
  }
}
```

делегатный тип

- Объекты делегатных типов предназначены служить обработчиками некоторых событий, т.е. при наступлении заданного события надо вызвать соответствующий делегат.
- Каждый объект-делегат представляет некоторый *список операций* (*invocation list*). Пустой список представляется как **null**.

делегатный тип

- добавлять элементы в конец списка можно при помощи операторов + и +=, применяемых к делегатам
- удалять операции из делегатов можно при помощи операторов – и –=

события

- **Событие** (**event**) представляет собой свойство специального вида, имеющее делегатный тип.
- У события методы доступа называются **add** и **remove** (добавление и удаление обработчиков событий при помощи операторов += и –=).

события

- Событие может быть реализовано как поле делегатного типа, помеченное модификатором **event**. В этом случае декларировать методы **add** и **remove** не обязательно.

```
public delegate void MouseEventHandler (object source,
        MouseEventArgs e); // делегатный тип

public class MouseEventSource {
        // событие нажатия кнопки «мыши»
        public event MouseEventHandler MouseDown;
        …
}
```

операторы

- Некоторые операторы можно переопределить (перекрыть) для данного пользовательского типа.
- Переопределяемый оператор всегда имеет модификатор **static**

операторы

- Переопределяемые унарные операторы (единственный параметр — пользовательский тип, в рамках которого он переопределяется):

1) +, –, !, ~ (в качестве типа результата могут иметь любой тип)
2) ++, -- (тип результата — только подтип объемлющего)
3) **true**, **false** (тип результата — **bool**)

операторы

- Переопределяемые бинарные операторы (хотя бы один из их параметров должен иметь объемлющий тип)

1) +, –, *, /, %, &, |, ^, ==, !=, <, >, <=, >= (тип результата — любой)
2) >>, << (второй параметр всегда типа **int**, тип результата — любой)

операторы

- Можно определять операторы приведения к другому типу или приведения из другого типа. Такое приведение можно объявлять:
  - *неявным* (**implicit**) — компилятор сам его вставляет где необходимо
  - *явным* (**explicit**) — программист всегда должен явно приводить один тип к другому

операторы

- Некоторые операторы можно переопределять только парами:
  - **true** и **false**
  - == и !=
  - < и >
  - <= и >=

операторы: пример

```
public class MyInt {
  int n = 0;
  public MyInt(int n) { this.n = n; }
  public static bool operator == (MyInt i1, MyInt i2)
    { return i1.n == i2.n; }
  public static bool operator != (MyInt i1, MyInt i2)
    { return i1.n != i2.n; }
  public static bool operator true (MyInt i)
    { return i.n > 0; }
  public static bool operator false (MyInt i)
    { return i.n <= 0; }
  public static MyInt operator ++ (MyInt i)
    { return new MyInt(i.n + 1); }
}
```

интерфейсы

- **Интерфейс** (*interface*) — программная сущность со следующими свойствами:
  - интерфейс подобен абстрактному базовому классу: любой неабстрактный тип, реализующий интерфейс, должен реализовать все его члены
  - невозможно создать экземпляр интерфейса напрямую
  - могут содержать события, индексаторы, методы и свойства
  - не содержат реализации методов
  - классы и структуры способны реализовывать несколько интерфейсов
  - интерфейс может быть унаследован от нескольких интерфейсов

интерфейсы

```
interface IStorable
{
    void Read( );
    void Write(object o);
}
```

```
class Document: IStorable
{
    // IStorable
    public void Read( ) { … }
    public void Write(object o)
    { … }
}
```

- Интерфейс не может содержать константы, поля, операторы, конструкторы экземпляров, деструкторы или типы.
- Он не может содержать статические члены.
- Члены интерфейсов автоматически являются открытыми, и они не могут включать никакие модификаторы доступа.

интерфейсы

- C# допускает наследование лишь от одного класса, к интерфейсам это не относится
- Класс может реализовывать несколько интерфейсов

интерфейсы

```
interface ICompressible
{
    void Compress( );
    void Decompress( );
}
```

```
class Document:
    IStorable, ICompressible
{
    // IStorable
    public void Read( ) { … }
    public void Write(object o)
    { … }
    // ICompressible
    public void Compress( ) { … }
    public void Decompress( )
    { … }
}
```

Шаблонные типы

- Классы и интерфейсы (а также отдельные операции) могут быть *шаблонными* (**generic**), т.е. иметь *типовые параметры* (начиная с C# 2.0)
- При создании объекта такого класса нужно указывать конкретные значения типовых параметров.

Шаблонные типы: пример

```
using System;
public interface IQueue<T>
{
    void Put(T o);
    T    Get();
    int  Size();
}
```

```
public class ArrayQueue<T>:
    IQueue<T>
{
    T[ ] arr = new T[10];
    int size = 0, first = 0;
    public void Put(T o)
        { arr[size++] = o; }
    public T Get()
        { return arr[first++]; }
    public int Size()
        { return size; }
}
```

Шаблонный тип List<T>

- List<T> описан в пространстве имён System.Collections.Generic
- Основные свойства:
  - Count — текущее количество хранящихся элементов
  - Item — индексер, возвращает i-й элемент
- Основные операции:
  - Add(T) — добавляет элемент в конец
  - Insert(int, T) — вставляет элемент в указанную позицию
  - Clear() — удаляет все элементы
  - Contains(T) — проверяет на вхождение
  - IndexOf(T) — возвращает индекс элемента
  - Remove(T) — удаляет первое вхождение
  - RemoveAll(T) — удаляет все вхождения
  - RemoveAt(int) — удаляет элемент по индексу

структуры

- Структуры (**struct**) — пользовательский **тип значений**, по декларации сильно напоминающий классы.
- Отличия от классов:
  - нельзя наследовать ни от классов, ни от структур, но могут реализовывать интерфейсы;
  - нельзя использовать инициализацию полей вне методов;
  - не обязательно создавать с помощью **new**, но можно;
  - в конструкторе (если он есть) обязаны инициализироваться все поля

наследование

- Отношение вложенности между типами определяется *наследованием*
- Класс может наследовать только одному классу (множественного наследования классов нет)
- Интерфейсы тоже могут наследовать, есть множественное наследование интерфейсов

наследование

- все классы, структурные, перечислимые и делегатые типы (но не интерфейсы!) — наследники класса System.Object.
- При этом типы значения преобразуются к типу System.Object с помощью упаковки, строящей каждый раз новый объект.
- Структурный тип может реализовывать один или несколько интерфейсов, но не может наследовать классу или другому структурному типу

наследование

- При наследовании (сужении типа) возможно определение дополнительных полей и операций.
- Возможно также определение в классе-потомке поля, имеющего то же имя, что и некоторое поле в классе-предке — в этом случае происходит ***перекрытие имён*** — в коде потомка доступно только новое поле.

наследование

- Доступ к полю или операции предка осуществляется через ключевое слово **base**:

  base.fieldName

  base.someOp()

наследование

- Основная выгода от использования наследования — возможность **перегружать** (***override***) реализации операций в типах-наследниках.
- Функции, которые допустимо перегружать, называются *виртуальными* и помечаются модификатором **virtual**.
- Новая реализация виртуальной функции в классе-потомке помечаются модификатором **override**.

наследование

- статические операции (относящиеся к классу в целом) **не могут** быть виртуальными
- нестатические операции могут быть как виртуальными (перегружаемыми), так и невиртуальными.