Повторение материала I семестра

- Основы языка C#
- Основы ООП
- Типы и структуры данных

Основы языка C#

Состав языка

- **Символы**:
  - буквы: A-Z, a-z, _, бук
  - цифры: 0-9, A-F
  - спец. символы: +, *, {, …
  - пробельные символы

Лексема (token, токен) – минимальная единица языка, имеющая самостоятельный смысл

- **Лексемы**:
  - константы “Вася”
  - имена Vasia a _11
  - ключевые слова double do if
  - знаки операций + <= new
  - разделители ; [ ] ,

- **Выражения**
  - выражение - правило вычисления значения: a + b
- **Операторы**
  - исполняемые: c = a + b;
  - описания: double a, b;

Константы (литералы) C#

Вид	Примеры
Булевские	**true** **false**
Целые десятич.	**8** 199226 0**Lu**
16-ричн.	0xA 0x1B8 0X00FF**L**
Веществ. с тчк	**5.7** .001f 35m
с порядком	**0.2E6** .11e–3 5E12
Символьные	**'A'** '**\x**74' '\0' '\\' '**\u**A81B'
Строковые	**"Здесь был Vasia"**
	"\t3начение r=\xF5\n"
	был \u0056\u0061"
	@"C:\temp\file1.txt"
Константа null	null

Суффикс типа

$5 \cdot 10^{12}$

Управляющий символ

Кодировка Unicode

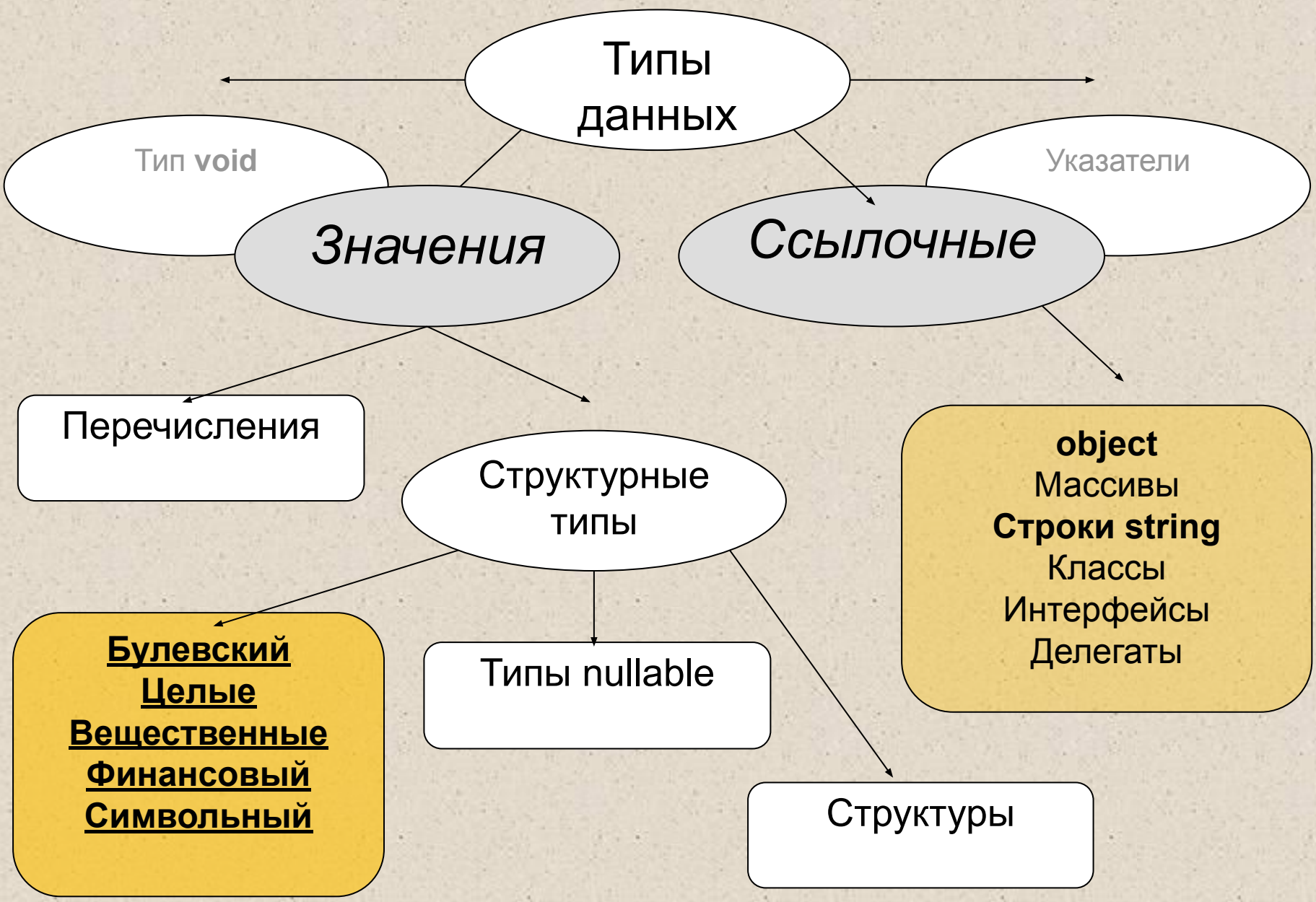
Концепция типа данных

Тип данных определяет:

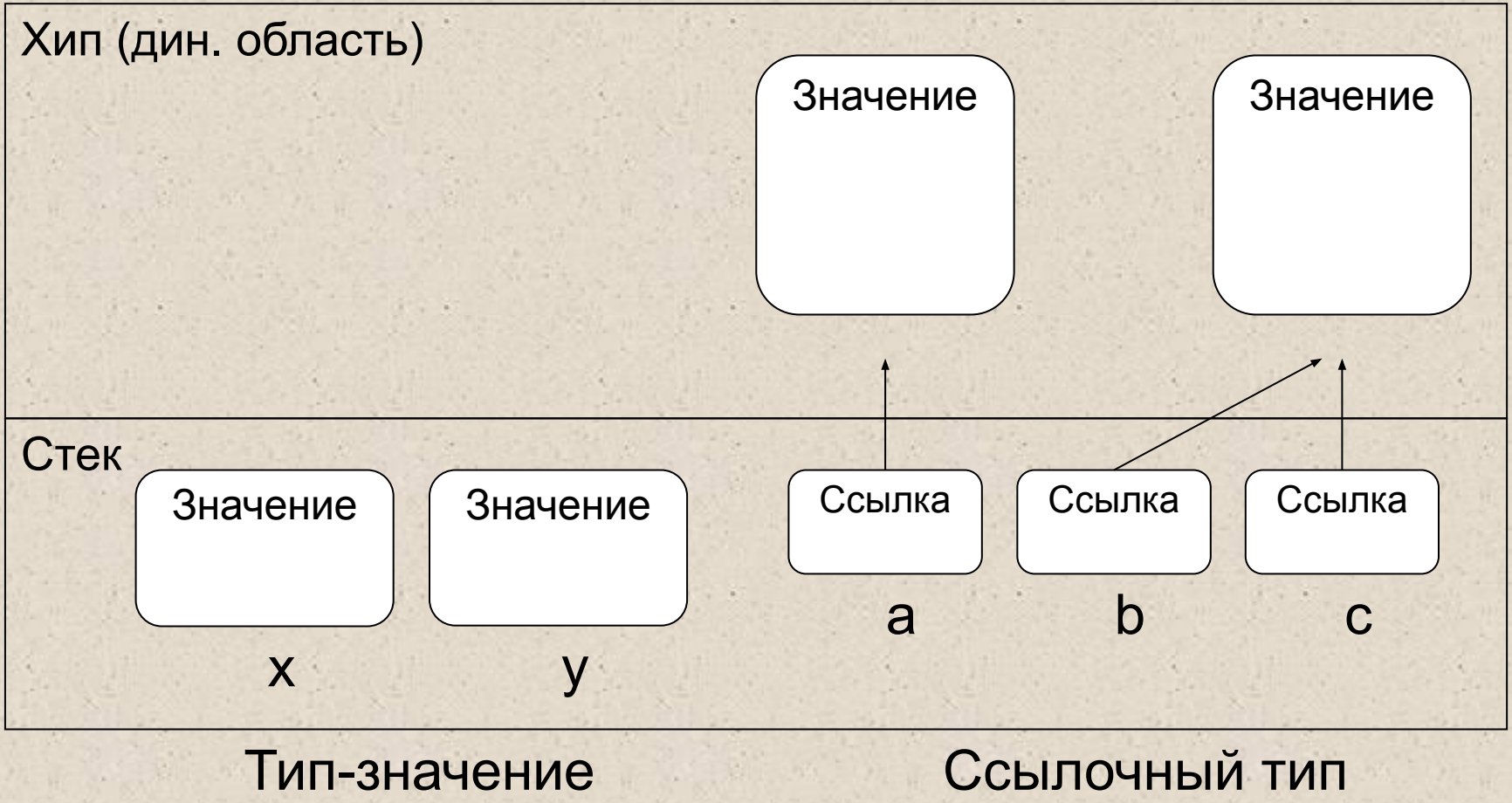
- **внутреннее представление данных** =>
  множество их возможных значений

- **допустимые действия над данными** =>
  операции и функции

Основная классификация типов C#



Хранение в памяти величин значимого и ссылочного типа



Логический (булевский) и целые

Название	Ключевое слово	Тип .NET	Диапазон значений	Описание	Размер в битах
Булевский	bool	**Boolean**	**true, false**		
Целые	sbyte	**SByte**	–128 — 127	знаковое	**8**
	byte	**Byte**	0 — 255	беззнаковое	**8**
	short	**Int16**	–32768 —32767	знаковое	**16**
	ushort	**UInt16**	0 — 65535	беззнаковое	**16**
	int	**Int32**	$\approx(-2\cdot10^9 — 2\cdot10^9)$	знаковое	**32**
	uint	**UInt32**	$\approx(0 — 4\cdot10^9)$	беззнаковое	**32**
	long	**Int64**	$\approx(-9\cdot10^{18} — 9\cdot10^{18})$	знаковое	**64**
	ulong	**UInt64**	$\approx(0— 18\cdot10^{18})$	беззнаковое	**64**

Остальные

Название	Ключевое слово	Тип .NET	Диапазон значений	Описание	Размер в битах
Символьный	char	**Char**	U+0000 — U+ffff	символ Unicode	16
Веществен-ные	float	**Single**	(+-)$1.5 \cdot 10^{-45}$ — $3.4 \cdot 10^{38}$	7 цифр	32
	double	**Double**	(+-) $5.0 \cdot 10^{-324}$ — $1.7 \cdot 10^{308}$	15-16 цифр	64
Финансовый	decimal	**Decimal**	(+-) $1.0 \cdot 10^{-28}$ — $7.9 \cdot 10^{28}$	28-29 цифр	128
Строковый	string	**String**	длина ограничена объемом доступной памяти	строка из символов Unicode	
object	object	**Object**	можно хранить все, что угодно	всеобщий предок	

Структура простейшей программы на C#

```
using System;
namespace A
{
    class Class1
    {
        static void Main()
        {
            // описания и операторы, например:
            Console.Write("Превед медвед");
        }

    // описания
    }
}
```

Переменные

- *Переменная* — это величина, которая во время работы программы может изменять свое значение.
- Все переменные, используемые в программе, должны быть описаны.
- *Для* каждой переменной задается ее ***имя*** *и* ***тип***:

```
int    number;
float  x, y;
char   option;
```

Тип переменной выбирается исходя из диапазона и требуемой точности представления данных.

Общая структура программы на C#

Область действия и время жизни переменных

- Переменные описываются внутри какого-л. блока:

  1) класса

  2) метода или блока внутри метода

Блок — код, заключенный в фигурные скобки. Основное назначение блока — группировка операторов.

  - Переменные, описанные *непосредственно внутри класса*, называются **полями класса**.
  - Переменные, описанные *внутри метода класса*, называются **локальными переменными**.

- Область действия переменной - область программы, где можно использовать переменную.
- Область действия переменной начинается в точке ее описания и длится до конца блока, внутри которого она описана.
- Время жизни: переменные создаются при входе в их область действия (блок) и уничтожаются при выходе.

Инициализация переменных

- При объявлении можно присвоить переменной начальное значение (*инициализировать*).

```
int number = 100;
float   x   = 0.02;
char    option = 'ю';
```

При инициализации можно использовать не только константы, но и выражения — главное, чтобы на момент описания они были вычислимыми, например:

```
int b = 1, a = 100;

int x = b * a + 25;
```

- **Поля класса** инициализируются «значением по умолчанию» (0 соответствующего типа).

- **Локальные переменные** автоматически НЕ инициализируются. Рекомендуется всегда явным образом инициализировать переменные при описании.

Тип результата выражения

- Если входящие в выражение **операнды одного типа** и операция для этого типа определена, то результат выражения будет иметь тот же тип.
- Если **операнды разного типа** и (или) операция для этого типа не определена, перед вычислениями автоматически выполняется преобразование типа по правилам, обеспечивающим приведение *более коротких типов к более длинным* для сохранения значимости и точности.

```
char   c = 'A';
int    i = 100;
double d = 1;
double summa = c + i + d;    // 166
```

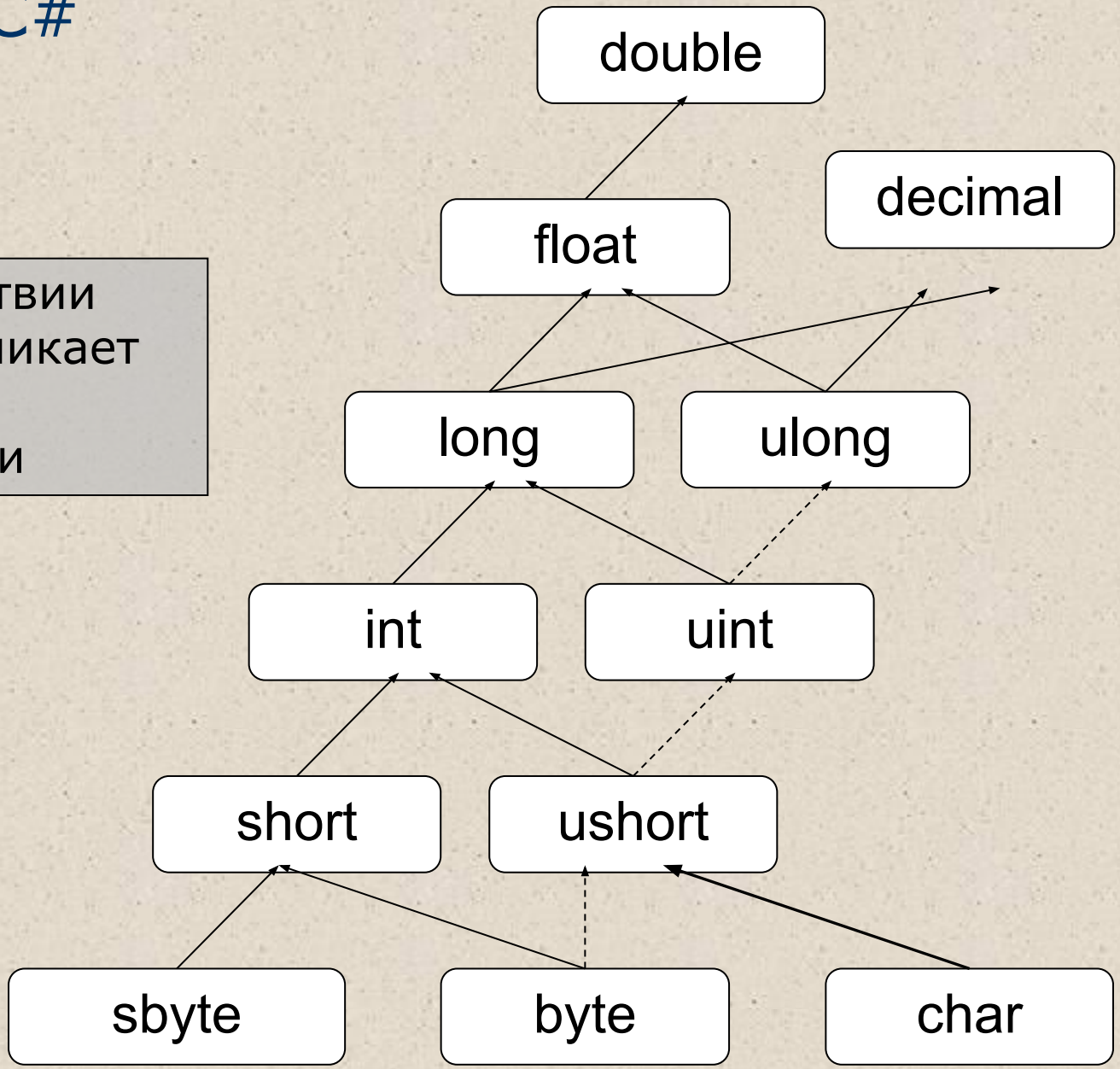
Явное преобразование типа

- Автоматическое (*неявное*) преобразование возможно не всегда, а только если при этом не может случиться потеря значимости.
- Если неявного преобразования из одного типа в другой не существует, программист может задать *явное* преобразование типа с помощью операции (тип) x.

```
char   c = 'A';
int    i = 100;
double d = 1;
c = (char) i;                  // 'd'
c = (char) d;
i = (int) d;
```

Неявные арифметические преобразования типов в C#

при отсутствии линии возникает ошибка компиляции



Вывод на консоль – 1/4

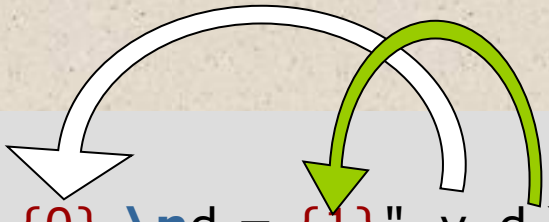
```
using System;
namespace A
{    class Class1
   {    static void Main()
      {
         int       i = 3;
         double  y = 4.12;
         decimal d = 600m;
      string    s = "Вася";

         Console.Write( i );
         Console.WriteLine( " y = "  + y);
         Console.WriteLine("d = " + d + " s = " + s );

      }
   }
}
```

Результат работы программы:
3 y = 4,12
d = 600 s = Вася

Вывод на консоль – 2/4

```
using System;
namespace A
{    class Class1
   {    static void Main()
      {
         int       i = 3;
         double  y = 4.12;
         decimal d = 600m;
      string    s = "Вася";

         Console.Write( i );
         Console.Write( " y = {0} \nd = {1}", y, d );
         Console.WriteLine( " s = " + s );

      }
   }
}
```

Результат работы программы:
3 y = 4,12
d = 600 s = Вася



Вывод на консоль – 3/4

```
using System;
namespace A
{    class Class1
   {    static void Main()
      {
         int       i = 3;
         double  y = 4.12;
         decimal d = 600m;
      string    s = "Вася";

         Console.Write( i );
         Console.Write( " y = {0:F2} \nd = {1:D3}", y, d );
         Console.WriteLine( " s = " + s );
      }
   }
}
```

Результат работы программы:
3 y = 4,12
d = 600 s = Вася

Формат

Формат

Ввод с консоли – 2/2

```
using System;
namespace A
{    class Class1
   {    static void Main()
      {
         string s = Console.ReadLine();          // ввод строки

         char c = (char)Console.Read();          // ввод символа
         Console.ReadLine();

         int i = Convert.ToInt32( Console.ReadLine() );

         double x = Convert.ToDouble( Console.ReadLine() );

         double y = double.Parse( Console.ReadLine() );
      }
   }
}
```

Пример: перевод температуры из F в C

$$C = \frac{5}{9}(F - 32)$$

```
using System;
namespace CA1
{    class Class1
  {    static void Main()
    {
      Console.WriteLine( "Введите температуру по Фаренгейту" );
      double fahr = Convert.ToDouble( Console.ReadLine() );

      double cels =5.0 / 9 * (fahr - 32);

      Console.WriteLine( "По Фаренгейту: {0} в градусах Цельсия: {1}",
                fahr, cels );
    }
  }
}
```

Управляющие операторы языка высокого уровня:

- следование
- ветвление
- цикл
- передача управления

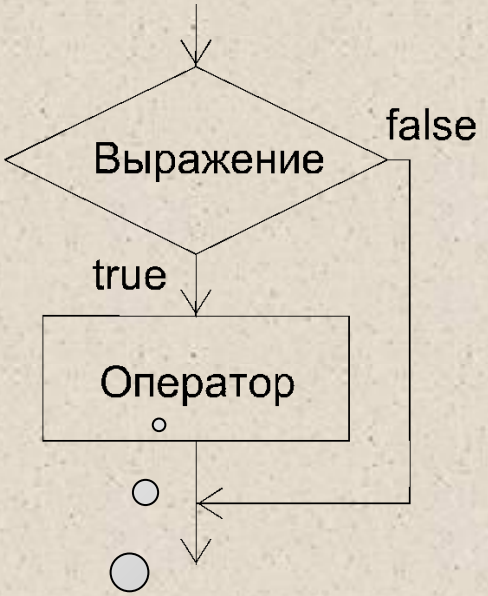
Реализуют логику выполнения программы

Блок (составной оператор)

- *Блок* — последовательность операторов, заключенная в операторные скобки:
  - begin end – в Паскале
  - { } - в С-подобных языках
- Блок воспринимается компилятором как один оператор и может использоваться **всюду, где синтаксис требует одного оператора, а алгоритм — нескольких**.
- Блок может содержать один оператор или быть пустым.

Условный оператор if

if (выражение) оператор_1;
[else оператор_2;]

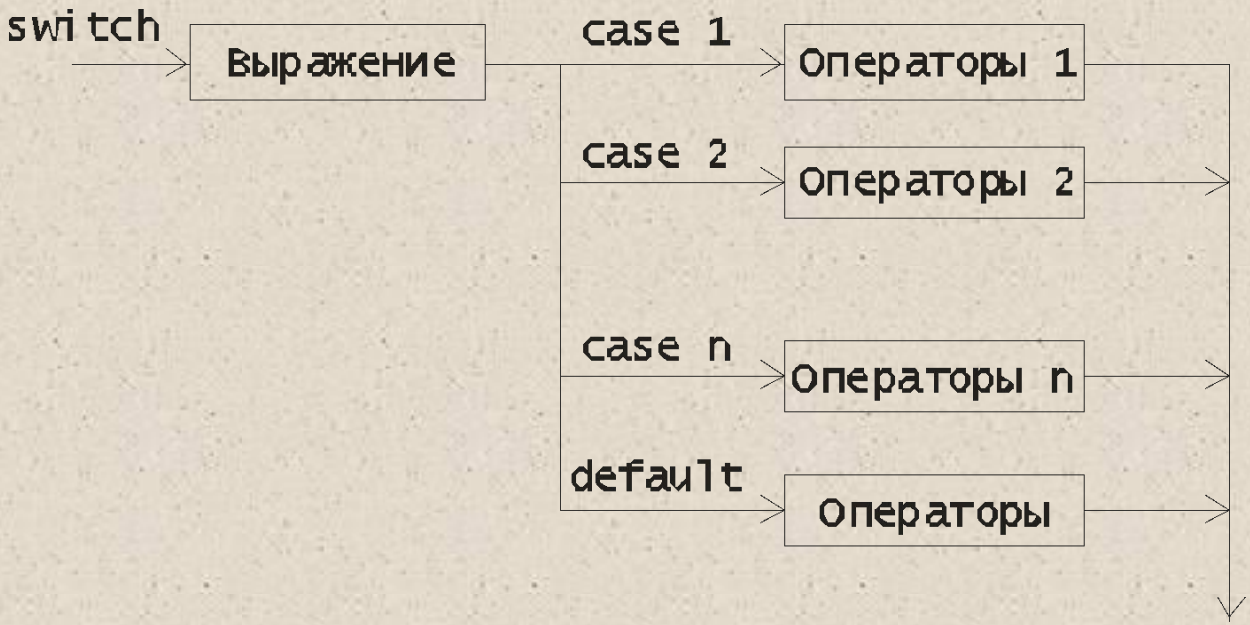



```
if ( a < 0 ) b = 1;

if ( a < b && (a > d || a == 0) )   ++b;

else { b *= a; a = 0; }

if ( a < b ) if ( a < c ) m = a;
             else         m = c;
else         if ( b < c ) m = b;
             else         m = c;
```

Простой или {блок}

Оператор выбора switch

```
switch ( выражение ){
   case константное_выражение_1: [ список_операторов_1 ]
   case константное_выражение_2: [ список_операторов_2 ]

   case константное_выражение_n: [ список_операторов_n ]
   [ default: операторы ]
}
```



Пример: Калькулятор на четыре действия

```
using System; namespace ConsoleApplication1
{   class Class1    {   static void Main()       {
        Console.WriteLine( "Введите 1й операнд:" );
       double a = double.Parse(Console.ReadLine());
       Console.WriteLine( "Введите знак" );
       char op = (char)Console.Read();
    Console.ReadLine();
       Console.WriteLine( "Введите 2й операнд:" );
       double b = double.Parse(Console.ReadLine());
        double res = 0;
        bool ok = true;
        switch (op)
           {  case '+' : res = a + b; break;
              case '-'  : res = a - b;  break;
              case '*' : res = a * b; break;
              case '/' : res = a / b; break;
              default  : ok = false; break;
           }
       if (ok) Console.WriteLine( "Результат: " + res );
       else    Console.WriteLine( "Недопустимая
    операция" );
}}}
```

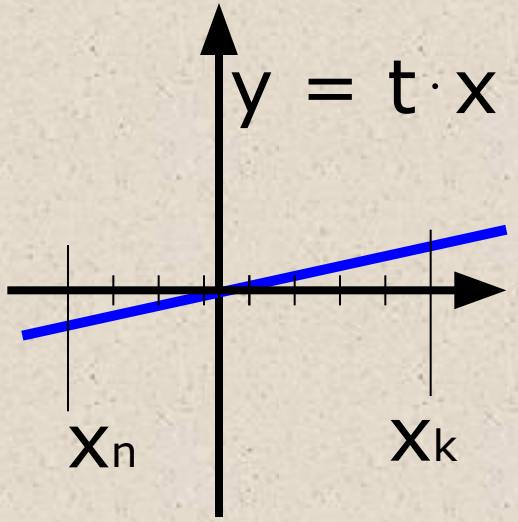
Операторы цикла:

- с предусловием - **while**
- с постусловием - **do**
- с параметром - **for**
- перебора - **foreach**

Цикл с предусловием

while (выражение) оператор

```
using System;
namespace ConsoleApplication1
{   class Class1
    {   static void Main()
        {
            double Xn = -2, Xk = 12, dX = 2, t = 2, y;
            Console.WriteLine( "|     x     |     y     |" );

            double x = Xn;
            while ( x <= Xk )
            {
                y = t * x;
                Console.WriteLine( "| {0,9} | {1,9} |", x, y );
                x += dX;
            }
        }
    }
}
```



Цикл с постусловием

Удобно использовать для проверки ввода

```
do
    оператор
while
    выражение
    ;
```

```
using System;
namespace ConsoleApplication1
{   class Program
    {   static void Main()
        {
            char answer;
            do
            {
                Console.WriteLine( "Купи слоника, а?" );
                answer = (char) Console.Read();
                Console.ReadLine();
            } while ( answer != 'y' );
        }
    }
}
```

Пример цикла с параметром

```
using System;
namespace ConsoleApplication1
{   class Class1
    {   static void Main()
        {
            double Xn = -2, Xk = 12, dX = 2, t = 2, y;
            Console.WriteLine( "|     x     |     y     |";
            for ( double x = Xn; x <= Xk; x += dX )
            {
                y = t * x;
                Console.WriteLine( "| {0,9} | {1,9} |", x, y );
            }
        }
    }
}
```

Передача управления

- оператор break — завершает выполнение цикла, внутри которого записан
- оператор continue — выполняет переход к следующей итерации цикла
- оператор return — выполняет выход из функции, внутри которой он записан
- оператор goto — выполняет безусловную передачу управления
- оператор throw — генерирует исключительную ситуацию.

Пример: вычисление суммы ряда

```
using System;
namespace ConsoleApplication1
{   class Class1
    {   static void Main()
        {  double e = 1e-6;                const int iterLimit = 500;
           Console.WriteLine( "Введите аргумент:" );
           double x = Convert.ToDouble(Console.ReadLine());

           bool error = false;          // признак ошибки
           double c = x, y = c;         // член ряда и сумма ряда
           for ( int n = 1; Math.Abs(c) > e; n++ )
           {    c *= - x * x / ((2 * n ) * ( 2 * n + 1 ));
                y += c;
                if ( n > iterLimit ) { error = true; break; }
           }
           if ( error ) Console.WriteLine( "Ряд расходится" );
           else         Console.WriteLine( "Сумма ряда - " + y );
}}}  end.
```

Простая проверка ввода

```
// пример проверки формата вводимого знач

double a;
if (! double.TryParse(Console.ReadLine(), out a) )
    {Console.WriteLine(" Неверный  формат "); return; }

// при вводе более одного значения предпочтительнее
// использовать механизм исключений
```

не гуманно!

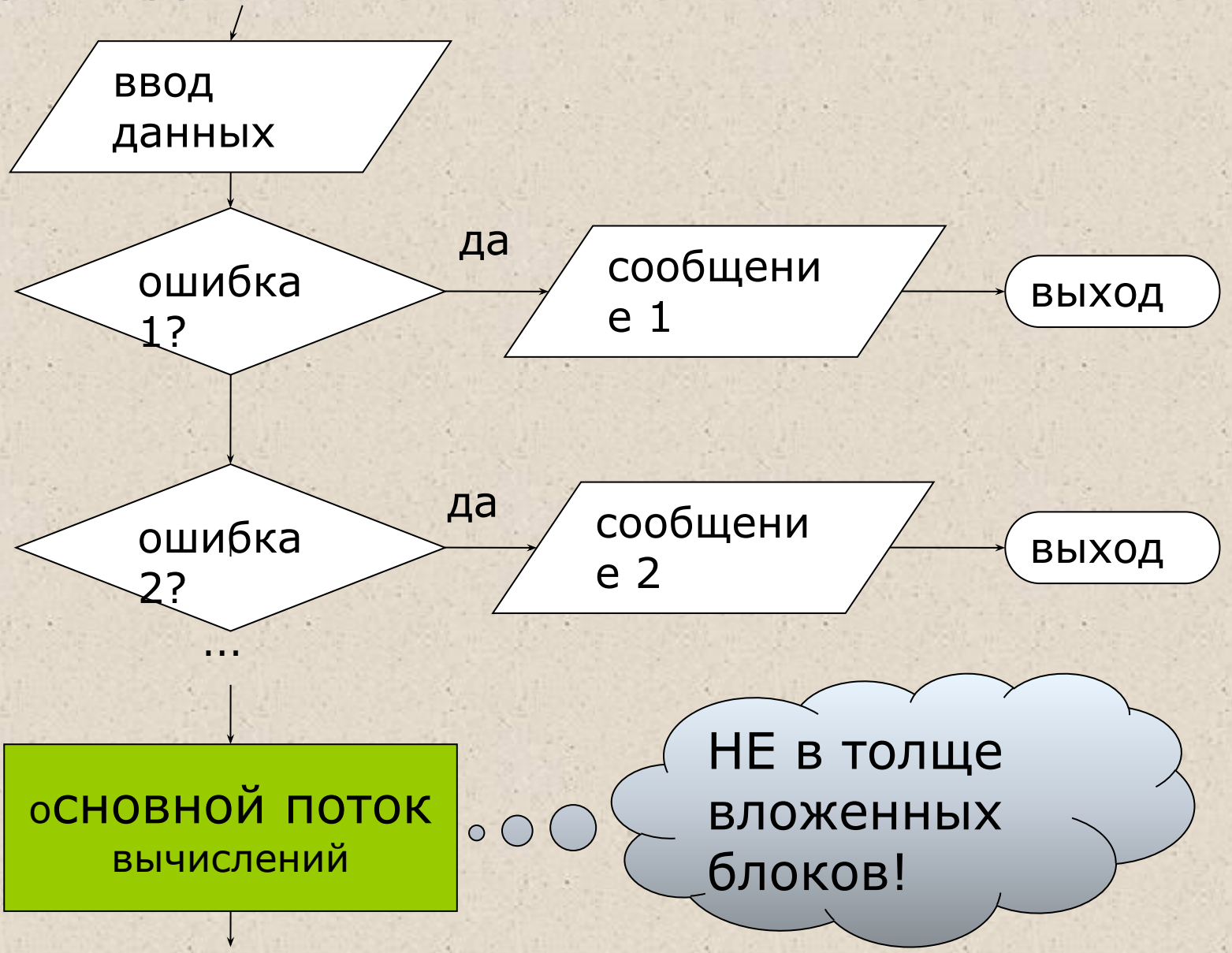
```
// пример проверки допустимости значения:

double a = double.Parse(Console.ReadLine());
...
if ( a <= 0 )
    { Console.WriteLine("Неверное  значение (<= 0)" );
      return;
    }
```

не человеколюбиво !

Проверка ввода с помощью цикла do-while

```
using System;
namespace ConsoleApplication1
{   class Program
    {   static void Main()  {
            const int max_attempts = 3;
            int i = 0;
            do
            {
                Console.WriteLine( "Введите  значение > 0:" );
                double a = double.Parse(Console.ReadLine());
                ++i; if ( i >= max_attempts ) { ... return; }
            } while ( a <= 0 );
        }
    }          // ограничивать кол-во попыток обязательно!
}
```

Рекомендуемая структура обработки ошибок исходных данных



Понятие «исключительная ситуация»

- При вычислении выражений могут возникнуть ошибки (переполнение, деление на ноль).
- В C# есть механизм **обработки исключительных ситуаций** (**исключений**), который позволяет избегать аварийного завершения программы.
- Если в процессе вычислений возникла ошибка, система сигнализирует об этом с помощью **выбрасывания** *(генерирования) исключения*.
- Каждому типу ошибки соответствует свое исключение. Исключения являются классами, которые имеют общего предка — класс **Exception**, определенный в пространстве имен System.
- Например, при делении на ноль будет выброшено исключение DivideByZeroException, при переполнении — исключение OverflowException.
- В программе необходимо предусмотреть **обработку** исключений.

Механизм обработки исключений

- Функция или операция, в которой возникла ошибка, генерируют исключение;
- Выполнение текущего блока прекращается, отыскивается соответствующий обработчик исключения, ему передается управление.
- В любом случае (была ошибка или нет) выполняется блок finally, если он присутствует.
- Если обработчик не найден, вызывается стандартный обработчик исключения.

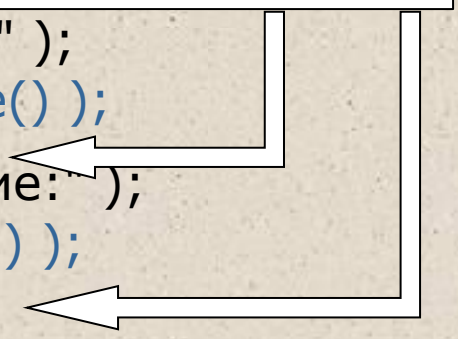
Пример 1:

```
try {

    // Контролируемый блок
}
catch ( OverflowException e ) {

            // Обработка переполнения
}
catch ( DivideByZeroException ) {

            // Обработка деления на 0
}
catch {
    // Обработка всех остальных исключений
}
```

Пример 2: проверка ввода

```
static void Main() {
        try
        {
            Console.WriteLine( "Введите напряжение:" );
            double u = double.Parse( Console.ReadLine() );

            Console.WriteLine( "Введите сопротивление:" );
            double r = double.Parse(Console.ReadLine() );

            double i = u / r;
            Console.WriteLine( "Сила тока - " + i );
        }
        catch ( FormatException )
        {
            Console.WriteLine( "Неверный формат ввода!" );
        }
        catch                                   // общий случай
        {
            Console.WriteLine( "Неопознанное исключение" );
        }
    }
```

```
if  (u < 0)
    { Console.WriteLine( "Недопустимое …" );
      return; }
```



Рекомендации по программированию – 1/2

- Главная цель, к которой нужно стремиться, — получить легко читаемую программу возможно более простой структуры.
- Создание программы начинают с определения ее исходных данных и результатов (тип, диапазон).
- Затем записывают на естественном языке (возможно, с применением обобщенных блок-схем), что именно и как должна делать программа.
- При кодировании необходимо помнить о принципах структурного программирования: программа должна состоять из четкой последовательности блоков — базовых конструкций.
- Имена переменных должны отражать их смысл. Переменные желательно инициализировать при их объявлении.
- Следует избегать использования в программе чисел в явном виде (кроме 0 и 1).
- Программа должна быть «прозрачна». Для записи каждого фрагмента алгоритма используются наиболее подходящие средства языка.

Рекомендации по программированию – 2/2

- В программе необходимо предусматривать реакцию на неверные входные данные.
- Необходимо предусматривать печать сообщений или выбрасывание исключения в тех точках программы, куда управление при нормальной работе программы передаваться не должно.
- Сообщение об ошибке должно быть информативным и подсказывать пользователю, как ее исправить.
- После написания программу следует тщательно отредактировать.
- Комментарии должны представлять собой правильные предложения без сокращений и со знаками препинания.

Основы ООП

Достоинства ООП

- использование при программировании понятий, близких к предметной области;
- возможность успешно управлять большими объемами исходного кода благодаря инкапсуляции, то есть скрытию деталей реализации объектов и упрощению структуры программы;
- возможность многократного использования кода за счет наследования;
- сравнительно простая возможность модификации программ;
- возможность создания и использования библиотек объектов.

Недостатки ООП

- идеи ООП не просты для понимания и в особенности для практического использования
- для эффективного использования существующих объектно-ориентированных систем и библиотек требуется **большой объем первоначальных знаний**
- **неграмотное применение ООП может привести к значительному ухудшению характеристик разрабатываемой программы**
- некоторое снижение быстродействия программы, связанное с использованием виртуальных методов

Абстрагирование и инкапсуляция

- При представлении реального объекта с помощью программного необходимо выделить в первом его существенные особенности и игнорировать несущественные. Это называется *абстрагированием*.
- Таким образом, программный объект — это абстракция.
- Детали реализации объекта скрыты, он используется через его *интерфейс* — совокупность правил доступа.
- Скрытие деталей реализации называется *инкапсуляцией.* Это позволяет представить программу в укрупненном виде — на уровне объектов и их взаимосвязей, а следовательно, управлять большим объемом информации.
- *Итак,* **объект — это инкапсулированная абстракция с четко определенным интерфейсом**.

Наследование

- Наследование (inheritance) - это процесс, посредством которого один объект может приобретать свойства другого.
- Важное значение имеет возможность многократного использования кода. Для объекта можно определить наследников, корректирующих или дополняющих его поведение.
- *Наследование* применяется для:
  - исключения из программы повторяющихся фрагментов кода;
  - упрощения модификации программы;
  - упрощения создания новых программ на основе существующих.
- Благодаря наследованию появляется возможность использовать объекты, исходный код которых недоступен, но в которые требуется внести изменения.
- Наследование позволяет создавать *иерархии объектов*. Иерархия представляется в виде дерева, в котором более общие объекты располагаются ближе к корню, а более специализированные — на ветвях и листьях.

Полиморфизм

- ООП позволяет писать гибкие, расширяемые и читабельные программы.
- Во многом это обеспечивается благодаря полиморфизму, под которым понимается возможность во время выполнения программы с помощью одного и того же имени выполнять разные действия или обращаться к объектам разного типа.
- Чаще всего понятие полиморфизма связывают с механизмом виртуальных методов.

Понятие класса

- *Класс* является **типом данных, определяемым пользователем**. Он должен представлять собой одну логическую сущность, например, являться моделью реального объекта или процесса. *Элементами* класса являются *данные* и *функции*, предназначенные для их обработки (*методы*).
- Все классы .NET имеют общего предка — класс object, и организованы в единую иерархическую структуру.
- Классы логически сгруппированы в пространства имен, которые служат для упорядочивания имен классов и предотвращения конфликтов имен: в разных пространствах имена могут совпадать. Пространства имен могут быть вложенными.
- Любая программа использует пространство имен System.

Описание класса

[атрибуты] [спецификаторы] **class** имя_класса [: предки] тело_класса

- Имя класса задается по общим правилам.
- Тело класса — список описаний его элементов, заключенный в фигурные скобки.
- Атрибуты задают дополнительную информацию о классе.
- Спецификаторы определяют свойства класса, а также доступность класса для других элементов программы.
- Простейшие примеры описания класса:

```
class Demo {}                          // пустой класс
public class Двигатель                 // класс с одним методом
{    public void Запуск()
     {     Console.WriteLine( " пыщь пыщь " ); }
}
```

Элементы класса

Описание объекта (экземпляра)

- Класс является обобщенным понятием, определяющим характеристики и поведение множества конкретных объектов этого класса, называемых экземплярами (объектами) класса.
- Объекты создаются явным или неявным образом (либо программистом, либо системой). Программист создает экземпляр класса с помощью операции new:

```
Demo a = new Demo();
Monster Vasia = new Monster();
Monster Petya = new Monster("Петя");
```

- Для каждого объекта при его создании в памяти выделяется отдельная область для хранения его данных.
- Кроме того, в классе могут присутствовать статические элементы, которые существуют в единственном экземпляре для всех объектов класса.
- Функциональные элементы класса всегда хранятся в единственном экземпляре.

Данные: поля и константы

- Данные, содержащиеся в классе, могут быть переменными или константами.
- Переменные, описанные в классе, называются **полями** класса.
- При описании полей можно указывать атрибуты и спецификаторы, задающие различные характеристики элементов:

[атрибуты] [спецификаторы] [const] тип имя **[= начальное_значение]**

```
public int a = 1;
public static string s = "Demo";
double y;
```

- Все поля сначала автоматически инициализируются нулем соответствующего типа (например, полям типа int присваивается 0, а ссылкам на объекты — значение null). После этого полю присваивается значение, заданное при его явной инициализации.

Методы

- Метод — функциональный элемент класса, реализующий вычисления или другие действия. Методы определяют поведение класса и составляют его **интерфейс**.
- Метод — законченный фрагмент кода, к которому можно обратиться по имени. Он описывается один раз, а вызываться может столько раз, сколько необходимо.
- Один и тот же метод может обрабатывать различные данные, переданные ему в качестве аргументов.

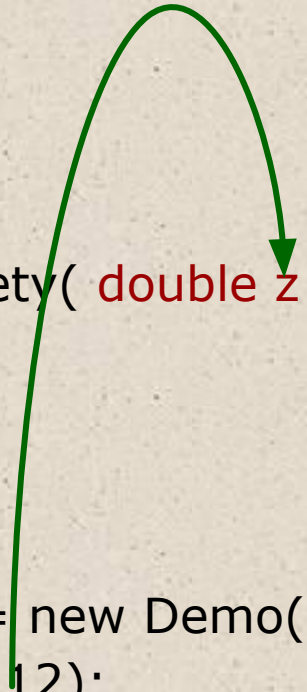
```
double a = 0.1;
double b = Math.Sin(a);
double c = Math.Sin(b-2*a);
```

```
Console.WriteLine
(a);
```

Синтаксис метода

[атрибуты] **[спецификаторы]** **тип** имя_метода **(** **[параметры]** **)** тело_метода

- Спецификаторы: new, **public**, protected, internal, protected internal, private, static, virtual, sealed, override, abstract, extern.
- Метод класса имеет непосредственный доступ к его полям.
- Пример:



```
class Demo {
    double y;                              // закрытое поле класса

    public void Sety( double z ) {         // открытый метод класса
        y = z;
    }
}

… Demo demo = new Demo();                  // где-то в методе другого класса
    demo.Sety(3.12); …                     // вызов метода
```

Параметры методов

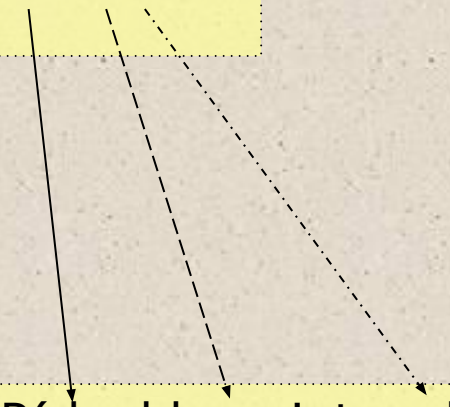
- Параметры определяют множество значений аргументов, которые можно передавать в метод.
- Список аргументов при вызове как бы накладывается на список параметров, поэтому они должны попарно соответствовать друг другу.
- Для каждого параметра должны задаваться его тип, имя и, возможно, вид параметра.
- Имя метода вкупе с количеством, типами и спецификаторами его параметров представляет собой сигнатуру метода — то, по чему один метод отличают от других.
- В классе не должно быть методов с одинаковыми сигнатурами.
- Метод, описанный со спецификатором static, должен обращаться только к статическим полям класса.
- Статический метод вызывается через имя класса, а обычный — через имя экземпляра.

Вызов метода

1. Вычисляются выражения, стоящие на месте аргументов.
2. Выделяется память под параметры метода.
3. Каждому из параметров сопоставляется соответствующий аргумент. При этом проверяется соответствие типов аргументов и параметров и при необходимости выполняется их преобразование. При несоответствии типов выдается диагностическое сообщение.
4. Выполняется тело метода.
5. Если метод возвращает значение, оно передается в точку вызова; если метод имеет тип void, управление передается на оператор, следующий после вызова.

Описание объекта: SomeObj obj = new SomeObj();
Описание аргументов: int b; double a, c;

Вызов метода: obj.P(a, b, c);



Заголовок метода P: public void P(double x, int y, double z);

Способы передачи аргументов в метод

Аргументы передаются:

- По значению
- По адресу (ссылке)

- *При передаче по значению* метод получает копии значений аргументов, и операторы метода работают с этими копиями.
- *При передаче по ссылке* (*по адресу*) метод получает копии адресов аргументов и осуществляет доступ к аргументам по этим адресам.

Типы параметров

В C# четыре типа параметров:

- параметры-значения - для исходных данных метода;
- параметры-ссылки (**ref**) - для изменения аргумента;
- выходные параметры (**out**) – для формирования аргумента;
- параметры-массивы (**params**) – для переменного кол-ва аргументов.

по адресу

Пример:

```
public int Calculate( int a, ref int b, out int c, params int[] d ) { …
```

параметр - значение

параметр -ссылка

выходной параметр

параметр -массив

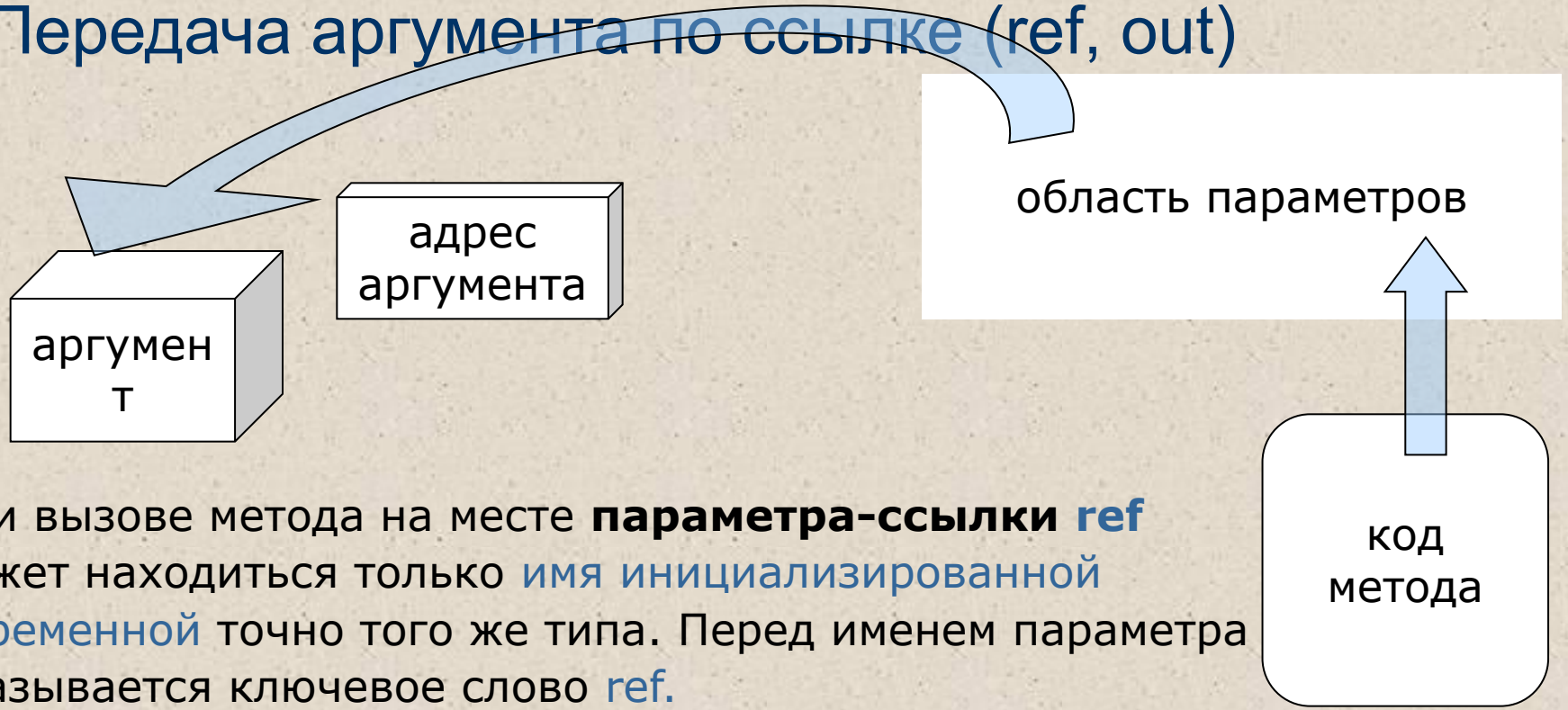
Передача аргумента по значению

- При вызове метода на месте параметра, передаваемого по значению, может находиться выражение (а также его частные случаи — переменная или константа).
- Должно существовать неявное преобразование типа выражения к типу параметра.

```
double a = 0.1;
double b = Math.Sin(a);
double c = Math.Sin(b-2*a);
```

```
static int Max(int a, int b)
{ … }
…
int x = Max(3, z);
```

Передача аргумента по ссылке (ref, out)



- При вызове метода на месте **параметра-ссылки ref** может находиться только имя инициализированной переменной точно того же типа. Перед именем параметра указывается ключевое слово ref.
- При вызове метода на месте **выходного параметра out** может находиться только имя переменной точно того же типа. Ее инициализация не требуется. Перед именем параметра указывается ключевое слово out.

```
int SomeMethod(ref int a, out int b)
{ … }
…
int s = 0; int z;
int x = SomeMethod(ref s, out z);
```

Пример: параметры-значения и ссылки ref

```
using System;
namespace ConsoleApplication1
{   class Class1
    {   static void P( int a, ref int b )
        {
            a = 44; b = 33;
            Console.WriteLine( "внутри метода {0} {1}", a, b );
        }
        static void Main()
        {
            int a = 2, b = 4;
            Console.WriteLine( "до вызова     {0} {1}", a, b );
            P( a, ref b );
            Console.WriteLine( "после вызова  {0} {1}", a, b );
}}}
```

Результат работы программы:

```
до вызова      2 4
внутри метода 44 33
после вызова   2 33
```

Пример: выходные параметры out

```
using System;
namespace ConsoleApplication1
{   class Class1
    {   static void P( int x, out int y )
        {
            x = 44; y = 33;
            Console.WriteLine( "внутри метода {0} {1}", x, y );
        }
        static void Main()
        {
            int a = 2, b;         // инициализация b не требуется

            P( a, out b );
            Console.WriteLine( "после вызова  {0} {1}", a, b );
}}}
```

Результат работы программы:

внутри метода 44 33
после вызова 2 33

Summary: Правила применения параметров

1. Для параметров-значений используется передача по значению. Этот способ применяется для исходных данных метода.

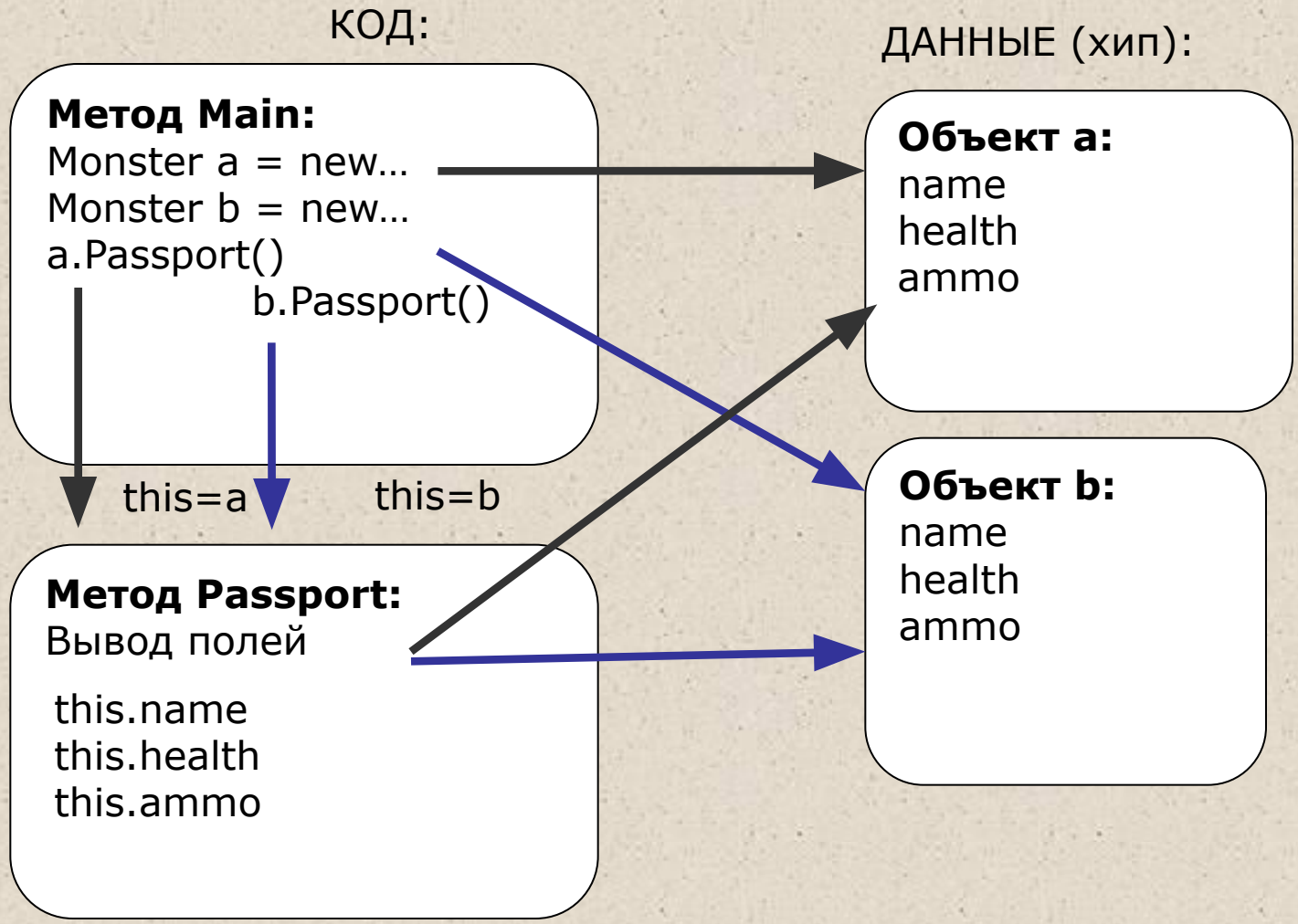
- При вызове метода на месте параметра, передаваемого по значению, может находиться выражение (а также его частные случаи — переменная или константа). Должно существовать неявное преобразование типа выражения к типу параметра.

2. Параметры-ссылки и выходные параметры передаются по адресу. Этот способ применяется для передачи побочных результатов метода.

- При вызове метода на месте параметра-ссылки **ref** может находиться только имя инициализированной переменной точно того же типа. Перед именем параметра указывается ключевое слово ref.
- При вызове метода на месте выходного параметра **out** может находиться только имя переменной точно того же типа. Ее инициализация не требуется. Перед именем параметра указывается ключевое слово out.

Ключевое слово this

Чтобы обеспечить работу метода с полями того объекта, для которого он был вызван, в метод автоматически передается скрытый параметр this, в котором хранится ссылка на вызвавший функцию объект.



Использование явного this

В явном виде параметр this применяется:

1) чтобы возвратить из метода ссылку на вызвавший объект:

```
class Demo
{       double y;
        public Demo T()    { return this; }

// 2) для идентификации поля, если его имя совпадает с
//      именем параметра метода:

        public void Sety( double y ) { this.y = y; }
}
```

Конструкторы

Конструктор – особый вид метода, предназначенный для инициализации **объекта** (конструктор экземпляра) или **класса** (статический конструктор).

Конструктор экземпляра инициализирует данные экземпляра, конструктор класса — данные класса.

Конструкторы экземпляра

Конструктор вызывается автоматически при создании объекта класса с помощью операции new. Имя конструктора совпадает с именем класса.

Свойства конструкторов:

- Конструктор не возвращает значение, даже типа void.
- Класс может иметь несколько конструкторов с разными параметрами для разных видов инициализации.
- Если программист не указал ни одного конструктора или какие-то поля не были инициализированы, полям значимых типов присваивается нуль, полям ссылочных типов — значение null.
- Конструктор, вызываемый без параметров, называется **конструктором по умолчанию**.

Сквозной пример класса

```
class Monster {
   public Monster()   // конс
   {
         name  = "Noname";
         health = 100;
         ammo = 100;
   }
  public Monster( string name ) : this()
   {
         this.name = name;
    }
  public Monster( int health, int ammo,
    string name )
    {
        this.name  = name;
        this.health = health > 0 ? health : 0 ;
        this.ammo = ammo > 0 ? ammo : 0 ;
    }
  public string GetName()     // метод
   {  return name;   }
  public int GetAmmo()       // метод
    {  return ammo;}
```

```
public int Health {                  // свойство
                                                  // метод
    {   Console.WriteLine(
  "Monster {0} \t health = {1} \
    ammo = {2}", name, health, ammo );
      }

public override string ToString(){
     string buf = string.Format(
       "Monster {0} \t health = {1} \
     ammo = {2}", name, health, ammo);
    return buf;  }

  string name;
  int health, ammo;
}
```

```
Monster Vasia = new Monster();
Monster Petya = new Monster("Петя");
Monster Masha = new Monster(150, 3000,
"Мария");
```

Свойства

- Свойства служат для организации доступа к полям класса. Как правило, свойство определяет методы доступа к закрытому полю.
- Свойства обеспечивают разделение между внутренним состоянием объекта и его интерфейсом.
- Синтаксис свойства:

[спецификаторы] тип имя_свойства

{

[get код_доступа]

[set код_доступа]

}

При обращении к свойству автоматически вызываются указанные в нем блоки чтения (**get)** и установки (**set)**.

- Может отсутствовать либо часть get, либо set, но не обе одновременно. Если отсутствует часть set, свойство доступно только для чтения (read-only), если отсутствует get - только для записи (write-only).

Пример: счетчик (свойства)

```
class Counter
{       public Counter() { }
        public Counter( int n ) { this.n = n > 0 ? n : 0; }
        public int N
        {  get { return n; }
           set { n = value > 0 ? value : 0; }
        }
// или: set { if (value > 0) n = value; else throw new Exception();}
        int n;        // поле, связанное со свойством N
}
class Program
{     static void Main(string[] args)
        {    Counter num = new Counter();
             num.N = 5;          //   работает set
             int a = num.N;      //   работает get
             num.N++;            //   работает get, а потом set
             ++num.N;            //   работает get, а потом set
}}
```

Сквозной пример класса: свойства

```
class Monster {
    public Monster()      // конструктор
    {
          this.name  = "Noname";
          this.health = 100;
          this.ammo = 100;
    }
  public Monster( string name ) : this()
    {
          this.name = name;
     }
  public Monster( int health, int ammo,
     string name )
     {
          this.name  = name;
          this.health = health;
          this.ammo = ammo;
     }
  public string GetName()     // метод
    {  return name;   }
  public int GetAmmo()        // метод
     {  return ammo;}
```

```
public int Health {          // свойство
   get { return health;  }
   set {health = value > 0 ? value : 0;
         }
 }
public string Name {         // свойство
    get { return name; }
}
 public void Passport()              // метод
     {   Console.WriteLine(
  "Monster {0} \t health = {1} \
    ammo = {2}", name, health, ammo );
      }
public override string ToString(){
     string buf = string.Format(
       "Monster {0} \t health = {1} \
     ammo = {2}", name, health, ammo);
    return buf;  }

  string name;
  int health, ammo;
}
```

Перегрузка методов

- Использование нескольких методов с одним и тем же именем, но различными типами параметров называется *перегрузкой методов*.
- Компилятор определяет, какой именно метод требуется вызвать, по типу фактических параметров. Это называется *разрешением* (resolution) перегрузки.

```
                    // Возвращает наибольшее из двух целых:
int max( int a, int b )
                   // Возвращает наибольшее из трех целых:
int max( int a, int b, int c )
                  // Возвращает наибольшее из первого параметра
                  // и длины второго:
int max ( int a, string b )
...
Console.WriteLine( max( 1, 2 ) );
Console.WriteLine( max( 1, 2, 3 ) );
Console.WriteLine( max( 1, "2" ) );
```

- Перегрузка методов является проявлением *полиморфизма*

Операции класса

- В C# можно переопределить для своих классов действие большинства операций. Это позволяет применять экземпляры объектов в составе выражений аналогично переменным стандартных типов:

```
MyObject a, b, c;  ...
c = a + b;                  // операция сложения класса MyObject
```

- Определение собственных операций класса называют ***перегрузкой операций***.
- Операции класса описываются с помощью методов специального вида (***функций-операций***):

public static имя_класса operator операция(параметры) {… }

Пример: public static MyObject operator --(MyObject m) { … }

В C# три вида операций класса: унарные, бинарные и операции преобразования типа.

Пример: счетчик (операция ++)

```
class Counter
{       public Counter() { }
        public Counter( int n ) { this.n = n > 0 ? n : 0; }
        public static Counter operator ++(Counter param)
        {
            Counter temp = new Counter(param.n + 1);
            return temp;
        }
        int n;
}
class Program
{       static void Main(string[] args)
        {    Counter num = new Counter();
             num++;    ++num;
             ...
        }
}
```

Пример: счетчик (операция +)

```
class Counter
{     ...
       public static Counter operator +(Counter param, int delta)
      {  Counter temp = new Counter(param.n + delta);
         return temp;
      }
      public static Counter operator +(int delta, Counter param)
      {  Counter temp = new Counter(param.n + delta);
         return temp;
      }
      int n;
}
class Program
{     static void Main(string[] args)
     {    Counter num = new Counter(); Counter num2 = new Counter();
          num2 = num + 3;      num2 = 3 + num;
           ...
      }
```

Проектирование класса

Summary

Интерфейс класса

- При создании класса следует хорошо продумать его *интерфейс* — средства работы с классом, доступные использующим его программистам.
- Интерфейс хорошо спроектированного класса интуитивно ясен, непротиворечив и обозрим. Как правило, он не должен включать поля данных.
- В идеале *интерфейс должен быть полным (*предоставлять возможность выполнять любые разумные действия с классом) и *минимально необходимым* (без дублирования и пересечения возможностей методов).

Состав класса

- Как правило, класс как тип, определенный пользователем, должен содержать скрытые (private) поля и следующие функциональные элементы:
  - *конструкторы*, определяющие, как инициализируются объекты класса;
  - набор *методов и свойств*, реализующих характеристики класса;
  - классы *исключений*, используемые для сообщений об ошибках путем генерации исключительных ситуаций.
  - Классы, моделирующие математические или физические понятия, обычно также содержат набор *операций*, позволяющих копировать, присваивать, сравнивать объекты и производить с ними другие действия, требующиеся по сути класса.

Элементы класса

- Методы определяют поведение класса. *Каждый метод класса должен решать только одну задачу.*
- *Создание любого метода следует начинать с его интерфейса (*заголовка). Необходимо четко представлять себе, какие параметры метод должен получать и какие результаты формировать. Входные параметры обычно перечисляют в начале списка параметров.
- Поля, характеризующие класс в целом, следует описывать как *статические*.
- Все литералы, связанные с классом, описываются как поля-константы с именами, отражающими их смысл.
- Необходимо стремиться к максимальному сокращению области действия каждой переменной. Это упрощает отладку программы, поскольку ограничивает область поиска ошибки.

Типы и структуры данных

Перечислимый тип данных

- *Перечисление* — отдельный тип-значение, содержащий совокупность именованных констант.
- Пример:

```
enum Color : long
{
    Red,
    Green,
    Blue
}
```

*Базовый класс - **System.Enum.***

*Перечисление может иметь **модификатор** (new, public, protected, internal, private). Он имеет такое же значение, как и при объявлении классов.*

- Каждый элемент перечисления имеет связанное с ним константное значение, тип которого определяется базовым типом перечисления.
- Базовые типы: byte, sbyte, short, ushort, int, uint, long и ulong. По умолчанию – int.

Массивы

- *Массив* — ограниченная совокупность однотипных величин
- Элементы массива имеют одно и то же имя, а различаются по порядковому номеру (*индексу*)
- **Виды** массивов в C#:
  - одномерные
  - многомерные (например, двумерные, или прямоугольные)
  - массивы массивов (др. термины: невыровненные, ступенчатые).

Создание массива

- **Массив относится к ссылочным типам данных** (располагается в хипе), поэтому *создание массива* начинается с выделения памяти под его элементы.
- *Элементами массива* могут быть величины как значимых, так и ссылочных типов (в том числе массивы), например:

```
int[] w = new int[10];                  // массив из 10 целых чисел
string[] z = new string[100];           // массив из 100 строк
Monster [] s = new Monster[5];          // массив из 5 монстров
double[,] t = new double[2, 10];        // прямоуг. массив 2x10
int[,,,] m = new int[2,2,2,2];          // 4-хмерный массив
int[][][] a = new int[2][][]; …         // массив массивов массивов
```

- Массив значимых типов хранит значения, массив ссылочных типов — ссылки на элементы.
- Всем элементам при создании массива присваиваются *значения по умолчанию*: нули для значимых типов и null для ссылочных.

Одномерные массивы

- Варианты описания массива:

тип[] имя;

тип[] имя = new тип [размерность];

тип[] имя = { список_инициализаторов };

тип[] имя = new тип [] { список_инициализаторов };

тип[] имя = new тип [размерность] { список_инициализаторов };

- Примеры описаний (один пример на каждый вариант описания, соответственно):

```
int[] a;                          //  память под элементы не выделена
int[] b = new int[4];                          //  элементы равны 0
int[] c = { 61, 2, 5, -9 };                    //  new подразумевается
int[] d = new int[] { 61, 2, 5, -9 };   // размерность вычисляется
int[] e = new int[4] { 61, 2, 5, -9 };       // избыточное описание
```

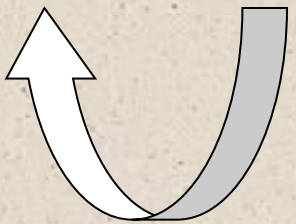
Оператор foreach (упрощенно)

- Применяется для перебора элементов массива. Синтаксис:

foreach (тип *имя* **in** имя_массива **) тело_цикла**

- *имя* задает локальную по отношению к циклу переменную, которая будет по очереди принимать все значения из массива, например:

```
int[] massiv =  { 24, 50, 18, 3, 16, -7, 9, -1 };
foreach ( int x in massiv ) Console.WriteLine( x );
```



Программа в true style ☺

```
class Mas_1                    // класс для работы с 1-мерным массивом
{
    int[] a = { 3, 12, 5, -9, 8, -4 };          // для простоты слайда

    public void PrintMas()                              // вывод массива
    {  Console.Write("Массив: ");
       foreach (int elem in a) Console.Write("  " + elem);
       Console.WriteLine();
    }

    public long SumOtr()        // сумма отрицательных элементов
    {
       long sum_otr = 0;
       foreach (int elem in a)
           if (elem < 0) sum_otr += elem;
       return sum_otr;
    }
```

```
public int NumOtr()                    // кол-во отрицательных элементов
      {
         int num_otr = 0;
         foreach (int elem in a)
            if (elem < 0) ++num_otr;
         return num_otr;
      }

public int MaxElem()                   // максимальный элемент
      {
         int max = a[0];
         foreach (int elem in a) if (elem > max) max = elem;
         return max;
      }
}
```

```
class Program                                   // класс-клиент
{   static void Main(string[] args)
    {
      Mas_1 mas = new Mas_1();
      mas.PrintMas();

      long sum_otr = mas.SumOtr();
      if (sum_otr != 0) Console.WriteLine("Сумма отриц. = " + sum_otr);
      else Console.WriteLine("Отриц-х эл-тов нет");

     int num_otr = mas.NumOtr();
     if (num_otr != 0) Console.WriteLine("Кол-во отриц. = " + num_otr);
     else Console.WriteLine("Отриц-х эл-тов нет");

     Console.WriteLine("Макс. элемент = " + mas.MaxElem());
   }
}
```

Пример анализа задания

Найти среднее арифметическое элементов, расположенных между минимумом и максимумом

- Варианты результата:
  - выводится среднее арифметическое
  - выводится сообщение «таких элементов нет» (мин. и макс. рядом или все элементы массива одинаковы)
- Вопрос: если макс. или мин. эл-тов несколько?
- Варианты тестовых данных:
  - минимум левее максимума
  - наоборот
  - рядом
  - более одного мин/макс
  - все элементы массива равны
  - все элементы отрицательные

Использование методов класса Array

```
static void Main()
        {
            int[] a =  { 24, 50, 18, 3, 16, -7, 9, -1 };
            PrintArray( "Исходный массив:", a );
            Console.WriteLine( Array.IndexOf( a, 18 ) );
            Array.Sort(a);     // Array.Sort(a, 1, 5);
            PrintArray( "Упорядоченный массив:", a );
            Console.WriteLine( Array.BinarySearch( a, 18) );
            Array.Reverse(a);     // Array.Reverse(a, 2, 4);
        }
public static void PrintArray( string header, int[] a ) {
            Console.WriteLine( header );
            for ( int i = 0; i < a.Length; ++i )
                Console.Write( "\t" + a[i] );
            Console.WriteLine();
        }
```

Что вы должны уметь найти в массиве:

- минимум/максимум [по модулю]
- номер минимума/максимума [по модулю]
- номер первого/второго/последнего положительного/отрицательного/нулевого эл-та
- сумма/произведение/количество/сред. арифм-е положительных/отрицательных/нулевых эл-тов

- упорядочить массив НЕ методом пузырька.

- анализировать все возможные варианты расположения исходных данных

Прямоугольные массивы

- *Прямоугольный массив* имеет более одного измерения. Чаще всего в программах используются двумерные массивы. Варианты описания двумерного массива:

тип[,] имя;

тип[,] имя = new тип [разм_1, разм_2];

тип[,] имя = { список_инициализаторов };

тип[,] имя = new тип [,] { список_инициализаторов };

тип[,] имя = new тип [разм_1, разм_2] { список_инициализаторов };

- Примеры описаний (один пример на каждый вариант описания):

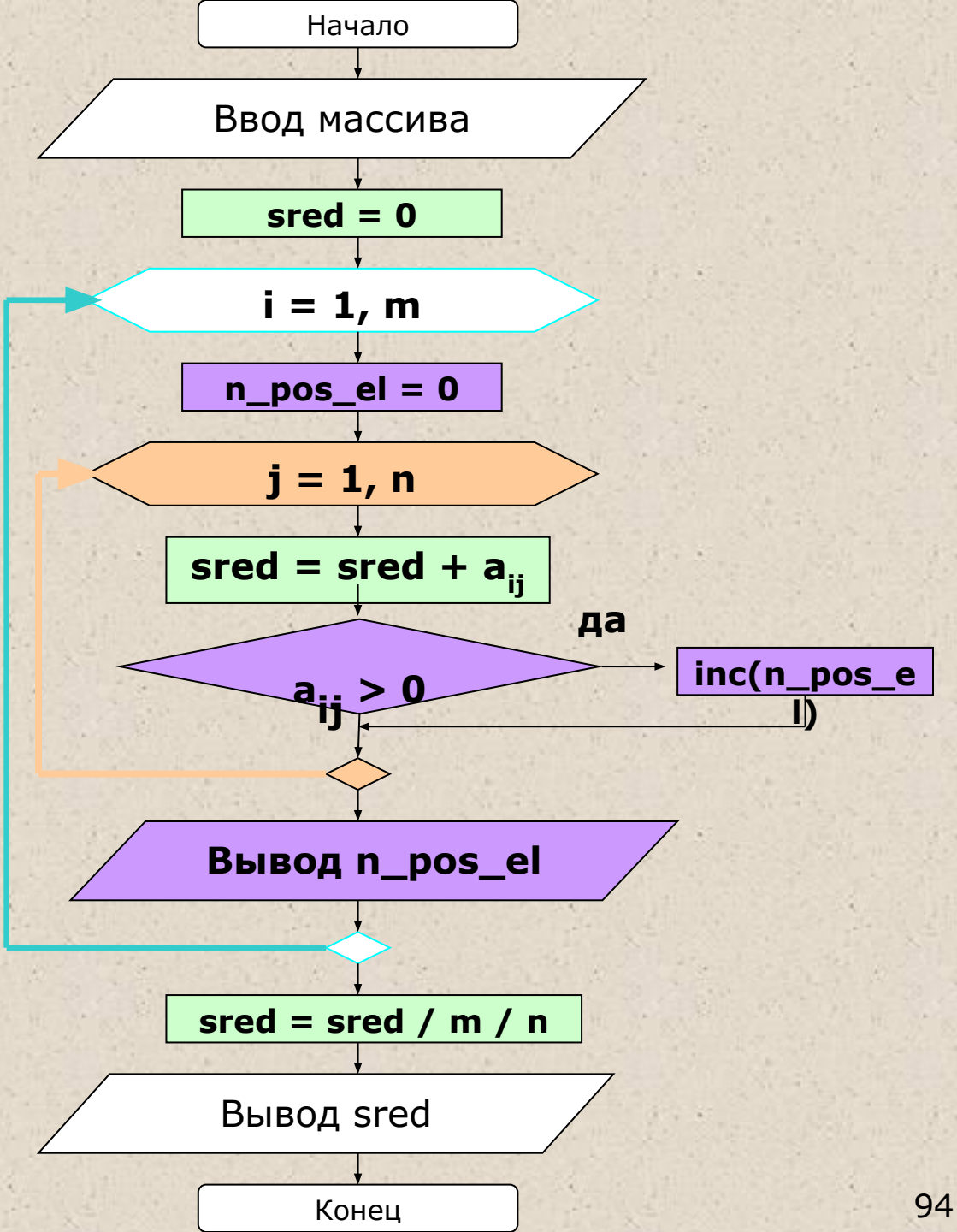
```
int[,] a;                                       // элементов нет
int[,] b = new int[2, 3];                       // элементы равны 0
int[,] c = {{1, 2, 3}, {4, 5, 6}};              // new подразумевается
int[,] c = new int[,] {{1, 2, 3}, {4, 5, 6}};   // разм-сть вычисляется
int[,] d = new int[2,3] {{1, 2, 3}, {4, 5, 6}}; // избыточное описание
```

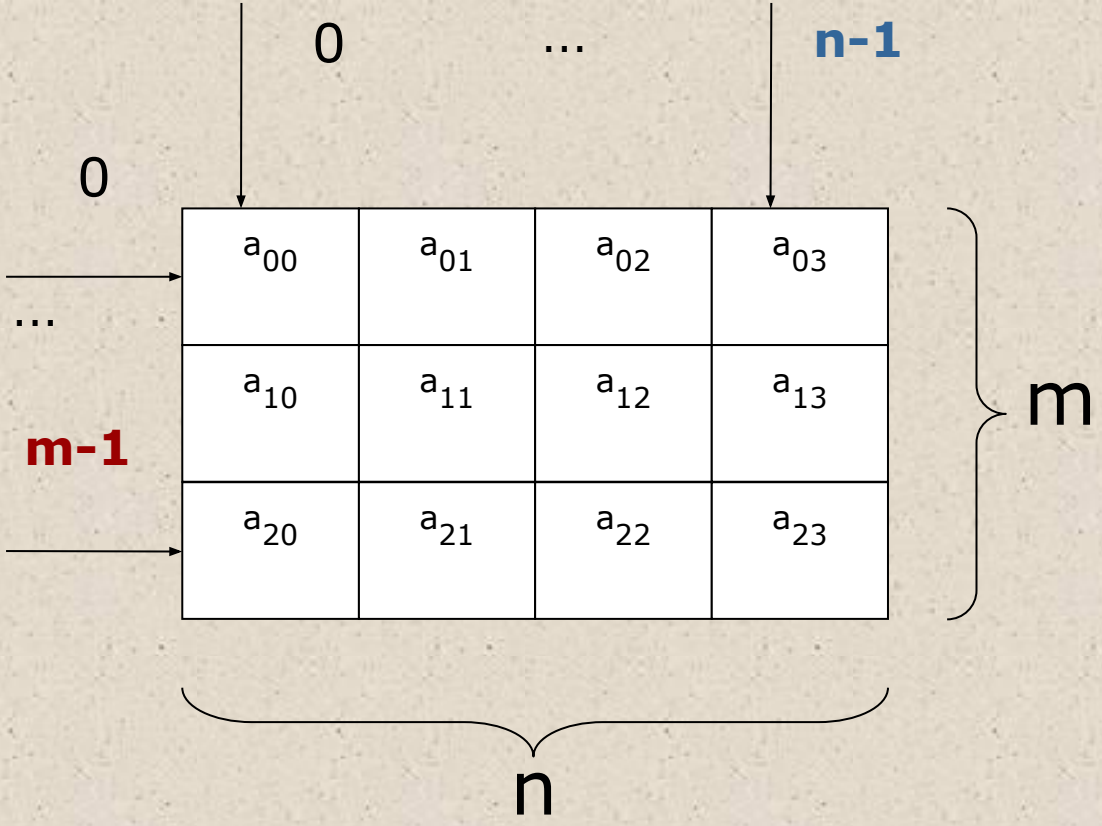
Пример

Программа определяет:

- среднее арифметическое всех элементов;
- количество положительных элементов в каждой строке

для целочисленной матрицы размером 3 x 4






```
const int m = 3, n = 4;
int[,] a = new int[m, n] {
            { 2,-2, 8, 9 },
            {-4,-5, 6,-2 },
            { 7, 0, 1, 1 }
        };
Console.WriteLine( "Исходный массив:" );
for ( int i = 0; i < m; ++i )
{   for ( int j = 0; j < n; ++j )
        Console.Write( "\t" + a[i, j] );
    Console.WriteLine();
}
```

```
int nPosEl;
for ( int i = 0; i < m; ++i )
{
    nPosEl = 0;
    for ( int j = 0; j < n; ++j )
        if ( a[i, j] > 0 ) ++nPosEl;
    Console.WriteLine( "В строке {0} {1} положит-х эл-в", i, nPosEl);
}

double sum = 0;
foreach ( int x in a ) sum += x;  // все элементы двумерного массива!
Console.WriteLine( "Среднее арифметическое всех элементов: "
                        + sum / m / n );
```

Строки типа string

Тип string предназначен для работы со строками символов в кодировке Unicode. Ему соответствует базовый класс System.String библиотеки .NET.

Создание строки:

```
1. string s;                                  // инициализация отложена
2. string t = "qqq";   // инициализация строковым литералом
3. string u = new string(' ', 20);            // с пом. конструктора
4. string v = new string( a ); // создание из массива символов
// создание массива символов:  char[] a = { '0', '0', '0' };
```

Операции для строк

- присваивание (=);
- проверка на равенство (==);
- проверка на неравенство (!=);
- обращение по индексу ([]);
- сцепление (конкатенация) строк (+).

- Строки равны, если имеют одинаковое количество символов и совпадают посимвольно.
- Обращаться к отдельному элементу строки по индексу можно только для получения значения, но не для его изменения. Это связано с тем, что строки типа string относятся к неизменяемым типам данных.
- Методы, изменяющие содержимое строки, на самом деле создают новую копию строки. Неиспользуемые «старые» копии автоматически удаляются сборщиком мусора.

Пример: разбиение текста на слова

```
StreamReader inputFile = new StreamReader("example.txt");
string text = inputFile.ReadToEnd();

char[] delims = ".,;:!?\n\xD\xA\" ".ToCharArray();
string[] words = text.Split(delims,
                            StringSplitOptions.RemoveEmptyEntries);
foreach (string word in words) Console.WriteLine(word);

Console.WriteLine("Слов в тексте: " + words.Length);

// слова, оканчивающиеся на «a»:
foreach (string word in words)
    if (word[word.Length-1] == 'a') Console.WriteLine(word);
```

Регулярные выражения

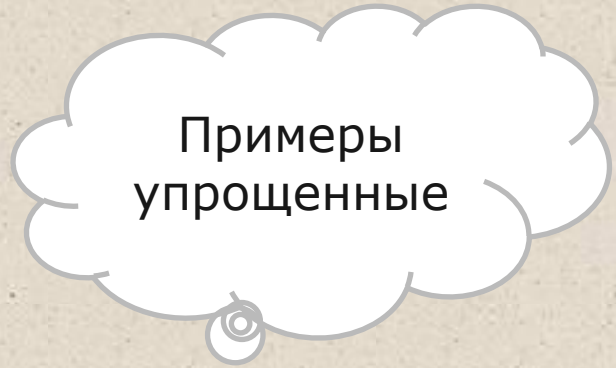
Регулярное выражение — шаблон (образец), по которому выполняется поиск соответствующего ему фрагмента текста.

- тег html:

```
<[^>]+>
```

- российский номер автомобиля:

```
[A-Z]\d{3}[A-Z]{2}\d\dRUS
```



- IP-адрес:

```
\d\d?\d?\.\d\d?\d?\.\d\d?\d?\.\d\d?\d?
```

Регулярные выражения предназначены для обработки текстовой информации и обеспечивают:

- эффективный **поиск** в тексте по заданному шаблону;
- **редактирование**, замену и удаление подстрок;
- формирование итоговых **отчетов** по результатам работы с текстом.

Язык описания регулярных выражений

Язык описания регулярных выражений состоит из символов двух видов: обычных и метасимволов.

- **Обычный символ** представляет в выражении сам себя.
- **Метасимвол**:
  - **класс символов** (например, любая цифра **\d** или буква **\w**)
  - **уточняющий символ** (например, **^**).
  - **повторитель** (например, **+**).

Примеры:

- выражение для поиска в тексте фрагмента «Вася» записывается с помощью четырех обычных символов «**Вася**»
- выражение для поиска двух цифр, идущих подряд, состоит из двух метасимволов «**\d\d**»
- выражение для поиска фрагментов вида «Вариант 1», «Вариант 2», …, «Вариант 9» имеет вид «**Вариант \d**»
- выражение для поиска фрагментов вида «Вариант 1», «Вариант 23», «Вариант 719», …, имеет вид «**Вариант \d+**»

Метасимволы - классы символов

Класс символов	Описание	Пример
.	любой символ, кроме \n	**c.t** соответствует фрагментам cat, cut, c1t, c{t и т.д.
[ ]	любой одиночный символ из последовательности внутри скобок.	**c[au1]t** соответствует фрагментам cat, cut и c1t. **c[a-z]t** соответствует фрагментам cat, cbt, cct, cdt, …, czt
[^]	любой одиночный символ, не входящий в последовательность внутри скобок.	**c[^au1]t** соответствует фрагментам cbt, c2t, cXt и т.д. **c[^a-zA-Z]t** соответствует фрагментам cит, c1t, cЧt, c3t и т.д.
\w	любой алфавитно-цифровой символ, то есть символ из множества прописных и строчных букв и десятичных цифр	**c\wt** соответствует фрагментам cat, cut, c1t, cЮt и т.д. Не соответствует c{t, c;t и т.д.

продолжение таблицы

`\W`	любой **не** алфавитно-цифровой символ, то есть символ, не входящий в множество прописных и строчных букв и десятичных цифр	`c\Wt` соответствует фрагментам `c{t`, `c;t`, `c t` и т.д. Не соответствует `cat`, `cut`, `c1t`, `cЮt` и т.д.
`\s`	любой пробельный символ, например, пробел, табуляция (`\t`, `\v`), перевод строки (`\n`, `\r`), новая страница (`\f`)	`\s\w\w\w\s` соответствует любому слову из трех букв, окруженному пробельными символами
`\S`	любой **не** пробельный символ, то есть символ, не входящий в множество пробельных	`\s\S\S\s` соответствует любым двум непробельным символам, окруженным пробельными.
`\d`	любая десятичная цифра	`c\dt` соответствует фрагментам `c1t`, `c2t`, …, `c9t`
`\D`	любой символ, не явдяющийся десятичной цифрой	`c\Dt` не соответствует фрагментам `c1t`, `c2t`, …, `c9t`.

Повторители

Мета-символ	Описание	Пример
*	0 или более повторений предыдущего элемента	**ca*t** соответствует фрагментам ct, cat, caat, caaaaaaaaaaat и т.д.
+	1 или более повторений предыдущего элемента	**ca+t** соответствует фрагментам cat, caat, caaaaaaaaaaat и т.д.
?	0 или 1 повторений предыдущего элемента	**ca?t** соответствует фрагментам ct и cat
{n}	ровно n повторений предыдущего элемента	**ca{3}t** соответствует фрагменту caaat. (cat){2} соответствует фрагменту catcat.
{n,}	по крайней мере n повторений предыдущего элемента	**ca{3,}t** соответствует фрагментам caaat, caaaat, caaaaaaaaaaat и т.д.
{n,m}	от n до m повторений предыдущего элемента	**ca{2,4}t** соответствует фрагментам caat, caaat и caaaat

Примеры простых регулярных выражений

- целое число (возможно, со знаком):

  [-+]?**\d**+

- вещественное число (может иметь знак и дробную часть, отделенную точкой):

  [-+]?\d+\.?\d*

- российский номер автомобиля (упрощенно):

  [A-Z]**\d**{3}**[A-Z]**{2}\d\dRUS

- ip-адрес (упрощенно):

  (\d{1,3}**\.**){3}\d{1,3}

Поддержка регулярных выражений в .NET

- Для поддержки регулярных выражений в библиотеку .NET включены классы, объединенные в пространство имен System.Text.RegularExpressions.
- Основной класс – **Regex**. Он реализует подсистему обработки регулярных выражений.
- Подсистеме требуется предоставить:
  - **Шаблон** (регулярное выражение), соответствия которому требуется найти в тексте.
  - **Текст**, который требуется проанализировать с помощью шаблона.

См.:

http://msdn.microsoft.com/ru-ru/library/hs600312.aspx?ppud=4

Методы класса Regex

позволяют выполнять следующие действия:

- Определить, **встречается ли** во входном тексте шаблон регулярного выражения (метод IsMatch).
- **Извлечь** из текста одно или все вхождения, соответствующие шаблону регулярного выражения (методы Match или Matches).
- **Заменить** текст, соответствующий шаблону регулярного выражения (метод Replace).
- **Разделить** строку на массив строк (метод Split).

Пример использования Regex.IsMatch

```
using System;
using System.Text.RegularExpressions;
public class Example
 { public static void Main()
    { string[] values = { "111-22-3333", "111-2-3333"};
      string pattern = @"^\d{3}-\d{2}-\d{4}$";
      foreach (string value in values)
       { if (Regex.IsMatch(value, pattern))
            Console.WriteLine("{0} is a valid SSN.", value);
         else Console.WriteLine("{0}: Invalid", value);
 } } }
 // Вывод:
 // 111-22-3333 is a valid SSN.
 // 111-2-3333: Invalid
```

Абстрактные структуры данных

- *Массив*

  конечная совокупность однотипных величин. Занимает непрерывную область памяти и предоставляет прямой (произвольный) доступ к элементам по индексу.
- *Линейный список*
- *Стек*
- *Очередь*
- *Бинарное дерево*
- *Хеш-таблица (ассоциативный массив, словарь)*
- *Граф*
- *Множество*

Контейнеры http://msdn.microsoft.com/ru-ru/library/ybcx56wz.aspx?ppud=4

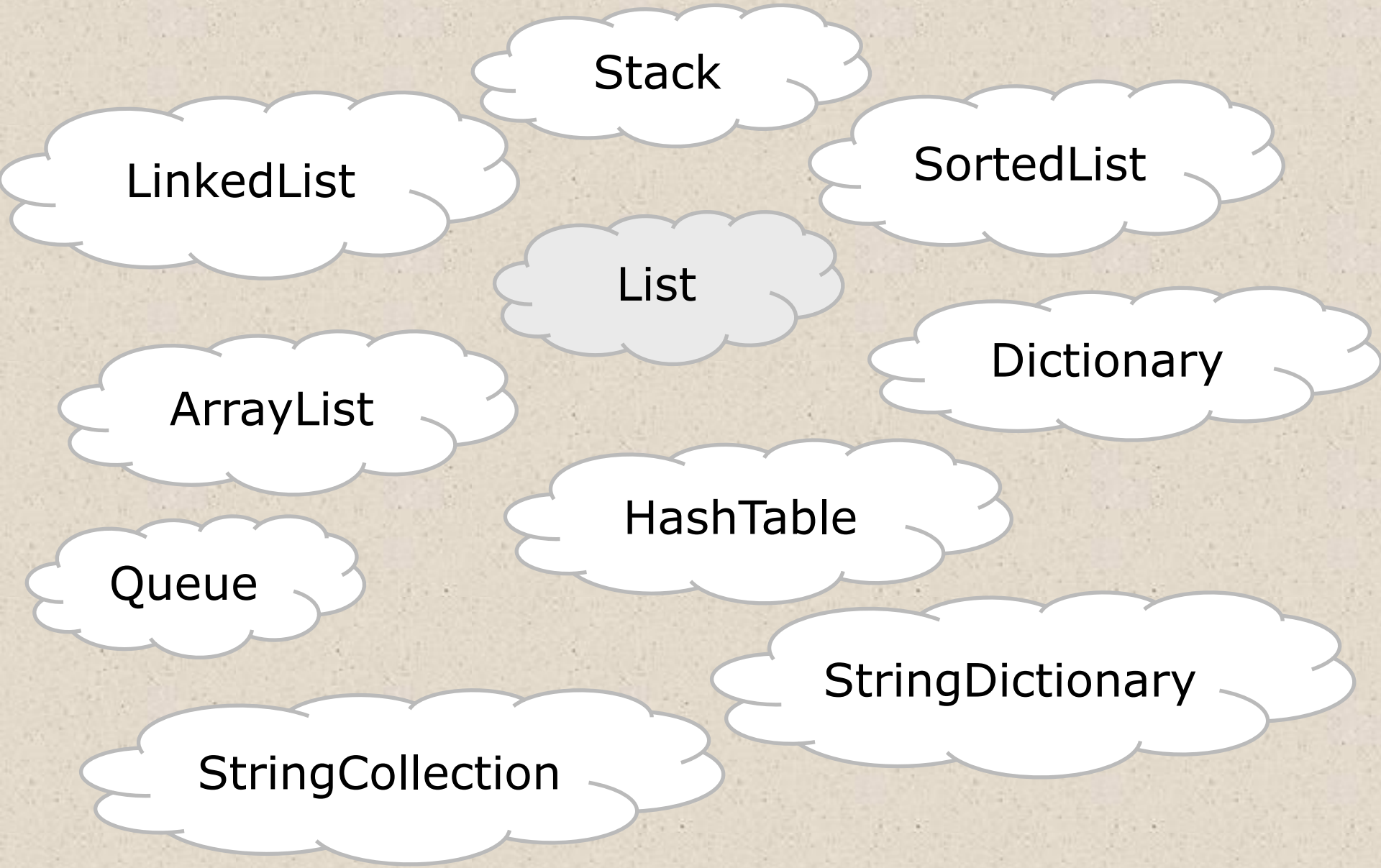
- *Контейнер (коллекция)* - стандартный класс, реализующий абстрактную структуру данных.
- Для каждого типа коллекции определены методы работы с ее элементами, не зависящие от конкретного типа хранимых данных.
- Использование коллекций позволяет сократить сроки разработки программ и повысить их надежность.
- 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

Параметризованные коллекции (классы-прототипы, generics)

- классы, имеющие типы данных в качестве параметров

Класс-прототип (версия 2.0)	**Обычный класс**
Dictionary<K,T>	HashTable
LinkedList<T>	—
List<T>	ArrayList
Queue<T>	Queue
SortedDictionary<K,T>	SortedList
Stack<T>	Stack

Повторение: контейнеры и файлы



Пример использования класса List

```
using System;
using System.Collections.Generic;
namespace ConsoleApplication1{
class Program {
        static void Main()  {
       List<int> lint = new List<int>();
            lint.Add( 5 ); lint.Add( 1 ); lint.Add( 3 );
            lint.Sort();
            int a = lint[2];
            Console.WriteLine( a );
            foreach ( int x in lint ) Console.Write( x + " ");
}}}
```

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.

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*** - очередной элемент можно прочитать (записать) только после аналогичной операции с предыдущим элементом
  - *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;
        }
        catch
        {
       Console.WriteLine( " Неопознанное исключение!" );
            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 – вар. 1

```
try {
        List<int> list_int = new List<int>();
        StreamReader file_in = new StreamReader( @"D:\FILES\1024" );
        Regex regex = new Regex( "[^0-9-+]+" );
        List<string> list_string = new List<string>(
                        regex.Split( file_in.ReadToEnd().TrimStart(' ') ) );
        foreach (string temp in list_string)
                list_int.Add( Convert.ToInt32(temp) );

        foreach (int temp in list_int) Console.WriteLine(temp);
        ...
}
catch (FileNotFoundException e)
        {  Console.WriteLine("Нет файла" + e.Message); return; }
catch (FormatException e)
        {  Console.WriteLine(e.Message); return;     }
catch { … }
```

Чтение чисел из текстового файла – вар. 2

```
try {
        StreamReader file_in = new StreamReader( @"D:\FILES\1024" );
        char[] delim = new char[] { ' ' };
        List<string> list_string = new List<string>(
                file_in.ReadToEnd().Split( delim,
                        StringSplitOptions.RemoveEmptyEntries ));
      List<int> list_int = list_string.ConvertAll<int>(Convert.ToInt32);
        foreach ( int temp in ist_int ) Console.WriteLine( temp );
        ...
}
catch (FileNotFoundException e)
        {  Console.WriteLine("Нет файла" + e.Message); return;  }
catch (FormatException e)
        {  Console.WriteLine(e.Message); return;   }
catch { … }
```

Организация справки MSDN

Для каждого элемента:

- Имя
- Назначение
- Пространство имен, сборка
- Синтаксис (Syntax)
- Описание (Remarks)
- Примеры (Examples)
- Иерархия наследования, платформы, версия, …
- Ссылки на родственную информацию (See also)