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.

Оператор «выражение»

- Любое выражение, завершающееся точкой с запятой, рассматривается как оператор, выполнение которого заключается в вычислении выражения.

```
i++;            // выполняется операция инкремента
a *= b + c;     // выполняется умножение с присваиванием
fun( i, k );    // выполняется вызов функции
```

Пустой оператор

- *пустой оператор* **;** используется, когда по синтаксису оператор требуется, а по смыслу — нет:
- while (true);

Это цикл, состоящий из пустого оператора (бесконечный)

- ;;;

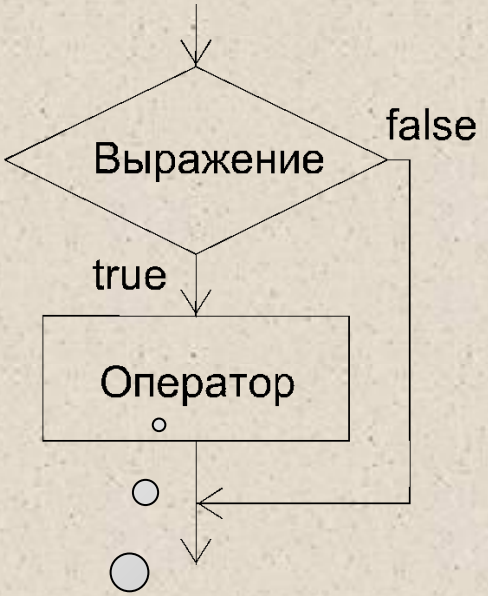
Три пустых оператора

Операторы ветвления:

- развилка (**if**)
- переключатель (**switch**)

Условный оператор if

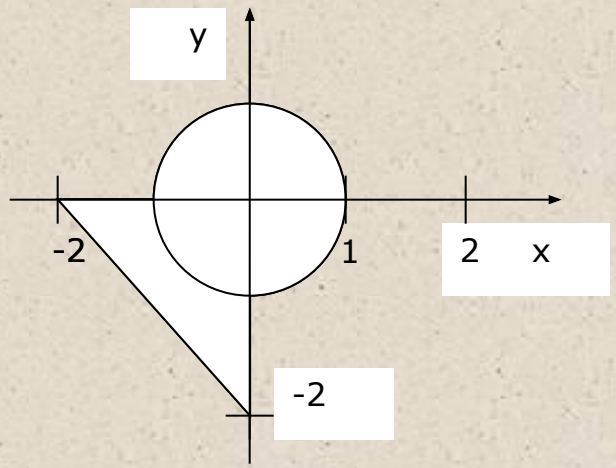
if (выражение) оператор_1;
[else оператор_2;]




```
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;
```

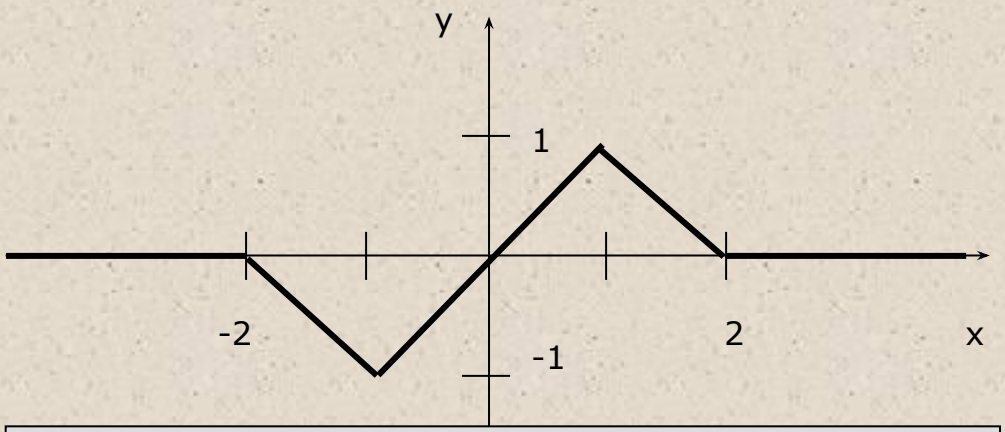
Простой или
{блок}

Пример




```
using System;
namespace ConsoleApplication1
{   class Class1
    {   static void Main()
        {
            Console.WriteLine( "Введите координату x" );
            double x = Convert.ToDouble(Console.ReadLine() );

            Console.WriteLine( "Введите координату y" );
            double y = double.Parse(Console.ReadLine() );

                    if  ( x * x + y * y <= 1 ||
                          x <= 0 && y <= 0 && y >= - x – 2 )
                        Console.WriteLine( " Точка попадает в область " );
                    else
                        Console.WriteLine( " Точка не попадает в область "
);
}}}
```

Пример 2



$$y = \begin{cases} 0, & x < -2 \\ -x - 2, & -2 \le x < -1 \\ x, & -1 \le x < 1 \\ -x + 2, & 1 \le x < 2 \\ 0, & x \ge 2 \end{cases}$$

```
if ( x < -2 )                  y = 0;
if ( x >= -2 && x < -1 ) y = -x - 2;
if ( x >= -1 && x <  1 ) y = x;
if ( x >=  1 && x <  2 ) y = -x + 2;
if ( x >=  2 )                 y = 0;
```

```
if      ( x <= -2 )  y = 0;
else if ( x < -1 )     y = -x - 2;
  else if ( x <  1 )     y = x;
    else if ( x <  2 )     y = -x + 2;
      else                   y = 0;
```

```
y = 0;
if ( x > -2 ) y = -x - 2;
if ( x > -1 ) y = x;
if ( x >  1 ) y = -x + 2;
if ( x >  2 ) y = 0;
```

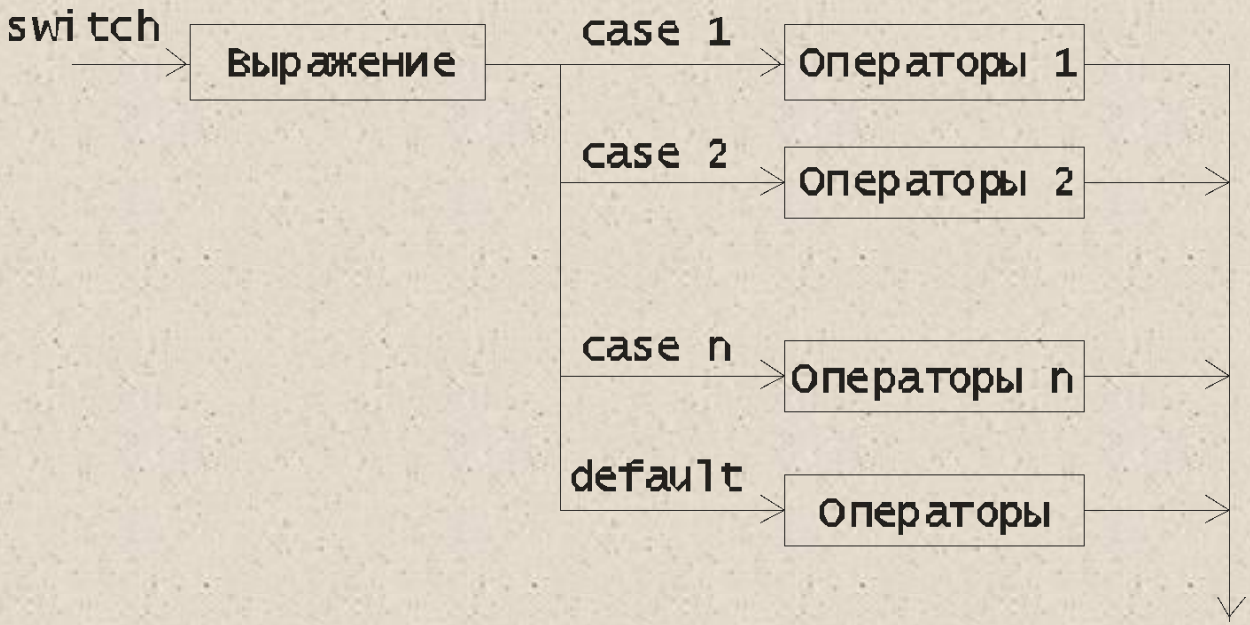
Проверка вещественных величин на равенство

- Из-за погрешности представления вещественных значений в памяти следует ее избегать, вместо этого лучше сравнивать модуль разности с некоторым малым числом.
- float a, b; …
- if (a == b) … // не рекомендуется!
- **if (Math.Abs(a - b) < 1e-6) … // надежно!**
- Значение величины, с которой сравнивается модуль разности, следует выбирать в зависимости от решаемой задачи и точности участвующих в выражении переменных.
- Снизу эта величина ограничена определенной в классах Single и Double константой **Epsilon**. Это минимально возможное значение переменной такое, что

  1.0 + Epsilon != 1.0

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 ]
}
```



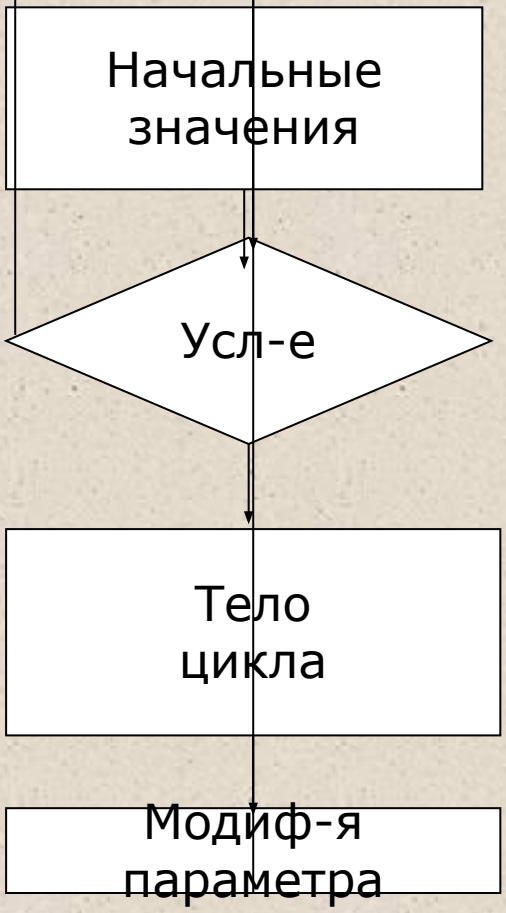
Пример: Калькулятор на четыре действия

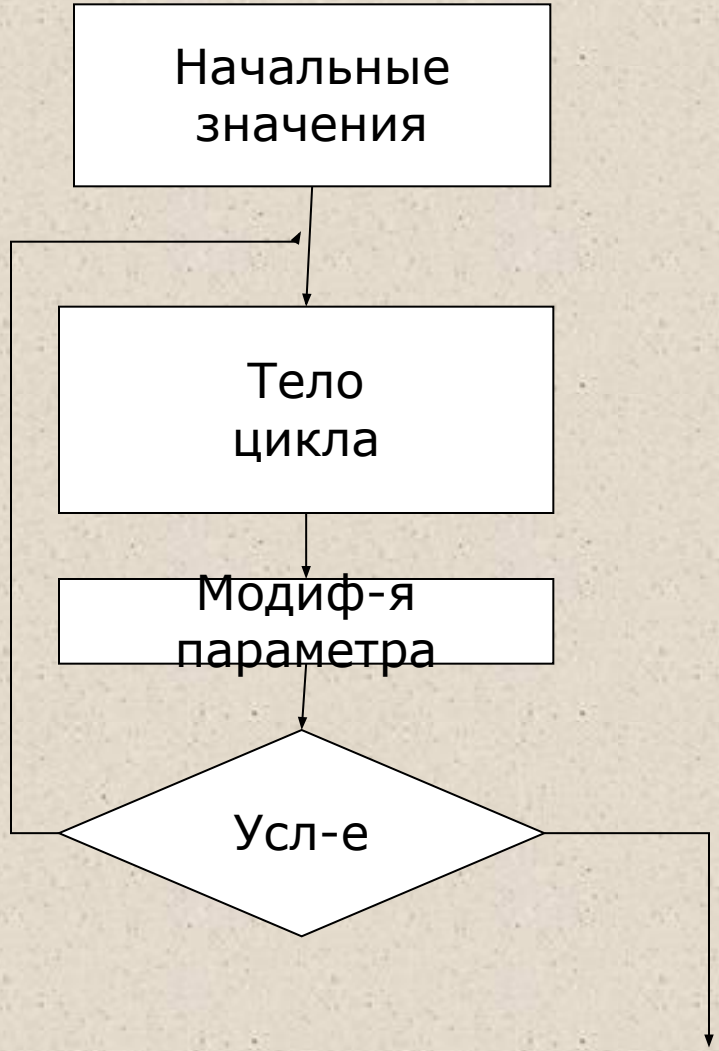
```
using System; namespace ConsoleApplication1
{   class Class1    {   static void Main()       {
        Console.WriteLine( "Введите 1й операнд:" );
         double a = double.Parse(Console.ReadLine());
         Console.WriteLine( "Введите знак" );
         char op = (char)Console.Read(); Console.ReadLine();
         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**

Структура оператора цикла

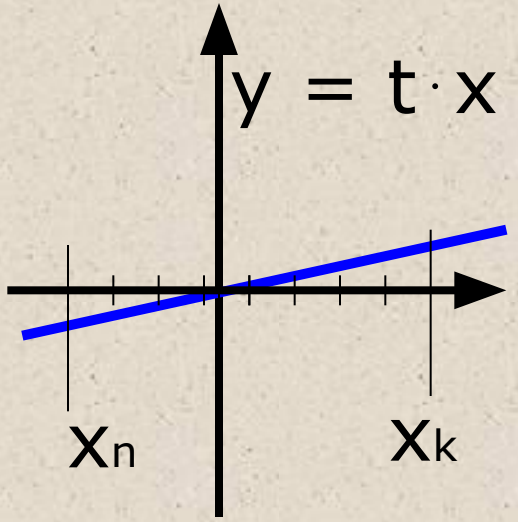




Цикл с предусловием

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' );
        }
    }
}
```

Цикл с параметром

for (инициализация; выражение; модификации) оператор;

```
int s = 0;
for ( int i = 1; i <= 100; i++ ) s += i;
for ( int i = 0, j = 20; i < 5 && j > 10; i++, j-- ) …
```

Несколько величин

Несколько величин

```
for ( double x = Xn; x <= Xk; x += dX ) { …; …; … }
```

Блок

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 );
            }
        }
    }
}
```

Рекомендации по написанию циклов

- Использовать do-while, если цикл обязательно требуется выполнить хотя бы один раз (например, при проверке ввода);
- в остальных случаях, как правило, применять for.

- Не забывать заключать в блок тело цикла, состоящее более чем из одного оператора;
- проверять, изменяется ли в теле цикла хотя бы одна переменная, входящая в условие продолжения цикла;
- предусматривать аварийный выход из *итеративного цикла** по достижению некоторого предельно допустимого количества итераций.

** цикл, количество повторений которого невозможно вычислить заранее*

Передача управления:

- break
- continue
- return
- goto
- throw

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

Написать программу вычисления значения функции sin с помощью степенного ряда с точностью ε по формуле:

$$y = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \ldots$$

$$|R_n| \le \varepsilon \qquad |C_{n+1}| \le \varepsilon$$

$$C_n = (-1)^n \frac{x^{2n-1}}{(2n-1)!} \qquad C_{n+1} = (-1)^{n+1} \frac{x^{2n+1}}{(2n+1)!}$$

$$C_{n+1} = -C_n \frac{x^2}{2n(2n+1)}$$

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.
```

Оператор return

завершает выполнение функции и передает управление в точку ее вызова:

return [выражение];

Оператор goto

1) **goto метка;**

В теле той же функции должна присутствовать ровно одна конструкция вида:

метка: оператор;

2) **goto case константное_выражение;**

3) **goto default;**

Обработка ошибок

Возможные действия при ошибке:

- прервать выполнение программы;
- возвратить значение, означающее «ошибка»;
- вывести сообщение об ошибке и вернуть вызывающей программе некоторое приемлемое значение, которое позволит ей продолжать работу;
- выбросить исключение.

Исключения генерирует либо система выполнения, либо программист с помощью оператора throw.

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!

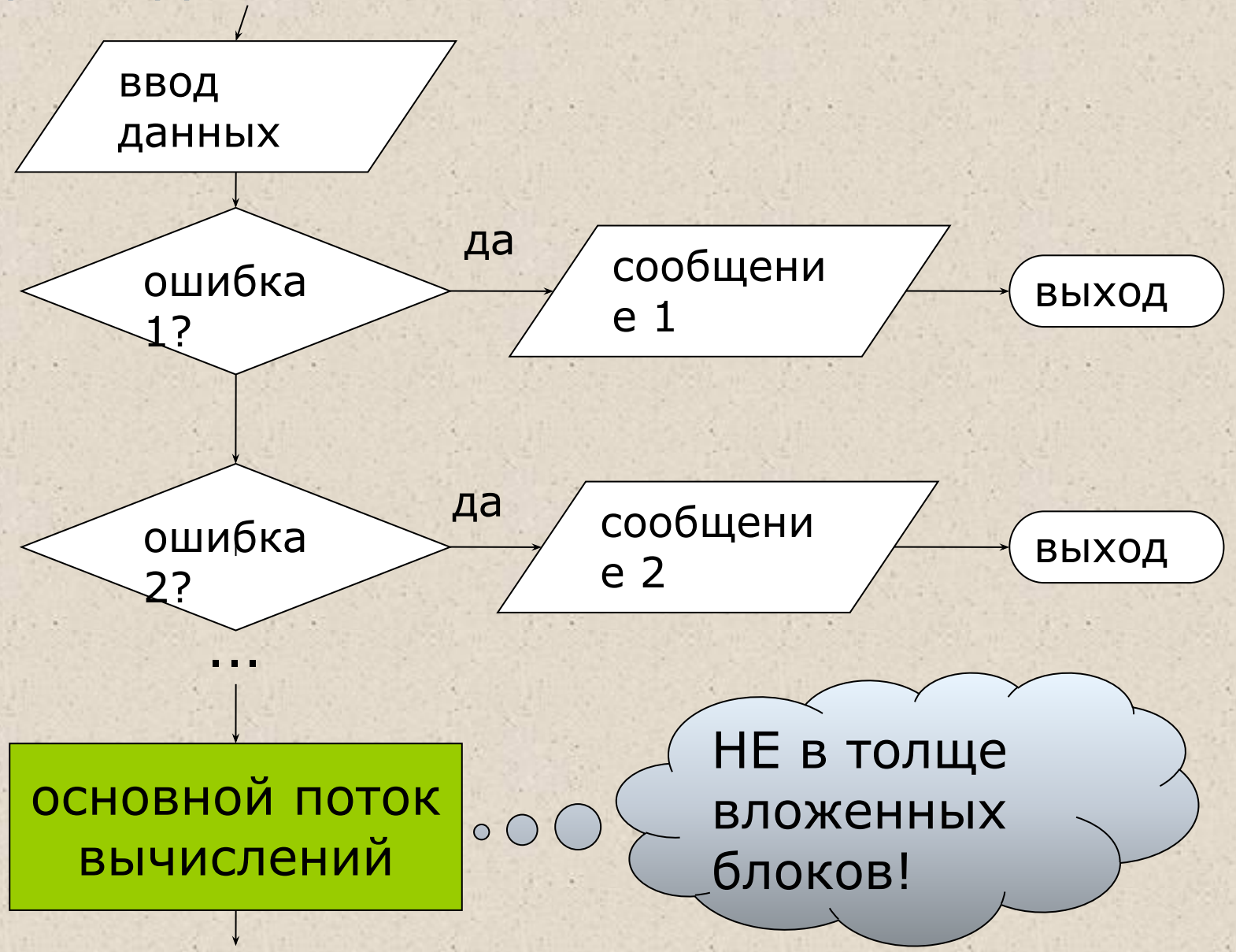
```
// пример проверки допустимости значения:

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

не хорошо!

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



Обработка исключений

Исключительная ситуация, или исключение — это возникновение непредвиденного или аварийного события, которое может порождаться некорректным использованием аппаратуры.

Например, это деление на ноль или обращение по несуществующему адресу памяти.

Исключения позволяют логически разделить вычислительный процесс на две части — обнаружение аварийной ситуации и ее обработка.

Некоторые стандартные исключения

Имя	Пояснение
ArithmeticException	Ошибка в арифметических операциях или преобразованиях (является предком DivideBeZeroException и OverFlowException)
DivideByZeroException	Попытка деления на ноль
FormatException	Попытка передать в метод аргумент неверного формата
IndexOutOfRangeException	Индекс массива выходит за границы диапазона
InvalidCastException	Ошибка преобразования типа
OutOfMemoryException	Недостаточно памяти для создания нового объекта
OverFlowException	Переполнение при выполнении арифметических операций
StackOverFlowException	Переполнение стека

Оператор try

Служит для обнаружения и обработки исключений.

Оператор содержит три части:

- *контролируемый блок* — составной оператор, предваряемый ключевым словом **try**. В контролируемый блок включаются потенциально опасные операторы программы. Все функции, прямо или косвенно вызываемые из блока, также считаются ему принадлежащими;
- один или несколько *обработчиков исключений* — блоков **catch**, в которых описывается, как обрабатываются ошибки различных типов;
- *блок завершения* **finally**, выполняемый независимо от того, возникла ли ошибка в контролируемом блоке.

Синтаксис оператора try:

try блок [catch-блоки] [finally-блок]

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
}
```

Иллюстрация генерации исключения

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; }
```

Оператор throw

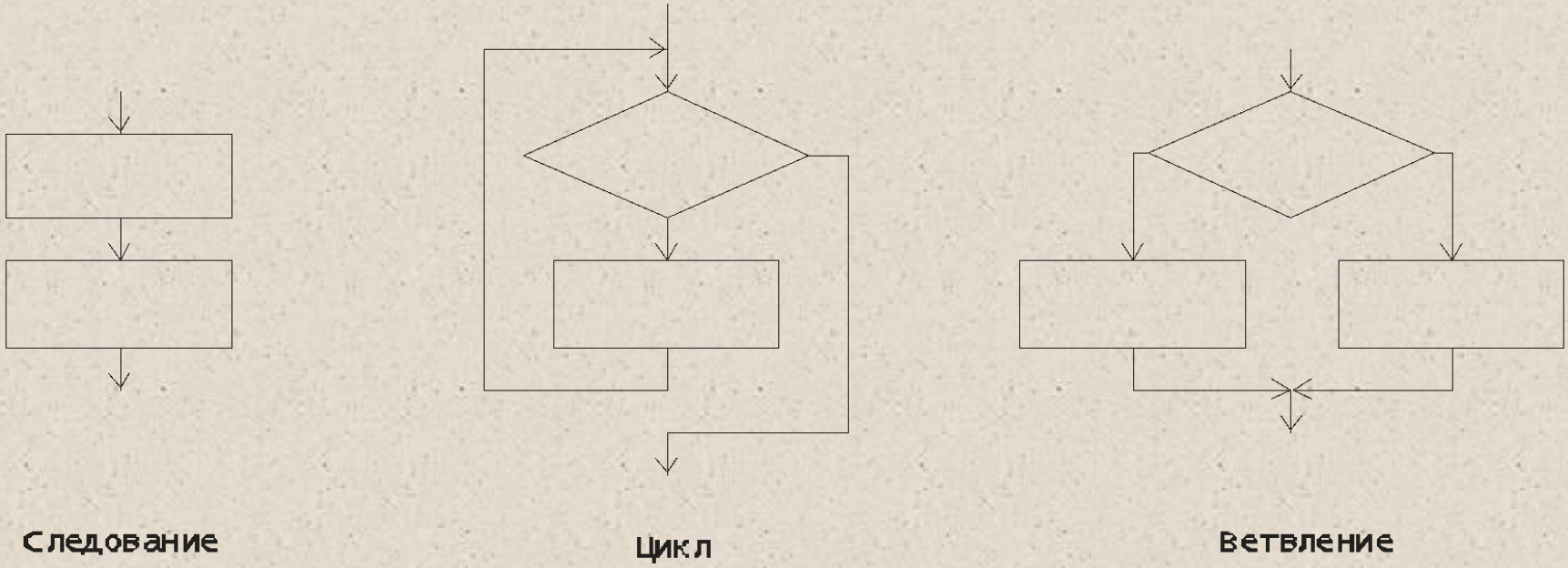
- **throw [выражение];**

Параметр должен быть объектом, порожденным от стандартного класса System.Exception. Этот объект используется для передачи информации об исключении его обработчику.

Пример:

```
throw new DivideByZeroException();
```

Базовые конструкции структурного программирования



Следование

Цикл

Ветвление

- Целью использования базовых конструкций является получение программы простой структуры. Такую программу легко читать, отлаживать и при необходимости вносить в нее изменения.
- Особенностью базовых конструкций является то, что любая из них имеет только один вход и один выход, поэтому конструкции могут вкладываться друг в друга

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.

Форматирование программы

- Вложенные блоки должны иметь отступ в 3–5 символов
- Форматируйте текст по столбцам везде, где это возможно:

```
string   buf       = "qwerty";
double   ex        = 3.1234;
int      number    = 12;
byte     z         = 0;
...
if ( done ) Console.WriteLine( "Сумма ряда - " + y );
else        Console.WriteLine( "Ряд расходится" );
...
if      ( x >= 0 && x < 10 ) y = t * x;
else if ( x >= 10 )          y = 2 * t;
else                         y = x;
```

- После знаков препинания должны использоваться пробелы:

```
f=a+b;                  // плохо! Лучше f = a + b;
```