Керов Леонид Александрович

ОПЕРАТОРЫ УПРАВЛЕНИЯ, ИСКЛЮЧЕНИЯ И ПОТОКИ В ЯЗЫКЕ C#

Аннотация

Данная статья является третьей из серии статей, посвященных изложению «нулевого уровня» языка C#. Рассматриваются операторы управления, обработка исключений и файловый ввод-вывод текстовых файлов.

Ключевые слова: язык C#, операторы управления, обработка исключений, ввод-вывод текстовых файлов.

1. ОПЕРАТОРЫ УПРАВЛЕНИЯ

Решение задач с помощью компьютера основывается на использовании алгоритмов. Алгоритм – это правило решения задачи, имеющее вид последовательности операций, выполняемых в некотором определенном порядке. Для решения простейших задач этот порядок представляет собой линейную последовательность. Для решения более сложных задач необходимы ветвления и циклы, реализацию которых и обеспечивают операторы управления.

1.1. ОПЕРАТОР IF

Оператор **if** позволяет реализовать ветвление в программе и имеет следующий вид (см. листинг 1).

Листинг 1

```
if ( условие )
  оператор_1
else
  оператор_2
```

Элемент **условие** представляет собой логическое выражение, значением которого является значение **true** или **false**. В качестве элементов **оператор_1** и **оператор_2** обычно указывается блок, который представляет собой последовательность операторов, заключенных в фигурные скобки, и имеет статус оператора. Элементы **else оператор_2** могут быть опущены. В качестве элемента **оператор_2** может быть указан оператор **if**. Семантика оператора **if** может быть определена посредством блок-схемы (см. рис. 1).

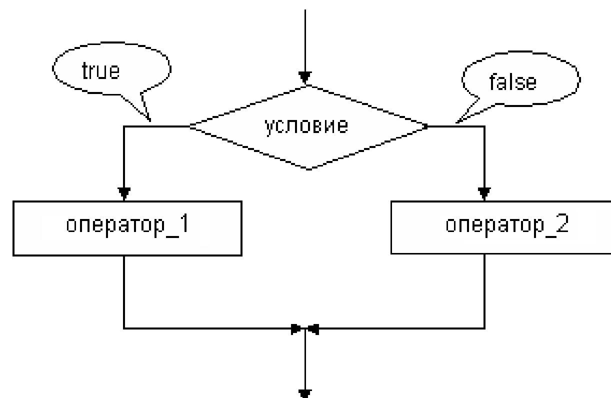

Рис. 1. Семантика оператора **if**

© Л.А. Керов, 2009

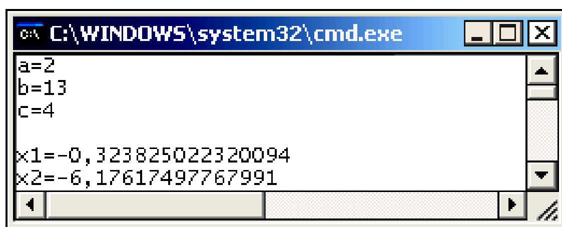

Рис. 2. Пример работы программы решения квадратного уравнения

В качестве примера использования оператора **if** приведем программу решения квадратного уравнения, параметры которого задаются случайным образом (см. листинг 2, рис. 2).

1.2. ОПЕРАТОР SWITCH

Оператор **switch** позволяет реализовать в программе ветвление по двум или более направлениям и имеет следующий вид (см. листинг 3).

Значение элемента **выражение** должно иметь тип **char**, **byte**, **short**, **int** или **string**. Тип элементов **константа_1**, **константа_2** и т. д. должен быть совместим с типом элемента **выражение**, а их значения должны быть различными.

Семантика оператора **switch** может быть определена посредством блок-схемы (см. рис. 3). Последовательно значение элемента **выражение** сравнивается со значениями констант. Выполняются операторы той ветви, для которой значения совпали. Если ни одного совпадения не было, то выполняются операторы ветви **default**.

В качестве примера использования оператора **switch** приведем программу, в которой с клавиатуры вводится значение некоторого дробного числа, а затем к нему применяется тригонометрическая функция, которая выбирается из списка предложенных вариантов (см. рис. 4).

Листинг 2

```
using System;
class P01
{
    public static void Main()
    {
        Random r = new Random();
        double a = r.Next(1, 5);
        double b = r.Next(10, 21);
        double c = r.Next(1, 5);
        double d = b * b - 4 * a * c;
        Console.WriteLine("a={0}\nb={1}\nc={2}\n", a, b, c);
        if (d < 0)
        {
            Console.WriteLine("Решений нет");
        }
        else if (d == 0)
        {
            Console.WriteLine("x={0}", -b / 2 / a);
        }
        else
        {
            Console.WriteLine("x1={0}\nx2={1}",
                (-b + Math.Sqrt(d)) / 2 / a,
                (-b - Math.Sqrt(d)) / 2 / a);
        }
    }
}
```

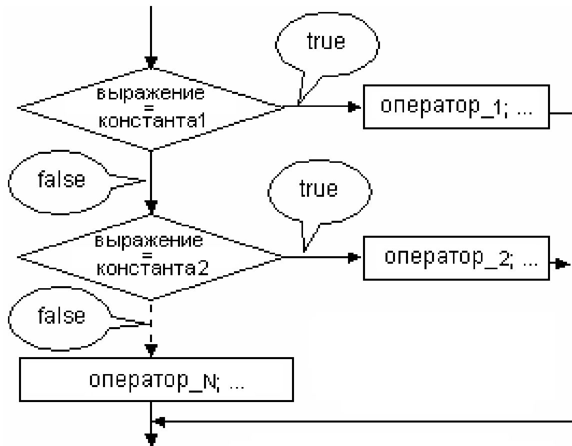

Рис. 3. Семантика оператора `switch`

Листинг 3

```
switch ( выражение )
{
    case константа_1:
         оператор_1;   ... break;
    case константа_2:
         оператор_2;   ... break;
            ...
    default:
          оператор_N; ... break;
}
```

1.3. ОПЕРАТОР FOR

Оператор **`for`** позволяет реализовать в программе цикл и имеет следующий вид (см. листинг 5).

Элементы **`инициализация`**, **`условие`** и **`продвижение`** – это выражения; элемент **`оператор`** называется телом цикла, это обыч-

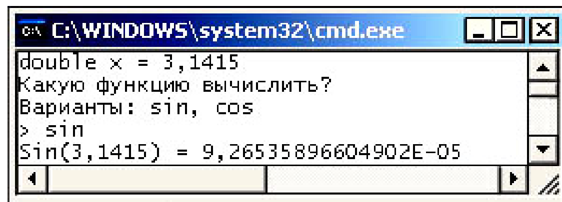

Рис. 4. Пример работы программы вычисления тригонометрической функции

Листинг 4

```
using System;
class P02
{
    public static void Main()
    {
           Console.Write("double x = ");
           double x = double.Parse(Console.ReadLine());
           Console.WriteLine("Какую функцию вычислить?");
           Console.WriteLine("Варианты: sin, cos");
           Console.Write("> ");
           string f = Console.ReadLine();
           switch (f)
           {
             case "sin":
                  Console.WriteLine("Sin({0}) = {1}", x, Math.Sin(x));
                   break;
             case "cos":
                  Console.WriteLine("Cos({0}) = {1}", x, Math.Cos(x));
                   break;
             default:
                  Console.WriteLine("Недопустимая функция");
                   break;
           }
    }
}
```

Листинг 5

```
for (инициализация; условие; продвижение)
         оператор
```

но блок. Если в разделе **инициализация** объявляется переменная, то областью ее действия является **оператор**. Семантика оператора **for** может быть определена посредством блок-схемы (см. рис. 5).

В качестве примера использования оператора **for** приведем программу вычисления первых двадцати значений функции «факториал» (см. листинг 6, рис. 6).

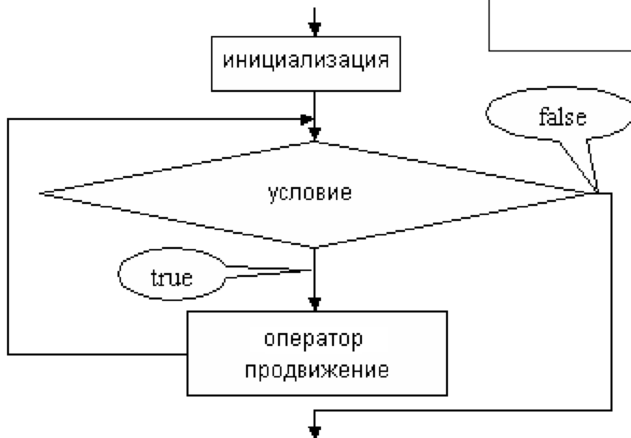

Рис. 5. Семантика оператора **for**

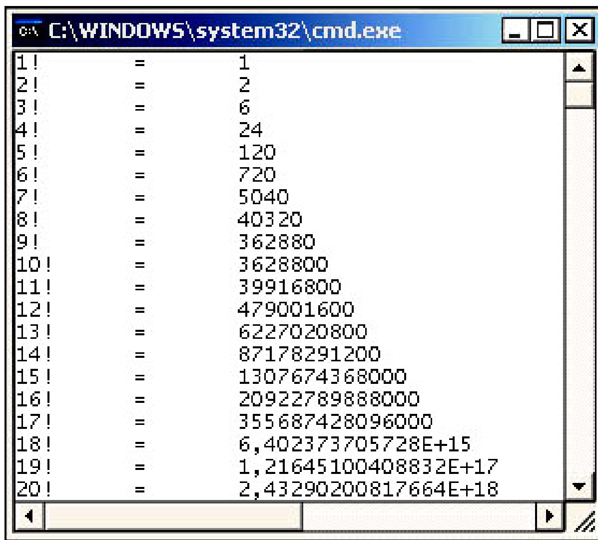

Рис. 6. Пример работы программы вычисления функции «факториал»

1.4. ОПЕРАТОР WHILE

Оператор **while** позволяет реализовать в программе цикл с предусловием и имеет следующий вид (см. листинг 7).

Элемент **условие** – это логическое выражение, элемент **оператор** называется телом цикла, это обычно блок. Семантика оператора **while** может быть определена посредством блок-схемы (см. рис. 7).

В качестве примера использования оператора **while** приведем программу вычисления степеней двойки (см. листинг 8, рис. 8).

Листинг 6

```
using System;
class P03
{
     public static void Main()
     {
          double f = 1;
          for (int x = 1; x <= 20; x++)
          {
               f *= x;
                Console.WriteLine("{0}! \t = \t {1}", x, f);
          }
     }
}
```

Листинг 7

```
while ( условие )
                 оператор
```

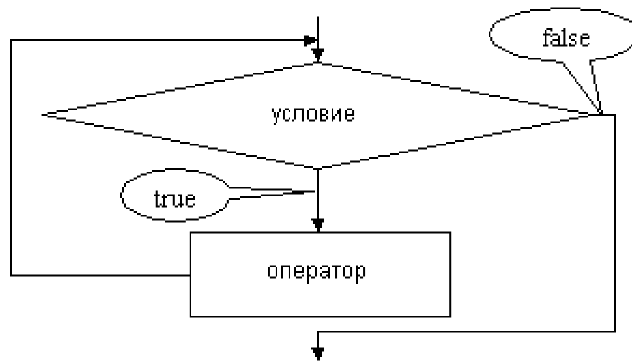

Рис. 7. Семантика оператора **`while`**

1.5. ОПЕРАТОР DO-WHILE

Оператор **`do-while`** позволяет реализовать цикл с постусловием в программе и имеет следующий вид (см. листинг 9).

Элемент **`условие`** – это логическое выражение, элемент **`оператор`** – это обычно блок. Тело цикла **`do-while`** выполняется хотя бы один раз, независимо от условия повторения цикла. Семантика оператора **`do-while`** может быть определена посредством блок-схемы (см. рис. 9).

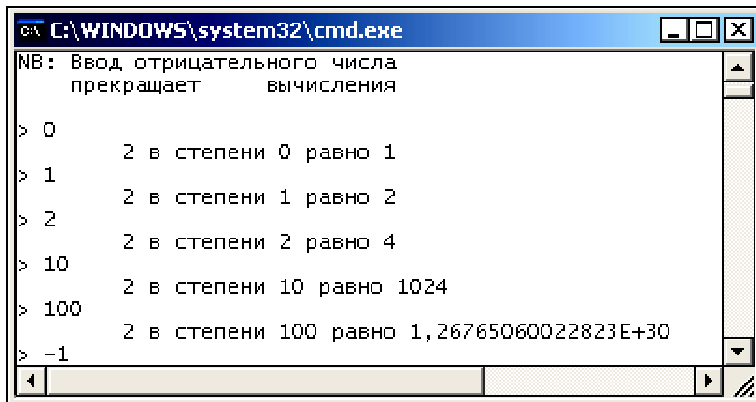

Рис. 8. Пример работы программы вычисления степеней двойки

Листинг 9

```
do
   оператор
while ( условие )
```

Листинг 8

```
using System;
class P04
{
     public static void Main()
     {
           Console.WriteLine("Введите натуральное число\n" +
                             "для возведения в него числа 2\n" +
                             "NB: Ввод отрицательного числа\n" +
                             "    прекращает      вычисления\n");
           Console.Write("> ");
           double n = double.Parse(Console.ReadLine());
           while (n >= 0)
           {
                Console.WriteLine("\t2 в степени {0} равно {1}",
                                  n, Math.Pow(2.0, n));
                Console.Write("> ");
                n = double.Parse(Console.ReadLine());
           }
     }
}
```

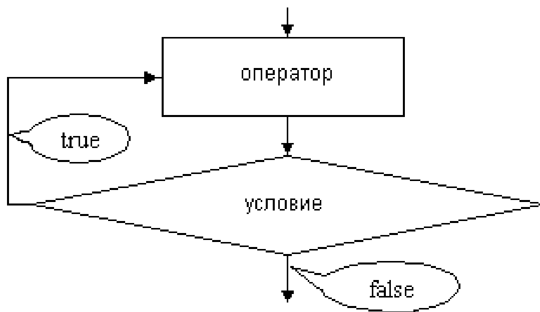

Рис. 9. Семантика оператора **`do-while`**

```
C:\WINDOWS\system32\cmd.exe
NB: Ввод отрицательного числа
    прекращает      вычисления

> 0
        2 в степени 0 равно 1
> 1
        2 в степени 1 равно 2
> 2
        2 в степени 2 равно 4
> 10
        2 в степени 10 равно 1024
> 100
        2 в степени 100 равно 1,26765060022823E+30
> -1
        2 в степени -1 равно 0,5
```

Рис. 10. Пример работы программы вычисления степеней двойки (второй вариант)

В качестве примера использования оператора **`do-while`** приведем второй вариант программы вычисления степеней двойки (см. листинг 10, рис. 10).

1.6. ОПЕРАТОР BREAK

Оператор **`break`** может использоваться не только в операторе **`switch`**, но и в циклах. Если в процессе выполнения тела цикла встретится оператор **`break`**, то он завершает выполнение цикла. В качестве примера использования оператора **`break`** приведем программу, в которой цикл **`for`** имеет пустой заголовок, а выполнение цикла прекращает оператор **`break`** (см. листинг 11, рис. 11).

1.7. ОПЕРАТОР «CONTINUE»

Оператор **`continue`** используется в операторах цикла и позволяет перейти на следующую итерацию цикла. В качестве примера использования оператора **`continue`** приведем программу, в которой используется цикл, игнорирующий нечетные числа (см. листинг 12, рис. 12).

Листинг 10

```
using System;
class P05
{
    public static void Main()
    {
        Console.WriteLine("Введите натуральное число\n" +
                          "для возведения в него числа 2\n" +
                          "NB: Ввод отрицательного числа\n" +
                          "    прекращает      вычисления\n");
        double n;
        do
        {
            Console.Write("> ");
           n = double.Parse(Console.ReadLine());
           Console.WriteLine("\t2 в степени {0} равно {1}",
                             n, Math.Pow(2.0, n));
        }
        while (n >= 0);
    }
}
```

Листинг 11

```
using System;
class P06
{
    public static void Main()
    {
        int i = 0;
        for (; ; )
        {
            if (i >= 10)
                break;
            Console.WriteLine("i = {0}", i);
            i++;
        }
    }
}
```

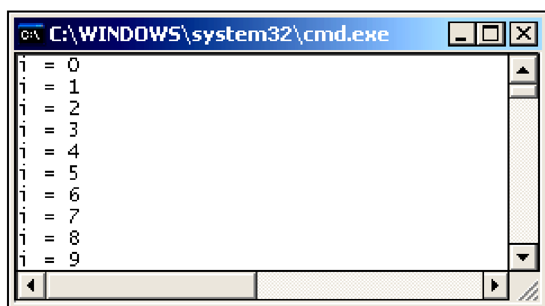

Рис. 11. Пример работы программы с оператором **`break`**

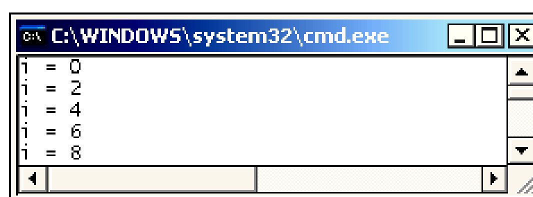

Рис. 12. Пример работы программы с оператором **`continue`**

Листинг 12

```
using System;
class P07
{
    public static void Main()
    {
        int i = 0;
        for (; ; i++)
        {
            if (i >= 10)
                break;
            if (i % 2 == 1)
                continue;
            Console.WriteLine("i = {0}", i);

        }
    }
}
```

2. ОБРАБОТКА ИСКЛЮЧЕНИЙ

2.1. ПОНЯТИЕ ИСКЛЮЧЕНИЯ

Если в программе допущена ошибка, которая делает невозможным ее выполнение, то C# рассматривает такую ситуацию как *исключение* (**Exception**). В результате аварийно прекращается выполнение программы и выдается соответствующее диагностическое сообщение. В качестве примера приведем программу, в которой может возникнуть исключительная ситуация в связи с попыткой деления на нуль (см. листинг 13, рис. 13, 14).

Листинг 13

```
using System;
class P08
{
    public static void Main()
    {
        Console.Write("int x = ");
        int x = int.Parse(Console.ReadLine());
        Console.Write("int y = ");
        int y = int.Parse(Console.ReadLine());
        int z = x / y;
        Console.WriteLine("{0} / {1} = {2}", x, y, z);
    }
}
```

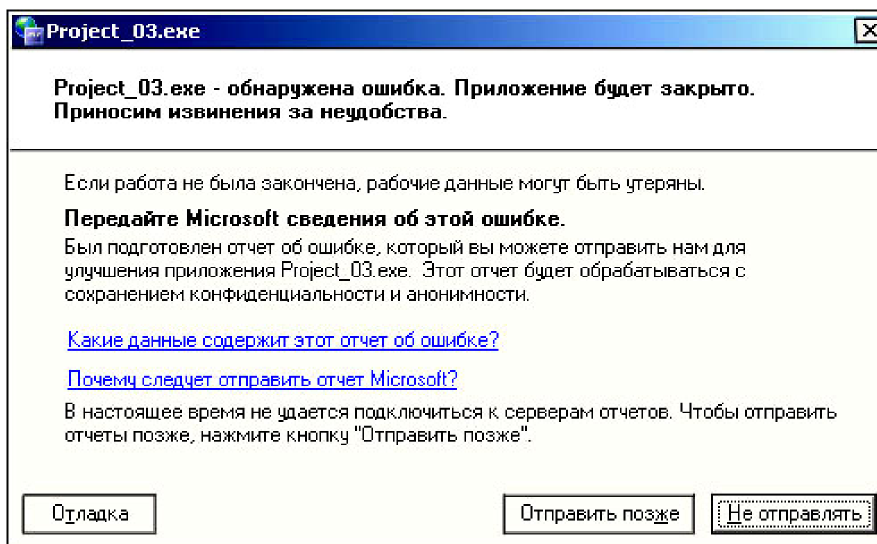

Рис. 13. Сообщение об аварийном завершении программы

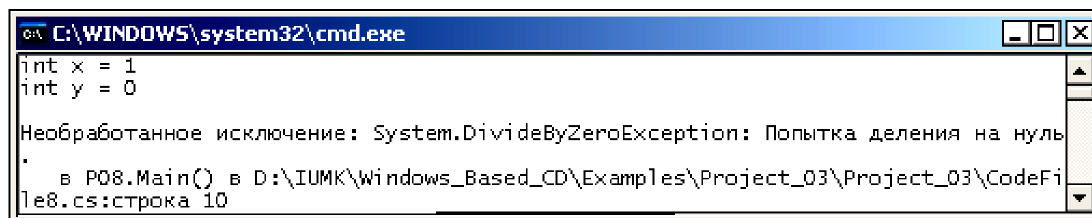

Рис. 14. Возникновение исключительной ситуации в связи с попыткой деления на нуль

2.2. ПЕРЕХВАТ ИСКЛЮЧЕНИЯ

Язык C# позволяет программе самой перехватывать исключения и обрабатывать их. В этом случае не происходит аварийного завершения программы. Операторы программы, которые могут вызвать исключение, помещаются в блок **try**. Операторы, которые выполняются при возникновении исключения, помещаются в блок **catch** (листинг 14).

Листинг 14

```
try
{ операторы
)
catch ( Exceptions e )
{ операторы
}
```

Блок **catch** принимает в качестве параметра объект **e** класса **Exception** (определенного в пространстве имен **System**) или объект его производного класса. Объект **e** содержит информацию об исключении. В качестве примера приведем программу, в которой перехватывается и обрабатывается исключительная ситуация в связи с попыткой деления на нуль (см. листинг 15, рис. 15).

После блока **try** можно указать несколько блоков **catch**. Если возникает исключение в блоке **try**, то это исключение последовательно проверяется в каждом блоке **catch**. В качестве примера приведем программу, которая запускалась два раза с различными исходными данными: в первый раз была попытка деления на ноль

Листинг 15

```
using System;
class P09
{
    public static void Main()
    {
        int x = 0, y = 0, z = 0;
        try
        {
            Console.Write(«int x = «);
            x = int.Parse(Console.ReadLine());
            Console.Write(«int y = «);
            y = int.Parse(Console.ReadLine());
            z = x / y;
            Console.WriteLine(«{0} / {1} = {2}», x, y, z);
        }
        catch (Exception e)
        {
            Console.WriteLine(e);
            Console.WriteLine(«{0} / {1} = {2}», x, y, «бесконечность»);
        }
    }
}
```

```
C:\WINDOWS\system32\cmd.exe
int x = 1
int y = 0
System.DivideByZeroException: Попытка деления на нуль.
   в P09.Main() в D:\IUMK\Windows_Based_CD\Examples\Project_03\Project_03\CodeFi
le9.cs:строка 13
1 / 0 = бесконечность
```

Рис. 15. Обработка исключительной ситуации в связи с попыткой деления на нуль

(см. рис. 16), во второй раз было введено недопустимо большое целое число (см. листинг 16, рис. 17).

2.3. БРОСАНИЕ ИСКЛЮЧЕНИЯ

При выполнении программы можно обратиться к операции **throw**, которая генерирует выброс указанного после нее исключения. Это приводит к прекращению выполнения программы и к переходу на поиск соответствующего блока **catch** (в этом же методе или выше по стеку вызовов). Ключевое слово **new** используется для создания нового объекта класса **ArgumentOutOfRangeException**. Соответствующий блок **catch** получает этот объект в качестве параметра. В качестве примера приведем программу, в которой генерируется и обрабатывается исключение при вводе отрицательного числа (см. листинг 17, рис. 18).

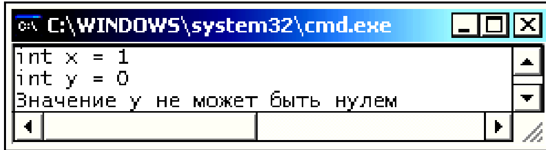

Рис. 16. Обработка исключения в связи с попыткой деления на нуль

2.4. БЛОК FINALLY

Допустим, что в программе открыто соединение с базой данных и происходит чтение данных. После завершения операции

Листинг 16

```
using System;
class P10
{   public static void Main()
    {   int x = 0, y = 0, z = 0;
        try
        {   Console.Write("int x = ");
            x = int.Parse(Console.ReadLine());
            Console.Write("int y = ");
            y = int.Parse(Console.ReadLine());
            z = x / y;
            Console.WriteLine("{0} / {1} = {2}",
                x, y, z);
        }
        catch (OverflowException e)
        {   Console.WriteLine(e);
        }
        catch (DivideByZeroException e)
        {   Console.WriteLine("Значение y не может быть нулем");
        }
    }
}
```

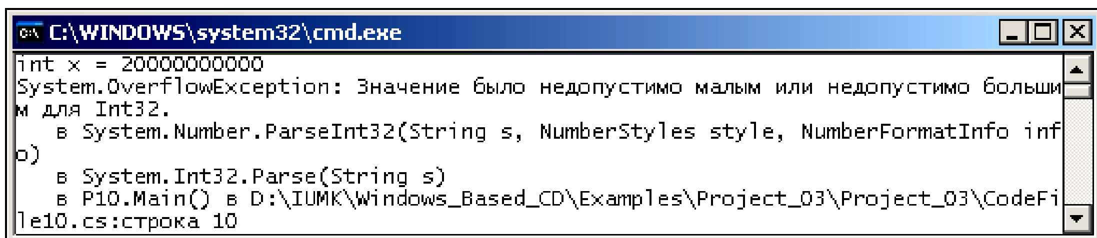

Рис. 17. Обработка исключения в связи с переполнением

Листинг 17

```
using System;
class P11
{
    public static void Main()
    {
        double x = 0, y = 0;
        Console.Write("double x = ");
        try
        {
            x = double.Parse(Console.ReadLine());
            if (x < 0)
            {
                string ex = "Отрицательное число: " + x;
                throw new ArgumentOutOfRangeException(ex);
            }
            else
            {
                y = Math.Sqrt(x);
                Console.WriteLine("Sqrt({0}) = {1}", x, y);
            }
        }
        catch (Exception e)
        {
            Console.WriteLine(e);
        }
    }
}
```

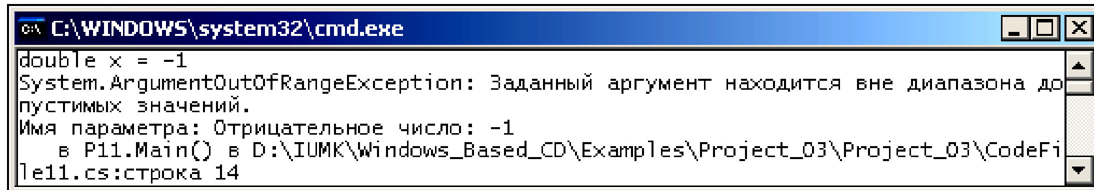

Рис. 18. Обработка сгенерированного исключения

чтения данных соединение с базой данных необходимо закрыть, чтобы освободить ресурсы сервера. Если при чтении данных возникло исключение, то соединение с базой данных также необходимо закрыть (листинг 18).

Чтобы не дублировать код, можно использовать блок **`finally`**, операторы в котором будут выполнены, независимо от того, завершился ли нормально блок **`try`** или управление было передано в блок **`catch`** (листинг 19).

2.5. ПРОВЕРКА АРИФМЕТИЧЕСКОГО ПЕРЕПОЛНЕНИЯ

По умолчанию C# не проводит проверку арифметического переполнения. Для демон-

Листинг 18

```
try
{ ...
  <закрыть соединение с базой данных>
}
catch
{ ...
  <закрыть соединение с базой данных>
}
```

Листинг 19

```
try
{   ...
}
catch
{   ...
}
finally
{   <закрыть соединение с базой данных>
}
```

страции этого факта приведем следующую программу (см. листинг 20, рис. 19).

Чтобы включить проверку переполнения, можно поместить в блок **checked** те операторы, которые способны вызвать переполнение. В этом случае проверка будет производиться независимо от настроек компилятора, и на переполнение будет выбрасываться исключение **System.OverflowException** (см. листинг 21, рис. 20).

Чтобы отключить проверку переполнения, можно поместить в блок **unchecked** те операторы, которые способны вызвать переполнение. В этом случае проверка не будет производиться независимо от настроек компилятора, и на переполнение не будет выбрасываться исключение **System.OverflowException** (см. листинг 22, рис. 21).

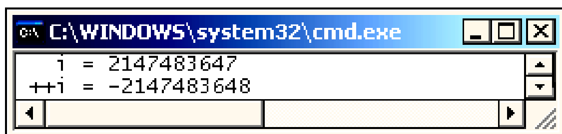

Рис. 19. Демонстрация игнорирования переполнения

3. ФАЙЛЫ И ПОТОКИ

3.1. ФАЙЛЫ И ПОТОКИ ВЫВОДА

Если данные, полученные в результате выполнения программы, нужно ис-

Листинг 20

```
using System;
class P12
{
    public static void Main()
    {
        int i = int.MaxValue;
        Console.WriteLine("   i = {0}", i);
        Console.WriteLine(" ++i = {0}", ++i);
    }
}
```

Листинг 21

```
using System;
class P13
{
    public static void Main()
    {
        int i = int.MaxValue;
        checked
        {
            Console.WriteLine("   i = {0}", i);
            Console.WriteLine(" ++i = {0}", ++i);
        }
    }
}
```

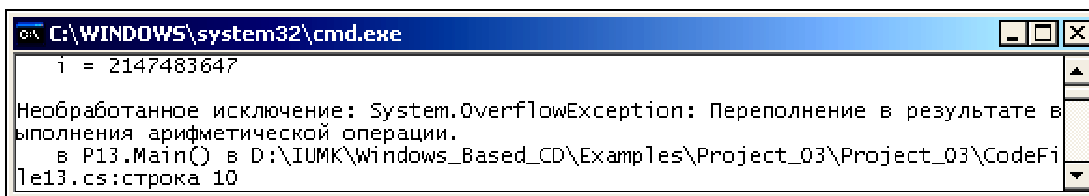

Рис. 20. Демонстрация включенной проверки переполнения

пользовать при следующем запуске программы или их нужно использовать в другой программе, то эти данные можно сохранить во внешней памяти компьютера. Данные во внешней памяти организованы в виде файлов. Файл имеет имя и ряд других характеристик: время создания файла, время его последней модификации и т. п. Простейшими видами файлов являются текстовые файлы, данные в которых хранятся в виде последовательности байтов.

Программа может выводить текстовые данные на экран монитора или в файл. Для всех устройств вывода используется обобщенное название поток вывода. Для вывода текстовых данных в любой поток вывода можно использовать одни и те же методы: **`Write()`**, **`WriteLine()`**. Библиотеки классов для создания файлов и ввода-вывода данных в файлы находятся в пространстве имен **`System.IO`**.

Для записи текста в файл программа должна создать поток вывода в виде объекта класса **`StreamWriter`**. Имя создаваемого файла (краткое или полное) указывается в конструкторе этого класса. Если файл с таким же именем уже существовал, то старый файл уничтожается и создается новый файл. При работе с файлами могут возникнуть исключения, например:

- **`IOException`**
- **`DirectoryNotFoundException`**

Если поток вывода успешно создан, можно записывать текст в файл с помощью методов **`Write()`**, **`WriteLine()`**. После окончания записи в поток его нужно закрыть посредством метода **`Close()`**: операционная система завершает все незаконченные операции вывода и освобождает память, выделенную для объекта класса **`StreamWriter`**. В качестве примера приведем программу, которая создает новый файл и записывает в него текст, который пост-

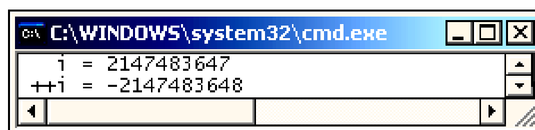

Рис. 21. Демонстрация отключенной проверки переполнения

Листинг 22

```
using System;
class P13
{
    public static void Main()
    {
        int i = int.MaxValue;
        unchecked
        {
            Console.WriteLine("   i = {0}", i);
            Console.WriteLine(" ++i = {0}", ++i);
        }
    }
}
```

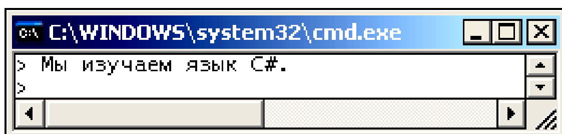

Рис. 22. Демонстрация ввода теста в файл с клавиатуры

рочно вводится с клавиатуры; программа завершает работу при вводе пустого текста путем нажатия клавиши «Enter» (см. листинг 23, рис. 22, 23).

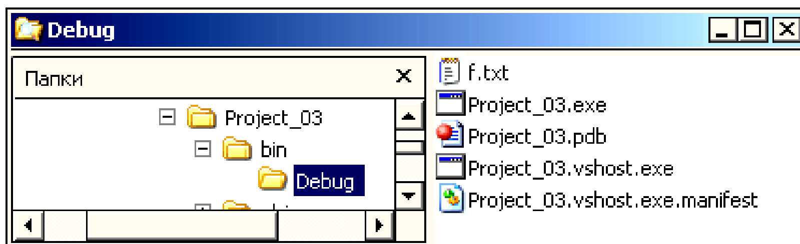

Рис. 23. Расположение созданного файла

3.2. ФАЙЛЫ И ПОТОКИ ВВОДА

Для чтения текста из файла следует создать поток ввода в виде объекта **`StreamReader`**. Имя файла (краткое или полное) указывается в конструкторе этого класса. Если файл открыт успешно, то можно читать из него строки текста с помощью метода **`ReadLine()`**. Если поток ввода исчерпан, то есть в файле нет больше данных, то метод **`ReadLine()`** возвращает значение **`null`**.

Можно читать из текстового потока по одному символу, используя метод **`Read()`**, который возвращает не символ, а 32-битовое целое число, соответствующее коду символа. Если файл уже прочитан до конца, то возвращается число –1 (листинг 24).

Листинг 23

```
using System;
using System.IO;
class P15
{
    public static void Main()
    {
        StreamWriter outStream = null;
        string f = "f.txt";
        try
        {
            outStream = new StreamWriter(f);
        }
        catch (Exception e)
        {
            Console.WriteLine("Невозможно создать файл {0}", f);
            Console.WriteLine("Причина: {0}", e.ToString());
            //здесь можно указать корректирующие действия
        }
        string s;
        do
        {
            Console.Write("> "); s = Console.ReadLine();
            outStream.WriteLine(s);
        } while (s != "");
        outStream.Close();
    }
}
```

Листинг 24

```
int k; while ((k=inStream.Read()) != -1)
{    char ch = (char)k; <обработка символа> }
```

Если текст из файла читается в объект класса **string**, то прочитанную строку нельзя изменять. Если строку требуется модифицировать, то ее нужно объявить как объект класса **StringBuilder**. Этот класс входит в пространство имен **System.Text** и содержит среди прочих следующие методы (табл. 1).

Табл. 1

```
result.Append(char ch)              // присоединить символ ch к строке
result.Append(string s1)            // присоединить строку s1 к строке
result.Insert(int i, char ch)       // вставить символ ch после позиции i
result.Insert(int i, string s1)     // вставить строку s1 после позиции i
result.Remove(int i, int len        // удалить len символов от позиции i
result.ToString()                   // преобразовать в обычную строку
```

Листинг 25

```
using System;
using System.Text;
using System.IO;
class P16
{   public static void Main()
    {   Console.Write("Введите имя файла: ");
        string inFileName = Console.ReadLine();
        inFileName = inFileName.TrimEnd();
        StreamReader inStream = null;
        try
        { inStream = new StreamReader(inFileName); }
        catch (Exception e)
        {   Console.WriteLine("Не открывается файл {0}\n{1}",
            inFileName, e.ToString()); return;
        }
        string outFileName = "CopyOf" + inFileName;
        StreamWriter outStream = new StreamWriter(outFileName);
        for (; ; )
        {   string s = inStream.ReadLine(); if (s == null) break;
            outStream.WriteLine(Modify(s.TrimEnd()));
        }
        outStream.Close(); inStream.Close();
    }
    public static string Modify(string s)
    {   StringBuilder result = new StringBuilder(s.Length);
        for (int i = 0; i < s.Length; i++)
        {  if (s[i] == ".") result.Append("!"); else result.Append(s[i]);
        }
        return result.ToString();
    }
}
```

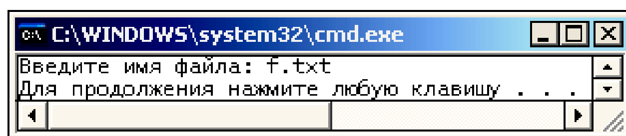

Рис. 24. Запрос имени исходного файла

В качестве примера приведем программу, которая читает строки из предварительно созданного текстового файла, заменяет все символы «.» на символы «!» и выводит результат в новый файл (см. листинг 25, рис. 24, 25).

Рис. 25. Содержимое результирующего файла

Литература

1. *Керов Л.А.* Методы объектно-ориентированного программирования на C# 2005: Учебное пособие. СПб: Изд. «ЮТАС», 2007. 164 с.
2. *Нэш Т.* C# 2008: ускоренный курс для профессионалов: Пер. с англ. М.: ООО «И.Д. Вильямс», 2008. 576 с.
3. *Павловская Т.А.* C#. Программирование на языке высокого уровня. Учебник для вузов. СПб: Питер, 2007. 432 с.
4. *Троелсен Э.* Язык программирования C# 2005 и платформа .NET 2.0. 3-е издание.: Пер. с англ. М.: ООО «И.Д. Вильямс», 2007. 1168 с.
5. *Шилдт Г.* C#: учебный курс. СПб: Питер; К.: Изд. гр. BHV, 2003. 512 с.

Abstract

The article is third of a series of articles, devoted to «a zero level» of language C#. Flow control statements, exception handling, input and output of text files are considered.

Наши авторы, 2009.
Our authors, 2009.

Керов Леонид Александрович,
кандидат технических наук, старший научный сотрудник, доцент, заведующий кафедрой бизнес-информатики Санкт-Петербургского филиала государственного университета – Высшей Школы Экономики при Правительстве РФ,

kerov@hse.spb.ru