Синтаксис языка C#

- Комментарии

```
// Однострочный комментарий

/* Можно комментировать
   много строк
 */

 /// <summary>Это однострочный комментарий для документации</summary>
 /// <remarks>
 /// Это однострочный комментарий
 /// </remarks>

 /**
 <summary> Документация </summary>
 *<remarks>
 *А это
 * многострочный
 *</remarks>
 */
```

Литералы

В C# существует четыре типа литералов:

- целочисленный литерал (8-й, 16-й и 10-й системе счисления);
- вещественный литерал;
- символьный литерал;
- строковый литерал.

Константы

- Литеральные константы

```
x =  100;   // литеральная константа
```

- Символические константы

Объявляются с дополнительным спецификатором const. Требуют непосредственной инициализации.

```
const dbl_pi = 3.1415926;
```

- Перечисления

```
enum Sizes: unint { Small=1, Middle, Large = 10 };
```

- Строковые константы

```
string strMessage = "Здравствуй Мир!";
```

Арифметические операции	+ – * / %
Логические (boolean и побитовые)	& | ^ ! ~ && ||
Строковые (конкатенаторы)	+
Increment, decrement	++ – –
Сдвига	>> <<
Сравнения	== != < > <= >=
Присвоения	= += –= *= /= %= &= |= ^= <<= >>=
Member access	.
Индексации	[ ]
Cast (приведение типа)	( )
Оператор условия (трехоперандная)	?:
Delegate concatenation and removal	+ –
Создание объекта	new( )
Информация о типе	is as sizeof typeof
Overflow exception control (управление исключениями)	checked unchecked
Indirection and Address (неуправляемый код)	* –> [ ] &

Приоритет	Категория	Операции
0	Первичные	f(x) a[x] x++ x-- -> checked unchecked new; typeof default(T) delegate sizeof
1	Унарные	+ - ! ~ ++x --x (T)x await &x *x
2	Мультипликативные (Умножение)	* / %
3	Аддитивные (Сложение)	+ -
4	Сдвиг	<< >>
5	Отношения, проверка типов	< > <= >= is as
6	Эквивалентность	== !=
7	Логическое И	&
8	Логическое исключающее XOR	^
9	Логическое ИЛИ (OR)	|
10	Условное И	&&
11	Условное ИЛИ	||
12	Условное выражение	? :
13	Присваивание и лямбда-выражение	= *= /= %= += -= <<= >>= &= ^= |= =>

Объявление переменных. Область видимости и время жизни

- Любая переменная используемая в программе вводится объявлением.
- Кроме переменных объявлению подлежат:
  - ***классы и структуры***. Класс (структура) может содержать вложенные объявления других классов и структур;
  - ***перечисления***;
  - ***объекты*** – переменные и константы, представляющие классы и структуры. Корректное объявление объекта предполагает, что информация, содержащая характеристики объявляемого объекта, доступна транслятору;
  - ***элементы перечислений*** – объекты, представляющие перечисления;
  - ***конструкторы классов и методы*** (функции) – члены классов (в том числе и специальные).

- Конфликта имен (проблемы с обращением к одноименным объектам) позволяет избежать принятая в языке дисциплина именования. В каждом языке программирования она своя

- Избежать конфликта имен в C# позволяет такая синтаксическая конструкция, как ***блок операторов***.

- Блок – это множество предложений (возможно пустое), заключенное в фигурные скобки.

- Переменные можно объявлять в любом месте блока. Точка объявления переменной в буквальном смысле соответствует месту ее создания. Обращение к объявляемой переменной или константе "выше" точки ее объявления лишено смысла.

Управляющие операторы

- Управляющие операторы определяют последовательность выполнения операторов в программе и являются основным средством реализации алгоритмов.

- Различаются следующие категории управляющих операторов:

  - ***Операторы выбора***. Вводятся ключевыми словами if, if ... else ..., switch.
  - ***Операторы цикла***. Вводятся ключевыми словами while, do ... while, for, foreach.
  - ***Операторы перехода***. Вводятся ключевыми словами goto, break, continue, return.

Операторы выбора

if, if ... else ...

- После ключевого слова **if** располагается взятое в круглые скобки условное выражение (булево выражение), следом за которым располагается оператор (блок операторов)

- Далее в операторе **if** ... **else** ... после ключевого слова **else** размещается еще один оператор

- В силу того, что в C# *отсутствуют предопределенные алгоритмы преобразования значений к булевскому типу*, условное выражение должно быть выражением типа **bool** – переменной, константой или выражением на основе операций сравнения и логических операций.

- З***амечание***: невозможно построить оператор if ... else ... на основе одиночного оператора объявления:

```
if (true) int XXX = 125;
if (true) int XXX = 125; else int ZZZ = 10;
```

- Такие конструкции воспринимаются как ошибочные. Ставить в зависимость от условия (пусть даже всегда истинного) создание объекта не принято.

- Совсем другое дело – при работе с блоками

```
if (true) {int XXX = 125;}
if (true) {int XXX = 125;} else {int ZZZ = 10;}
```

- Даже в таких примитивных блоках определена своя область видимости, и создаваемые в них объекты, никому не мешая, существуют по своим собственным правилам

switch

- switch оператор имеет следующий вид:

```
int val;
:::::::::
switch (val)
{  case 0:    …    break;
   case 100: …    break;
   default:       break;          // Список операторов пуст.
 }
```

- Списки операторов в **case** элементах не могут включать операторы объявления.

- Каждый case элемент в обязательном порядке завершается оператором-терминатором break , goto, return

```
class MyClass
{
    static void Main (string[]  args)
      {
      string user;
      user = Console.ReadLine();
      switch (user)
        {
          case "user1":
             Console.WriteLine("Здравствуйте user1 ");                    break;
          case "user2":
             Console.WriteLine ("Здравствуйте user2 ") ;                  break;
          case "user3":
            Console . WriteLine ("Здравствуйте user3 ") ;                  break;
          default:
            Console.WriteLine("Здравствуйте новый пользователь");  break;
        }
      }
}
```

Операторы цикла

while

```
while ( …  ) {      };
```

- Оператор

```
while (true) int XXX = 0;
```

с самого первого момента своего существования (еще до начала трансляции!) сопровождается предупреждением:

Embedded statement cannot be a declaration or labeled statement

```
String Name;
while (true)
 {
    Console.Write ("Введите ваш имя " );
    Name = Console.ReadLine();
    if(Name ==” ”)   break;
    Console.WriteLine("Здравствуйте {0}", Name);
 }
```

do ... while

- Разница с ранее рассмотренным оператором цикла состоит в том, что здесь сначала выполняется оператор (блок операторов), а затем проверяется условие продолжения оператора

```
string password;
do {
      password=Console.ReadLine();
     }
while(password!=“Admin");
```

for

- Оператор for также невозможно построить на основе одиночного оператора объявления

foreach

- Этим оператором обеспечивается повторение множества операторов, составляющих тело цикла, для каждого элемента массива или коллекции. После перебора ВСЕХ элементов массива или коллекции и применения множества операторов для каждого элемента массива или коллекции, управление передается следующему за оператором foreach оператору.

```
int[ ] array = new int[10];              // Объявили и определили массив
…
foreach (int i in array) { /*:::::*/ };   // Для каждого элемента массива надо сделать...
```

- Специализированный оператор, приспособленный для работы с массивами и коллекциями. Конструкция экзотическая и негибкая. Предполагает выполнение примитивных последовательностей действий над массивами и коллекциями (начальная инициализация или просмотр ФИКСИРОВАННОГО количества элементов). Действия, связанные с изменениями размеров и содержимого коллекций, в рамках этого оператора могут привести к непредсказуемым результатам

Обработка исключений

Как правило программист в своем проекте не может предсказать все действия пользователя, входные значения и многие другие параметры. А чем крупнее и серьезнее проект тем таких мест в приложении становится все больше, поэтому перед программистом встает задача отловить и по возможность обработать все возможные ситуации неправильного выполнения кода.

Любое действие которое не может быть выполнено по той или иной причине называется Exception (исключение).

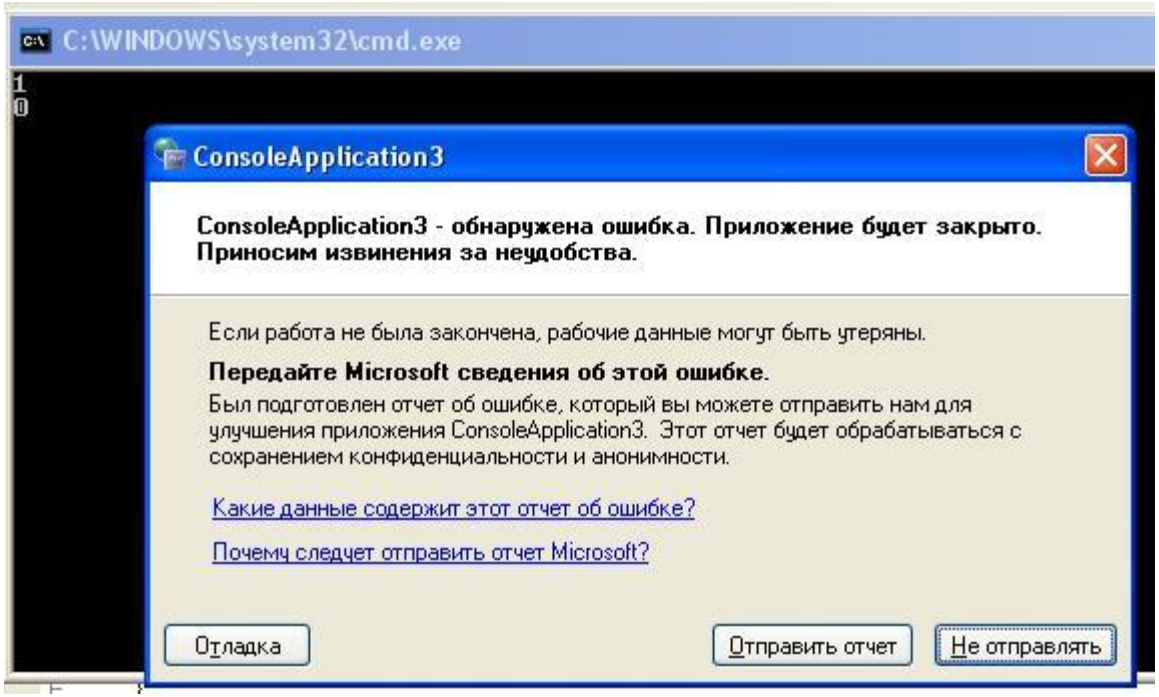
Данное исключение и будем отлавливать. Для этого в C# есть конструкция

try … catch

Рассмотрим небольшой пример:

```
int a = Convert.ToInt32(Console.ReadLine());
int b = Convert.ToInt32(Console.ReadLine());
Console.WriteLine(a / b);
```

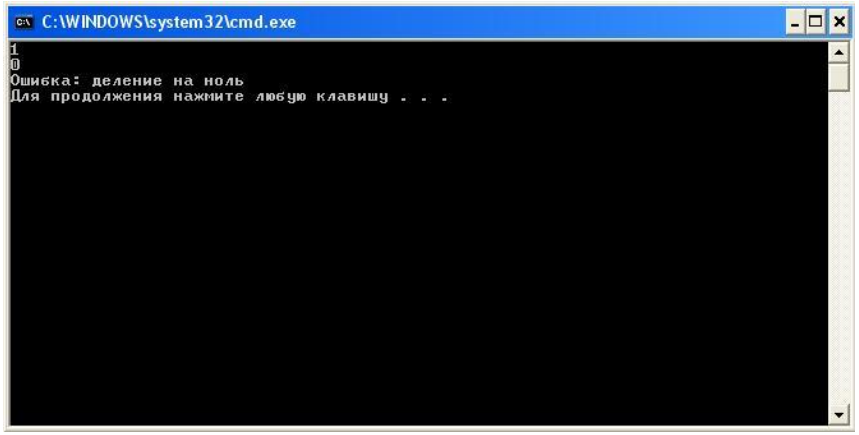
В данном примере вводятся два числа, далее идет деление и вывод результата. Все бы ничего, а что если пользователь введет во втором случае 0? Ведь на ноль делить нельзя - будет выведено исключение



Возможно пользователь даже не сможет понять что произошло.

Заключим критическую область в try, а в catch выведем ошибку

```
try
    {
        int a = Convert.ToInt32(Console.ReadLine());
        int b = Convert.ToInt32(Console.ReadLine());
        Console.WriteLine(a / b);
    }
    catch
    {
        Console.WriteLine("Ошибка: деление на ноль");
    }
```



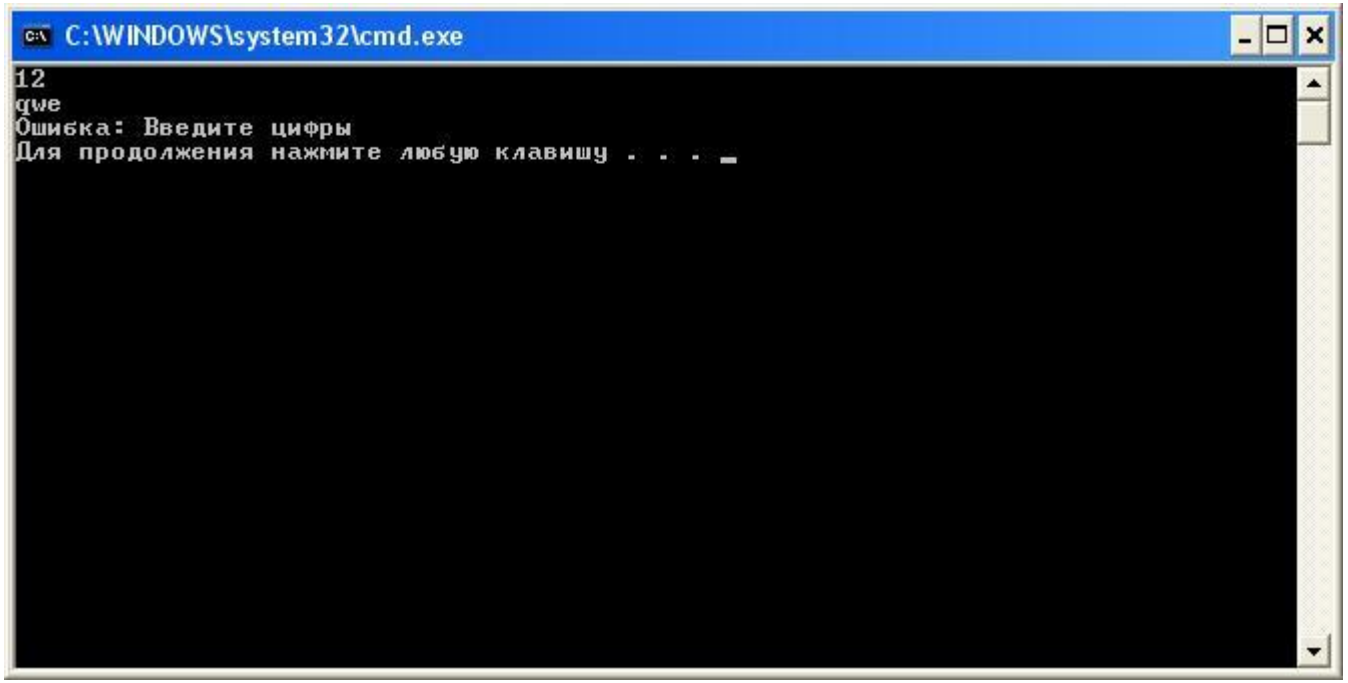
Теперь как видите пользователю все видно.

Но часто так бывает что в критической секции могут возникнуть разные исключения и их надо по разному обработать тогда нам надо как можно точнее описать секцию catch то есть для какого именно исключения она будет отрабатывать.

Для этого рядом с catch пишем имя исключения. Так как пользователь может ввести не только цифры но и буквы, то обработаем и это исключение. Оно будет происходить в момент конвертации string в int

```
try
        {
            int a = Convert.ToInt32(Console.ReadLine());
            int b = Convert.ToInt32(Console.ReadLine());
            Console.WriteLine(a / b);
        }
        catch (System.DivideByZeroException)
        {
            Console.WriteLine("Ошибка: деление на ноль");
        }
        catch (System.FormatException)
        {
            Console.WriteLine("Ошибка: Введите цифры");
        }
```

Теперь будут все исключения обрабатываться как надо.



Осталось только рассмотреть последний блок данной конструкции - блок finally Он выполняется всегда: произошла ошибка в catch или нет.

Пример может быть следующим: вы открываете файл что то делаете и в любом случае произошла или нет ошибка вы должны закрыть файл это можно как раз сделать в finally

Директива using

Директива using используется в двух случаях:

- разрешает использование типов в пространстве имен, поэтому уточнение использования типа в этом пространстве имен не требуется;

```
using System.Text;
```

- позволяет создавать псевдонимы пространства имен или типа. Это называется *директива* using alias

```
using Project = PC.MyCompany.Project;
```

Область директивы using ограничивается файлом, в котором она появляется.

Создание псевдонима using упрощает определение идентификатора для пространства имен или типа. Правая часть директивы using alias должна всегда быть полным именем, независимо от предшествовавших ей директив using.

Создание директивы using позволяет использовать типы из пространства имен без указания этого пространства. Директива using не предоставляет доступ к пространствам имен, которые вложены в указанное пространство.

Существует две категории пространства имен: определенные пользователем и определенные системой. Пространства, определенные пользователем, — это те, которые определены в коде. Список пространств имен, определенных системой в библиотеке классов .NET Framework.

```
using System;
// Using alias directive for a class.
using AliasToMyClass = NameSpace1.MyClass;
// Using alias directive for a generic class.
using UsingAlias = NameSpace2.MyClass<int>;

namespace NameSpace1
{
    public class MyClass
     {
         public override string ToString()
                { return "You are in NameSpace1.MyClass."; }
     }
}

namespace NameSpace2
{
     class MyClass<T>
      {
          public override string ToString()
                 { return "You are in NameSpace2.MyClass."; }
      }
}
```

```
namespace NameSpace3
{
    using NameSpace1; // Using directive:
    using NameSpace2;

    class MainClass
     {
       static void Main()
         {
            AliasToMyClass instance1 = new AliasToMyClass();
            Console.WriteLine(instance1);
            UsingAlias instance2 = new UsingAlias();
            Console.WriteLine(instance2);
         }
    }
}

// Output:
// You are in NameSpace1.MyClass.
// You are in NameSpace2.MyClass.
```

```
using System;
using AliasToMyClass = NameSpace1.MyClass;
using UsingAlias = NameSpace2.MyClass<int>;

namespace NameSpace1
{
    public class MyClass
     {
         public override string ToString()
                { return "You are in NameSpace1.MyClass."; }
     }
}

namespace NameSpace2
{
     class MyClass<T>
      {
          public override string ToString()
                 { return "You are in NameSpace2.MyClass."; }
      }
}
```

Ключевое слово using также используется для создания операторов using, которые обеспечивают правильную обработку объектов IDisposable

Класс Math и его функции

```
public  void MathFunctions(){
   double a, b,t,t0,dt,y;
   string NameFunction;
   Console.WriteLine("Введите имя исследуемой функции  a*F(b*t)" + " (sin, cos, tan, cotan)");
   NameFunction = Console.ReadLine();
   Console.WriteLine("Введите параметр a (double)");              a= double.Parse(Console.ReadLine());
   Console.WriteLine("Введите параметр b (double)");              b= double.Parse(Console.ReadLine());
   Console.WriteLine("Введите начальное время t0(double)");   t0= double.Parse(Console.ReadLine());
   const int points = 10;
   dt = 0.2;
   for(int i = 1; i<=points; i++)   {
     t = t0 + (i-1)* dt;
     switch (NameFunction)      {
        case ("sin"):     y = a*Math.Sin(b*t);            break;
        case ("cos"):     y = a*Math.Cos(b*t);          break;
        case ("tan"):     y = a*Math.Tan(b*t);          break;
        case ("cotan"):  y = a/Math.Tan(b*t);          break;
        case ("ln"):        y = a*Math.Log(b*t);          break;
        case ("tanh"):    y = a*Math.Tanh(b*t);         break;
        default:              y=1;                                 break;
     }
     Console.WriteLine ("t = " + t + "; " + a +"*" +  NameFunction +"(" + b + "*t)= " + y + ";");
   }
   double u = 2.5, v = 1.5, p,w;
   p=   Math.Pow(u,v);
   w = Math.IEEERemainder(u,v);                        // остаток от деления
   Console.WriteLine ("u = " + u + "; v= " + v +  "; power(u,v)= " + p + "; reminder(u,v)= " + w);
}
```

Класс Random и его функции

Конструкторы:

public Random()
public Random (int)

- public int Next( ) - метод без параметров выдает целые положительные числа во всем положительном диапазоне типа int;
- public int Next(int max) - выдает целые положительные числа в диапазоне [0,max];
- public int Next(int min, int max) - выдает целые положительные числа в диапазоне [min,max].
- public double NextDouble( ) - выдается новое случайное число, равномерно распределенное в интервале [0-1).
- public void NextBytes(byte[ ] buffer) - генерирует случайные числа в диапазоне [0, 255] и заполняет массив байт.

```
   public  void Rand() {
   Random realRnd = new Random();
   Random repeatRnd = new Random(100);

//  случайные числа в диапазоне [0,1]
   Console.WriteLine("случайные числа в диапазоне[0,1)");
   for(int i =1; i <= 5; i++)   {
      Console.WriteLine("Число " + i + "= " + realRnd.NextDouble() );
   }
//  случайные числа в диапазоне[min,max]
   int min = -100, max=-10;
   Console.WriteLine("случайные числа в диапазоне [" +  min +"," + max + "]");
   for(int i =1; i <= 5; i++)   {
     Console.WriteLine("Число " + i + "= " + realRnd.Next(min,max) );
   }
//  массив случайных чисел
   byte[] bar = new byte[10];
   repeatRnd.NextBytes(bar);
   Console.WriteLine("Массив случайных чисел в диапазоне [0, 255]");
   for(int i =0; i < 10; i++)   {
      Console.WriteLine("Число " + i + "= " +bar[i]);
   }
}
```