Лекция 13

Синтаксис операторов передачи управления

Оператор **Выражение**

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

Пустой оператор

Частным случаем выражения является **пустой оператор** (;) - он используется, когда по синтаксису оператор требуется, а по смыслу — нет.

```
i++;       // выполняется операция инкремента
a *= b + c;    // выполняется умножение с
               // присваиванием
fun( i, k );     // выполняется вызов функции
while( true );   // цикл из пустого оператора -
                 // - бесконечный цикл
```

Составной оператор

Блок, или **составной оператор**, — это последовательность описаний и операторов, заключенная в *фигурные скобки*. 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.

Операторы передачи управления

- безусловного перехода **goto**;
- выхода из цикла **break**;
- перехода к следующей итерации цикла **continue**;
- возврата из функции **return**;
- генерации исключения **throw**.

Эти *операторы* могут передать управление в пределах блока, в котором они использованы, и за его пределы. Передавать управление внутрь другого блока запрещается.

Оператор goto

Оператор безусловного перехода **goto** используется в одной из трех форм:

goto метка;

goto case константное_выражение;

goto default;

Для выполнения оператора **goto** требуется **метка** — действительный в **C#** идентификатор с двоеточием.

метка: оператор;

Оператор goto

Метка — это обычный идентификатор, областью видимости которого является функция, в теле которой он задан. Метка должна находиться в той же области видимости, что и оператор перехода.

Оператор ***goto*** *метка* передает управление на помеченный **меткой** оператор.

```
x = 1;
loop1:   // Цикл суммирует числа от 1 до 100
x++;
if (x < 100) goto loop1;
```

Оператор goto

Вторая и третья формы оператора **goto** используются в *теле оператора* выбора **switch**. Оператор ***goto case*** *константное_выражение* передает управление на соответствующую константному выражению ветвь, а оператор **goto default** — на ветвь **default**. Формально ветви **case** или **default** выполняют в операторе **switch** роль меток. Поэтому они могут служить адресатами оператора **goto**. Тем не менее оператор **goto** должен выполняться в пределах оператора **switch**. Это означает, что его нельзя использовать как внешнее средство для безусловного перехода в оператор **switch**.

Пример 1

```
using System;
class SwitchGoto {
    static void Main() {
        for (int i = 1; i < 5; i++) {
            switch(i) {
                case 1:
                    Console.WriteLine("В ветви case 1");
                    goto case 3;
                case 2:
                    Console.WriteLine("В ветви case 2");
                    goto case 1;
```

Пример 1

```
            case 3:
                Console.WriteLine("В ветви case 3");
                goto default;
            default:
                Console.WriteLine("В ветви default");
                break;
        }
        Console.WriteLine() ;
    }
      // goto case 1; // Ошибка! Безусловный переход
// к оператору switch недопустим
  }
}
```

Пример 2

```
using System;
class Use_goto {
   static void Main() {
      int i=0, j=0, k=0;
      for (i=0; i < 10; i++) {
         for (j=0; j < 10; j++ ) {
            for (k=0; k < 10; k++) {
               Console.WriteLine("i, j, k: " + i + " " + j + " " + k);
               if (k == 3) goto stop;
            }
         }
      }
      stop:
      Console.WriteLine("Стоп! i, j, k: " + i + ", " + j + " " + k);
   }
}
```

Оператор break

С помощью оператора **break** можно специально организовать немедленный выход из цикла в обход любого кода, оставшегося в теле цикла, а также минуя проверку условия цикла. Когда в теле цикла встречается оператор **break**, цикл завершается, а выполнение программы возобновляется с оператора, следующего после этого цикла.

Пример 3

```
using System;
class BreakDemo {
    static void Main() {
        for (int i = -10; i <= 10; i++) {
            if (i > 0) break;
            Console.Write (i + " ");
        }
        Console.WriteLine ("Готово !");
    }
}
```

Пример 4

```
class BreakDemo2 {
    static void Main() {
        int i;
        i = -10;
        do {
            if (i > 0) break;
            Console. Write (i + " ");
            i++;
        } while (i <= 10);
        Console.WriteLine("Готово!");
    }
}
```

Пример 5

```
class FindSmallestFactor {
    static void Main() {
        int factor = 1;          int num = -1000;
        for (int i = 2; i <= num/i; i++) {
            if ((num%i) == 0) {
                factor = i;      break;
            }
        }
        Console.WriteLine("Наименьший
  множитель равен " + factor);
    }
}
```

Пример 6

```
static void Main() {
    for (int i=0; i<3; i++) {
        Console.WriteLine("Внешний цикл: " + i);
        Console.Write("Внутренний цикл: ");
            int t = 0;
            while (t < 100) {
                if (t == 10) break;
                Console.Write(t + " ");
                t++;
            }
            Console.WriteLine ();
    }
    Console.WriteLine("Циклы завершены.");
}
```

Пример 6

Выполнение программы дает следующий результат.

```
Подсчет во внешнем цикле: 0
Подсчет во внутреннем цикле: 0 1 2 3 4 5 6 7 8 9
Подсчет во внешнем цикле: 1
Подсчет во внутреннем цикле: 0 1 2 3 4 5 6 7 8 9
Подсчет во внешнем цикле: 2
Подсчет во внутреннем цикле: 0 1 2 3 4 5 6 7 8 9
Циклы завершены
```

Таким образом, оператор **break** из внутреннего цикла вызывает прерывание только этого цикла, а на выполнение внешнего цикла он не оказывает никакого влияния.

Оператор continue

С помощью оператора **continue** можно организовать **преждевременное завершение шага итерации цикла** в обход обычной структуры управления циклом. Оператор **continue** осуществляет **принудительный переход к следующему шагу цикла (**итерации), пропуская любой код, оставшийся до конца тела цикла. Таким образом, оператор **continue** служит своего рода дополнением оператора **break**.

Пример 7

```
using System;
class ContDemo {
    static void Main() {
        for (int i = 0; i <= 100; i++) {
            if ((i%2) != 0) continue;
            Console.WriteLine (i);
        }
    }
}
```

Пример 8

Программа вычисляет значение функции **Sin x** (синус) с точностью $\varepsilon = 10^{-6}$ с помощью бесконечного ряда Тейлора по формуле:

$y = x - x^3/3! + x^5/5! - x^7/7! + \ldots$

Этот ряд сходится при $|x| < \infty$. Точность достигается при $|Rn| < \varepsilon$, где Rn — остаточный член ряда, который для данного ряда можно заменить величиной C_n очередного члена ряда, прибавляемого к сумме.

Пример 8

Алгоритм решения задачи выглядит так: задать начальное значение *суммы ряда*, а затем многократно вычислять очередной член ряда и добавлять его к ранее найденной сумме. Вычисления заканчиваются, когда абсолютная величина очередного члена ряда станет меньше заданной точности.

Пример 8

Для уменьшения количества выполняемых действий следует воспользоваться рекуррентной формулой получения последующего члена ряда через предыдущий: $C_{n+1} = C_n$ x T, где T — некоторый множитель. Если подставить в эту формулу C_n и C_{n+1}, получится выражение для вычисления T:

$$T = 1 + \frac{c_{n+1}}{c_n} + \frac{2n! \cdot x^{2(n+1)}}{x^{2n} \cdot (2(n+1))!} = \frac{x^2}{(2n+1)(2n+2)}$$

Пример 8

```
using System;namespace ConsoleApplication1{
   class Class1{
     static void Main(){
       double e = 10e-6;
       const int MaxIter = 500;
       Console.WriteLine( "Введите аргумент:" );
       double x =
                Convert.ToDouble(Console.ReadLine());
       bool done = true;
       double ch = x, y = ch;
```

Пример 8

```
        for ( int n = 0; Math.Abs(ch) > e; n++ ) {
            ch *= x * x / ( 2 * n + 1 ) / ( 2 * n + 2 );
            y += ch;
            if ( n <= MaxIter ) continue;
            done = false;
            break;
        }
        if (done) Console.WriteLine( "Сумма ряда - " + y);
        else Console.WriteLine( "Ряд расходится");
    }
  }
}
```

Оператор return

Оператор возврата из функции **return** завершает выполнение функции и передает управление в точку ее вызова. Синтаксис оператора:

return [выражение];

Тип выражения должен иметь неявное преобразование к типу функции. Если тип возвращаемого функцией значения описан как **void**, ***выражение*** должно отсутствовать.

Пример 9

```
class Program{
   public static string Hello() {
       return "Hell to World";
   }
   public static void Ho(int k){
       if (k == 0){
          Console.WriteLine("k = " + k);
          return;
       }
       Console.WriteLine("k = " + k);
       Console.WriteLine("До свидания!");
   }
```

Пример 9

```
  static void Main(string[] args){
     string message = Hello(); // вызов первого метода
     Console.WriteLine(message);
     Console.WriteLine( "Вызов метода \"Ho\" с
   параметром 0");
     Ho(0);
     Console.WriteLine("Вызов метода \"Ho\" с
   параметром 1");
     Ho(1);
     Console.ReadKey();  // остановка экрана
    }
  }
}
```

Обработка исключительных ситуаций

Исключительная ситуация, или ***исключение***, — это возникновение аварийного события, которое может порождаться некорректным использованием аппаратуры или неправильной работой программы, например, делением на ноль или переполнением. Исключения генерирует либо среда выполнения, либо программист с помощью оператора **throw**.

Обработка исключительных ситуаций

Часто используемые стандартные исключения - потомки класса **SystemException**, который является потомком класса **Exception**.	
Имя	**Описание**
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

Обработка исключительных ситуаций

Исключения обнаруживаются и обрабатываются в операторе **try**, который содержит три части:

1. ***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;

Обработка исключительных ситуаций

2. один или несколько ***обработчиков исключений*** — блоков **catch**, в которых описывается, как обрабатываются ошибки различных типов;

3. ***блок завершения*** **finally** выполняется независимо от того, возникла ошибка в контролируемом блоке или нет.

Синтаксис оператора **try**:

try блок [блоки **catch**] [блок **finally**]

Отсутствовать могут либо блоки **catch**, либо блок **finally**, но не оба одновременно.

Обработка исключительных ситуаций

Порядок обработки исключительных ситуаций.

1. Обработка исключения начинается с появления ошибки в блоке **try**. Функция или операция, в которой возникла ошибка, генерирует исключение.

2. Выполнение текущего блока **try** прекращается, отыскивается соответствующий обработчик исключения **catch** , и ему передается управление.

3. Выполняется блок **finally**, если он присутствует.

4. Если обработчик не найден, вызывается стандартный обработчик исключения. Обычно он выводит на экран окно с информацией об исключении и завершает текущий процесс.

Обработка исключительных ситуаций

Обработчики исключений должны располагаться непосредственно за блоком **try**. Они начинаются с ключевого слова **catch**, за которым в скобках следует тип обрабатываемого исключения.
Блоки **catch** просматриваются в том порядке, в котором они записаны, пока не будет найден соответствующий типу выброшенного исключения.

Существуют три формы записи обработчиков:
catch(тип имя) { ... /* тело обработчика */ }
catch(тип) { ... /* тело обработчика */ }
catch { ... /* тело обработчика */ }

Пример 10

```
try {
      ...  // Контролируемый блок
}
catch ( OverflowException e ) {
   ... // Обработка исключений класса
       // OverflowException (переполнение)
}
catch ( DivideByZeroException ) {
   ... // Обработка исключений класса
       // DivideByZeroException (деление на 0)
}
catch {
     ...    // Обработка всех остальных исключений
}
```

Обработка исключительных ситуаций

Если исключение в контролируемом блоке **try** не возникло, все обработчики **catch** пропускаются.

В любом случае, произошло исключение или нет, управление передается в блок завершения **finally** (если он существует), а затем — первому оператору, находящемуся непосредственно за оператором **try**.

Операторы **try** могут многократно вкладываться друг в друга. Исключение, которое возникло во внутреннем блоке **try** и не было перехвачено соответствующим блоком **catch**, передается на верхний уровень, где продолжается поиск подходящего обработчика. Этот процесс называется ***распространением исключения***.

Оператор throw

Для генерации исключения используется оператор **throw** с параметром, определяющим вид исключения. Параметр должен быть объектом, порожденным от стандартного класса **System.Exception**. Этот объект используется для передачи информации об исключении его обработчику.

Синтаксис оператора **throw**:

throw [выражение];

Форма без параметра применяется только внутри блока **catch** для повторной генерации исключения. Тип выражения, стоящего после **throw**, определяет тип исключения, например:

throw new DivideByZeroException();

Контрольные вопросы

1 В составе каких операторов чаще всего применяется составной оператор?

2 В каких случаях применяется пустой оператор?

3 Какие операторы передачи управления (безусловного перехода) вы знаете?

4 Чем отличается оператор **continue** от **break**?