Объектно-ориентированное программирование

Лекция №4. Массивы, исключения, перечисления

# Темы:

1. Массивы. Одномерные массивы. Двумерные массивы. Прямоугольные и ступенчатые массивы. Класс System.Array. Оператор foreach.

2. Исключительные ситуации. Операторs try, catch, finally. Оператор throw. Класс Exception

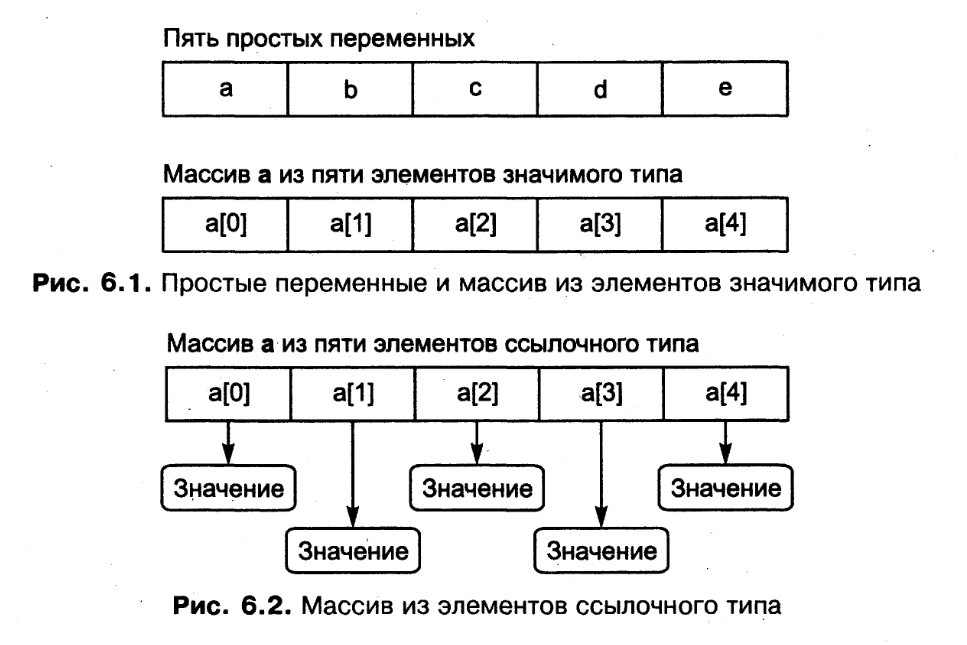
3. Перечисления. Объявление перечислений. Использование перечислений.

### Массивы

До настоящего момента мы использовали в программах простые переменные. При этом каждой области памяти, выделенной для хранения одной величины, соответствует свое имя. Если переменных много, программа, предназначенная для их обработки, получается длинной и однообразной. Поэтому практически в любом языке есть понятие массива — ограниченной совокупности однотипных величин.

Элементы массива имеют одно и то же имя, а различаются порядковым номером (индексом). Это позволяет компактно записывать множество операций с помощью циклов.

Массив относится к ссылочным типам данных, то есть располагается в динамической области памяти, 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). 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.



На рис. 6.1 представлен массив, состоящий из пяти элементов любого значимого типа, например int или double, а рис. 6.2 иллюстрирует организацию массива из элементов ссылочного типа.

Вот, например, как выглядят операторы, создания массива из 10 целых чисел и массива из 100 строк:

int[] w = new int[10];

string[] z = new string[100];

В первом операторе описан массив w типа int[].Операция new выделяет память под 10 целых элементов, и они заполняются нулями. Во втором операторе описан массив z типа string[].Операция new выделяет память под 100 ссылок на строки, и эти ссылки заполняются значением null. Память под сами строки, составляющие массив, не выделяется — это будет необходимо сделать перед заполнением массива.

Количество элементов в массиве (размерность) не является частью его типа, это количество задается при выделении памяти и не может быть изменено впоследствии. Размерность может задаваться не только константой, но и выражением. Результат вычисления этого выражения должен быть неотрицательным, а его тип должен иметь неявное преобразование к int, uint, long или ulong.

Элементы массива нумеруются с нуля, поэтому максимальный номер элемента всегда на единицу меньше размерности (например, в описанном выше массиве w элементы имеют индексы от 0 до 9). Для обращения к элементу массива после имени массива указывается номер элемента в квадратных скобках.

С элементом массива можно делать все, что допустимо для переменных того же типа. При работе с массивом автоматически выполняется контроль выхода за его границы: если значение индекса выходит за границы массива, генерируется исключение IndexOutOfRangeException.

Массивы одного типа можно присваивать друг другу. При этом происходит присваивание ссылок, а не элементов, как и для любого другого объекта ссылочного типа, например:

int[] а = new int[10];

int[] b = а; // b и а указывают на один и тот же массив

Все массивы в C# имеют общий базовый класс Array, определенный в пространстве имен System. В нем есть несколько полезных методов, упрощающих работу с массивами, например методы получения размерности, сортировки и поиска. Мы рассмотрим эти методы немного позже в разделе «Класс System.Array».

В C# существуют три разновидности массивов: одномерные, прямоугольные и ступенчатые (невыровненные).

#### Одномерные массивы

Одномерные массивы используются в программах чаще всего. 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 для каждого варианта описания):

int[] а; // 1 элементов нет

int[] b = new int[4]: // 2 элементы равны 0

int[] с = {61,2,5,-9}; // 3 new подразумевается

int[] d = new int[] {61,2,5,-9}; // 4 размерность вычисляется

int[] e = new int[4] {61,2,5,-9}; // 5 избыточное описание

Здесь описано пять массивов. Отличие первого оператора от остальных состоит в том, что в нем, фактически, описана только ссылка на массив, а память под элементы массива не выделена. Если список инициализации не задан, размерность может быть не только константой, но и выражением типа, приводимого к целому.

В каждом из остальных массивов по четыре элемента целого типа. Как видно из операторов 3—5, массив при описании можно инициализировать. Если при этом не задана размерность (оператор 4), количество элементов вычисляется по количеству инициализирующих значений. Для полей объектов и локальных переменных можно опускать операцию new, она будет выполнена по умолчанию (оператор 3). Если присутствует и размерность, и список инициализаторов, размерность должна быть константой (оператор 5).

using System;

namespace ConsoleApplication

{

class Class1

{

static void Main()

{

const int n = 6;

int[] a = new int[n] { 3, 12, 5, -9, 8, -4 };

Console.WriteLine("Исходный массив:");

for (int i = 0; i < n; ++i)

Console.Write("\t" + a[i]);

Console.WriteLine();

long sum = 0; // сумма отрицательных элементов

int num = 0; // количество отрицательных элементов

for (int i = 0; i < n; ++i)

if (a[i] < 0)

{

sum += a[i];

++num;

}

Console.WriteLine("Сумма отрицательных = " + sum);

Console.WriteLine("Кол-во отрицательных = " + num);

int max = a[0]; // максимальный элемент

for (int i = 1; i < n; ++i)

if (a[i] > max) max = a[i];

Console.WriteLine("Максимальный элемент = " + max);

}

}

}

#### 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 для каждого варианта описания):

int[,] а; // 1 элементов нет

int[,] b = new int[2, 3]; // 2 элементы равны 0

int[,] с = { { 1, 2, 3 }, { 4, 5, 6 } }; // 3 new подразумевается

int[,] d = new int[,] { { 1, 2, 3 }, { 4, 5, 6 } }; // 4 размерность вычисляется

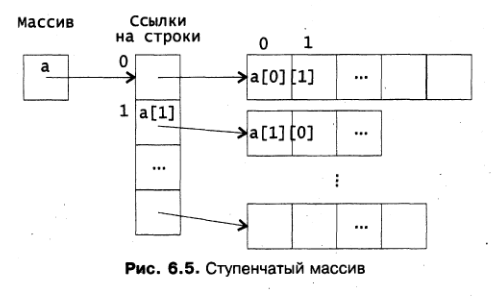
int[,] e = new int[2, 3] { { 1, 2, 3 }, { 4, 5, 6 } }; // 5 избыточное описание

Если список инициализации не задан, размерности могут быть не только константами, но и выражениями типа, приводимого к целому. К элементу двумерного массива обращаются, указывая номера строки и столбца, на пересечении которых он расположен, например:

а[1, 4] b[i, j] b[j, i]

#### Ступенчатые массивы

В ступенчатых массивах количество элементов в разных строках может различаться. В памяти ступенчатый массив хранится иначе, чем прямоугольный: в виде нескольких внутренних массивов, каждый из которых имеет свой размер. Кроме того, выделяется отдельная область памяти для хранения ссылок на каждый из внутренних массивов. Организацию ступенчатого массива иллюстрирует рис. 6.5.



Описание ступенчатого массива:

тип[][] имя;

Под каждый из массивов, составляющих ступенчатый массив, память требуется выделять явным образом, например:

int[][] а = new int[3][]; //выделение памяти под ссылки на три строки

а[0] = new int[5]; //выделение памяти под 0-ю строку (5 элементов)

а[1] = new int[3]; //выделение памяти под 1-ю строку (3 элемента)

а[2] = new int[4]; //выделение памяти под 2-ю строку (4 элемента)

Здесь а[0], а[1] и а [2] — это отдельные массивы, к которым можно обращаться по имени (пример приведен в следующем разделе). Другой способ выделения памяти:

int[][] а = { new int[5], new int[3], new int[4] };

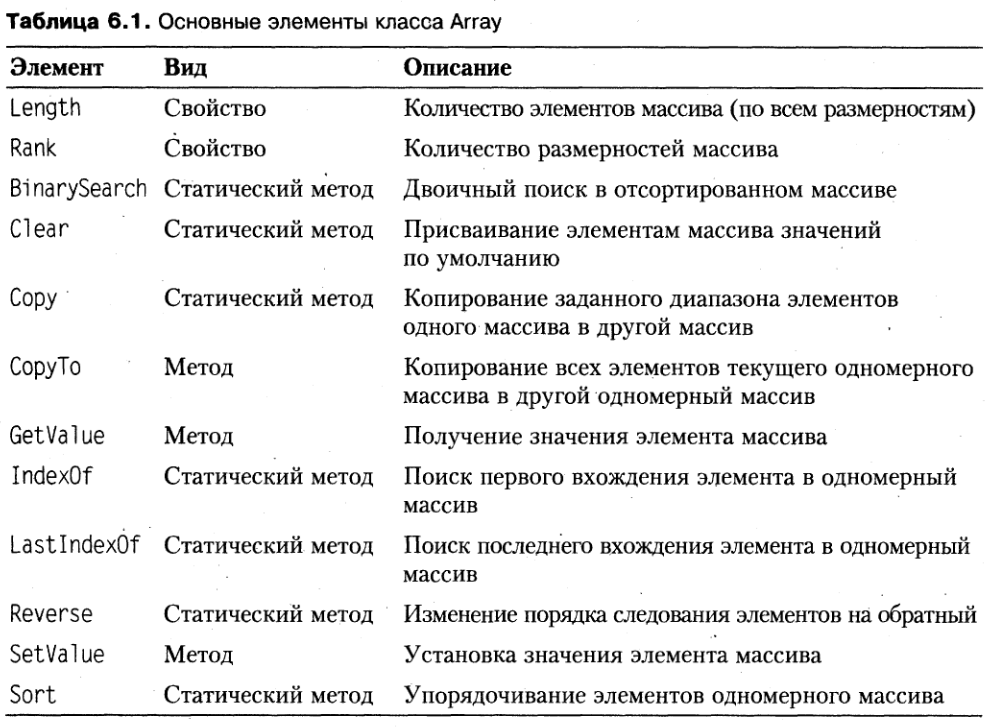
К элементу ступенчатого массива обращаются, указывая каждую размерность в своих квадратных скобках, например:

а[1][2] a[i][j] a[j][i]

В остальном использование ступенчатых массивов не отличается от использования прямоугольных. Невыровненные массивы удобно применять, например, для работы с треугольными матрицами большого объема.

#### Класс System.Array

Ранее уже говорилось, что все массивы в C# построены на основе базового класса Array, который содержит полезные для программиста свойства и методы, часть из которых перечислены в табл. 6.1.



Свойство Length позволяет реализовывать алгоритмы, которые будут работать с массивами различной длины или, например, со ступенчатым массивом. Использование этого свойства вместо явного задания размерности исключает возможность выхода индекса за границы массива. Ниже продемонстрировано применение элементов класса Array при работе с одномерным массивом.

using System;

namespace ConsoleApplication1

{

class Class1

{

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);

PrintArray("Упорядоченный массив;", a);

Console.WriteLine(Array.BinarySearch(a, 18));

}

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();

}

}

}

Методы Sort, IndexOf и BinarySearch являются статическими, поэтому к ним обращаются через имя класса, а не экземпляра, и передают в них имя массива. Двоичный поиск можно применять только для упорядоченных массивов. Он выполняется гораздо быстрее, чем линейный поиск, реализованный в методе IndexOf. В листинге поиск элемента, имеющего значение 18, выполняется обоими этими способами.

#### Оператор foreach

Оператор foreach применяется для перебора элементов в специальным образом организованной группе данных. Массив является именно такой группой. Удобство этого вида цикла заключается в том, что нам не требуется определять количество элементов в группе и выполнять их перебор по индексу: мы просто указываем на необходимость перебрать все элементы группы. Синтаксис оператора:

foreach ( тип имя in выражение ) тело цикла

Имя задает локальную по отношению к циклу переменную, которая будет по очереди принимать все значения из массива выражение (в качестве выражения чаще всего применяется имя массива или другой группы данных). В простом или составном операторе, представляющем собой тело цикла, выполняются действия с переменной цикла. Тип переменной должен соответствовать типу элемента массива.

Например, пусть задан массив:

int[] а = { 24, 50, 18, 3, 16. -7, 9, -1 }:

Вывод этого массива на экран с помощью оператора foreach выглядит следующим образом:

foreach (int х in а) Console.WriteLine(х);

Этот оператор выполняется так: на каждом проходе цикла очередной элемент массива присваивается переменной х и с ней производятся действия, записанные в теле цикла.

Ступенчатый массив вывести на экран с помощью оператора foreach немного сложнее, чем одномерный, но все же проще, чем с помощью цикла for:

foreach (int[] х in а)

{

foreach (int у in x)

Console.Write("\t" + у);

Console.WriteLine();

}

### Обработка исключительных ситуаций

В языке C# есть операторы, позволяющие обнаруживать и обрабатывать ошибки (исключительные ситуации), возникающие в процессе выполнения программы.

Исключительная ситуация, или исключение, — это возникновение аварийного события, которое может порождаться некорректным использованием аппаратуры или неправильной работой программы, например делением на ноль или переполнением. Обычно эти события приводят к завершению программы с системным сообщением об ошибке. C# дает программисту возможность восстановить работоспособность программы и продолжить ее выполнение.

Исключения C# не поддерживают обработку асинхронных событий, таких как ошибки оборудования или прерывания, например нажатие клавиш Ctrl+C. Механизм исключений предназначен только для событий, которые могут произойти в результате работы самой программы и указываются явным образом. Исключения возникают тогда, когда некоторая часть программы не смогла сделать то, что от нее требовалось. При этом другая часть программы может попытаться сделать что-нибудь иное.

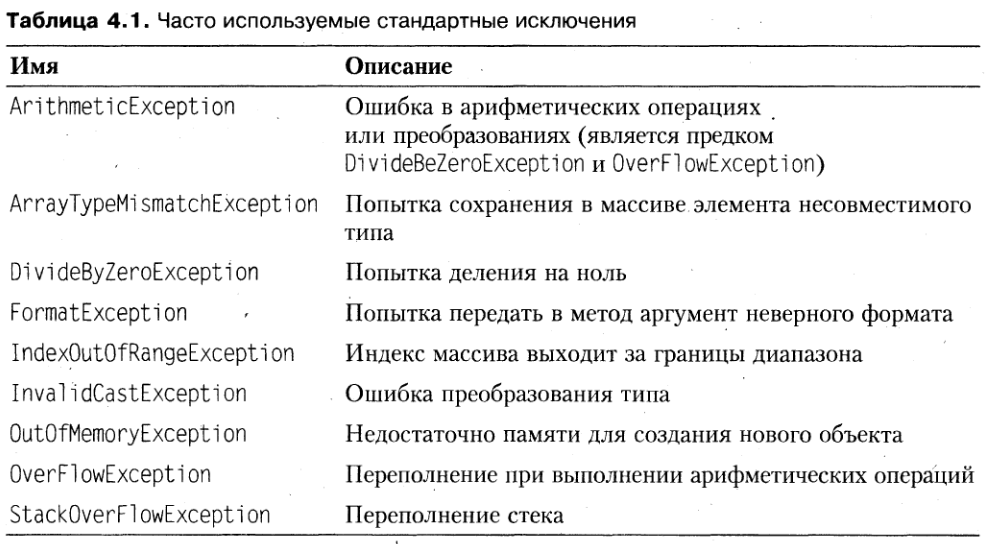
Исключения позволяют логически разделить вычислительный процесс на две части —

обнаружение аварийной ситуации и ее обработка. Это важно не только для лучшей структуризации программы. Главное то, что функция, обнаружившая ошибку, может не знать, что предпринимать для ее исправления, а использующий эту функцию код может знать, что делать, но не уметь определить место возникновения. Это особенно актуально при использовании библиотечных функций и программ, состоящих из многих модулей.

Другое достоинство исключений состоит в том, что для передачи информации об ошибке в вызывающую функцию не требуется применять возвращаемое значение или параметры, поэтому заголовки функций не разрастаются.

Исключения генерирует либо среда выполнения, либо программист с помощью оператора throw. В табл. 4Л приведены наиболее часто используемые стандартные исключения, генерируемые средой. Они определены в пространстве имен System. Все они являются потомками класса Exception, а точнее, потомками его потомка SystemException.

Исключения обнаруживаются и обрабатываются в операторе try.



#### Оператор try

Оператор try 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 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.

Рассмотрим, каким образом реализуется обработка исключительных ситуаций.

1. Обработка исключения начинается с появления ошибки. Функция или операция, в которой возникла ошибка, генерирует исключение. Как правило, исключение генерируется не непосредственно в блоке try, а в функциях, прямо или косвенно в него вложенных.
2. Выполнение текущего блока прекращается, отыскивается соответствующий обработчик исключения, и ему передается управление.
3. Выполняется блок final1у, если он присутствует (этот блок выполняется и в том случае, если ошибка не возникла).
4. Если обработчик не найден, вызывается стандартный обработчик исключения. Его действия зависят от конфигурации среды. Обычно он выводит на экран окно с информацией об исключении и завершает текущий процесс.

Обработчики исключений должны располагаться непосредственно за блоком try. Они начинаются с ключевого слова catch, за которым в скобках следует тип обрабатываемого исключения. Можно записать один или несколько обработчиков в соответствии с типами обрабатываемых исключений. Блоки catch просматриваются в том порядке, в котором они записаны, пока не будет найден соответствующий типу выброшенного исключения.

Синтаксис обработчиков напоминает определение функции с одним параметром — типом исключения. Существуют три формы записи:

catch( тип имя ){ ... /\* тело обработчика \*/ }

catch( тип ) {.../\* тело обработчика \*/ }

catch { ... /\* тело обработчика \*/ }

Первая форма применяется, когда имя параметра используется в теле обработчика для выполнения каких-либо действий, например вывода информации об исключении.

Вторая форма не предполагает использования информации об исключении, играет роль только его тип.

Третья форма применяется для перехвата всех исключений. Так как обработчики просматриваются в том порядке, в котором они записаны, обработчик третьего типа (он может быть только один) следует помещать после всех остальных. Пример:

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

}

Если исключение в контролируемом блоке не возникло, все обработчики 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. В завершающем блоке обычно записываются операторы, которые необходимо выполнить независимо от того, возникло исключение или нет, например, закрытие файлов, с которыми выполнялась работа в контролируемом блоке, или вывод информации.

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.

Распространение исключений предоставляет программисту интересные возможности. Например, если на внутреннем уровне недостаточно информации для того, чтобы провести полную обработку ошибки, можно выполнить частичную обработку и сгенерировать исключение повторно, чтобы оно было обработано на верхнем уровне. Генерация исключения выполняется с помощью оператора throw.

#### Оператор throw

До сих пор мы рассматривали исключения, которые генерирует среда выполнения С#, но это может сделать и сам программист. 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.

Оператор throw употребляется либо с параметром, либо без него:

throw [ выражение ];

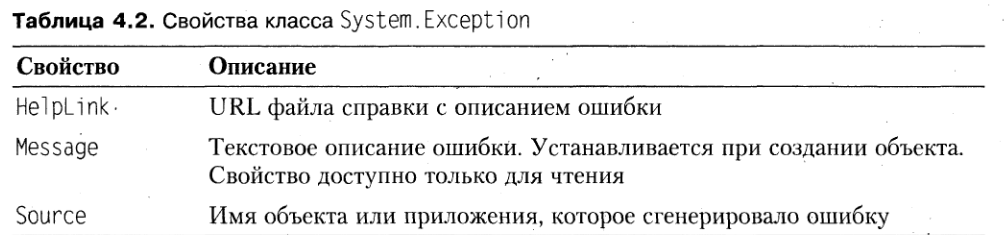
Форма без параметра применяется только внутри блока catch для повторной генерации исключения. Тип выражения, стоящего после throw, определяет тип исключения, например:

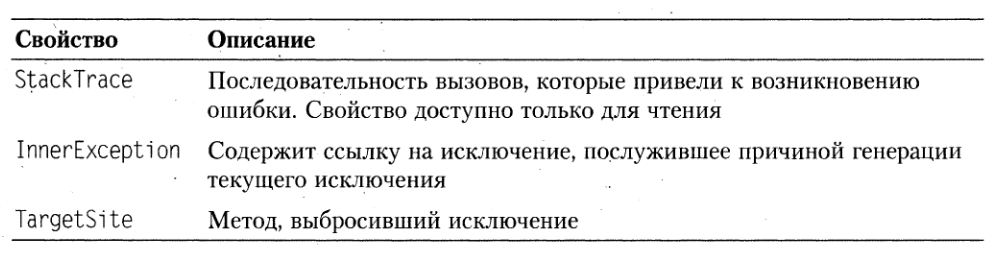
throw new DivideByZeroException();

Здесь после слова throw записано выражение, создающее объект стандартного класса «ошибка при делении на 0» с помощью операции new. При генерации исключения выполнение текущего блока прекращается и происходит поиск соответствующего обработчика с передачей ему управления. Обработчик считается найденным, если тип объекта, указанного после throw, либо тот же, что задан в параметре catch, либо является производным от него.

#### Класс Exception

Класс Exception содержит несколько полезных свойств, с помощью которых можно получить информацию об исключении. Они перечислены в табл. 4.2.





### Перечисления

Перечисление (Enumeration) – это определяемый пользователем целочисленный тип, который позволяет специфицировать набор допустимых значений, и назначить каждому понятное имя.

Для объявления перечисления используется ключевое слово enum. Общая структура объявления перечисления выглядит так:

enum [имя\_перечисления] { [имя1], [имя2], … };

Например, создадим перечисление Directions, которое будет соответствовать направлениям движения:

enum Directions { Left, Right, Forward, Back };

Объявив таким образом перечисление, каждой символически обозначаемой константе присваивается целочисленное значение, начиная с 0 (Left = 0, Right = 1 …). Это целочисленное значение можно задавать и самому:

enum Directions { Left, Right = 5, Forward = 10, Back };

Back в этом примере будет иметь значение 11.

Пример программы с использованием перечисления:

using System;

enum Directions { Left, Right, Forward, Back }; // объявление перечисления

class Program

{

public static void GoTo(Directions direction)

{

switch (direction)

{

case Directions.Back:

Console.WriteLine("Go back");

break;

case Directions.Forward:

Console.WriteLine("Go forward");

break;

case Directions.Left:

Console.WriteLine("Turn left");

break;

case Directions.Right:

Console.WriteLine("Turn right ");

break;

}

}

static void Main(string[] args)

{

Directions direction = Directions.Forward;

GoTo(direction); // "Go forward"

Console.ReadKey();

}

}

Чтобы получить целое значение определенного элемента перечисления, достаточно этот элемент явно привести к целому типу:

enum Directions : byte { Left, Right, Forward, Back };

class Program

{

static void Main(string[] args)

{

Console.WriteLine((int)Directions.Forward); // 2

Console.ReadKey();

}

}

По умолчанию в качестве целого типа для enum используется int. Этот тип можно изменить на любой другой целый тип (кроме char), указав после имени перечисления необходимый тип и разделив двоеточием:

enum Directions : byte { Left, Right, Forward, Back };

Зачем нужны перечисления и где они используются

Главные преимущества, которые нам дают перечисления это:

* Гарантия того, что переменным будут назначаться допустимые значения из указанного набора;
* Когда вы пишите код программы в Visual Studio, благодаря средству IntelliSense будет выпадать список с допустимыми значениями, что позволит сэкономить некоторое время, и напомнить, какие значения можно использовать;
* Код становится читабельнее, когда в нем присутствуют понятные имена, а не ни о чем не говорящие числа.

Перечисления очень широко используются в самой библиотеке классов .NET. Например, при создании файлового потока (FileStream) используется перечисление FileAccess, при помощи которого мы указываем с каким режимом доступа открыть файл (чтение/запись).

# Использованная литература

1. Павловская Т.А. C# Программирование на языке высокого уровня, Питер 2014г. 432 с.