

**СОЦИАЛЬНО-ГУМАНИТАРНЫЙ КОЛЛЕДЖ  
УО «МОГИЛЕВСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ  
ИМЕНИ А.А. КУЛЕШОВА»**



**Дисциплина  
«Конструирование программ и языки программирования»**

**Обработка исключительных ситуаций  
(2 часа)**

Методические рекомендации к лабораторной работе №14

Могилев 2019

Понятие «исключительная ситуация». Методические указания по лабораторной работе №14 «Конструирование программ и языки программирования». Для учащихся очной формы обучения специальности 1–40 01 01 «Программное обеспечение информационных технологий».

## Оглавление

1 Цель работы.....	4
2 Ход работы .....	5
3 Краткие теоретические сведения .....	6
3.1 Исключительные ситуации .....	6
3.1.1 Генерация исключений .....	7
3.1.2 Последствия возникновения перехватываемых исключений .....	9
3.1.3 Возможность красиво выходить из ошибочных ситуаций .....	11
3.1.4 Возврат из исключения .....	12
3.1.5 Использование нескольких catch-конструкций.....	13
3.1.6 Перехват всех исключений .....	14
3.1.7 Вложение try-блоков.....	15
3.2 Создание собственных классов и генерация исключений .....	17
3.2.1 Наследование классов исключений.....	20
4 Задания.....	21
5 Контрольные вопросы.....	22

## **1 Цель работы**

Целью лабораторной работы является:

1. Познакомиться с механизмами событийно-ориентированного программирования на языке C#, такими как механизм обработка исключительных ситуаций.

## **2   Ход работы**

1. Изучение теоретического материала.
2. Выполнение практических индивидуальных заданий по вариантам (вариант уточняйте у преподавателя).
3. Оформление отчета.
  - 3.1.Отчет оформляется индивидуально каждым студентом. Отчет должен содержать задание, алгоритм и листинг программы.
  - 3.2.Отчет по лабораторной работе выполняется на листах формата А4. В состав отчета входят:
    - 1) титульный лист;
    - 2) цель работы;
    - 3) текст индивидуального задания;
    - 4) выполнение индивидуального задания.

## **3   4. Контрольные вопросы.**

### 3 Краткие теоретические сведения

#### 3.1 Исключительные ситуации

Исключение представляет собой ошибку, происходящую во время выполнения программы. С помощью подсистемы обработки исключений для C# можно обрабатывать такие ошибки, не вызывая краха программы.

Обработка исключений в C# выполняется с применением четырех ключевых слов: `try`, `catch`, `throw` и `finally`. Эти ключевые слова образуют взаимосвязанную подсистему, в которой использование одного из ключевых слов влечет за собой использование других.

Основа обработки исключений основана на использовании блоков `try` и `catch`. Синтаксис:

```
try {  
    //Блок кода для которого выполняется мониторинг ошибок  
} catch (Exception ex) {  
    //Обработчик исключений Exception  
}  
catch (Exception2 ex) {  
    //Обработчик исключений Exception2  
}
```

Основные системные исключения приведены в следующей таблице:

Таблица 2.1 Основные системные исключения

Исключение	Значение
<code>ArrayTypeMismatchException</code>	Тип сохраненного значения несовместим с типом массива.
<code>DivideByZeroException</code>	Предпринята попытка деления на ноль
<code>StackOverflowException</code>	Переполнение стека.
<code>OutOfMemoryException</code>	Вызов <code>new</code> был неудачным из-за недостатка памяти.
<code>InvalidCastException</code>	Некорректное преобразование в процессе выполнения.
<code>IndexOutOfRangeException</code>	Индекс массива выходит за пределы диапазона.
<code>OverflowException</code>	Переполнение при выполнении арифметической операции

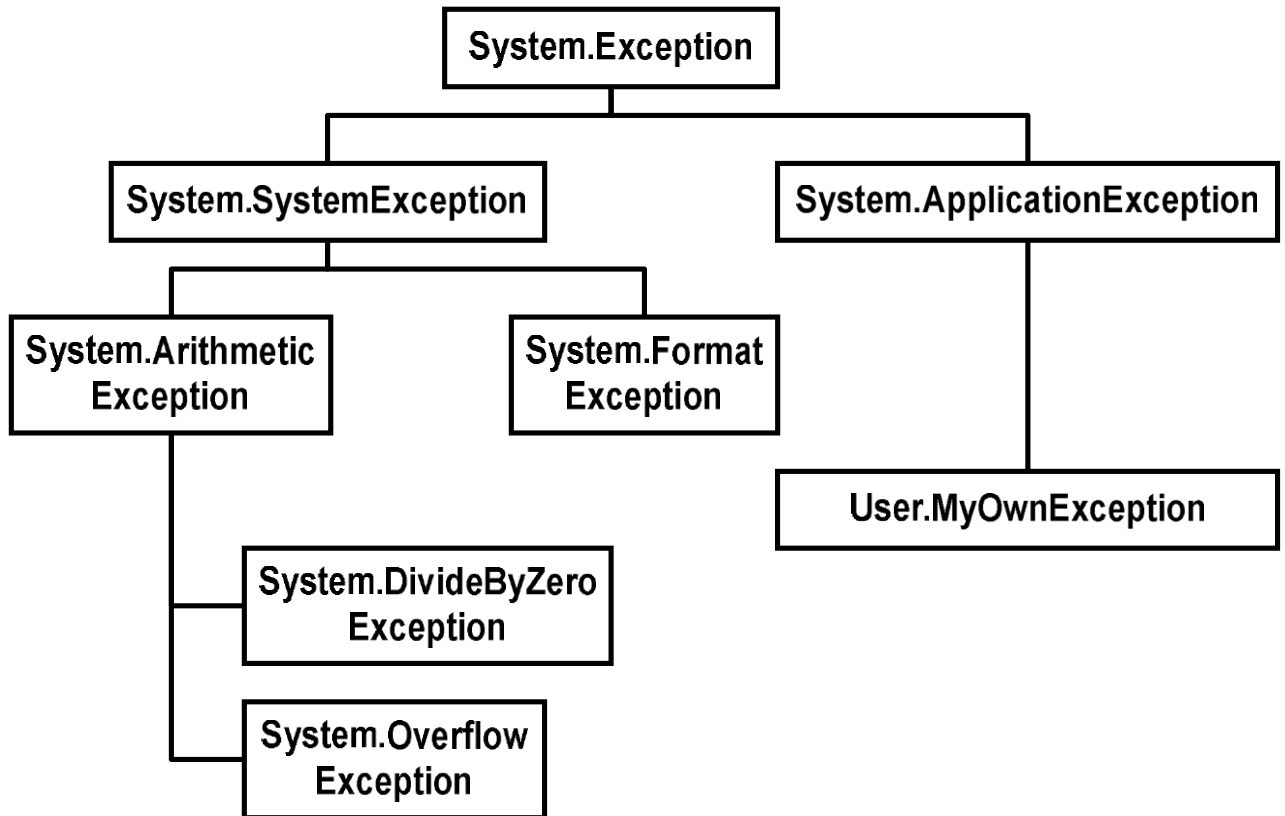


Рис 2.1 Иерархия исключений

Тип исключения в операторе `catch` должен соответствовать типу перехватываемого исключения. Неперехваченное исключение непременно приводит к досрочному прекращению выполнения программы.

Для выполнения перехвата исключений вне зависимости от их типа (перехват всех исключений) возможно использование `catch` без параметров.

### 3.1.1 Генерация исключений

Исключения автоматически генерируются системой. Однако исключение может быть сгенерировано и посредством оператора `throw`.

Формат ее записи таков:

```
throw [параметр];
```

Параметр – это объект класса исключений, производного от класса `Exception`.

Пример `double x`;

```
if (x == 0) throw new DivideByZeroException();
```

Исключение, перехваченное одной `catch`-инструкцией, можно регенерировать, чтобы обеспечить возможность его перехвата другой (внешней) `catch`-инструкцией. Чтобы повторно сгенерировать исключение, достаточно использовать ключевое слово `throw`, не указывая параметра. Приведем пример генерирования исключения вручную:

```
try
```

```

{
try
{
int v = Convert.ToInt32(Console.ReadLine());
Console.WriteLine("v=" + v);
}
catch (FormatException)
{ Console.WriteLine("Неверный ввод"); throw; }
}
catch (FormatException)
{ Console.WriteLine("Это очень плохо"); }

```

Один try-блок можно вложить в другой.

Исключение, сгенерированное во внутреннем try-блоке и не перехваченное catch-инструкцией, которая связана с этим try-блоком, передается во внешний try-блок. Часто внешний try-блок используют для перехвата самых серьезных ошибок, позволяя внутренним try-блокам обрабатывать менее опасные (см. п. 2.2.5).

Весь код, выполняемый внутри try-блока, проверяется на предмет возникновения исключительной ситуации. Сюда также относятся исключения, которые могут сгенерировать методы, вызываемые из блока try. Исключение, сгенерированное методом, вызванным из try-блока, может быть перехвачено этим try-блоком, если, конечно, метод сам не перехватит это исключение. Рассмотрим пример.

```

/* Исключение может сгенерировать один метод, а перехватить его — другой. */
using System;

class ExcTest
{
    // Генерируем исключение.
    public static void genException()
    {
        int[] nums = new int[4];
        Console.WriteLine("Перед генерированием исключения.");
        // Генерируем исключение, связанное с попаданием
        // индекса вне диапазона.
        for (int i=0; i < 10; i++) {
            nums[i] = i;
            Console.WriteLine("nums[{0}]: {1}", i, nums[i]);
        }
        Console.WriteLine("Этот текст не будет отображаться.");
    }
}

class ExcDemo2
{

```



```

public static void Main()
{
    try
    {
        ExcTest.genException();
    }
    catch (IndexOutOfRangeException)
    {
        // Перехватываем исключение.
        Console.WriteLine("Индекс вне диапазона!");
    }
    Console.WriteLine("После catch-инструкции.");
}
}

```

Результат:



```

C:\Windows\system32\cmd.exe
Перед генерированием исключения.
nums[0]: 0
nums[1]: 1
nums[2]: 2
nums[3]: 3
После catch-инструкции.

```

Поскольку метод `genException()` вызывается из блока `try`, исключение, которое он генерирует (и не перехватывает), перехватывается инструкцией `catch` в методе `Main()`. Но если бы метод `genException()` перехватывал это исключение, оно бы никогда не вернулось в метод `Main()`.

### 3.1.2 Последствия возникновения неперехватываемых исключений

Перехват одного из стандартных C#-исключений, как показала предыдущая программа, имеет побочный эффект: он предотвращает аварийное окончание программы. При генерировании исключения оно должно быть перехвачено программным кодом. Если программа не перехватывает исключение, оно перехватывается C#-системой динамического управления. Но дело в том, что система динамического управления сообщит об ошибке и завершит программу. Например, в следующем примере исключение, связанное с нарушением границ диапазона, программой не перехватывается.

```

// Предоставим возможность обработать ошибку
// C#-системе динамического управления.
using System;

class NotHandled
{
    public static void Main()
    {
        int[] nums = new int[4];

        Console.WriteLine("Перед генерированием исключения.");
    }
}

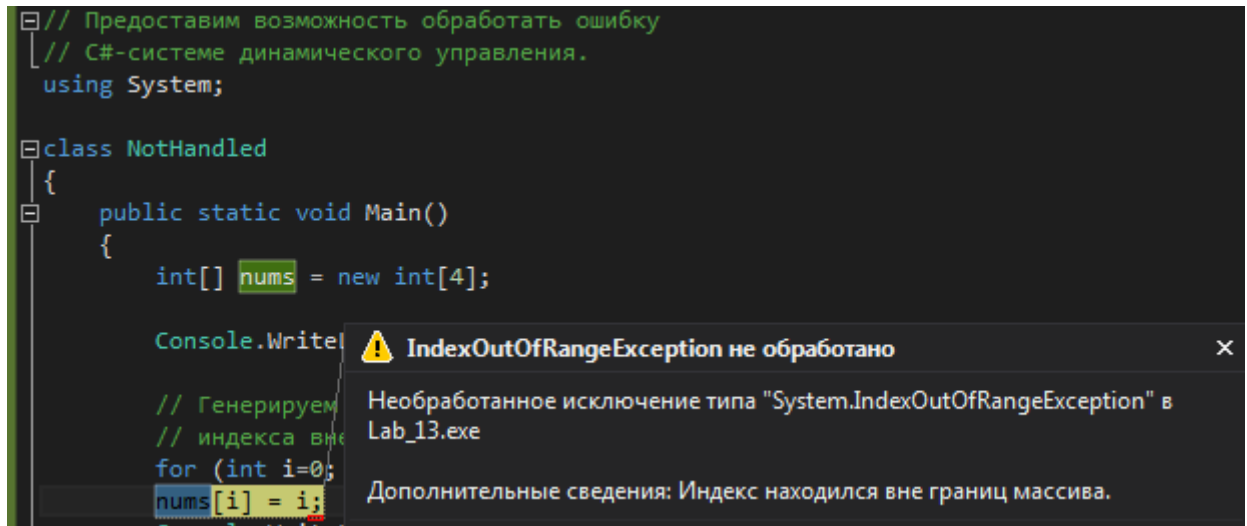
```

```

        // Генерируем исключение, связанное с попаданием
        // индекса вне диапазона.
for(int i=0; i < 10; i++) {
    nums[i] = i;
    Console.WriteLine("nums[{0} ] : {1}", i, nums[i]);
}
}
}

```

При неверном индексировании массива выполнение программы останавливается, и на экране отображается следующее сообщение об ошибке:



Несмотря на то что такое сообщение может быть полезным во время отладки программы, вряд ли вы захотите, чтобы его увидели пользователи! Поэтому важно, чтобы программы сами обрабатывали подобные исключения.

Как упоминалось выше, тип исключения должен совпадать с типом, заданным в `catch`-инструкции. В противном случае это исключение не будет перехвачено. Например, в следующей программе делается попытка перехватить ошибку нарушения индексом массива границ диапазона с помощью `catch`-инструкции для класса `DivideByZeroException` (это еще одно из встроенных C#-исключений). При нарушении границ диапазона, допустимого для индекса массива, генерируется исключение типа `IndexOutOfRangeException`, которое не перехватывается предусмотренной в программе `catch`-инструкцией. В результате программа завершается аварийно.

// Эта программа работать не будет!

```

using System;

class ExcTypeMismatch
{
    public static void Main()
    {
        int[] nums = new int[4]; try
        {
            Console.WriteLine("Перед генерированием исключения.");
        }
    }
}

```

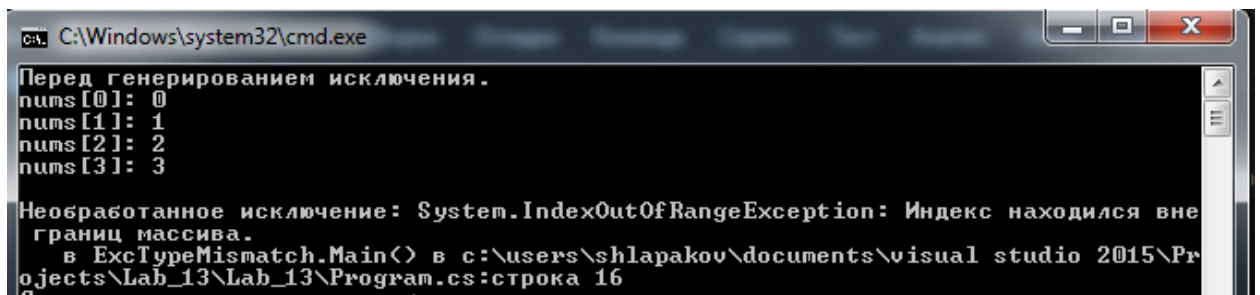
```

        // Генерируем исключение, связанное с попаданием
        // индекса вне диапазона.
        for (int i=0; i < 10; i++) {
            nums[i] = i;
            Console.WriteLine("nums[{0}]: {1}", i, nums[i]);
        }
        Console.WriteLine("Этот текст не отображается.");
    }

    /* Если в catch-инструкции указан тип исключения
    DivideByZeroException, то с ее помощью невозможно перехватить ошибку
    нарушения границ массива. */
    catch (DivideByZeroException) {
        // Перехватываем исключение.
        Console.WriteLine("Индекс вне границ диапазона!");
    }
    Console.WriteLine("После catch-инструкции.");
}
}
}

```

Вот как выглядят результаты выполнения этой программы:



```

C:\Windows\system32\cmd.exe
Перед генерированием исключения.
nums[0]: 0
nums[1]: 1
nums[2]: 2
nums[3]: 3
Необработанное исключение: System.IndexOutOfRangeException: Индекс находился вне
границ массива.
в ExcTypeMismatch.Main() в c:\users\shlapakov\documents\visual studio 2015\Pr
ojects\Lab_13\Lab_13\Program.cs:строка 16

```

Как подтверждают результаты выполнения этой программы, catch-инструкция, предназначенная для перехвата исключения типа `DivideByZeroException`, не в состоянии перехватить исключение типа `IndexOutOfRangeException`.

### 3.1.3 Возможность красиво выходить из ошибочных ситуаций

Одно из основных достоинств обработки исключений состоит в том, что она позволяет программе отреагировать на ошибку и продолжить выполнение. Рассмотрим, например, следующую программу, которая делит элементы одного массива на элементы другого. Если при этом встречается деление на нуль, генерируется исключение типа `DivideByZeroException`. В программе это исключение обрабатывается выдачей сообщения об ошибке, после чего выполнение программы продолжается. Следовательно, попытка разделить на нуль не вызывает внезапную динамическую ошибку, в результате которой прекращается выполнение программы. Вместо аварийного останова исключение позволяет красиво выйти из ошибочной ситуации и продолжить выполнение программы.

```

// Достойная реакция на ошибку и продолжение работы
// вот что значит с толком использовать исключения!

```

```
using System;
```

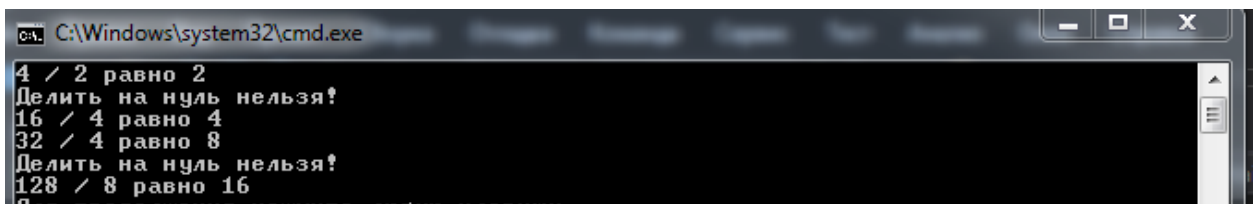
```

class ExcDemo3
{
    public static void Main()
    {
        int[] numer = { 4, 8, 16, 32, 64, 128 };
        int[] denom = { 2, 0, 4, 4, 0, 8 };

        for (int i = 0; i < numer.Length; i++)
        {
            try
            {
                Console.WriteLine(numer[i] + " / " + denom[i] + " равно " + numer[i] / denom[i]);
            }
            catch (DivideByZeroException)
            {
                // Перехватываем исключение.
                Console.WriteLine("Делить на ноль нельзя!");
            }
        }
    }
}

```

При выполнении эта программа демонстрирует следующие результаты:



```

C:\Windows\system32\cmd.exe
4 / 2 равно 2
Делить на ноль нельзя!
16 / 4 равно 4
32 / 4 равно 8
Делить на ноль нельзя!
128 / 8 равно 16

```

Эта программа демонстрирует еще один важный аспект обработки исключений.

После обработки исключение удаляется из системы. Таким образом, в этой программе

### 3.1.4 Возврат из исключения

Так как оператор `catch` не вызывается из программы, то после выполнения блока `catch` управление не передается обратно оператору программы, при выполнении которого возникло исключение. Выполнение программы продолжается с операторов, находящихся после блока `catch`.

С целью предотвращения этой ситуации возможно указание блока кода, который вызывается после выхода из блока `try/catch`, с помощью блока `finally` в конце последовательности `try/catch`. Общая форма конструкции `try/catch`, включающей блок `finally`, показана ниже:

```

try {
    // Блок кода, выполняющий мониторинг ошибок }

```

```

catch (Exception exObj)    {
//I Обработка исключения Exception1. }
catch (Exception2 exObj2)  {
//II Обработка исключения Exception2.
finally { // Код блока finally. }

```

Блок `finally` будет вызываться независимо от того, появится исключение или нет, и независимо от причин возникновения такового.

### 3.1.5 Использование нескольких `catch`-конструкций

С `try`-блоком можно связать не одно, а несколько `catch`-инструкций. И это – довольно распространенная практика программирования. Однако все `catch`-инструкции должны перехватывать исключения различного типа. Например, следующая программа перехватывает как ошибку нарушения границ массива, так и ошибку деления на нуль.

```

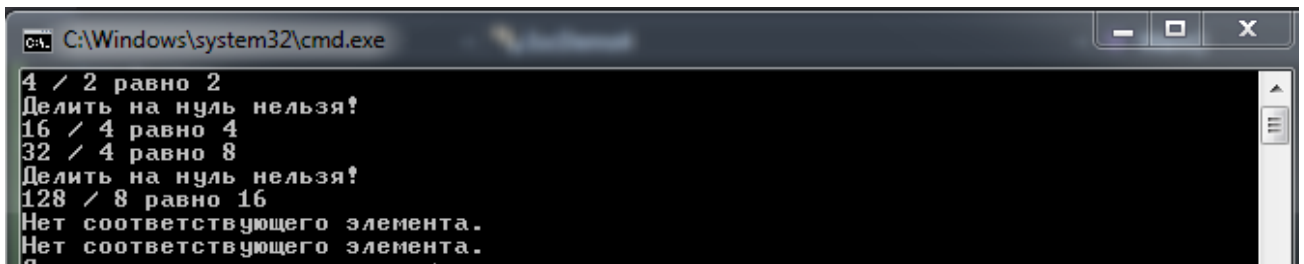
// Использование нескольких catch-инструкций.
using System;

class ExcDemo4
{
    public static void Main()
    {
        // Здесь массив number длиннее массива denom
        int[] number = { 4, 8, 16, 32, 64, 128, 256, 512 };
        int[] denom = { 2, 0, 4, 4, 0, 8 };

        for (int i = 0; i < number.Length; i++)
        {
            try
            {
                Console.WriteLine(number[i] + " / " +
                    denom[i] + " равно " + number[i] / denom[i]);
            }
            catch (DivideByZeroException)
            {
                // Перехватываем исключение.
                Console.WriteLine("Делить на нуль нельзя!");
            }
            catch (IndexOutOfRangeException)
            {
                // Перехватываем исключение.
                Console.WriteLine("Нет соответствующего элемента.");
            }
        }
    }
}

```

Эта программа генерирует следующие результаты:



```
C:\Windows\system32\cmd.exe
4 / 2 равно 2
Делить на ноль нельзя!
16 / 4 равно 4
32 / 4 равно 8
Делить на ноль нельзя!
128 / 8 равно 16
Нет соответствующего элемента.
Нет соответствующего элемента.
```

Как подтверждают результаты выполнения этой программы, каждая `catch`-инструкция реагирует только на собственный тип исключения.

В общем случае `catch`-выражения проверяются в том порядке, в котором они встречаются в программе. Выполняется только инструкция, тип исключения которой совпадает со сгенерированным исключением. Все остальные `catch`-блоки игнорируются.

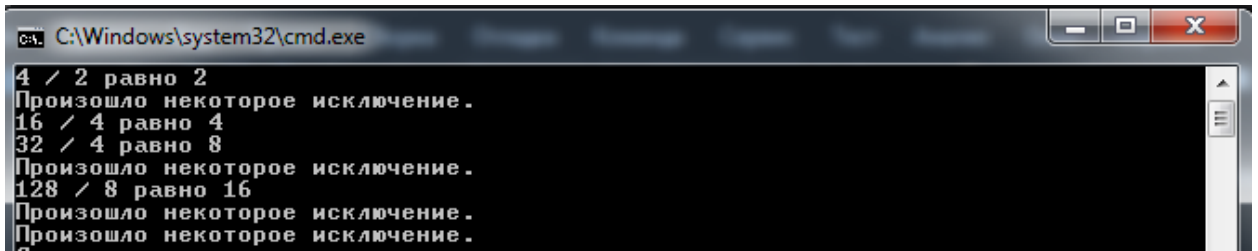
### 3.1.6 Перехват всех исключений

Иногда требуется перехватывать все исключения, независимо от их типа. Для этого используйте `catch`-инструкцию без параметров. В этом случае создается обработчик “глобального перехвата”, который используется, чтобы программа гарантированно обработала все исключения. В следующей программе приведен пример использования такого обработчика, который успешно перехватывает генерируемые здесь исключение типа `IndexOutOfRangeException` и исключение типа `DivideByZeroException`.

```
// Использование catch-инструкции для
// "глобального перехвата".
using System;
class ExcDemo5
{
    public static void Main()
    {
        // Здесь массив numer длиннее массива denom.
        int[] numer = { 4, 8, 16, 32, 64, 128, 256, 512 };
        int[] denom = { 2, 0, 4, 4, 0, 8 };

        for (int i = 0; i < numer.Length; i++)
        {
            try
            {
                Console.WriteLine(numer[i] + " / " + denom[i] + " рав-
но " + numer[i] / denom[i]);
            }
            catch
            {
                Console.WriteLine("Произошло некоторое исключение.");
            }
        }
    }
}
```

Вот как выглядят результаты выполнения этой программы:



```
C:\Windows\system32\cmd.exe
4 / 2 равно 2
Произошло некоторое исключение.
16 / 4 равно 4
32 / 4 равно 8
Произошло некоторое исключение.
128 / 8 равно 16
Произошло некоторое исключение.
Произошло некоторое исключение.
```

В отношении `catch`-инструкции, предназначенной для “глобального перехвата”, необходимо запомнить следующее: она должна быть последней в последовательности `catch`-инструкций.

### 3.1.7 Вложение `try`-блоков

Один `try`-блок можно вложить в другой. Исключение, сгенерированное во внутреннем `try`-блоке и не перехваченное `catch`-инструкцией, которая связана с этим `try`-блоком, передается во внешний `try`-блок. Например, в следующей программе исключение типа `IndexOutOfRangeException` перехватывается не внутренним `try`-блоком, а внешним.

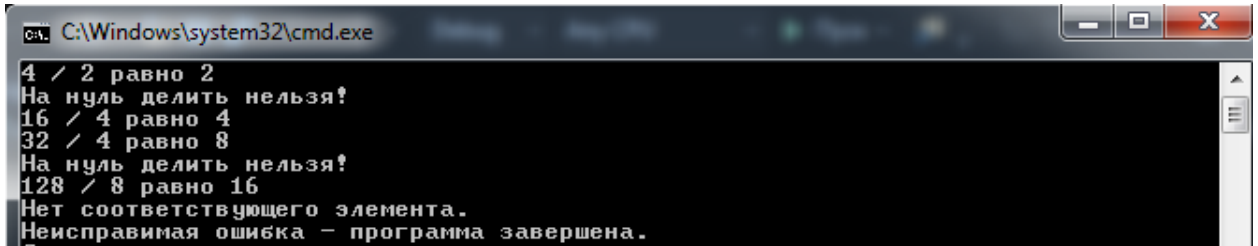
```
// Использование вложенного try-блока.
using System;
class NestTrys
{
    public static void Main()
    {
        // Здесь массив numer длиннее массива denom.
        int[] numer = { 4, 8, 16, 32, 64, 128, 256, 512 };
        int[] denom = { 2, 0, 4, 4, 0, 8 }; try
        {
            // Внешний try-блок.
            for (int i = 0; i < numer.Length; i++)
            {
                try
                {
                    // Вложенный try-блок.
                    Console.WriteLine(numer[i] + " / " +
                        denom[i] + " равно " + numer[i] / denom[i]);
                }
                catch (DivideByZeroException)
                {
                    // Перехватываем исключение.
                    Console.WriteLine("На нуль делить нельзя!");
                }
            }
        }
        catch (IndexOutOfRangeException)
        {
            // Перехватываем исключение.
            Console.WriteLine("Нет соответствующего элемента.");
        }
    }
}
```

```

Console.WriteLine("Неисправимая ошибка – программа завершена.");
    }
}
}

```

Вот результаты выполнения этой программы:



```

C:\Windows\system32\cmd.exe
4 / 2 равно 2
На ноль делить нельзя!
16 / 4 равно 4
32 / 4 равно 8
На ноль делить нельзя!
128 / 8 равно 16
Нет соответствующего элемента.
Неисправимая ошибка – программа завершена.

```

Исключение, которое может быть обработано внутренним `try`-блоком (в данном случае это деление на ноль), позволяет программе продолжать работу. Однако нарушение границ массива перехватывается внешним `try`-блоком и заставляет программу завершиться.

В предыдущей программе хочется обратить ваше внимание вот на что. Чаще всего использование вложенных `try`-блоков обусловлено желанием обрабатывать различные категории ошибок различными способами. Одни типы ошибок носят катастрофический характер и не подлежат исправлению. Другие – неопасны для дальнейшего функционирования программы, и с ними можно справиться прямо на месте их возникновения. Многие программисты используют внешний `try`-блок для перехвата самых серьезных ошибок, позволяя внутренним `try`-блокам обрабатывать менее опасные. Внешние `try`-блоки можно также использовать в качестве механизма “глобального перехвата” для обработки тех ошибок, которые не перехватываются внутренним блоком.

Одно из основных достоинств обработки исключений состоит в том, что она позволяет программе отреагировать на ошибку и продолжить выполнение. Рассмотрим, например, следующую программу, которая делит элементы одного массива на элементы другого. Если при этом встречается деление на ноль, генерируется исключение типа `DivideByZeroException`. В программе это исключение обрабатывается выдачей сообщения об ошибке, после чего выполнение программы продолжается. Следовательно, попытка разделить на ноль не вызывает внезапную динамическую ошибку, в результате которой прекращается выполнение программы. Вместо аварийного останова исключение позволяет красиво выйти из ошибочной ситуации и продолжить выполнение программы.

```

// Достойная реакция на ошибку и продолжение работы
// вот что значит с толком использовать исключения!

```

```

using System;

class ExcDemo3
{
    public static void Main()
    {
        int[] numer = { 4, 8, 16, 32, 64, 128 };
        int[] denom = { 2, 0, 4, 4, 0, 8 };
    }
}

```



```

        for (int i = 0; i < numer.Length; i++)
        {
            try
            {
                Console.WriteLine(numer[i] + " / " + denom[i] + " равно "
+ numer[i] / denom[i]);
            }
            catch (DivideByZeroException)
            {
                // Перехватываем исключение.
                Console.WriteLine("Делить на нуль нельзя!");
            }
        }
    }
}

```

При выполнении эта программа демонстрирует следующие результаты:



```

C:\Windows\system32\cmd.exe
4 / 2 равно 2
Делить на нуль нельзя!
16 / 4 равно 4
32 / 4 равно 8
Делить на нуль нельзя!
128 / 8 равно 16

```

Эта программа демонстрирует еще один важный аспект обработки исключений.

После обработки исключение удаляется из системы. Таким образом, в этой программе при каждом проходе через цикл заново вводится `try`-блок, обеспечивая полную “готовность” к обработке следующих исключений. Такая организация позволяет обрабатывать в программах повторяющиеся ошибки.

### 3.2 Создание собственных классов и генерация исключений

При использовании исключений разработчик может создать код для реакции на многие стандартные ошибки. Однако, в сложных программах ошибки порождаются не только кодом, но и логикой самой программы. Поэтому для более эффективной реакции на потенциальные ошибки создаются собственные классы исключений, которые содержат специализированную информацию о произошедшем событии и дают возможность отреагировать на него с максимальной пользой.

Рассмотрим пример генерации исключения стандартного класса. Такие исключения можно использовать в программе, чтобы выполнить стандартную обработку ошибок, которая может быть реализована в других библиотеках платформы. Аналогичным образом генерируются исключения, созданные самим пользователем.

В данном примере производится контроль диапазона значений для выполнения операции извлечения квадратного корня. В случае нарушения границ диапазона порождается исключение, которое обрабатывается программой. Для этого используется служебное слово `throw`. Данное исключение с тем же успехом может быть передано в другой модуль, вызвавший функцию вычисления квадратного корня, и обработано вызывающим модулем. В этом случае необходимо явно указать вызывающему модулю возможность по-

явления исключения.

```
public static double Sqrt(double aValue)
{
    if (aValue < 0)
        throw new System.ArgumentOutOfRangeException(
            "Функция не выполняется для отрицательных чисел!");
    return Math.Sqrt(aValue);
}
static void Main(string[] args)
{
    try
    {
        Sqrt(1);
    }
    catch (ArgumentOutOfRangeException ex)
    {
        Console.Error.WriteLine("Error: " + ex.Message); throw;
    }
}
```

Здесь порождается стандартное исключение класса `System.ArgumentOutOfRangeException`, в котором задается собственное сообщение пользователя, поясняющее причину ошибки.

Для определения собственного исключения наследуется класс `System.ApplicationException`, для которого создаются подходящие конструкторы. Ему также можно добавить дополнительные свойства, дающие информацию о возникшей проблеме (информационное сообщение, информацию о причинах исключения в виде значений некоторых параметров).

```
class ParseFileException : ApplicationException
{
    private string mFileName;
    private long mLineNumber;
    public string FileName
    {
        get
        {
            return mFileName;
        }
    }
    public long LineNumber
    {
        get
        {
            return mLineNumber;
        }
    }
}
```

```

        {
        }
    }
    return mLineNumber;
    public ParseFileException(string aMessage, string aFileName, long
aLineNumber, Exception aCauseException) : base(aMessage, aCauseExcep-
tion)
    {
        mFileName = aFileName; mLineNumber = aLineNumber;
    }
    public ParseFileException(string aMessage, string aFile-
Name, Exception aCauseException) : this(aMessage, aFileName, 0, aCau-
seException)
    {
    }
    public ParseFileException(string aMessage,
string aFileName) : this(aMessage, aFileName, null)
    {
    }
}

```

В приведенном выше примере определен класс исключения, возникающего при разборе файла. Это исключение содержит информацию о причине его возникновения, а также параметры, указывающие, в каком месте файла возникла ошибка разбора. Пример использования исключения приведен ниже:

```

static long CalculateSumOfLines(string aFileName)
{
    StreamReader inF; try
    {
        inF = File.OpenText(aFileName);
    }
    catch (IOException ioe)
    {
    }
    try
    {
        aFileName), aFileName, ioe);
        long sum = 0;
        long lineNumber = 0; while (true)
        {
            lineNumber++; string line; try
            {
                line = inF.ReadLine();

```

```

    }
    catch (IOException ioe)
    {
        throw new ParseFileException("Ошибка чтения
файла.", aFileName, lineNumber, ioe);
    }
    if (line == null)
        break; // конец файла
    try
    {
    }
    sum += Int32.Parse(line);
    catch (SystemException se)
    {
        throw new ParseFileException(String.Format("Ошибка разбора
строки '{0}'.", line), aFileName, lineNumber, se);
    }
    }
    return sum;
}
finally
{
    inF.Close();
}
}
static void Main(string[] args)
{
    long sumOfLines = CalculateSumOfLines(@"c:\test.txt");
    Console.WriteLine("Количество строк={0}", sumOfLines);
}

```

Данная программа подсчитывает количество строк в текстовом файле.

### 3.2.1 Наследование классов исключений

Можно создавать заказные исключения, выполняющие обработку ошибок в пользовательском коде. Генерирование исключений не представляет особых сложностей. Просто определите класс, наследуемый из класса `Exception`. В качестве общего правила руководствуйтесь тем, что определенные пользователем исключения наследуются из класса `ApplicationException`, так как они представляют собой иерархию зарезервированных исключений, связанных с приложениями. Наследуемые классы не нуждаются в фактической реализации в каком-либо виде, поскольку само их существование в системе типов данных позволяет воспользоваться ими в качестве исключений.

Создаваемые пользователем классы исключений автоматически получают доступные для них свойства и методы, определенные в классе `Exception`.

#### 4 Задания

Создайте программу, содержащую обработку системных и собственных типов исключений. Искусственно сгенерируйте исключения, проиллюстрируйте срабатывание обработчиков. Реализуйте несколько обработчиков по иерархии типов исключений, покажите, как влияет тип обработчика на порядок обработки исключений.

1. `ArrayTypeMismatchException`
2. `DivideByZeroException`
3. `IndexOutOfRangeException`
4. `InvalidCastException`
5. `OutOfMemoryException`
6. `OverflowException`

## 5 Контрольные вопросы

1. Что понимается под термином «событие»?
2. Являются ли события членами классов?
3. Какое ключевое слово языка C# используется для описания событий?
4. На каком механизме языка C# основана поддержка событий?
5. Приведите синтаксис описания события в общем виде. Проиллюстрируйте его фрагментом программы на языке C#.
6. Что понимается под термином «широковещательное событие»?
7. На основе какого механизма языка C# строятся широковещательные события?
8. Приведите синтаксис описания широковещательного события в общем виде. Проиллюстрируйте его фрагментом программы на языке C#.
9. Что понимается под термином «исключительная ситуация (исключение)»?
10. В чем состоит значение механизма исключений в языке C#?
11. Какие операторы языка C# используются для обработки исключений?
12. Какие операторы языка C# являются важнейшими для обработки исключений?
13. Приведите синтаксис блока try... catch в общем виде. Проиллюстрируйте его фрагментом программы на языке C#.
14. Приведите пять видов основных системных исключений.
15. Необходимо ли обеспечивать соответствие типов исключения в операторе catch типу перехватываемого исключения?
16. Что происходит в случае неудачного перехвата исключения?
17. В каком случае возможно использование оператора языка C# catch без параметров?
18. Каким образом осуществляется возврат в программу после обработки исключительной ситуации?
19. Какой оператор языка C# используется для обеспечения возврата в программу после обработки исключения?
20. Приведите синтаксис блока finally (в составе оператора try.catch) в общем виде. Проиллюстрируйте его фрагментом программы на языке C#.
21. Зависит ли вызов блока finally от наличия исключения?
22. Какие способы генерации исключений Вам известны?
23. Что является источником автоматически генерируемых (неявных) исключений?
24. Каким образом возможно осуществить явную генерацию исключений?
25. Какой оператор языка C# используется для явной генерации исключений?
26. Приведите синтаксис оператора throw в общем виде. Проиллюстрируйте его фрагментом программы на языке C#.
27. Каким образом осуществляется повторный перехват исключений в языке C#?
28. Возможно ли создавать специализированные исключения для обработки ошибок в коде пользователя?
29. Какой системный класс является базовым для создания исключений?
30. На основе какого системного класса осуществляется генерация пользовательских исключений?
31. Необходима ли явная реализация классов, наследуемых от системных исключений?
32. Каким образом обеспечивается обращение к свойствам и методам системных исключений?