**№ 7 Исключения**

**Курносенко Софья**

**Задание**

Дополнить предыдущую лабораторную работу № 6.

Создать иерархию классов исключений (собственных) – 3 типа и более. Сделать наследование пользовательских типов исключений от стандартных классов .Net (например, Exception, IndexOutofRange).

Сгенерировать и обработать, как минимум, пять различных исключительных ситуаций на основе своих и стандартных исключений. Например, не позволять при инициализации объектов передавать неверные данные, обрабатывать ошибки при работе с памятью и ошибки работы с файлами, деление на ноль, неверный индекс, нулевой указатель и т. д.

В конце поставить универсальный обработчик catch.

Обработку исключений вынести в main. При обработке выводить специфическую информацию о месте, диагностику и причине исключения. Последним должен быть блок, который отлавливает все исключения (finally).

Добавьте код в одной из функций макрос Assert. Объясните что он проверяет, как будет выполняться программа в случае не выполнения условия. Объясните назначение Assert.

Ознакомьтесь с классами Debug и Debugger:

<https://docs.microsoft.com/en-us/dotnet/api/system.diagnostics.debugger?view=netframework-4.8>

<https://docs.microsoft.com/en-us/dotnet/api/system.diagnostics.debug?view=netframework-4.8>

Пример:

int[] aa= null;

Debug.Assert(aa !=null, "Values array cannot be null");

Не забудьте подключить using System.Diagnostics;

**Обработка исключений**

**► Конструкция try..catch..finally**

Иногда при выполнении программы возникают ошибки, которые трудно предусмотреть или предвидеть, а иногда и вовсе невозможно. Например, при передаче файла по сети может неожиданно оборваться сетевое подключение, такие ситуации называются **исключениями**. Язык C# предоставляет разработчикам возможности для обработки таких ситуаций. Для этого в C# предназначена конструкция **try...catch...finally**.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | try  {    }  catch  {    }  finally  {    } |

При использовании блока **try...catch..finally** вначале выполняются все инструкции в блоке **try**. Если в этом блоке не возникло исключений, то после его выполнения начинает выполняться блок **finally**. И затем конструкция try..catch..finally завершает свою работу.

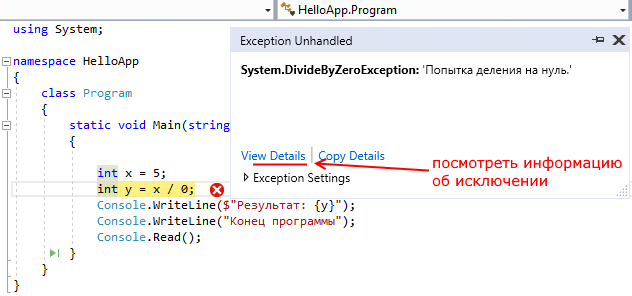
Если же в блоке try вдруг возникает исключение, то обычный порядок выполнения останавливается, и среда CLR начинает искать блок **catch**, который может обработать данное исключение. Если нужный блок catch найден, то он выполняется, и после его завершения выполняется блок finally.

Если нужный блок catch не найден, то при возникновении исключения программа аварийно завершает свое выполнение.

Рассмотрим следующий пример:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | class Program  {      static void Main(string[] args)      {          int x = 5;          int y = x / 0;          Console.WriteLine($"Результат: {y}");          Console.WriteLine("Конец программы");          Console.Read();      }  } |

В данном случае происходит деление числа на 0, что приведет к генерации исключения. И при запуске приложения в режиме отладки мы увидим в Visual Studio окошко, которое информирует об исключении:



В этом окошке мы видим, что возникло исключение, которое представляет тип **System.DivideByZeroException**, то есть попытка деления на ноль. С помощью пункта **View Details** можно посмотреть более детальную информацию об исключении.

И в этом случае единственное, что нам остается, это завершить выполнение программы.

Чтобы избежать подобного аварийного завершения программы, следует использовать для обработки исключений конструкцию **try...catch...finally**. Так, перепишем пример следующим образом:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22 | class Program  {      static void Main(string[] args)      {          try          {              int x = 5;              int y = x / 0;              Console.WriteLine($"Результат: {y}");          }          catch          {              Console.WriteLine("Возникло исключение!");          }          finally          {              Console.WriteLine("Блок finally");          }          Console.WriteLine("Конец программы");          Console.Read();      }  } |

В данном случае у нас опять же возникнет исключение в блоке try, так как мы пытаемся разделить на ноль. И дойдя до строки

|  |  |
| --- | --- |
| 1 | int y = x / 0; |

выполнение программы остановится. CLR найдет блок **catch** и передаст управление этому блоку.

После блока catch будет выполняться блок finally.

Возникло исключение!

Блок finally

Конец программы

Таким образом, программа по-прежнему не будет выполнять деление на ноль и, соответственно, не будет выводить результат этого деления, но теперь она не будет аварийно завершаться, а исключение будет обрабатываться в блоке catch.

Следует отметить, что в этой конструкции обязателен блок **try**. При наличии блока catch мы можем опустить блок finally:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | try  {      int x = 5;      int y = x / 0;      Console.WriteLine($"Результат: {y}");  }  catch  {      Console.WriteLine("Возникло исключение!");  } |

И, наоборот, при наличии блока finally мы можем опустить блок catch и не обрабатывать исключение:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | try  {      int x = 5;      int y = x / 0;      Console.WriteLine($"Результат: {y}");  }  finally  {      Console.WriteLine("Блок finally");  } |

Однако, хотя с точки зрения синтаксиса C# такая конструкция вполне корректна, тем не менее, поскольку CLR не сможет найти нужный блок catch, то исключение не будет обработано, и программа аварийно завершится.

### Обработка исключений и условные конструкции

Ряд исключительных ситуаций может быть предвиден разработчиком. Например, пусть программа предусматривает ввод числа и вывод его квадрата:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | static void Main(string[] args)  {      Console.WriteLine("Введите число");      int x = Int32.Parse(Console.ReadLine());        x \*= x;      Console.WriteLine("Квадрат числа: " + x);      Console.Read();  } |

Если пользователь введет не число, а строку, какие-то другие символы, то программа выпадет в ошибку. С одной стороны, здесь как раз та ситуация, когда можно применить блок try..catch, чтобы обработать возможную ошибку. Однако, гораздо более, оптимально было бы проверить допустимость преобразования:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | static void Main(string[] args)  {      Console.WriteLine("Введите число");      int x;      string input = Console.ReadLine();      if (Int32.TryParse(input, out x))      {          x \*= x;          Console.WriteLine("Квадрат числа: " + x);      }      else      {          Console.WriteLine("Некорректный ввод");      }      Console.Read();  } |

Метод Int32.TryParse() возвращает true, если преобразование можно осуществить, и false - если нельзя. При допустимости преобразования переменная x будет содержать введенное число. Так, не используя try...catch можно обработать возможную исключительную ситуацию.

С точки зрения производительности использование блоков try..catch более накладно, чем применение условных конструкций. Поэтому по возможности вместо try..catch лучше использовать условные конструкции на проверку исключительных ситуаций.

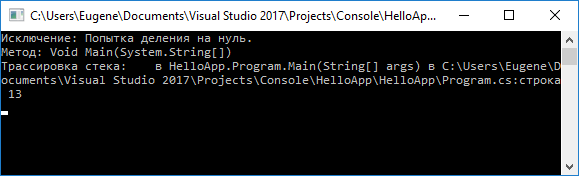
## ► Типы исключений. Класс Exception

Базовым для всех типов исключений является тип **Exception**. Этот тип определяет ряд свойств, с помощью которых можно получить информацию об исключении.

* **InnerException**: хранит информацию об исключении, которое послужило причиной текущего исключения
* **Message**: хранит сообщение об исключении, текст ошибки
* **Source**: хранит имя объекта или сборки, которое вызвало исключение
* **StackTrace**: возвращает строковое представление стека вызовов, которые привели к возникновению исключения
* **TargetSite**: возвращает метод, в котором и было вызвано исключение

Например, обработаем исключения типа Exception:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | static void Main(string[] args)  {      try      {          int x = 5;          int y = x / 0;          Console.WriteLine($"Результат: {y}");      }      catch (Exception ex)      {          Console.WriteLine($"Исключение: {ex.Message}");          Console.WriteLine($"Метод: {ex.TargetSite}");          Console.WriteLine($"Трассировка стека: {ex.StackTrace}");      }        Console.Read();  } |



Однако так как тип Exception является базовым типом для всех исключений, то выражение catch (Exception ex) будет обрабатывать все исключения, которые могут возникнуть.

Но также есть более специализированные типы исключений, которые предназначены для обработки каких-то определенных видов исключений. Их довольно много, я приведу лишь некоторые:

* **DivideByZeroException**: представляет исключение, которое генерируется при делении на ноль
* **ArgumentOutOfRangeException**: генерируется, если значение аргумента находится вне диапазона допустимых значений
* **ArgumentException**: генерируется, если в метод для параметра передается некорректное значение
* **IndexOutOfRangeException**: генерируется, если индекс элемента массива или коллекции находится вне диапазона допустимых значений
* **InvalidCastException**: генерируется при попытке произвести недопустимые преобразования типов
* **NullReferenceException**: генерируется при попытке обращения к объекту, который равен null (то есть, по сути, не определен)

И при необходимости мы можем разграничить обработку различных типов исключений, включив дополнительные блоки catch:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22 | static void Main(string[] args)  {      try      {          int[] numbers = new int[4];          numbers[7] = 9;     // IndexOutOfRangeException            int x = 5;          int y = x / 0;  // DivideByZeroException          Console.WriteLine($"Результат: {y}");      }      catch (DivideByZeroException)      {          Console.WriteLine("Возникло исключение DivideByZeroException");      }      catch (IndexOutOfRangeException ex)      {          Console.WriteLine(ex.Message);      }        Console.Read();  } |

В данном случае блоки catch обрабатывают исключения типов IndexOutOfRangeException, DivideByZeroException и Exception. Когда в блоке try возникнет исключение, то CLR будет искать нужный блок catch для обработки исключения. Так, в данном случае, на строке

|  |  |
| --- | --- |
| 1 | numbers[7] = 9; |

происходит обращение к 7-му элементу массива. Однако поскольку в массиве только 4 элемента, то мы получим исключение типа IndexOutOfRangeException. CLR найдет блок catch, который обрабатывает данное исключение, и передаст ему управление.

Следует отметить, что в данном случае в блоке try есть ситуация для генерации второго исключения - деление на ноль. Однако поскольку после генерации IndexOutOfRangeException управление переходит в соответствующий блок catch, то деление на ноль int y = x / 0 в принципе не будет выполняться, поэтому исключение типа DivideByZeroException никогда не будет сгенерировано.

Однако рассмотрим другую ситуацию:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19 | static void Main(string[] args)  {      try      {          object obj = "you";          int num = (int)obj;     // InvalidCastException          Console.WriteLine($"Результат: {num}");      }      catch (DivideByZeroException)      {          Console.WriteLine("Возникло исключение DivideByZeroException");      }      catch (IndexOutOfRangeException)      {          Console.WriteLine("Возникло исключение IndexOutOfRangeException");      }        Console.Read();  } |

В данном случае в блоке try генерируется исключение типа InvalidCastException, однако соответствующего блока catch для обработки данного исключения нет. Поэтому программа аварийно завершит свое выполнение.

Мы также можем определить для InvalidCastException свой блок catch, однако суть в том, что теоретически в коде могут быть сгенерированы сами различные типы исключений. А определять для всех типов исключений блоки catch, если обработка исключений однотипна, не имеет смысла. И в этом случае мы можем определить блок catch для базового типа Exception:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22 | static void Main(string[] args)  {      try      {          object obj = "you";          int num = (int)obj;     // InvalidCastException          Console.WriteLine($"Результат: {num}");      }      catch (DivideByZeroException)      {          Console.WriteLine("Возникло исключение DivideByZeroException");      }      catch (IndexOutOfRangeException)      {          Console.WriteLine("Возникло исключение IndexOutOfRangeException");      }      catch (Exception ex)      {          Console.WriteLine($"Исключение: {ex.Message}");      }      Console.Read();  } |

И в данном случае блок catch (Exception ex){} будет обрабатывать все исключения кроме DivideByZeroException и IndexOutOfRangeException. При этом блоки catch для более общих, более базовых исключений следует помещать в конце - после блоков catch для более конкретный, специализированных типов. Так как CLR выбирает для обработки исключения первый блок catch, который соответствует типу сгенерированного исключения. Поэтому в данном случае сначала обрабатывается исключение DivideByZeroException и IndexOutOfRangeException, и только потом Exception (так как DivideByZeroException и IndexOutOfRangeException наследуется от класса Exception).

## ► Создание классов исключений

Если нас не устраивают встроенные типы исключений, то мы можем создать свои типы. Базовым классом для всех исключений является класс Exception, соответственно для создания своих типов мы можем унаследовать данный класс.

Допустим, у нас в программе будет ограничение по возрасту:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35 | class Program  {      static void Main(string[] args)      {          try          {              Person p = new Person { Name = "Tom", Age = 17 };          }          catch (Exception ex)          {              Console.WriteLine($"Ошибка: {ex.Message}");          }          Console.Read();      }  }  class Person  {      private int age;      public string Name { get; set; }      public int Age      {          get { return age; }          set          {              if (value < 18)              {                  throw new Exception("Лицам до 18 регистрация запрещена");              }              else              {                  age = value;              }          }      }  } |

В классе Person при установке возраста происходит проверка, и если возраст меньше 18, то выбрасывается исключение. Класс Exception принимает в конструкторе в качестве параметра строку, которая затем передается в его свойство Message.

Но иногда удобнее использовать свои классы исключений. Например, в какой-то ситуации мы хотим обработать определенным образом только те исключения, которые относятся к классу Person. Для этих целей мы можем сделать специальный класс PersonException:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | class PersonException : Exception  {      public PersonException(string message)          : base(message)      { }  } |

По сути, класс, кроме пустого конструктора, ничего не имеет, и то в конструкторе мы просто обращаемся к конструктору базового класса Exception, передавая в него строку message. Но теперь мы можем изменить класс Person, чтобы он выбрасывал исключение именно этого типа и соответственно в основной программе обрабатывать это исключение:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30 | class Program  {      static void Main(string[] args)      {          try          {              Person p = new Person { Name = "Tom", Age = 17 };          }          catch (PersonException ex)          {              Console.WriteLine("Ошибка: " + ex.Message);          }          Console.Read();      }  }  class Person  {      private int age;      public int Age      {          get { return age; }          set          {              if (value < 18)                  throw new PersonException("Лицам до 18 регистрация запрещена");              else                  age = value;          }      }  } |

Однако необязательно наследовать свой класс исключений именно от типа Exception, можно взять какой-нибудь другой производный тип. Например, в данном случае мы можем взять тип ArgumentException, который представляет исключение, генерируемое в результате передачи аргументу метода некорректного значения:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | class PersonException : ArgumentException  {      public PersonException(string message)          : base(message)      { }  } |

Каждый тип исключений может определять какие-то свои свойства. Например, в данном случае мы можем определить в классе свойство для хранения устанавливаемого значения:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | class PersonException : ArgumentException  {      public int Value { get;}      public PersonException(string message, int val)          : base(message)      {          Value = val;      }  } |

В конструкторе класса мы устанавливаем это свойство и при обработке исключения мы его можем получить:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32 | class Person  {      public string Name { get; set; }      private int age;      public int Age      {          get { return age; }          set          {              if (value < 18)                  throw new PersonException("Лицам до 18 регистрация запрещена", value);              else                  age = value;          }      }  }  class Program  {      static void Main(string[] args)      {          try          {              Person p = new Person { Name = "Tom", Age = 13 };          }          catch (PersonException ex)          {              Console.WriteLine($"Ошибка: {ex.Message}");              Console.WriteLine($"Некорректное значение: {ex.Value}");          }          Console.Read();      }  } |

# Assert. Что это?

# [Assert](http://en.wikipedia.org/wiki/Assertion_(computing)) — это специальная конструкция, позволяющая проверять предположения о значениях произвольных данных в произвольном месте программы. Эта конструкция может автоматически сигнализировать при обнаружении некорректных данных, что обычно приводит к аварийному завершению программы с указанием места обнаружения некорректных данных. Странная, на первый взгляд, конструкция — может завалить программу в самый неподходящий момент. Какой же в ней смысл? Давайте подумаем, что произойдет, если во время исполнения программы в какой-то момент времени некоторые данные программы стали некорректными и мы не «завалили» сразу же программу, а продолжили ее работу, как ни в чем не бывало. Программа может еще долго работать после этого без каких-либо видимых ошибок. А может в любой момент времени в будущем «завалиться» сама по известной только ей причине. Или накачать вам полный винчестер контента с гей-порносайтов. Это называется [неопределенное поведение (undefined behavior)](http://en.wikipedia.org/wiki/Undefined_behavior) и, вопреки расхожему мнению, оно свойственно не только языкам программирования с произвольным доступом к памяти (aka C, C++). Т.к. assert завершает программу сразу же после обнаружения некорректных данных, он позволяет быстро локализировать и исправить баги в программе, которые привели к некорректным данным. Это его основное назначение. Assert'ы доступны во многих языках программирования, включая java, c#, c и python.

# *Assert – метод класса Debug. Debug определен в пространстве имен* [*System.Diagnostics*](https://docs.microsoft.com/en-us/dotnet/api/system.diagnostics?view=net-5.0)*.*

# Метод Assert принимает логическое значение и выдает исключение, если значение ложно. Второй параметр выдает сообщение об ошибке, которое должно отображаться в исключении.

# Например, следующий метод Average использует Debug.Assert для проверки входного параметра метода.

// Return the average of the numbers.

private float Average(float[] values)

{

Debug.Assert(values != null,

"Values array cannot be null");

Debug.Assert(values.Length > 0,

"Values array cannot be empty");

Debug.Assert(values.Length < 100,

"Values array should not contain more than 100 items");

// If there are no values, return NaN.

if (values == null || values.Length < 1) return float.NaN;

// Calculate the average.

return values.Average();

# }

# Перед тем, как начать свои вычисления, метод проверяет, что параметр values не равен нулю, содержит хотя бы один элемент и содержит не более 99 элементов. Первые два условия необходимы для расчета значимого среднего.

# Последнее условие (не более 99 элементов) - это «проверка работоспособности», чтобы увидеть, имеет ли смысл вызов метода. Если во время тестирования приложения вы обнаружите, что программа вызывает метод с более чем 99 элементами, вы можете увидеть, имеет ли это смысл. Если это так, вы можете увеличить верхний предел до 199 или до другого подходящего значения. Это позволяет выявлять неожиданные ситуации, когда вызывающий код может содержать ошибку.

**Преимущества C # Assert**

С каждым конкретным методом или функцией на языке программирования у нас есть несколько преимуществ, как и следующие преимущества использования метода assert в C #:

◦ Одно из самых больших преимуществ - возможность обнаруживать ошибки в программе, которые могли не заметить.

◦ Помимо поиска ошибок, реализация метода assert может быть полезна для более быстрого обнаружения этих ошибок, что ускоряет решение проблемы.

◦ Всегда верно: с помощью метода assert у вас есть утверждение, объясняющее влияние конкретного кода, который гарантированно истинен.

◦ Метод assert гарантирует, что у программиста будет достаточно времени для обнаружения, понимания и устранения ошибки.

**Заключение**

Метод Assert просто используется для выявления ошибок во время выполнения. Метод Assert принимает два аргумента: первый - это логическое выражение, в котором проверяется условие, а второй - сообщение, отображаемое в зависимости от результата выполнения условия. Мы продемонстрировали два примера, чтобы понять работу метода assert.

# Класс Debug

Namespace: [System.Diagnostics](https://docs.microsoft.com/en-us/dotnet/api/system.diagnostics?view=net-5.0)

Предоставляет набор методов и свойств, которые помогают отлаживать ваш код.

# С двумя методами данного класса мы уже знакомы:

# *Assert(Boolean)*

# Checks for a condition; if the condition is false, displays a message box that shows the call stack.

# *Assert(Boolean, String)*

# Checks for a condition; if the condition is false, outputs a specified message and displays a message box that shows the call stack.

# Класс Debugger

Пространство имен: [System.Diagnostics](https://docs.microsoft.com/ru-ru/dotnet/api/system.diagnostics?view=net-5.0)

# Разрешает взаимодействие с отладчиком. Этот класс не наследуется.

**Вопросы**

**1. Расскажите, как генерируется исключение.**

## Генерация исключения и оператор throw

Обычно система сама генерирует исключения при определенных ситуациях, например, при делении числа на ноль. Но язык C# также позволяет генерировать исключения вручную с помощью оператора **throw**. То есть с помощью этого оператора мы сами можем создать исключение и вызвать его в процессе выполнения.

Например, в нашей программе происходит ввод строки, и мы хотим, чтобы, если длина строки будет больше 6 символов, возникало исключение:

static void Main(string[] args)

{

try

{

Console.Write("Введите строку: ");

string message = Console.ReadLine();

if (message.Length > 6)

{

throw new Exception("Длина строки больше 6 символов");

}

}

catch (Exception e)

{

Console.WriteLine($"Ошибка: {e.Message}");

}

Console.Read();

}

После оператора **throw** указывается объект исключения, через конструктор которого мы можем передать сообщение об ошибке. Естественно вместо типа Exception мы можем использовать объект любого другого типа исключений.

Затем в блоке catch сгенерированное нами исключение будет обработано.

Подобным образом мы можем генерировать исключения в любом месте программы. Но существует также и другая форма использования оператора throw, когда после данного оператора не указывается объект исключения. В подобном виде оператор throw может использоваться только в блоке catch:

try

{

try

{

Console.Write("Введите строку: ");

string message = Console.ReadLine();

if (message.Length > 6)

{

throw new Exception("Длина строки больше 6 символов");

}

}

catch

{

Console.WriteLine("Возникло исключение");

throw;

}

}

catch (Exception ex)

{

Console.WriteLine(ex.Message);

}

В данном случае при вводе строки с длиной больше 6 символов возникнет исключение, которое будет обработано внутренним блоком catch. Однако поскольку в этом блоке используется оператор throw, то исключение будет передано дальше внешнему блоку catch.

**2. Расскажите методику обработки исключений.**

(где-то в заданиях--)

**3. Какое ключевое слово служит для обозначения блока кода, в котором можно генерировать исключение?**

Ключевое слово throw служит для генерации исключений, **try - для выделения фрагмента кода, в котором генерируется исключение**, а catch - для обработки исключения из предыдущего фрагмента try. При этом блок catch следует сразу после try, а самих блоков catch может быть сколько угодно.

**4. Какие ключевые слова используются для обработки и генерации исключений? Расскажите о механизме обработке исключения?**

(1 вопрос – выше, 2 вопрос – где-то в заданиях--)

**5. Что будет, если в программе нет предложения catch, способного обработать исключение?**

Если нужный блок catch найден, то он выполняется, и после его завершения выполняется блок finally.

**Если нужный блок catch не найден**, то при возникновении исключения программа аварийно завершает свое выполнение.

**6. Что такое фильтры исключения? Приведите пример**

### Фильтры исключений

Фильтры исключений позволяют обрабатывать исключения в зависимости от определенных условий. Для их применения после выражения catch идет выражение **when**, после которого в скобках указывается условие:

catch when (условие)

{

}

В этом случае обработка исключения в блоке catch производится только в том случае, если условие в выражении when истинно. Например:

int x = 1;

int y = 0;

try

{

int result = x / y;

}

catch (DivideByZeroException) when (y == 0 && x == 0)

{

Console.WriteLine("y не должен быть равен 0");

}

catch (DivideByZeroException ex)

{

Console.WriteLine(ex.Message);

}

В данном случае будет выброшено исключение, так как y=0. Здесь два блока catch, и оба они обрабатывают исключения типа DivideByZeroException, то есть по сути все исключения, генерируемые при делении на ноль. Но поскольку для первого блока указано условие **y == 0 && x == 0**, то оно не будет обрабатывать исключение - условие, указанное после оператора **when** возвращает false. Поэтому CLR будет дальше искать соответствующие блоки catch далее и для обработки исключения выберет второй блок catch. В итоге если мы уберем второй блок catch, то исключение вообще не будет обрабатываться.

**7. Могут ли исключения быть вложенными?**

Операторы try-catch могут быть вложенными. Внутри блока try либо catch может быть еще один try-catch.

**8. Какой синтаксис нужно использовать в C# для отлова любого возможного исключения?**

Блок catch, отлавливающий исключения типа System.Exception. Также можно просто использовать catch{}, не указывая тип исключения.

**9. Чем следует руководствоваться при размещении обработчиков исключения?**

(благоразумием?)

**10. Что будет выведено на консоль в результате выполнения фрагмента листинга?**

using System;

using System.Collections.Generic;

using System.Text;

namespace pract

{

class Program

{

static void Main(string[] args)

{

string[] str = new string[5];

try

{

str[4] = "anything";

Console.WriteLine("It's OK");

}

catch (IndexOutOfRangeException e)

{

Console.WriteLine("IndexOutOfRangeException");

}

catch (Exception e)

{

Console.WriteLine("Exception");

}

}

}

}



**11. Как повторно сгенерировать то же самое исключение в блоке обработчике catch?**

Если возникает необходимость снова сгенерировать исключения из блока, который обрабатывает исключения, можно сделать это путем вызова throw без указания исключения. В результате теку­щее исключение будет передано во внешнюю последовательность try/catch обработки исключений. Причиной для этого может послужить желание обрабатывать исключения несколькими обработ­чиками. Например, один обработчик может заниматься одним аспектом исключения, а второй обработчик — другим. Исключение может быть снова сгенерировано или изнутри блока catch, или из функции, вызванной в этом блоке. Когда повторно генерируется исключение, оно не бу­дет перехвачено той же самой инструкцией catch. Оно будет распространяться до следующей внешней инструкции catch.

**12. Какие методы содержаться в классе Exception? Где и как их можно использовать?**

|  |  |
| --- | --- |
| **МЕТОДЫ** | |
| [Equals(Object)](https://docs.microsoft.com/ru-ru/dotnet/api/system.object.equals?view=net-5.0#System_Object_Equals_System_Object_) | Определяет, равен ли указанный объект текущему объекту.  (Унаследовано от [Object](https://docs.microsoft.com/ru-ru/dotnet/api/system.object?view=net-5.0)) |
| [GetBaseException()](https://docs.microsoft.com/ru-ru/dotnet/api/system.exception.getbaseexception?view=net-5.0#System_Exception_GetBaseException) | При переопределении в производном классе возвращает исключение [Exception](https://docs.microsoft.com/ru-ru/dotnet/api/system.exception?view=net-5.0), которое является первопричиной одного или нескольких последующих исключений. |
| [GetHashCode()](https://docs.microsoft.com/ru-ru/dotnet/api/system.object.gethashcode?view=net-5.0#System_Object_GetHashCode) | Служит хэш-функцией по умолчанию.  (Унаследовано от [Object](https://docs.microsoft.com/ru-ru/dotnet/api/system.object?view=net-5.0)) |
| [GetObjectData(SerializationInfo, StreamingContext)](https://docs.microsoft.com/ru-ru/dotnet/api/system.exception.getobjectdata?view=net-5.0#System_Exception_GetObjectData_System_Runtime_Serialization_SerializationInfo_System_Runtime_Serialization_StreamingContext_) | При переопределении в производном классе задает объект [SerializationInfo](https://docs.microsoft.com/ru-ru/dotnet/api/system.runtime.serialization.serializationinfo?view=net-5.0) со сведениями об исключении. |
| [GetType()](https://docs.microsoft.com/ru-ru/dotnet/api/system.exception.gettype?view=net-5.0#System_Exception_GetType) | Возвращает тип среды выполнения текущего экземпляра. |
| [MemberwiseClone()](https://docs.microsoft.com/ru-ru/dotnet/api/system.object.memberwiseclone?view=net-5.0#System_Object_MemberwiseClone) | Создает неполную копию текущего объекта [Object](https://docs.microsoft.com/ru-ru/dotnet/api/system.object?view=net-5.0).  (Унаследовано от [Object](https://docs.microsoft.com/ru-ru/dotnet/api/system.object?view=net-5.0)) |
| [ToString()](https://docs.microsoft.com/ru-ru/dotnet/api/system.exception.tostring?view=net-5.0#System_Exception_ToString) | Создает и возвращает строковое представление текущего исключения. |

# КОД ВСЕЙ ПРОГРАММЫ