МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ
«ХАРКІВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ»

Методические указания к лабораторной работе «Обработка исключительных ситуаций в языке C#»

по дисциплине «Технологии программирования»

для студентов специальностей

122 – Компьютерные науки и информационные технологии,

124 – Системный анализ

Утверждено
редакционно-издательским
советом университета,
протокол №2 от 23.06.16 г

Харьков
НТУ ”ХПИ”
2017

Методические указания к лабораторной работе «Обработка исключительных ситуаций в языке C#» по дисциплинам «Технологии программирования» для студентов специальностей 122 – Компьютерные науки и информационные технологии, 124 – Системный анализ /сост. Ю. Н. Кожин, О. Н. Малых, В. Ф. Прокопенков. – Харьков: НТУ “ХПИ”, 2017.– 32 с. – на рус.яз.

Составители: Ю. Н. Кожин,
О. Н. Малых,
В. Ф. Прокопенков,

Рецензент О.В. Горелый

Кафедра системного анализа и информационно-аналитических технологий

ВВЕДЕНИЕ

Язык C# и технология программирования .NET Framework пришли на смену языку C/C++ и обычному программированию для Windows. Возможности, предлагаемые платформой .NET, позволяют радикально облегчить жизнь программистов и разрабатывать программные приложения разного назначения.

Любая программа представляет собой закодированный алгоритм обработки данных, подчиненный цели решения определенной задачи.

Программа разработана правильно, если она решает возложенную на неё задачу и устойчиво работает в разных, возможно, непредвиденных ситуациях. Для устойчивой работы программа должна иметь возможность диагностировать возможную аварийную ситуацию, адекватно реагировать на неё, по возможности устранять её последствия с целью восстановления своей работоспособности.

Платформа .NET Framework и язык C# для решения указанных задач предлагают специальный механизм обработки исключений (exception handling).

Предлагаемые методические указания помогут студентам познакомиться с механизмом обработки исключений и использовать его в дальнейшем для разработки своих устойчивых к сбоям программ.

Методические указания содержат необходимые теоретические сведения, а также рекомендации для выполнения лабораторных работ.

1. ИСКЛЮЧИТЕЛЬНЫЕ СИТУАЦИИ

Ошибки разрабатываемого программного обеспечения делят на два типа: ошибки компиляции и ошибки времени выполнения.

Ошибки первого типа связаны с этапом кодирования алгоритма решения задачи и выявляются на стадии получения двоичного кода приложения. После устранения таких ошибок приложение может быть запущено на исполнение. Такие ошибки ещё называются синтаксическими, поскольку они выявляются на этапе синтаксического анализа исходного кода программы, представленного на выбранном языке программирования.

Ошибки второго типа невозможно обнаружить на этапе разработки программы, так как они проявляются только во время исполнения. Такие ошибки могут быть следствием разных причин, например, логическими ошибками алгоритмов обработки данных или аварийными ситуациями вычислительной системы (возможно как следствие логических ошибок), когда дальнейшее исполнение кода невозможно. В любом случае такие ошибки можно отнести к непредвиденным ошибкам.

Любая программа разрабатывается при определенных допущениях, о которых разработчик часто даже не задумывается. Вызывая метод, мы предполагаем, что в стеке хватает свободного места. Выполняя операцию деления, мы не допускаем, что делитель может быть величиной, близкой к нулю, и привести к результату, переполняющему разрядную сетку. Читая из файла, мы должны считать 200 записей, а их всего 90 и т.д. Конечно, при разработке программы всё предвидеть невозможно. Но программа должна быть устойчивой к возможным ошибкам и способной сохранять свою работоспособность.

Тестирование и отладка приложения позволяют устранить основные логические ошибки, но не исключают возможности возникновения ошибок во время его эксплуатации.

Ошибки времени исполнения в современной терминологии программирования называют исключительными ситуациями (или просто исключения). Они возникают, как правило, тогда, когда нарушаются принятые допущения о работе программы. Такое название связано с использованием специального механизма выявления и обработки ошибок.

2. МЕХАНИЗМ ОБРАБОТКИ ИСКЛЮЧЕНИЙ

Рассмотрим, как устроен механизм обработки исключений.

2.1. Пример исключения

Обнаружив ошибку времени выполнения, приложение генерирует исключение (exception), т.е. создается объект соответствующего класса исключения, который передается от метода, обнаружившего ошибку (сгенерировавшего исключение), к методу вверх по стеку вызовов (вызвавший этот метод) для обработки аварийной ситуации.

Выбранный класс объекта исключения определяется методом, который его генерируют, и зависит от типа обнаруженной ошибки. Все исключения представляют собой объекты специализированных классов исключений, потомков класса System.Exception.

Первым сгенерированное исключение получает метод, из которого был вызван метод, сгенерировавший исключение. Если этот метод предусматривает обработку этого исключения, то оно обрабатывается; в противном случае оно передается дальше следующему методу по стеку вызовов и т.д.

Если обработка исключения в приложении не предусмотрена, то вызывается обработчик исключений, заданный по умолчанию. Этот обработчик отображает сообщение с указанием типа исключения и возможные дополнительные сведения, предоставленные объектом исключения, и закрывает приложение, не позволяя сохранить результаты работы или исправить ошибку.

Помните, необработанное в программе исключение обработкой по умолчанию приводит к завершению работы приложения.

Рассмотрим приложение, в которое намеренно заложена ошибка: массив *array* имеет размер 10 элементов, а мы пытаемся присвоить значения ста элементам массива.

```
using System;
using System.Text;
```

```
namespace exceptions_example
{
   class Program
   {
      static void Main(string[] args)
      {
         int[] array = new int[10];
         int i = 0;

         for (i = 0; i < 100; i++)
            array[i] = i;

         Console.WriteLine("End of program!");
      }
   }
}
```

Во время исполнения этой программы будет обнаружена ошибка и сгенерировано исключение. Поскольку в данном примере не предусмотрена обработка исключений, то сработает обработчик исключений по умолчанию, результат которого приведет к появлению окна, показанного на рис.2.1. Будет подсвечен оператор, вызвавший ошибку, а в специальном окне будет дано описание исключительной ситуации.

В данном случае из сообщения в окне понятно, что индекс вышел за пределы границ массива, а из заголовка окна следует, что сгенерированное по ошибке исключение IndexOutOfRangeException не было обработано.

Кроме этого, через это окно можно выполнить дополнительные действия, связанные с изучением исключительной ситуации, например, окно View Detail позволяет ознакомиться с объектом сгенерированного исключения детально.

В любом случае после срабатывания обработчика по умолчанию работа приложения дальше продолжена быть не может.

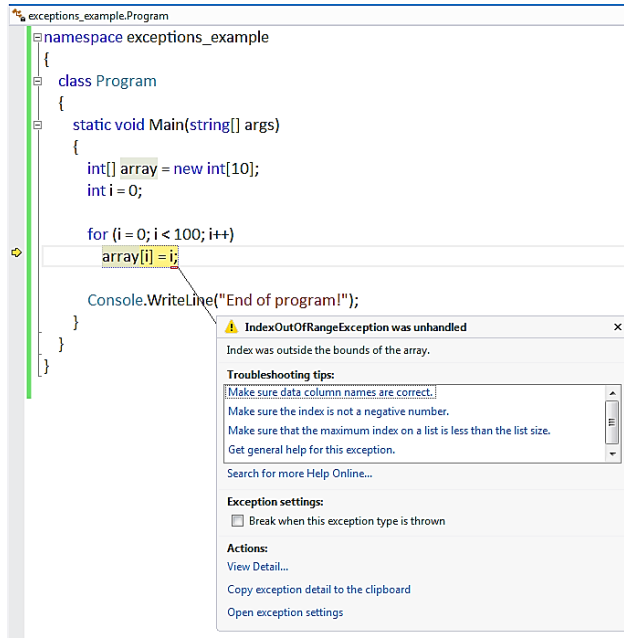

Рис.2.1. Результат обработчика исключения по умолчанию

2.2. Структурная обработка исключений

Механизм генерации и обработки исключений, предоставляемый языком C# и платформой .NET Framework, называется структурной обработкой исключений.

Его назначение – устранить последствия непредвиденной ошибки, возможно, продолжить работу приложения или, как минимум, безаварийно её завершить (например, сохранить данные перед завершением).

У каждого метода должен быть собственный обработчик исключений, ориентированный на особенности данного метода и приспособленный для обработки исключений, которые он обычно генерирует. Обработку исключений обязательно необходимо реализовать в методах, которые часто генерируют исключения, например, использующих файловый доступ.

Для организации обработчиков исключений используют оператор, структура которого предполагает наличие частей *try, catch и finally*.

```
try
{
  // Здесь находится обычный код приложения,
  // который может вызывать ошибки
}
catch( Exception e)
{
  // в этом блоке размещается код обработки исключения (ошибки)
  // параметр e – это объект класса исключения, передаваемый из
  // блока try, который несет в себе информацию об ошибке
}
finally
{
    // действия, выполняемые
    // независимо от наличия или отсутствия ошибок
}
```

Создание обработчика начинается с заключения некоторого кода в блок *try*. Если при исполнении любого из операторов блока *try* возникнет исключение, то оно будет передано в *catch* блок.

За блоком *try* может следовать один или несколько блоков catch, каждый из которых отличается фильтром исключения.

Фильтр исключения – это указание типа и\или объекта исключения, который передается в блок *catch*, и на который блок *catch* будет реагировать. Объект исключения, передаваемый в блок catch, несет информации об ошибке и может использоваться для генерации нового исключения.

Если объект исключения не используется, в фильтре указывается только тип исключения.

Все типы исключений являются производными от класса System.Exception.

Если фильтр за словом *catch* опущен или образован типом System.Exception, то такой блок называется универсальным (он реагирует на все типы исключений).

Если фильтр образован типом – потомком System.Exception, то блок catch является специализированным для этого типа исключений.

Блоки catch просматриваются сверху вниз, поэтому если мы хотим использовать одновременно несколько блоков *catch* после блока *try*, то сначала мы должны указывать блоки с типами, которые являются более далекими потомками класса System.Exception, иначе они не сработают.

Пример использования исключений:

```
using System;
using System.Text;

namespace try_catch
{
   class Program
   {
      static void Main(string[] args)
      {
         int[] array = new int[10];
         int i = 0;

         try
         { // шаг с индексом  i=10 вызовет исключение IndexOutOffRange
            for (i = 0; i < 100; i++)
               array[i] = i;
         }

         catch (IndexOutOfRangeException e) // специализированный блок
         {
            Console.WriteLine("Исключение: {0}\nЭлемент с индексом
                                          i={1} не существует!", e.Message,i);
         }

         catch (Exception e)// универсальный  блок
         {
```

```
            Console.WriteLine(e);
        }
        finally
        {   // этот блок выполняется независимо от наличия исключения
            Console.WriteLine("Block finally");
        }

        Console.WriteLine("Last program operator");
    }
  }
}
```

При выполнении приложения будет выдан результат, представленный на рис.2.2.

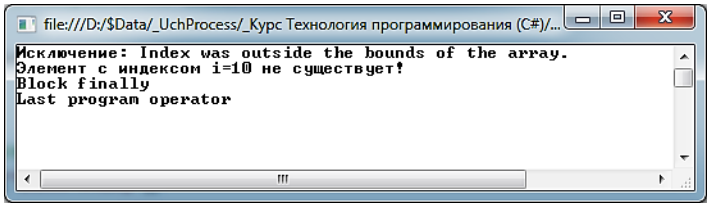

Рис.2.2. Результат обработки исключения

Из рис.2.2 видно, что работа кода, указанного в части *try*, завершится в цикле при попытке обработать несуществующий элемент массива с индексом 10, это мы определили по состоянию переменной *i*. Информацию об исключении мы получаем из свойства *Message* объекта исключения *e*.

Свойство *Message* содержит описание ошибки в доступной для человека форме и другие важные сведения об ошибке.

После выполнения блока *catch* был выполнен блок *finally*, а также оператор – последний оператор программы. В нашем примере блок *catch* реально ничего полезного не делает. Обычно этот блок должен выполнять действия по восстановлению нормальной работы программы после ошибки. В завершение этих действий можно выполнить следующее на выбор:

– позволить программному потоку покинуть блок *catch* (как в нашем примере, просто не указывать никаких действий);

– сгенерировать то же исключение повторно и позволить методу выше по стеку вызовов среагировать на исключение;

– сгенерировать исключение другого типа, чтобы передать методу выше по стеку вызовов больше сведений об ошибке.

Таким образом, применение обработчика исключений позволяет выполнять обработку возникающих ощибок и при возможности продолжить вывполнение программы.

Заметим, если бы ошибки в блоке *try* не было, то блок *catch* не выполнялся бы, а блок *finally* был бы выполен.

2.3. Генерация исключений

При написании своих приложений мы также можем генерировать ислючения. Для генерации исключения служит оператор *throw*. Возможные варианты применения оператора рассмотрим на примере программы:

```
using System;
using System.Text;

namespace throw_example
{
   class ExeptionTester
   {
      public void GenerateException( )
      {
         int[] array = new int[10];
         int i = 0;

         try
         {
            for (i = 0; i < 100; i++)
               array[i] = i;
         }
```

```
        catch (IndexOutOfRangeException iore)
        {
           //throw;

           //throw new Exception( );
           //throw new Exception( "Текст сообщения");
          throw new Exception("Текст сообщения с привязкой
                                    исходного исключения ", iore);
        }
        catch (Exception e)
        {
           Console.WriteLine("ExceptionTester: Исключение: {0}",
                                                    e.Message);
           throw new Exception("My throw2. ", e);
        }
        finally
        {  // этот блок выполняется независимо от наличия исключения
           Console.WriteLine("ExceptionTester: Block finally");
        }
     }
  }

  class Program
  {
     static void Main(string[] args)
     {
        ExeptionTester test_ob = new ExeptionTester( );

        try
        {
           test_ob.GenerateException( );
        }
        catch (Exception e)
        {
```

```
            Console.WriteLine("Main: All exceptions:");

            while (e != null)
            {
                Console.WriteLine("{0}", e.Message);
                e = e.InnerException;
            }
        }
        finally
        {   // этот блок выполняется независимо от наличия исключения
            Console.WriteLine("Main: Block finally");
        }

        Console.WriteLine("Main: Last operator");
    }
  }
}
```

2.3.1. Применение оператора throw без параметров

При использовании блока *catch* без получения объекта исключения оператор *throw* можно применять без параметров, например:

```
catch
{
    // код...
     throw;
}
```

Такое применение *throw* приводит к тому, что перехваченное исключение передается (транслируется) на уровень выше по стеку вызовов (рис.2.3).

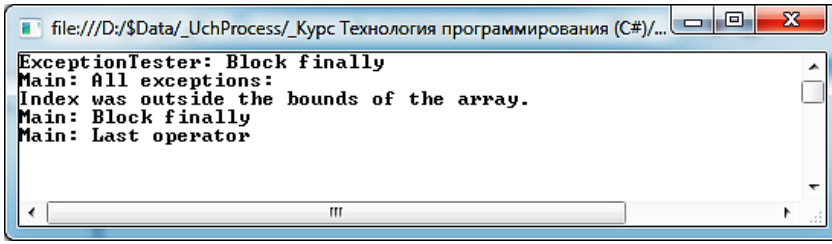

Рис.2.3. Результат применения *throw* без параметров

2.3.2 Применение оператора throw с параметром без сообщения

В некоторых случаях необходимо передать на следующий уровень приложения также и дополнительные сведения об исключительной ситуации.

Для этого после слова *throw* указывается объект исключения, через который эта инфомация передается. То есть вместо повторной генерации возникшего исключения генерируется другое исключение необходимого типа:

```
catch
{

    throw new Exception( );
}
```

Создается новое пустое исключение (рис.2.4).

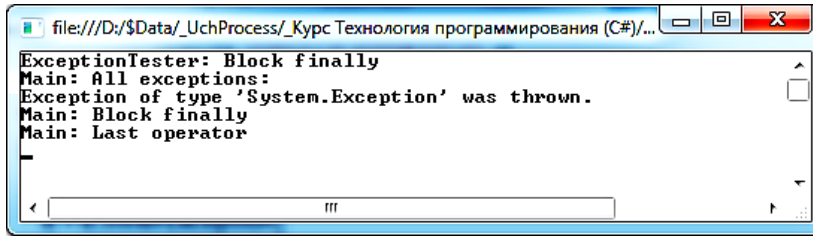

Рис.2.4. Результат применения *throw* с параметром без сообщения

2.3.3. Применение оператора throw с параметром сообщением

В этом случае используется следущий синтаксис опреатора (рис.2.5):
catch (IndexOutOfRangeException)

{

throw new Exception("Exception text");

}

Создается исключение с заданным сообщением.

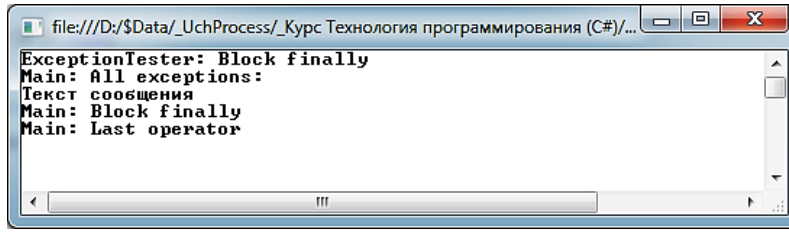

Рис.2.5. Результат применения *throw* с параметром сообщением

2.3.4. Применение оператора throw с параметром сообщением и привязкой исходного исключения

При формировании нового исключения к нему можно привязать информацию об исключении его вызвавшем, записав необходимые данные в соответствующие свойства нового исключения и поместив исходное исключение в свойство *InnerException*. Ссылка на объект исходного исключе-

ния записывается в свойство *InnerException* объекта нового исключения через конструктор.

```
catch (IndexOutOfRangeException  iore)
{
throw new Exception("Exception text", iore);
}
```

В параметрах конструктора Exception первый параметр – это сообщение нового выбрасываемого исключения, а второй параметр – *iore* это исключение, которое вызвало работу этого блока *catch*.

Этот второй аргумент становится внутренним исключением (*InnerException*) для вновь сформированного исключения. Таким образом можно формировать цепочки исключений, чтобы создавать картину процесса обработки исключений в целом.

Для обработки такой цепочки можно впоследствии использовать код:

```
catch (Exception e)
{
   Console.WriteLine("Main: All exceptions:");

   while (e != null)
   {
      Console.WriteLine("{0}", e.Message);
      e = e.InnerException;
   }
}
```

Через свойство *InnerException* можно получать вложенное по цепочке исключение, которое вызвало текущее исключение.

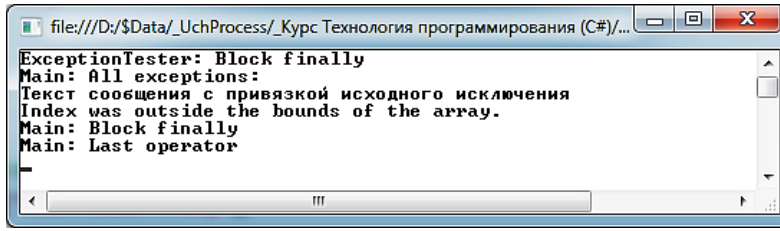

Рис.2.5. Результат применения *throw* с параметром сообщением привязкой исходного исключения

2.4. Восстановление работы после исключений

Для иллюстрации возможности восстановления нормальной работы после исключения используем уже рассмотренное исключение выход за пределы границ массива.

В примере программы в методе *GenerateException( )* предполагается возможность ошибки. Поэтому код этого метода написан так, чтобы, реагируя на исключения, устранить ошибку и продолжить работу программы. Поэтому выполняемый код этого метода помещен в цикл, условием выхода из которого служит отсутствие исключений во время выполнения обработки (см. оператор *break*). Если в цикле изменения массива не возникает исключительной ситуации, то этим оператором прекращается работа цикла. В противном случае мы попадаем в *catch* блок и, догадываясь о причине, декрементируем переменную max. Поскольку *try…catch* блок помещен в цикл, то снова предпринимается попытка выполнения действий, вызвавших ошибку, и так до ее устранения.

Результат программы, которая представлена, ниже смотрите на рис.2.5.

```
using System;
using System.Text;
```

```
namespace try_catch
{
   class ExeptionTester
   {
      public void GenerateException( )
      {
         int[] array = new int[10];
         int i = 0;
         int max = 15;   // преднамеренно задаём больше чем возможно

         do // предполагая ошибку организуем цикл для её исправления
         {
            try
            {
               for (i = 0; i < max; i++)
                  array[i] = i;

               break;   // Для выхода из цикла, в этой точке ошибок нет
            }
            catch (IndexOutOfRangeException iore)
            {
               Console.WriteLine("ExceptionTester: IndexOutOffRange: {0}",
                                                              iore.Message);

               max--;   // пытаемся исправить ошибку
            }
            catch (Exception e)
            {
               Console.WriteLine("ExceptionTester: Other exceptions: {0}",
                                                          e.Message);
            }
            finally
            {
               Console.WriteLine("ExceptionTester: Block finally");
```

```
            }
        }
        while (true);

        for (i = 0; i < max; i++)
            Console.WriteLine(array[i]);
    }
}

class Program
{
    static void Main(string[] args)
    {
        ExeptionTester test_ob = new ExeptionTester( );


        try
        {
            test_ob.GenerateException( );
        }
        catch (Exception e)
        {
            Console.WriteLine("Main: All exceptions:");
            while (e != null)
            {
                Console.WriteLine("{0}", e.Message);
                e = e.InnerException;
            }
        }
        finally
        {
            Console.WriteLine("Main: Block finally");
        }
```

```
            Console.WriteLine("Last operator");
        }
    }
}
```

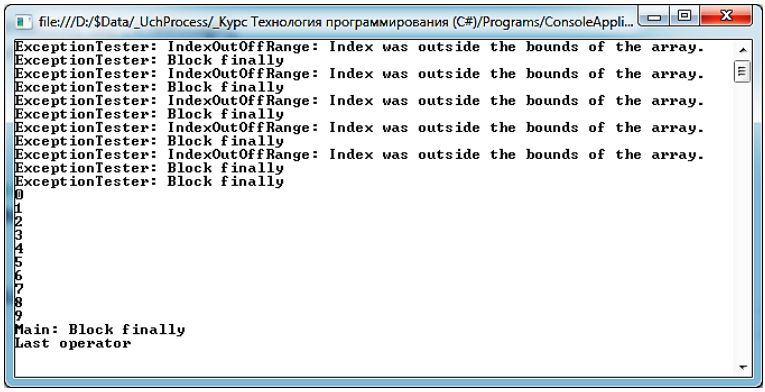

Рис.2.5. Результат применения *throw* с восстановлением правильной работы программы

Необходимо отметить, что приведенный пример является иллюстрацией процесса восстановления. В реальных программах исправление ошибок может быть намного сложнее. При выбросе исключений рекомендуется использовать уже имеющиеся классы исключений, такие, которые не имеют потомков.

3. РАЗРАБОТКА СОБСТВЕННЫХ КЛАССОВ ИСКЛЮЧЕНИЙ

Механизм исключений позволяет повысить надежность разрабатываемого программного обеспечения, не усложняя читаемость исходного кода, выделяя в отдельные составные части основной исполняемый код и код обработки ошибок, облегчает поиск и исправление ошибок в коде.

Все классы базовой библиотеки платформы .NET Framework разработаны так, что включают в себя средства внутренней диагностики обработки данных и необходимые для сигнализации об ошибках классы исключений, производные от класса System.Exception. Информацию об исключениях,

выбрасываемых во время обработки объектами классов, можно найти в справочной системе MSDN.

Класс System.Exception – очень простой тип. При возникновении ошибки выбрасывает исключение, содержащее сведения об ошибке. После создания исключения оно обрабатывается приложением или обработчиком исключений по умолчанию. Некоторые его члены представлены в табл. 3.1.

Таблица 3.1 – Члены класса *System.Exception*

Член класса	Спецификация	Описание
Конструкторы	*public Exception( )*	Инициализирует новый экземпляр класса с параметрами или без: *message* – сообщение, описывающее ошибку; *innerException* – исключение, которое вызвало текущее исключение
	public Exception(string message)	
	public Exception(string message, Exception innerException)	
InnerException	*public Exception InnerException { get; }*	Возвращает экземпляр объекта *Exception*, который вызвал текущее исключение
Message	*public string Message { get; }*	Возвращает сообщение, описывающее текущее исключение
StackTrace	*public string StackTrace { get; }*	Получает строковое представление непосредственных кадров в стеке вызова, которое позволяет проследить стек вызовов до номера строки метода, в котором происходит исключение
TargetSite	*public MethodBase TargetSite { get; }*	Возвращает объект класса *MethodBase*, описывающий метод, создавший текущее исключение.
GetBaseException( )	*public Exception GetBaseException( )*	Возвращает исключение *Exception*, которое является корневой причиной одного серии исключений

Создавать собственные классы исключений не сложно, их необходимо наследовать от базового класса, как в примере ниже.

```
using System;
using System.Text;

namespace my_class_exception
{
   class MyClassException : Exception
   {
      public MyClassException( )
         : base(@"***<MyException>***: Try smaller size!")
      { }
   }

   class ExeptionTester
   {
      public void GenerateException(int sz)
      {
         int[] array = new int[10];
         int i = 0;


         try
         {
            if (sz > array.Length) // диагностируем ошибку
            {
               throw new MyClassException( );  // выбрасывем исключение
            }

            for (i = 0; i < sz; i++)
               array[i] = i;

         }
```

```
        finally
        {   // этот блок выполняется независимо от наличия исключения
            Console.WriteLine("ExceptionTester: Block finally");
        }

        for (i = 0; i < sz; i++)
            Console.WriteLine(array[i]);
    }
}

class Program
{
    static void Main(string[] args)
    {
        ExeptionTester test_ob = new ExeptionTester( );

        try
        {
            test_ob.GenerateException(20);
        }
        catch (MyClassException e)
        {
            Console.WriteLine("{0}", e.Message);
        }
        finally
        {
            Console.WriteLine("Main: Block finally");
        }

        Console.WriteLine("Last operator");
    }
}
}
```

Результат представлен на рис.3.1.

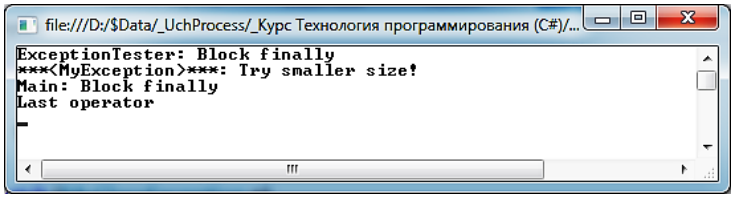

Рис.3.1. Результат программы с использованием своего класса исключения

В завершение сделаем следующее замечание. При выполнении выражений над целыми данными с использованием операций инкремента, декремента, унарного вычитания, сложения, вычитания, умножения и деления возможно переполнение разрядной сетки результата. Это может быть как ошибкой, так и возможно допустимым результатом обработки.

Для контроля выполнения операций в целочисленной арифметике можно настраивать контекст среды выполнения. Операторы C# могут выполняться в проверяемом или непроверяемом контексте. В проверяемом контексте арифметическое переполнение вызовет исключение. В непроверяемом контексте арифметическое переполнение будет проигнорировано, а результат усечен.

Установить контекст можно установкой флагов checked и unchecked соответственно ключевыми словами *checked* и *unchecked*.

Если не установлено ни checked, ни unchecked, то по умолчанию контекст зависит от внешних факторов, например параметров компилятора.

С помощью параметра компилятора */checked* можно указать проверяемый или непроверяемый контекст для всех целочисленных арифметических операторов, которые явно не выражены в области действия ключевого слова *checked* или *unchecked*.

Ключевое слово *checked* используется для явного включения проверки переполнения при выполнении арифметических операций и преобразований с данными целого типа.

Ключевое слово *unchecked* используется для подавления проверки переполнения при выполнении арифметических операций и преобразований с данными целого типа.

Проиллюстрируем на примере:

```
using System;
using System.Text;

namespace checked_unchecked
{

   class Program
   {
      static void Main(string[] args)
      {

         int ten = 10;

         Console.WriteLine(unchecked(int.MaxValue + ten));

         checked
         {
            int i = int.MaxValue + ten;
            Console.WriteLine(i);
         }

      }
   }
}
```

Как видно на рис.3.2–3.3 одна и та же операция в разных контекстах приводит или не приводит к исключению переполнения.

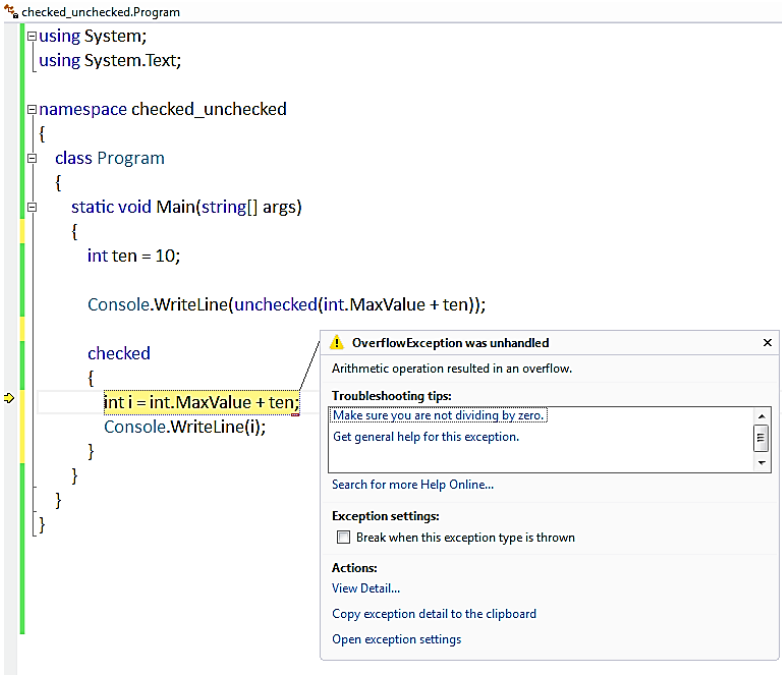

Рис.3.2. Исключение переполнения

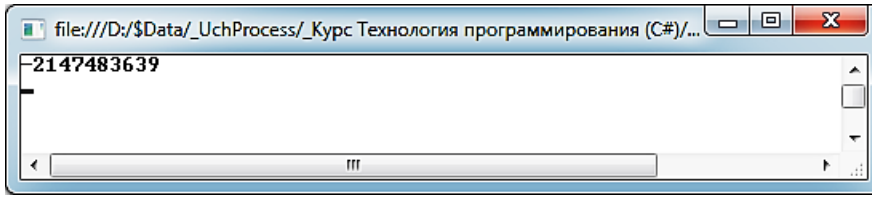

Рис.3.3. Окно выполнения программы

4. ЗАДАНИЕ

Цель работы – изучение и освоение механизма обработки исключений языка C# и платформы .Net Framework для выявления ошибок времени выполнения.

Для выполнения работ могут быть использованы системы программирования MS Visual Studio .Net 2010, 2012, 2013 MS Visual C# Express Edition 2010, 2012, 2013.

Задание

1) Изучить механизм обработки исключений и возможности его использования в языке C#.

2) Изучить возможности класса System.Exception.

3) По справочной системе MSDN ознакомиться с имеющимися типами исключений для целей дальнейшего их использования.

4) Решить задачу на закрепление полученных знаний и оформить отчет.

В файле, имя которого вводится с консоли, записаны по одному в строку целые числа a_i. Разработать консольное приложение для обработки последовательности чисел и сформировать файл результата, который имеет то же имя, что и исходный, а расширение имени «.res». В этом файле столько же чисел, сколько и в исходном. Каждое число r_i в результирующем файле с индексом i (начиная с 0) вычисляется по формуле:

$$r_i = \left(\sum_{k=1,i-1} a_k \right) / a_i .$$

Требования к программной реализации

1. Для хранения чисел использовать обычный массив (размер const1).

2. Чтение чисел из исходного файла совместить во времени с обработкой и записью в файл результата.

3. Количество чисел в файле может быть больше числа const1. Поэтому необходимо отслеживать исключение выхода за границы массива, при обработке которого выделять память под новый массив в два раза больше, чем она была в старом, переписывать в новый массив элементы из старого, продолжать обработку исходного и формирование результирующего файла (восстановление работоспособности программы).

4. В файле могут встречаться числа, равные 0. Поэтому отслеживать исключение деления на 0. В блоке обработки этого исключения сообщить об ошибке на консоль, а в файл результата – записать “бесконечность”.

5. Сформированный файл результата вывести на консоль.

СПИСОК ЛИТЕРАТУРЫ

1. Рейли Д. Создание приложений Microsoft ASP.Net / Д.Рейли : пер.с англ.– М.: Изд.-торгов.дом. «Русская редакция», 2002. – 480с., ил.

2. Петцольд Ч. Программирование для Microsoft Windows на C#: В 2-х т. Т.1. Ч. Петцольд : пер. с англ. – М.: Изд.-торгов.дом «Русская Редакция», 2002. – 576 с., ил.

3. Петцольд Ч. Программирование для Microsoft Windows на C#: В 2-х т. Т.2. Ч. Петцольд : пер. с англ. – М.: Изд.-торгов.дом «Русская Редакция», 2002.– 624 с., ил.

4. Лабор В. В. Си Шарп: Создание приложений для Windows / В. В. Лабор.– Мн.: Харвест, 2003. – 384 с.

5. Рихтер Дж. Программирование на платформе Microsoft .NET Framework / Дж. Рихтер : пер. с англ. – 2-е изд., испр. – М.: Изд.-торгов.дом «Русская Редакция», 2003. – 512 стр., ил.

6. Разработка Windows-приложений на Microsoft Visual Basic .NET и Microsoft Visual C# -NET: Учеб.курс MCAD/MCSD : пер. с англ. – М.: Изд.-торгов.дом «Русская Редакция», 2003. – 512 стр., ил.

7. Троелсен Э. C# и платформа NET. Библиотека программиста / Э. Троелсен. – Спб.: Питер, 2003. – 800с.,ил.

8. Анализ требований и создание архитектуры решений на основе Microsoft .NET: Учеб.курс MCSD: пер. с англ.– М.: Изд.-торгов.дом “Русская Редакция”, 2004. – 416 стр., ил.

9. Шилдт Г. Полный справочник по C# / Шилдт Г. : пер. с англ. – М.: Изд.дом «Вильямс», 2004. – 752 с., ил.

10. Бишоп Дж. C# в кратком изложении / Дж. Бишоп, Н. Хорспул : пер. с англ. – М.: «Бином», Лаборатория знаний, 2005. – 472с., ил.

Содержание

Учет использования методички

	Ф.И.О. студента	Дата	Роспись

Навчальне видання

Методичні вказівки
до лабораторної роботи за темою
«Обробка виключних ситуацій в мові C#» з дисципліни
«Технології програмування»
для студентів спеціальностей
122 – Комп'ютерні науки та інформаційні технології,
124 – Системний аналіз

Російською мовою

Укладачі: МАЛИХ Олег Миколайович
КОЖИН Юрій Миколайович
ПРОКОПЕНКОВ Володимир Пилипович

Відповідальний за випуск *О.С. Куценко*
Роботу до друку рекомендував *О.В. Горілий*
Редактор *О.С. Самініна*

План 2016 , поз. 88

Підп. до друку 23.03.2017 Формат 60x84 1/16 Папір офсетний.
Riso-друк. Гарнітура Таймс. Ум.друк.арк.1,26
Наклад 25 прим. Зам. № Ціна договірна.

Видавничий центр НТУ "ХПІ",
вул.Кирпичова, 21, м.Харків-2, 61002
Свідоцтво суб'єкта видавничої справи ДК №3657 від 24.12.2009 р.

ООО Планета Прінт