

Платформа Microsoft .NET и язык программирования C#



Урок №7

Обработка исключений. Сборщик мусора

Содержание

1.	Иерархия исключений	4
	Базовый класс System.Exception.....	5
	Анализ иерархии стандартных исключений	6
	Наследование и исключения.....	7
	Наследование от стандартных классов исключений	8
2.	Основы обработки исключений	12
	Ключевое слово try	12
	Ключевое слово catch	14
	Ключевое слово finally	15
	Ключевое слово throw.....	18
3.	Тонкости обработки исключений	21
	Перехват всех исключений	24
	Вложенные блоки try.....	29
	Повторное генерирование исключений.....	32

4. Применение конструкций checked и unchecked	39
5. Фильтры исключений	44
6. Использование nameof	47
7. Жизненный цикл объектов	49
8. Понятие сборщика мусора	50
9. Понятие поколений при сборке мусора	52
10. Класс System.GC	54
11. Финализатор и метод Finalize	58
12. Метод Dispose и интерфейс IDisposable	62
13. Использование using при работе с классами, реализующими интерфейс IDisposable	67
14. Домашнее задание	73

1. Иерархия исключений

Код, не содержащий ошибок — это идеал, который редко встречается на практике, но к которому все-таки стоит стремиться. Существует даже специальная область в теории надежности, которая применяется для теоретической оценки количества ошибок в проекте в зависимости от количества модулей, используемой технологии разработки и других факторов.

На самом деле причина сбоев при выполнении приложения не всегда является следствием ошибок в коде программы. Ошибки могут быть вызваны неправильными действиями пользователя или внешними причинами — аппаратные сбои, недоступность некоторых ресурсов (например, сетевого диска или сервера базы данных).

Однако это ни в коей мере не слагает ответственности с программиста, который разрабатывал данный программный продукт. Поэтому, профессионально разработанное приложение должно обеспечивать корректную обработку всех возникших ошибок.

До появления технологии .Net Framework для обработки ошибок использовались различные подходы. Так, например в WinAPI многие функции при неуспешном выполнении возвращают признак ошибки (например, FALSE, INVALID_HANDLE_VALUE, NULL). Далее с помощью функции GetLastError() можно получить код ошибки и найти по этому коду ее описание. Такой способ обработки ошибок является трудоемким, так как после вызова функции надо проверять результат возврата, и не вполне надежным, потому

что можно продолжить работу без проверки результата так, как будто функция завершилась успешно.

В .Net Framework используется так называемая структурированная обработка исключений — методика для генерации и выявления ошибок в исполняющей среде. Преимущества данной методики заключается в однотипной обработке ошибок независимо от языка семейства .Net. Еще одно достоинство данного похода состоит в идентичности обработки исключительных ситуаций вне зависимости от применяемой технологии, будь то служба, десктопное или web-приложение. Кроме того любая исключительная ситуация содержит набор свойств, позволяющих получить детальную информацию о ней. Это возможно по той причине, что каждая исключительная ситуация представлена классом наследником от базового класса `System.Exception` или его потомка.

Базовый класс `System.Exception`

В C# базовым классом для всех исключений является класс `Exception`, который содержит ряд виртуальных методов и свойств только для чтения — дополнительная информация об ошибке, которая облегчает отладку программы (Таблица 1.1).

Таблица 1.1. Свойства класса `System.Exception`

Название свойства	Описание
<code>string Message</code>	Содержит текст сообщения с указанием причины возникновения исключения.
<code>IDictionary Data</code>	Ссылка на набор пар «параметр-значение». Обычно код, генерирующий исключение, добавляет записи в этот

Название свойства	Описание
	набор. Код, перехвативший исключение, может использовать эти данные для получения дополнительной информации о причине возникновения исключения.
string Source	Содержит имя сборки, сгенерировавшей исключение.
string StackTrace	Содержит последовательность методов, вызов которых привел к возникновению исключения.
MethodBase TargetSite	Содержит объект MethodBase, описывающий метод, который сгенерировал исключение.
string HelpLink	Содержит URL документа с описанием исключения
Exception InnerException	Содержит предыдущее исключение, которое послужило причиной возникновения текущего исключения

Исключение является объектом, который создается и «выбрасывается» (*throw*) в случае возникновения ошибки. Любой CLS-совместимый язык должен быть способен генерировать и перехватывать типы исключений, производные от базового класса `System.Exception`. Эти исключения называются CLS-совместимыми.

Анализ иерархии стандартных исключений

В BCL существует множество классов исключений, которые может использовать разработчик, а также создавать собственные классы исключений. На рисунке 1.1 частично представлена диаграмма классов исключений.

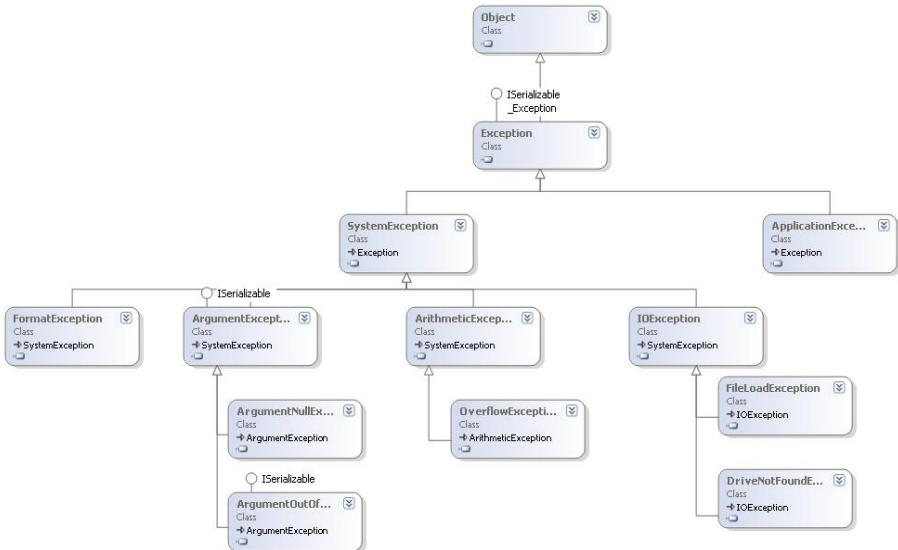


Рисунок 1.1. Иерархии классов исключений (частично)

Следует подчеркнуть, что на рисунке 1.1. представлены далеко не все исключения. Основное назначение этой диаграммы — показать общие закономерности иерархии классов исключений.

Вы, наверное, обратили внимание на то, что от базового класса `Exception` напрямую наследуются только два класса: `SystemException` и `ApplicationException`. Они же являются базовыми для остальных классов в иерархии наследования исключений.

Наследование и исключения

Класс `SystemException` — это базовый класс для исключений, которые обычно генерируются CLR или являются исключениями общей природы и могут быть сгенерированы любым приложением. Например, исключение

`StackOverflowException` генерируется CLR при переполнении стека, исключение `ArgumentException` (и производные от него) могут быть сгенерированы любым приложением, если метод получает недопустимые значения аргументов.

От класса `ApplicationException` должны наследоваться все, специфичные по своей природе, исключения, созданные разработчиками.

Наследование классов исключений является в некоторой степени необычным, так как производные классы в основном не добавляют новую функциональность к возможностям базового класса, а используются для указания более специфических причин возникновения ошибки. Например, от класса `ArgumentException` наследуются классы `ArgumentNullException` (генерируется при передаче значения `null` в качестве параметра методу, который не принимает его, как допустимое) и `ArgumentOutOfRangeException` (генерируется при выходе аргумента за допустимый диапазон значений).

Наследование от стандартных классов исключений

При создании специального исключения в своих программах, разработчик должен создавать класс наследник от `ApplicationException`, что является рекомендацией, направленной на определение причины возникновения ошибки. При обработке такого исключения будет понятно, что его причиной является некорректная работа классов самого приложения, а не базовых классов .Net Framework.

Существуют три способа создания специального исключения. Рассмотрим каждый из них на примере создания исключения, свойства которого будут содержать время и некоторую информацию.

Так как исключения, по сути, являются классами, то первый способ заключается в создании класса с членами, необходимыми для работы с ним. Для этого мы переопределим свойство Message базового класса.

```
public class MyException : ApplicationException
{
    private string _message;
    public DateTime TimeException { get; private set; }

    public MyException()
    {
        _message = «Мое исключение»;
        TimeException = DateTime.Now;
    }

    public override string Message
    {
        get
        {
            return _message;
        }
    }
}
```

На самом деле в переопределении свойств базового класса особой необходимости нет, потому что для передачи нашей информации мы можем использовать вызов конструктора базового класса. В этом и заключается второй способ создания специального исключения.

```
public class MyException : ApplicationException
{
    public DateTime TimeException { get; private set; }

    public MyException() : base("Мое исключение")
    {
        TimeException = DateTime.Now;
    }
}
```

Третий способ заключается в реализации рекомендаций .Net Framework по созданию специального класса исключения, которые заключаются в следующем:

- исключение должно быть наследником ApplicationException;
- у исключения должен быть указан атрибут [Serializable] (будет рассмотрен в последующем уроке);
- у исключения должен быть конструктор по умолчанию;
- у исключения должен быть перегруженный конструктор для установки свойства Message;
- должен быть перегруженный конструктор для обработки внутренних исключений;
- должен быть перегруженный конструктор, выполняющий сериализацию типа.

Для выполнения всех этих рекомендаций нет необходимости писать большое количество кода, достаточно воспользоваться специальной возможностью Visual Studio. Для этого необходимо написать слово Exception и два раза нажать клавишу табуляции (Tab) (Рисунок 1.2).

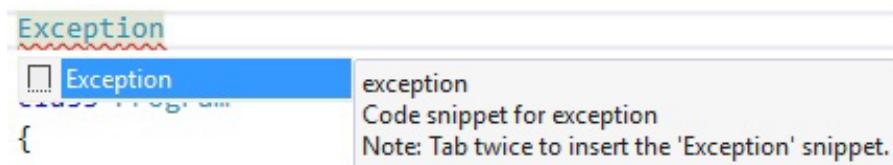


Рисунок 1.2. Создание специального класса исключений

Visual Studio сгенерирует класс, а нам останется только внести в него необходимые изменения, после чего мы получим туже функциональность, что и в первых двух случаях.

```
[Serializable]
public class MyException : Exception
{
    public DateTime TimeException { get; private set; }
    public MyException() : this(«Мое исключение»)
        { TimeException = DateTime.Now; }
    public MyException(string message) : base(message) { }
    public MyException(string message, Exception inner)
        : base(message, inner) { }
    protected MyException(System.Runtime.Serialization.
        SerializationInfo info,
        System.Runtime.Serialization.
        StreamingContext context)
        : base(info, context) { }
}
```

Хочется отметить, что полученный класс является наследником класса `Exception`, а не `ApplicationException`, как мы того ожидали. Загадка...

2. Основы обработки исключений

Для обработки всех типов исключений применяется специальная конструкция — блок `try-catch-finally`, синтаксис которого представлен ниже:

```
try
{
// код, в котором может возникнуть исключение
}
catch(тип _ исключения)
{
// обработка исключения
}
finally
{
// освобождение ресурсов
}
```

Как Вы видите, данная конструкция состоит из трех разных блоков, каждый из которых создается при помощи одноименных ключевых слов, опишем их.

Ключевое слово `try`

Блок `try` предназначен для размещения в нем операторов, выполнение которых потенциально может привести к возникновению исключительной ситуации.

Блок `try` должен использоваться только совместно с любым из блоков `catch` или `finally` или с ними обоими. Одиночное использование блока `try` не допустимо, следующий код приведет к ошибке на рисунке 2.1.

```

using static System.Console;

namespace SimpleProject
{
    class Program
    {
        static void Main(string[] args)
        {
            int number1, number2, result = 0;

            WriteLine(«Введите два числа»);

            try
            {
                number1 = int.Parse(ReadLine());
                number2 = int.Parse(ReadLine());

                result = number1 / number2;

                WriteLine($»Результат деления чисел: { result }»);
            }
        }
    }
}

```

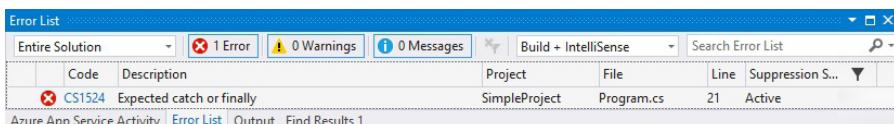


Рисунок 2.1. Ошибка: ожидается ключевое слово `catch` или `finally`

Процесс создания блоков обработки исключений также упрощен благодаря специальной возможности Visual Studio. Для этого необходимо написать слово `try`, после чего появится список возможных вариантов (Рисунок 2.2).

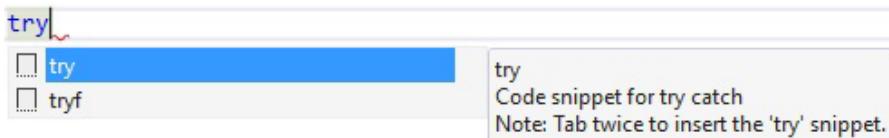


Рисунок 2.2. Создание блока обработки исключений

После выбора одного из пунктов необходимо два раза нажать клавишу табуляции. Если в списке был выбран пункт `try`, тогда сгенерируется шаблон блока `try-catch`, выбор пункта `tryf` приведет к генерации блока `try-finally`.

Ключевое слово `catch`

Блок `catch` содержит код, который выполнится только при возникновении исключения. При объявлении блока `catch` обычно указывается тип исключения, для обработки которого он предназначен. Если блок `try` завершился без генерации исключения, блок `catch` не выполняется (Рисунок 2.3).

```
using System;
using static System.Console;

namespace SimpleProject
{
    class Program
    {
        static void Main(string[] args)
        {
            int number1, number2, result = 0;

            WriteLine(«Введите два числа»);

            try
```

```
    {
        number1 = int.Parse(ReadLine());
        number2 = int.Parse(ReadLine());

        result = number1 / number2;

        WriteLine($"Результат деления чисел: { result }");
    }
    catch (DivideByZeroException de)
    {
        WriteLine(de.Message);
    }
}
```

Возможный результат работы программы.

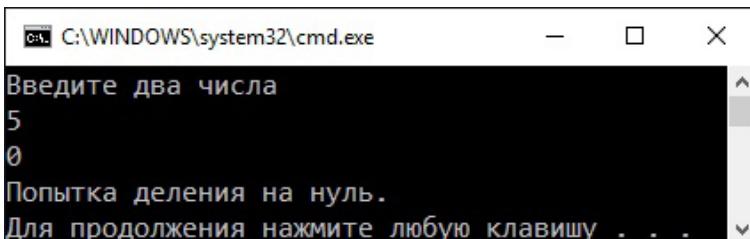


Рисунок 2.3. Обработка запрещенной операции

Ключевое слово finally

В блоке `finally` обычно размещаются действия, которые необходимо гарантировано выполнить, вне зависимости от того произошла исключительная ситуация в блоке `try` или нет, то есть блок `finally` выполняется всегда, независимо от возникновения исключения (Рисунок 2.4).

Урок №7

```
using System;

using static System.Console;

namespace SimpleProject
{
    class Program
    {
        static void Main(string[] args)
        {
            WriteLine("Введите два числа");

            int number1, number2, result = 0;

            try
            {
                number1 = int.Parse(ReadLine());
                number2 = int.Parse(ReadLine());

                result = number1 / number2;

                WriteLine($"Результат деления чисел
{result}");
            }

            catch (DivideByZeroException de)
            {
                WriteLine(de.Message);
            }

            finally
            {
                WriteLine("Очистка ресурсов");
            }
        }
    }
}
```

Возможный результат работы программы.

```
C:\WINDOWS\system32\cmd.exe
Введите два числа
48
6
Результат деления чисел 8
Очистка ресурсов
Для продолжения нажмите любую клавишу . . .
```

Рисунок 2.4. Пример работы блока finally

В блоке `finally` может находиться код очистки каких-то ресурсов, например, закрытие файла или соединения с базой данных. Если нет необходимости выполнять очистку ресурсов, блок `finally` может отсутствовать.

Следует учитывать, что использование блоков `try` и `finally` без блока `catch` синтаксически допустимо, однако в таком случае обработка исключительной ситуации не происходит и в случае ее возникновения программа заканчивается аварийно (Рисунок 2.5).

```
using static System.Console;
namespace SimpleProject
{
    class Program
    {
        static void Main(string[] args)
        {
            WriteLine("Введите два числа");
            int number1, number2, result = 0;

            try
            {
                number1 = int.Parse(ReadLine());
                number2 = int.Parse(ReadLine());
            }
            finally
            {
                Console.WriteLine("Результат деления чисел {0}", result);
            }
        }
    }
}
```

```
        result = number1 / number2;
        WriteLine($»Результат деления чисел {result}»);
    }
    finally
    {
        WriteLine(«Очистка ресурсов»);
    }
}
}
```

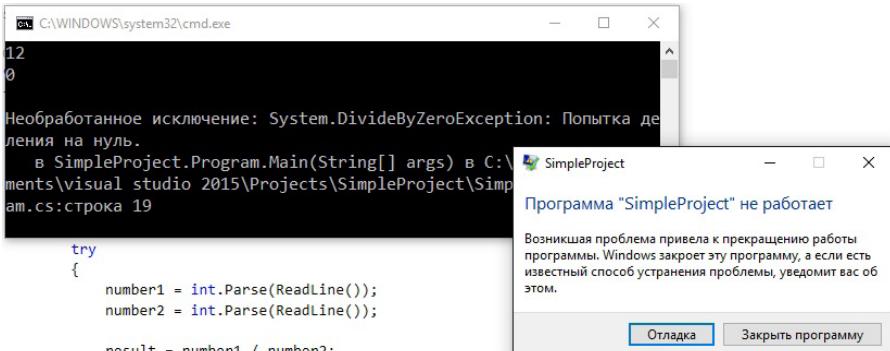


Рисунок 2.5. Ошибка времени выполнения

Ключевое слово `throw`

Ключевое слово `throw` используется для явной генерации исключения в ходе выполнения программы. Синтаксис явной генерации исключения представлен ниже:

```
throw new Тип_исключения();
```

В качестве типа исключения надо использовать класс производный от `Exception`, который наиболее полно описывает возникшую проблему. Не рекомендуется

использовать базовые классы `SystemException` или `ApplicationException`, так как в этом случае при обработке исключения трудно будет точно определить причину его возникновения.

Приведем пример явной генерации исключения, с использованием специального исключения `MyException`, примеры создания которого мы рассматривали в разделе 1.

```
using System;
using static System.Console;

namespace SimpleProject
{
    public class MyException : ApplicationException
    {
        public DateTime TimeException { get; private set; }
        public MyException() : base("«Мое исключение»")
        {
            TimeException = DateTime.Now;
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
            WriteLine("«Введите два числа»");

            int number1, number2, result = 0;

            try
            {
                number1 = int.Parse(ReadLine());
                number2 = int.Parse(ReadLine());
                result = number1 / number2;
            }
            catch (MyException ex)
            {
                Console.WriteLine(ex.Message);
            }
        }
    }
}
```

```
if (result % 2 != 0)
{
    throw new MyException();
}

WriteLine($>>Результат деления чисел:
          { result }>>);
}
catch (MyException my)
{
    WriteLine(my.Message);
    WriteLine(my.TimeException);
}
}
}
```

Результат работы программы (Рисунок 2.6).

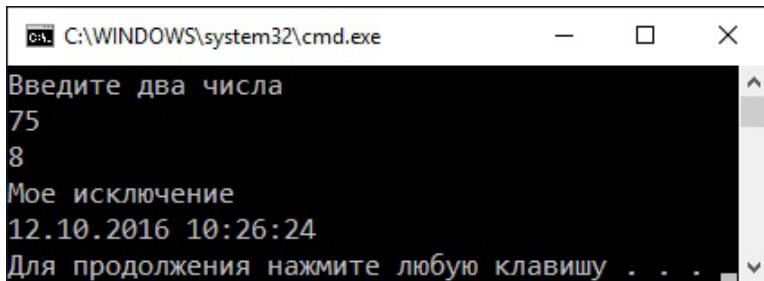


Рисунок 2.6. Явная генерация специального исключения

3. Тонкости обработки исключений

Процесс обработки исключительной ситуации происходит следующим образом. При возникновении исключения в блоке `try` выполнение этого блока прекращается и CLR выполняет поиск блока `catch`, предназначенного для обработки исключений такого типа. Если этот блок найден, он выполняется, затем выполняется блок `finally` (если он есть). Если же подходящий блок `catch` не найден, тогда исключение передается вверх по стеку вызовов в поисках необходимого блока `catch` у предыдущих методов. Последним в этой цепочке является метод `Main()`, если и в нем обработки исключения не происходит, тогда данное исключение перехватывает CLR, которая аварийно завершает выполнение программы (Рисунок 2.5).

Последовательность выполнения программы при возникновении исключительной ситуации демонстрирует следующий код (Рисунок 3.1).

```
using System;
using static System.Console;

namespace SimpleProject
{
    class Program
    {
        static void Main(string[] args)
        {
            try
```

```
{  
    WriteLine(«Код до исключения»); //1  
    throw new Exception(«Test Exception»); //2  
  
    // Эта строка никогда не появится  
    WriteLine(«Код после исключения»);  
}  
catch (Exception e)  
{  
    WriteLine($»Ошибка: { e.Message }»); //3  
}  
finally  
{  
    WriteLine(«Кода блока finally»); //4  
}  
}  
}  
}
```

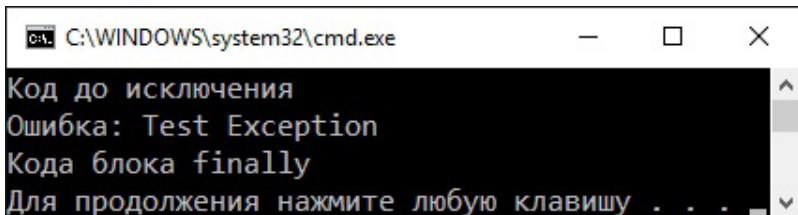


Рисунок 3.1. Последовательность выполнения операторов блока try-catch-finally

Текст «Код после исключения» не будет выведен, так как после генерации исключения выполнение блока `try` прекращается.

Также необходимо понимать, что после успешной обработки возникшей исключительной ситуации, Ваша программа продолжит свое выполнение. И Вы должны учитывать, что например, использование не

проинициализированной переменной за пределами блока `try-catch` приведет к аварийному завершению Вашей программы (Рисунок 3.2).

```
using System;
using static System.Console;

namespace SimpleProject
{
    class Program
    {
        static void Main(string[] args)
        {
            WriteLine(«Введите два числа»);
            int number1, number2, result = 0;

            string str = null;

            try
            {
                number1 = int.Parse(ReadLine());
                number2 = int.Parse(ReadLine());

                result = number1 / number2;

                str = «Проверка»;

                WriteLine(«Результат деления чисел: « + result);
            }
            catch (DivideByZeroException de)
            {
                WriteLine(de.Message);
            }
            WriteLine(str.Contains(«удалась»));
        }
    }
}
```

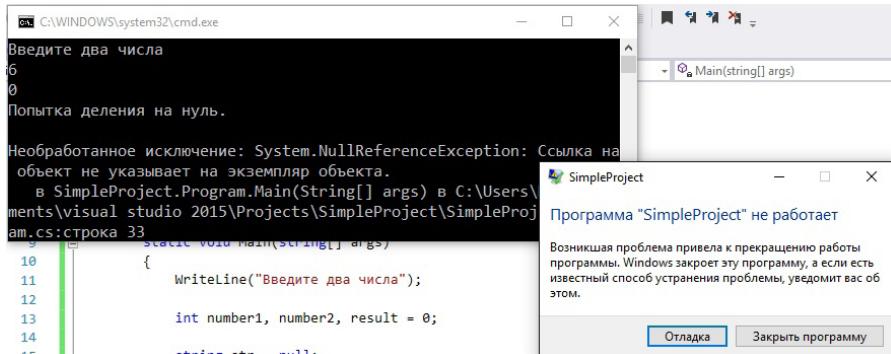


Рисунок 3.2. Необработанное исключение
после блока try-catch

Перехват всех исключений

Так как класс `System.Exception` является базовым для всех классов исключений, то для обработки всех типов CLS совместимых исключений можно использовать блок `catch`, который перехватывает именно этот класс:

```
catch (Exception e)
{
    WriteLine(e.Message);
}
```

Как уже отмечалось, в C# разрешается генерировать только исключения классов производных от `System.Exception`. Однако методы, разработанные на других языках программирования, например на C++, могут генерировать исключения любых типов. Для обработки всех типов исключений (в том числе и не CLS совместимых) в блоке `catch` можно не указывать тип перехватываемого исключения:

```
catch
{
    WriteLine(«Ошибка!!!»);
}
```

Однако в этом случае отсутствует возможность получения детальной информации о возникшей ошибке, и Вы можете указать только универсальную информацию.

Не редки ситуации, когда в блоке `try` находится код, выполнение которого может привести к генерации различного типа ошибок. В таких случаях необходимо использовать несколько блоков `catch`, каждый из которых будет обрабатывать свой тип ошибок в случае необходимости. Как уже говорилось выше, CLR осуществляет поиск подходящего для обработки ошибки блока `catch`, путем перебора всех блоков сверху вниз. Поэтому располагать их необходимо от более частных случаев к более общим, то есть последним должен быть блок обработки базового класса `System.Exception`. Неправильное расположение блоков `catch` приведет к ошибке на этапе компиляции (Рисунок 3.3).

```
using System;
using static System.Console;

namespace SimpleProject
{
    class Program
    {
        static void Main(string[] args)
        {
            WriteLine(«Введите два числа»);
            int number1, number2, result = 0;
```

```
try
{
    number1 = int.Parse(ReadLine());
    number2 = int.Parse(ReadLine());

    result = number1 / number2;

    WriteLine(<<Результат деления чисел:
              << + result);
}

catch (Exception e)
{
    WriteLine(e.Message);
}

catch (DivideByZeroException de)
{
    WriteLine(de.Message);
}

}
```

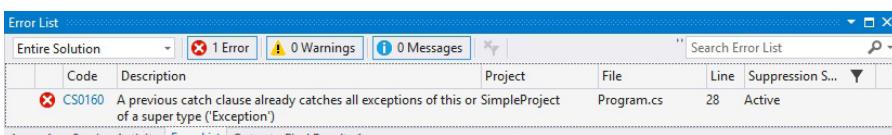


Рисунок 3.3. Ошибка: предшествующий блок catch уже перехватывает все исключения

Для демонстрации использования нескольких блоков `catch` рассмотрим следующий пример — вычисление выражение $100/\ln(n)$, где n вводится пользователем. В блоках `catch` будут перехватываться следующие типы исключений:

- `FormatException` — возникает, если введенную пользователем строку невозможно преобразовать в число;
- `DivideByZeroException` — возникает, если $\text{Ln}(n) = 0$, то есть $n = 1$;
- `Exception` — все остальные исключения.

```
using System;
using static System.Console;

namespace SimpleProject
{
    class Program
    {
        static void Main(string[] args)
        {
            do
            {
                try
                {
                    Write($"\\nВведите число: <>");
                    // чтение ввода пользователя
                    string str = ReadLine();
                    // условие выхода из цикла

                    if (string.IsNullOrEmpty(str))
                    {
                        return;
                    }
                    // преобразование строки в число
                    int number = Convert.ToInt32(str);
                    // проверка, что полученное число
                    // принадлежит области определения
                    // функции ln()
                    if (number <= 0)
                    {
                
```

```
        throw new
            ArgumentException
                («Число <= 0»);
        }
        double log = Math.Log(number);
        WriteLine($>>ln({number}) =
                    {log}\n100/ln({number}) =
                    {100 / (int)log}»);
    }

    catch (FormatException fe)
    {
        //происходит, если введенное
        //пользователем значение
        //невозможно преобразовать в целое число
        WriteLine(fe.Message);
    }

    catch (DivideByZeroException de)
    {
        //происходит, если Log(n) = 0 (т.е. n = 1)
        WriteLine(de.Message);
    }

    catch (Exception e)
    {
        //прехват всех остальных исключений
        WriteLine($>>Exception: { e.Message }»);
    }

    } while (true);
}
}
```

Скриншот работы программы (Рисунок 3.4).

C:\WINDOWS\system32\cmd.exe

```

Введите число: 0
Exception: Заданный аргумент находится вне диапазона допустимых значений.

Имя параметра: число <= 0

Введите число: 1
Попытка деления на нуль.

Введите число: 5
ln(5) = 1,6094379124341
100/ln(5) = 100

```

Рисунок 3.4. Использования нескольких блоков `catch`

Вложенные блоки `try`

Блоки `try` могут быть вложенными, общая форма записи представлена ниже:

```

try                                // внешний блок
{
    // точка А
    try                            // внутренний блок
    {
        // точка В
    }

    catch
    {
        // точка С
    }
    finally
    {
        // очистка
    }
    // точка D
}

```

```
catch
{
    // обработка ошибок
}
finally
{
    // очистка
}
```

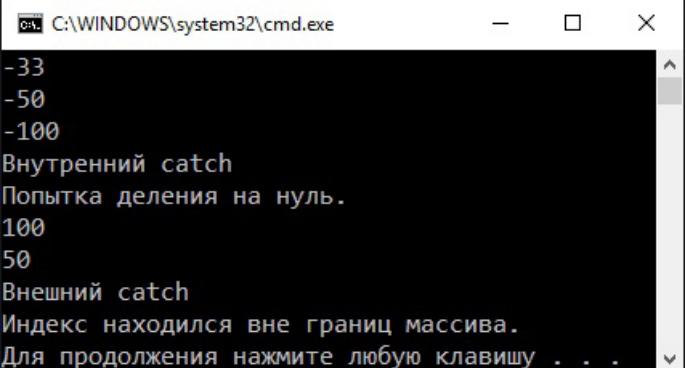
Рассмотрим возможные сценарии возникновения и обработки исключения в этом блоке. Если исключение возникает в точке В, то выполняется поиск подходящего обработчика исключения. Если исключение может быть обработано внутренним блоком `catch`, оно перехватывается и обрабатывается, после чего продолжается выполнение кода (точка D). Если в точке D не возникло нового исключения, блок `catch` внешнего блока игнорируется.

Если исключение не может быть перехвачено внутренним блоком `catch` или возникла ошибка в точке D, то выполняется проверка внешнего блока `catch`. Если этот блок не в состоянии обработать исключение, поиск подходящего обработчика выполняется выше по стеку вызовов.

Блоки `finally` вложенного и внешнего блока выполняются в любых случаях.

Рассмотрим пример использования вложенных блоков `try`. Во внутреннем блоке происходит два вида исключений: деление на 0 и обращение к массиву по недопустимому индексу. Первое исключение перехватывается внутренним блоком `catch`, второе — внешним (Рисунок 3.5).

```
using System;
using static System.Console;
namespace SimpleProject
{
    class Program
    {
        static void Main(string[] args)
        {
            int[] a = new int[5];
            int n = 0;
            try // внешний блок try
            {
                for (int i = -3; i <= 3; i++)
                {
                    //при делении на 0 не происходит выход
                    //из цикла: исключение перехватывается и
                    //обрабатывается внутренним блоком try
                    try //внутренний блок try
                    {
                        a[n] = 100 / i;
                        WriteLine(a[n]);
                        n++;
                    }
                    catch (DivideByZeroException e)
                    {
                        WriteLine(«Внутренний catch»);
                        WriteLine(e.Message);
                    }
                }
            }
            catch (IndexOutOfRangeException e)
            {
                WriteLine(«Внешний catch»);
                WriteLine(e.Message);
            }
        }
    }
}
```



The screenshot shows a command prompt window titled 'C:\WINDOWS\system32\cmd.exe'. The output of the program is as follows:

```
-33
-50
-100
Внутренний catch
Попытка деления на нуль.
100
50
Внешний catch
Индекс находился вне границ массива.
Для продолжения нажмите любую клавишу . . .
```

Рисунок 3.5. Пример вложенных блоков try

Повторное генерирование исключений

Повторное генерирование исключений осуществляется только в блоке `catch` с использованием оператора `throw`.

В первом случае оператор `throw` используется без указания типа исключения, такой синтаксис можно применить, когда из блока `catch` необходимо переслать перехваченное исключение вызывающему коду (Рисунок 3.6).

```
using System;
using static System.Console;

namespace SimpleProject
{
    class Program
    {
        static int DivisionNumbers(int n1, int n2)
        {
            int result = 0;

            try
            {
```

```
        result = n1 / n2;
    }

    catch (DivideByZeroException)
    {
        throw; //передача исключения вверх
                //по стеку вызовов
    }
    return result;
}

static void Main(string[] args)
{
    WriteLine(«Введите два числа»);

    int number1, number2, result = 0;

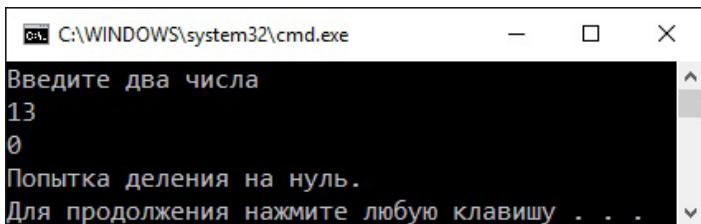
    try
    {
        number1 = int.Parse(ReadLine());
        number2 = int.Parse(ReadLine());

        result = DivisionNumbers(number1, number2);

        WriteLine($»Результат деления чисел:
                  { result }»);
    }

    catch (Exception e)
    {
        WriteLine(e.Message);
    }
}
}
```

Возможный результат работы программы.



The screenshot shows a Windows Command Prompt window titled 'cmd.exe' with the path 'C:\WINDOWS\system32\cmd.exe'. The window contains the following text:
Введите два числа
13
0
Попытка деления на нуль.
Для продолжения нажмите любую клавишу . . .

Рисунок 3.6. Повторное генерирование исключения

На самом деле при повторной генерации исключения рекомендуется передавать еще и дополнительную информацию, чтобы процесс отладки стал более информативным. Для генерации исключения такого типа используется обычный синтаксис генерации исключения. Исходное исключение, перехваченное блоком `catch`, сохраняется в поле `InnerException` нового исключения (Рисунок 3.7).

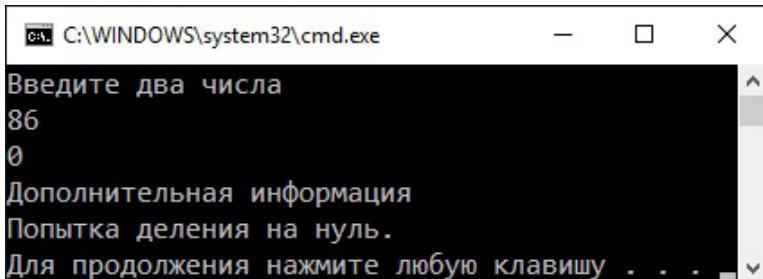
```
using System;
using static System.Console;

namespace SimpleProject
{
    class Program
    {
        static int DivisionNumbers(int n1, int n2)
        {
            int result = 0;

            try
            {
                result = n1 / n2;
            }
            catch (DivideByZeroException de)
```

```
{  
    throw new Exception(«Дополнительная  
        информация», de);  
}  
return result;  
}  
  
static void Main(string[] args)  
{  
    WriteLine(«Введите два числа»);  
    int number1, number2, result = 0;  
  
    try  
    {  
        number1 = int.Parse(ReadLine());  
        number2 = int.Parse(ReadLine());  
        result = DivisionNumbers(number1, number2);  
        WriteLine($»Результат деления чисел:  
            { result }»);  
    }  
  
    catch (Exception e)  
    {  
        WriteLine(e.Message);  
        //дополнительная информация  
  
        WriteLine(e.InnerException.Message);  
        //предыдущее исключение  
    }  
}  
}
```

Возможный результат работы программы.



The screenshot shows a command prompt window titled 'cmd' with the path 'C:\WINDOWS\system32\cmd.exe'. The window contains the following text:
Введите два числа
86
0
Дополнительная информация
Попытка деления на нуль.
Для продолжения нажмите любую клавишу . . .

Рисунок 3.7. Повторная генерация исключения с дополнительной информацией

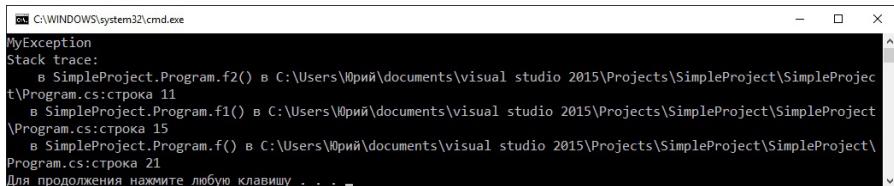
При генерации исключения CLR регистрирует метод, в котором была выполнена команда `throw` или сгенерировано исключение. Когда обнаруживается блок `catch`, способный обработать это исключение, CLR регистрирует место перехвата исключения. Теперь, если в блоке `catch` обратиться к свойству `StackTrace` сгенерированного объекта исключения, код этого свойства вызовет CLR. При этом CLR построит строку, идентифициирующую все методы, вызванные между возникновением исключения и его обработкой. Свойство `StackTrace` дает информацию об обработке исключения, включая метод, его сгенерировавший.

Пример трассировки стека при возникновении исключения приведен ниже, исключение возникает в методе `f2()`, а перехватывается в методе `f()` (Рисунок 3.8).

```
using System;
using static System.Console;

namespace SimpleProject
{
    class Program
    {
        static void f2()
        {
            throw new Exception(`MyException`);
            //генерация исключения
        }
        static void f1()
        {
            f2();
        }
        static void f()
        {
            try
            {
                f1();
            }
            catch (Exception e)
            //перехват исключения
            {
                WriteLine(e.Message);
                //вывод стека трассировки
                WriteLine(`Stack trace:\n {0} `, e.StackTrace);
            }
        }
        static void Main(string[] args)
        {
            f();
        }
    }
}
```

Возможный результат работы программы.



```
PS C:\WINDOWS\system32\cmd.exe
MyException
Stack trace:
   в SimpleProject.Program.f2() в C:\Users\Юрий\documents\visual studio 2015\Projects\SimpleProject\SimpleProject\Program.cs:строка 11
   в SimpleProject.Program.f1() в C:\Users\Юрий\documents\visual studio 2015\Projects\SimpleProject\SimpleProject\Program.cs:строка 15
   в SimpleProject.Program.f() в C:\Users\Юрий\documents\visual studio 2015\Projects\SimpleProject\SimpleProject\Program.cs:строка 21
Для продолжения нажмите любую клавишу . . .
```

Рисунок 3.8. Пример трассировки стека

Методы f(), f1(), f2() выводятся в трассировке, а метод Main() не выводится, так как он расположен в стеке вызовов выше, чем f(), в котором было перехвачено исключение.

4. Применение конструкций checked и unchecked

При выполнении арифметических операций с целочисленными типами данных может возникать переполнение, то есть ситуация при которой количество двоичных разрядов полученного результата превышает разрядность переменной, в которую этот результат записывается.

Переполнение может возникнуть в следующих ситуациях:

- в выражениях, которые используют арифметические операторы;
- при выполнении явного преобразования целочисленных типов.

При возникновении переполнения CLR может использовать один из двух вариантов:

- игнорирование переполнения и отбрасывание старших разрядов;
- генерация исключение `OverflowException`.

По умолчанию при возникновении переполнения старшие разряды отбрасываются (Рисунок 4.1).

```
using static System.Console;

namespace SimpleProject
{
    class Program
    {
```

```
static void Main(string[] args)
{
    byte b = 100;
    b = (byte)(b + 200);
    //разряды, которые не попадают в диапазон Byte
    //отбрасываются
    WriteLine($"(byte)300 = {b}"); // 44
    int n = 65536;
    short s = (short)n;
    //разряды, которые не попадают в диапазон
    //Short отбрасываются
    WriteLine($"(short)65536 = {s}"); // 0
}
}
```

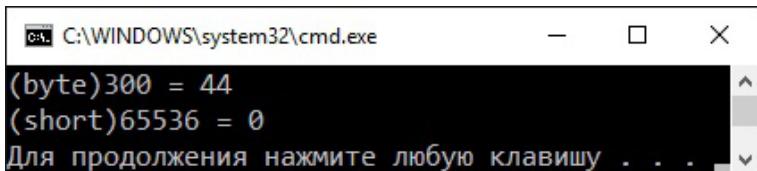


Рисунок 4.1. Переполнение при явном преобразовании

Режим контроля переполнения можно задать явно с помощью ключевых слов `checked` и `unchecked`.

Ключевое слово `checked` задает режим контроля переполнения с генерацией исключения.

Ключевое слово `unchecked` задает игнорирование возникновения переполнения.

В следующем примере демонстрируется использование этих ключевых слов на примере увеличения переменной с начальным значением 255 на единицу: в блоке `checked` — возникает исключение, которое перехватывается в блоке `catch`, а в блоке `unchecked` — исключение не возникает, в переменную записывается значение 0 (Рисунок 4.2).

4. Применение конструкций checked и unchecked

```
using System;
using static System.Console;

namespace SimpleProject
{
    class Program
    {
        static void Main(string[] args)
        {
            byte b = 255;
            try
            {
                checked
                {
                    b++; // генерация OverflowException
                }
                WriteLine(b);
            }
            catch (OverflowException e)
            {
                WriteLine(e.Message);
            }
            try
            {
                unchecked
                {
                    b++; // переполнение игнорируется
                }
                WriteLine(b); //0
            }
            catch (OverflowException e)
            {
                WriteLine(e.Message);
            }
        }
    }
}
```

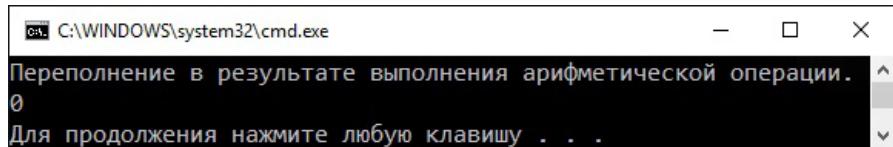


Рисунок 4.2. Использование ключевых слов checked и unchecked

Ключевые слова `checked` и `unchecked` могут использоваться как операторы в строке преобразования. В следующем примере выполняется явное приведение целого числа к типу Byte (Рисунок 4.3).

```
using System;
using static System.Console;

namespace SimpleProject
{
    class Program
    {
        static void Main(string[] args)
        {
            byte b = 100;
            WriteLine(unchecked((byte)(b + 200))); // b = 44
            try
            {
                WriteLine(checked((byte)(b + 200)));
                //генерация исключения
            }
            catch (OverflowException oe)
            {
                WriteLine(oe.Message);
            }
        }
    }
}
```

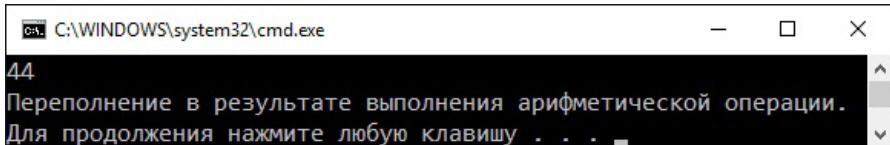


Рисунок 4.3. Стrocное использование ключевых слов checked и unchecked

Если режим контроля переполнения не указан явно, он определяется опцией компилятора /checked. Эту опцию можно задать с помощью пункта Properties меню Project. Для этого на вкладке Build необходимо нажать кнопку Advanced..., и в открывшейся форме установить или сбросить флажок Check for arithmetic overflow/underflow. Если флажок установлен, то при возникновении переполнения генерируется исключение (Рисунок 4.4).

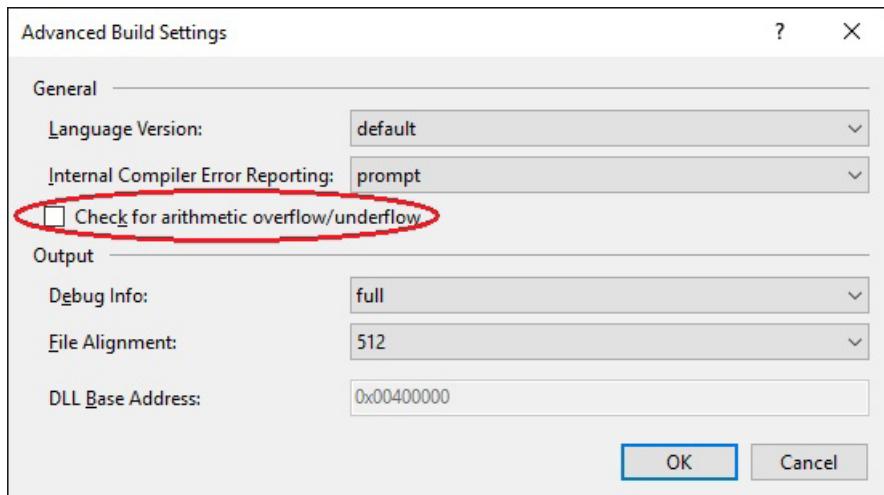


Рисунок 4.4. Окно Advanced Build Settings

5. Фильтры исключений

В C# 6.0 возможности работы с исключениями были расширены благодаря введению такой функциональности, как фильтры исключений. При помощи этих фильтров появилась возможность обрабатывать исключения исходя из дополнительно условий, заданных в блоке `catch`. Общая форма записи выглядит следующим образом:

```
try
{
    //код, в котором может возникнуть исключение
}
catch(тип_исключения) when (условие)
{
    //обработка исключения
}
```

Приведем пример применения фильтра исключения, в нем первый блок `catch` выполниться только в том случае если причиной возникновения текущего исключения является другое исключение, а второй блок `catch` выполниться во всех остальных случаях (Рисунок 5.1).

```
using System;
using static System.Console;

namespace SimpleProject
{
    class Program
    {
```

```
static int DivisionNumbers(int n1, int n2)
{
    int result = 0;

    try
    {
        result = n1 / n2;
    }
    catch (DivideByZeroException de)
    {
        throw new Exception(«Проверка фильтров
                            исключений», de);
    }
    return result;
}

static void Main(string[] args)
{
    WriteLine(«Введите два числа»);
    int number1, number2, result = 0;
    try
    {
        number1 = int.Parse(ReadLine());
        number2 = int.Parse(ReadLine());
        result = DivisionNumbers(number1, number2);

        WriteLine($»Результат деления чисел:
                  { result }»);
    }
    catch (Exception e) when (e.InnerException != null)
    {
        WriteLine(e.Message); //дополнительная
                             //информация
        WriteLine(e.InnerException.Message);
        //предыдущее исключение
    }
}
```

```
        catch (Exception e)
        {
            WriteLine(e.Message);
        }
    }
}
```

Возможный результат работы программы.

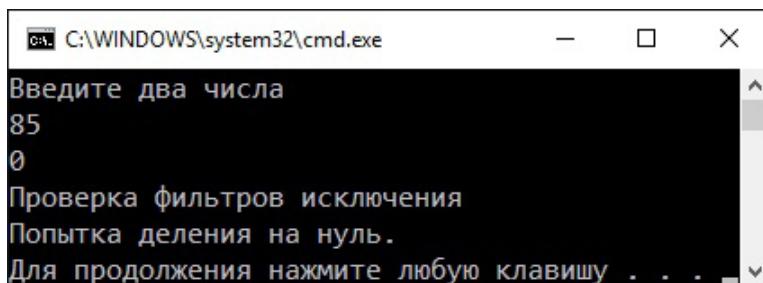


Рисунок 5.1. Использование фильтра исключения

6. Использование nameof

Еще одним улучшением в C# 6.0 является появление оператора `nameof`, при помощи которого можно получить имя класса или свойства в виде строки.

До этого введения в коде приходилось использовать так называемые «волшебные строки», в частности при генерации исключения `ArgumentException` (или его наследников) в некоторых случаях необходимо передавать имя параметра, который вызвал это исключение (нечто подобное было продемонстрировано в разделе 3). К сожалению, при написании «волшебной строки» существует большая вероятность допустить синтаксическую ошибку. Еще одной немаловажной причиной является то, что переименование элемента не будет автоматически обновлять его строковое представление, что приведет к несогласованности, которая проявится только на этапе выполнения программы.

Применение оператора `nameof` позволяет избегать синтаксических ошибок в случаях, когда необходимо указывать строковый литерал. Также использование оператора `nameof` помогает сохранить код действительным при переименовании элементов, что очень существенно в случае дальнейшего рефакторинга. Пример использования оператора `nameof` приведен на рисунке 6.1.

```
using System;
using static System.Console;
namespace SimpleProject
{
```

Урок №7

```
class ExampleNameOf
{
    public string Name { get; set; }
    public ExampleNameOf(string name)
    {
        if (name == null)
        {
            throw new ArgumentNullException(nameof(name));
        }

        Name = name;
    }
}

class Program
{
    static void Main(string[] args)
    {
        try
        {
            ExampleNameOf example = new ExampleNameOf(null);
        }
        catch (Exception e)
        {
            WriteLine(e.Message);
        }
    }
}
```

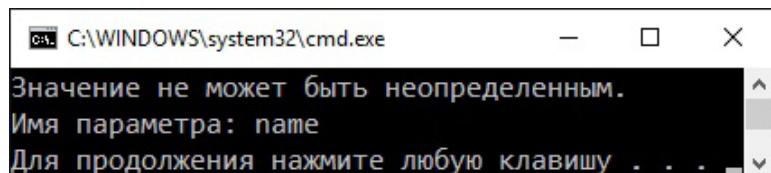


Рисунок 6.1. Пример использования оператора nameof

7. Жизненный цикл объектов

Практически любая программа для своей работы использует различные объекты, работа с которыми приводит к расходованию оперативной памяти. При объектно-ориентированном подходе программирования, каждый тип описывает некий объект, с которым после его создания может работать программа.

Жизненный цикл любого объекта можно представить следующим образом:

- выделение памяти для типа;
- инициализация выделенной памяти (установка объекта в начальное значение — вызов конструктора);
- использование объекта в программе;
- разрушение состояния объекта;
- освобождение занятой памяти.

Управление жизненным циклом объектов представляет собой утомительную задачу и является постоянным источником ошибок. Для облегчения работы с объектами в C# используется автоматический механизм управления памятью — сборщик мусора (Garbage Collector).

8. Понятие сборщика мусора

Сборщик мусора освобождает программиста от необходимости постоянно следить за использованием и своевременным освобождением памяти.

Выделение памяти под новый объект происходит автоматически самой средой CLR, при этом используется управляемая динамически распределяемая область оперативной памяти — куча (heap). Так как выделяемая память не безгранична, то необходимо периодически освобождать память, выделенную под объекты, которые больше не используются в приложении (на них нет действительных ссылок). Эту работу выполняет сборщик мусора, который является недерминированным процессом, то есть невозможно заранее узнать, когда CLR «решит» запустить процесс сборки мусора в Вашем приложении. Однако Вы можете принудительно запустить сборку мусора, вызвав в приложении метод `Collect()` класса `System.GC`.

Сборка мусора состоит из следующих шагов:

- Сборщик мусора осуществляет поиск в управляемой куче объекты, на которые нет ссылок.
- Сборщик мусора пытается завершить объекты, на которые нет ссылок.
- Сборщик мусора освобождает память, выделенную для этих объектов.

В процессе работы программы сборщик мусора не очищает выделенную под объект память, пока этот объект

еще используется приложением, то есть является достижимым. Для определения достижимости объекта CLR создает граф объектов, причем для каждого объекта свой график. Граф объектов — это совокупность графов объектов, каждый из которых хранит в себе все наборы зависимостей конкретного объекта. В каждом графике объекта фиксируется зависимость данного объекта от другого объекта или ссылка его на другой объект. До тех пор пока существует хотя бы один график объекта со ссылкой на определенный объект, этот объект считается достижимым.

Следует понимать, что сборщик мусора не запустится, пока в куче будет достаточно места для размещения новых объектов, даже если большинство объектов в памяти являются недоступными.

9. Понятие поколений при сборке мусора

Так как процесс проверки всех объектов на достижимость занял бы много времени, что естественно сказалось бы на производительности приложений, было принято решение разделить управляемую кучу на три поколения (0, 1, 2) и проверять объекты только в том из них, объем памяти, которого критичен для размещения новых объектов.

При работе приложения все вновь созданные объекты размещаются в поколение 0, объем памяти этого поколения приблизительно 256 Кbyte. Как только объем свободной памяти достигнет определенного уровня, запустится сборщик мусора, который произведет анализ объектов нулевого поколения. Все объекты, на которые нет ни одной ссылки, будут уничтожены, все остальные объекты будут перемещены в поколение 1. При этом сборщик мусора осуществляет дефрагментацию памяти и обновляет ссылки на эти объекты.

На рисунке 9.1 этот процесс представлен графически, на объекты, которые выделены серым цветом, нет ссылок (C, E, F).

В поколении 1 находятся объекты, которые остались после очистки нулевого поколения, объем памяти первого поколения приблизительно 2 Mbyte. В какой-то момент времени этот объем памяти тоже заполнится, и сборщик мусора вызовется для очистки памяти поколения 1. Все

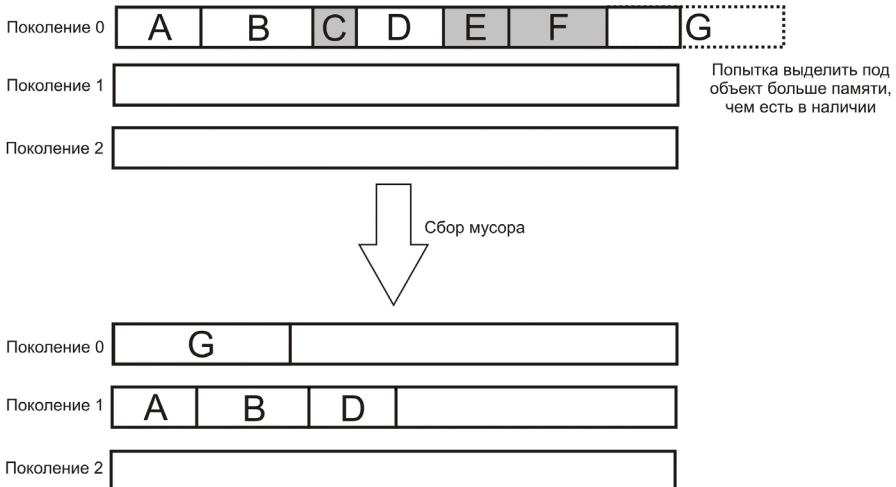


Рисунок 9.1. Работа сборщика мусора с объектами поколения 0

используемые приложением объекты переместятся в поколение 2, а остальные будут уничтожены.

Поколение 2 предназначено для хранения объектов, которые остались после процесса сборки мусора в первом поколении, объем памяти поколения 2 приблизительно 10 Mbyte. Это последнее поколение и в случае нехватки в нем памяти для размещения объектов также запускается сборщик мусора, однако объекты уже никуда не перемещаются.

Размеры памяти для всех поколений указаны, основываясь на информации из открытых источников, хотя на самом деле являются приблизительными, так как могут быть динамически увеличены CLR в случае необходимости.

10. Класс System.GC

Для программного взаимодействия со сборщиком мусора в .NET Framework предусмотрен класс `System.GC`. В основном этот класс необходимо применять при работе с неуправляемыми ресурсами, принудительно же вызывать сборщик мусора при работе с управляемым кодом следует только в самых крайних случаях.

Первый из таких случаев может произойти, когда Ваше приложение начинает выполнение некоего кода и его прерывание приведет к непоправимым последствиям. Сборка мусора может стать одной из причин такого прерывания, так как, несмотря на то, что работа сборщика мусора достаточно оптимизирована, при дефрагментации памяти происходит приостановка потока выполнения приложения, хотя и на непродолжительное время.

Вторая ситуация может возникнуть, когда для работы приложения требуется выделить большой объем памяти под определенные объекты и его необходимо освободить как можно скорее.

Принудительный запуск сборщика мусора осуществляется при помощи вызова метода `Collect()` класса `System.GC`. В перегруженном варианте этот метод принимает номер поколения, в котором необходимо произвести сборку мусора.

Методы класса `System.GC` оказывают влияние не только на то, когда выполняется сборка мусора, но и когда высвобождаются ресурсы, выделенные объектом. Свойства этого класса предоставляют сведения об общем объеме

доступной памяти системы и о том, к какому поколению относится память, выделенная объекту.

Пример использования основных методов класса System.GC показан ниже (Рисунок 10.1).

```
using System;
using static System.Console;

namespace SimpleProject
{
    class Program
    {
        static void Main(string[] args)
        {
            WriteLine("Демонстрация System.GC");
            WriteLine($"Максимальное поколение:
{GC.MaxGeneration}");

            GarbageHelper hlp = new GarbageHelper();
            //узнаем поколение, в котором находится объект
            WriteLine($"»Поколение объекта:
{GC.GetGeneration(hlp)}");
            // количество занятой памяти
            WriteLine($"»Занято памяти (байт):
{GC.GetTotalMemory(false)}");

            hlp.MakeGarbage(); //создаем мусор
            WriteLine($"»Занято памяти (байт):
{GC.GetTotalMemory(false)}");

            GC.Collect(0); //вызываем явный сбор мусора
                            //в поколении 0

            WriteLine($"Занято памяти (байт):
{GC.GetTotalMemory(false)}");

            WriteLine($"Поколение объекта:
{GC.GetGeneration(hlp)}");
```

```
GC.Collect(); //вызываем явный сбор мусора
              //во всех поколениях

WriteLine($"Занято памяти (байт):
{GC.GetTotalMemory(false)}");

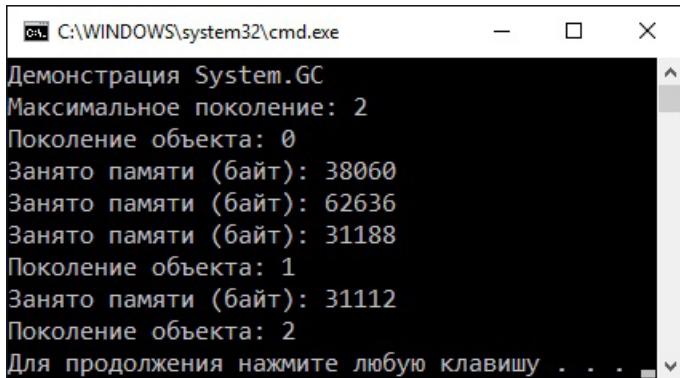
WriteLine($"Поколение объекта:
{GC.GetGeneration(hlp)}");

}

//Вспомогательный класс для создания мусора
class GarbageHelper
{
    //Метод, создающий мусор
    public void MakeGarbage()
    {
        for (int i = 0; i < 1000; i++)
        {
            Person p = new Person();
        }
    }

    class Person
    {
        string _name;
        string _surname;
        byte _age;
    }
}

}
```



```
C:\WINDOWS\system32\cmd.exe
Демонстрация System.GC
Максимальное поколение: 2
Поколение объекта: 0
Занято памяти (байт): 38060
Занято памяти (байт): 62636
Занято памяти (байт): 31188
Поколение объекта: 1
Занято памяти (байт): 31112
Поколение объекта: 2
Для продолжения нажмите любую клавишу . . .
```

Рисунок 10.1. Пример работы с основными методами класса System.GC

11. Финализатор и метод Finalize

Жизненный цикл объекта завершается очисткой памяти, выделенной при его создании. Если в таком объекте используются только управляемые типы, то разработчику нет необходимости создавать дополнительную логику по очистке памяти, сборщик мусора освободит занимаемую ими память автоматически.

Для объектов, которые используют неуправляемые ресурсы, CLR поддерживает специальный механизм, называемый финализацией объектов (finalization). Для реализации этого механизма в объекте необходимо переопределить метод `Finalize()` класса `System.Object`. Вызвать этот метод напрямую через экземпляр класса нельзя, так как он имеет модификатор доступа `protected`. Этот метод вызывается автоматически сборщиком мусора непосредственно перед уничтожением объекта.

Попытка перегрузки метода `Finalize()` с использованием ключевого слова `override` приведет к ошибке на этапе компиляции (Рисунок 11.1).

```
namespace SimpleProject
{
    class FinalizeExample
    {
        protected override void Finalize() { }
    }
}
```

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

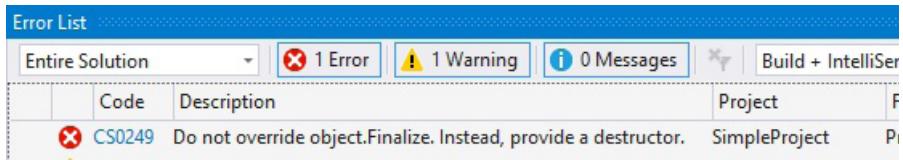


Рисунок 11.1. Ошибка: не перегружайте метод Finalize(), создайте финализатор

На самом деле для перегрузки метода `Finalize()` используется специальная конструкция — финализатор, название которого совпадает с именем класса и выделяется при помощи тильды (~). Финализатор нельзя перегружать, он не имеет модификатора доступа, не принимает параметры и не имеет типа возврата.

```
namespace SimpleProject
{
    class FinalizeExample
    {
        ~FinalizeExample()
        { }
    }
    class Program
    {
        static void Main(string[] args)
        {
        }
    }
}
```

В отличии от деструктора C++, финализатор не поддерживает детерминированное уничтожение объекта и хотя финализатор синтаксически подобен деструктору C++ — это только внешнее сходство. Объясняется это тем, что в версии C# 1.0 изначально использовался деструктор, как наследие от C++, в последующих версиях C# деструктор был заменен на финализатор, однако синтаксис оставил прежним для совместимости версий.

После компиляции данного кода мы можем посмотреть полученный CIL-код при помощи дизассемблера (Ildasm.exe), где мы обнаружим, что финализатор в нашем классе `FinalizeExample` был заменен на метод `Finalize()` (Рисунок 11.2).

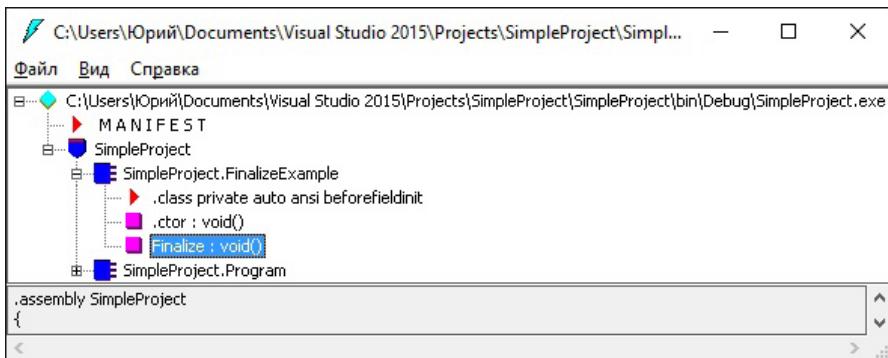


Рисунок 11.2. Отражение метода `Finalize()` типа `FinalizeExample`

Специфичным является и процесс выделения и очистки памяти у классов, имеющих финализатор.

При выделении памяти в куче каждый объект проверяется на наличие метода `Finalize()`, если он есть, тогда объект помечается как финализируемый и ссылка на него сохраняется в специальной очереди — очереди

финализации. Когда запускается сборщик мусора, он проверяет, является ли текущий объект финализируемым и выполнена ли для него финализация. Если финализация не выполнена, то сборщик мусора помещает этот объект в специальную таблицу объектов, доступных для финализации. После чего он создает отдельный поток, в котором для всех объектов в этой таблице вызывается метод `Finalize()`, а сам тем временем очищает память объектов не требующих финализации. И только при следующем вызове сборщика мусора финализируемые объекты будут уничтожены. Таким образом, получается, что для очистки памяти, выделенной под финализируемые объекты, требуется два вызова сборщика мусора.

Поэтому настоятельно не рекомендуется использовать финализатор без веских на то оснований, так как для финализации объекта требуется дополнительное время.

12. Метод Dispose и интерфейс IDisposable

Все неуправляемые объекты освобождаются сборщиком мусора в процессе финализации, при этом вызвать метод `Finalize()` напрямую нельзя. Однако не редки ситуации, когда необходимо освобождать ресурсы сразу же после их использования, например соединения с базой данных или файловые дескрипторы. Такое поведение реализуется при помощи интерфейса `IDisposable`, в котором определен единственный метод `Dispose()`. Интерфейс `IDisposable` может быть реализован как, классами так и структурами, в отличие от финализатора, который можно определять только в классах. Как Вы думаете почему?

Если объект реализует интерфейс `IDisposable`, то это уведомляет пользователя этого объекта, что для корректного завершения работы с ним необходимо вызвать метод `Dispose()`, так как требуется очистка каких-то ресурсов. Простой пример класса, реализующего интерфейс `IDisposable`, представлен на рисунке 12.1.

```
using System;
using static System.Console;

namespace SimpleProject
{
    class DisposeExample : IDisposable
    {
```

```
public void Dispose()
{
    WriteLine("Очистка ресурсов");
}

class Program
{
    static void Main(string[] args)
    {
        DisposeExample de = new DisposeExample();
        de.Dispose();
    }
}
```

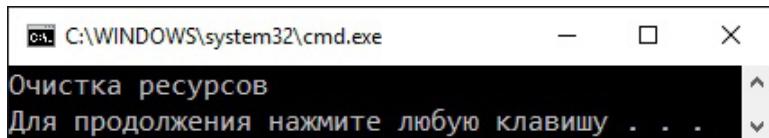


Рисунок 12.1. Пример класса, реализующего интерфейс IDisposable

Мы могли бы и дальше усовершенствовать свой код, однако в корпорации Microsoft был создан универсальный шаблон очистки ресурсов, как комбинация реализации классом интерфейса `IDisposable` и использования в нем же финализатора. Использование данного шаблона позволяет правильно очищать машинные ресурсы как явно, так и посредством автоматического сбора мусора (Рисунок 12.2).

```
using System;
using static System.Console;

namespace SimpleProject
{
    class DisposeExample : IDisposable
    {
        //используется для того, чтобы выяснить,
        //вызывался ли метод Dispose()
        private bool isDisposed = false;

        private void Cleaning(bool disposing)
        //вспомогательный метод
        {
            //убедиться, что ресурсы ещё не освобождены
            if (!isDisposed) //очищать только один раз
            {
                //если true, то освобождаем все
                //управляемые ресурсы
                if (disposing)
                {
                    WriteLine(«Освобождение управляемых
                               ресурсов»);
                }
                WriteLine(«Освобождение неуправляемых
                               ресурсов»);
            }
            isDisposed = true;
        }

        public void Dispose()
        {
            //вызов вспомогательного метода
            //true - очистка инициирована пользователем
            //объекта
            Cleaning(true);
        }
    }
}
```

12. Метод Dispose и интерфейс IDisposable

```
//запретить сборщику мусора осуществлять
//финализацию
GC.SuppressFinalize(this);
}

~DisposeExample()
{
    //false указывает на то, что очистку
    //инициировал сборщик мусора
    Cleaning(false);
}

class Program
{
    static void Main(string[] args)
    {
        DisposeExample test = new DisposeExample();
        test.Dispose();

        DisposeExample test1 = new DisposeExample();
    }
}
```

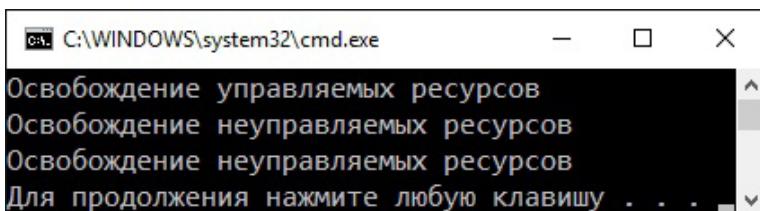


Рисунок 12.2. Шаблон реализации метода Dispose()

Обратите внимание, что даже при наличии явного управления с помощью метода Dispose() следует

предоставить и неявное освобождение ресурсов с использованием метода `Finalize()`. Метод `Finalize()` предоставляет дополнительный способ предотвращения потери ресурсов в случае, если программист забыл вызвать метод `Dispose()`.

13. Использование using при работе с классами, реализующими интерфейс IDisposable

При работе с объектами, реализующими интерфейс `IDisposable`, необходимо гарантировать очистку ресурсов в случае возникновения исключительной ситуации, поэтому следует использовать блок `try-finally` (Рисунок 13.1).

```
using System;
using static System.Console;

namespace SimpleProject
{
    class DisposeExample : IDisposable
    {
        //используется для того, чтобы выяснить, вызывался ли метод Dispose()
        private bool isDisposed = false;

        private void Cleaning(bool disposing)
        //вспомогательный метод
        {
            //убедиться, что ресурсы ещё не освобождены
            if (!isDisposed) //очищать только один раз
            {
                //если true, то освобождаем все
                //управляемые ресурсы
                if (disposing)
                {
```

```
        WriteLine(«Освобождение управляемых
                  ресурсов»);
    }
    WriteLine(«Освобождение неуправляемых
                  ресурсов»);
}
isDisposed = true;
}

public void Dispose()
{
    //вызов вспомогательного метода
    //true - очистка инициирована пользователем объекта
    Cleaning(true);
    //запретить сборщику мусора осуществлять
    //финализацию
    GC.SuppressFinalize(this);
}

~DisposeExample()
{
    //false указывает на то, что очистку
    //инициировал сборщик мусора
    Cleaning(false);
}

public void DoSomething()
{
    WriteLine(«Выполнение определенных операций»);
}

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

```
DisposeExample test = new DisposeExample();
try
{
    test.DoSomething();
}
finally
{
    test.Dispose();
}
}
```

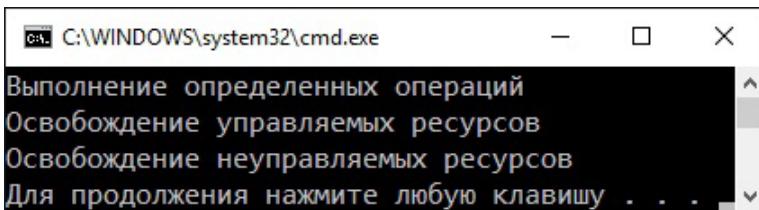


Рисунок 13.1. Использование блока try-finally

В качестве альтернативы использования блока `try-finally` для работы с классами, реализующими интерфейс `IDisposable`, можно воспользоваться специальным синтаксисом с применением ключевого слова `using`.

```
using System;
using static System.Console;
namespace SimpleProject
{
    class DisposeExample : IDisposable
    {
        //используется для того, чтобы выяснить,
        //вызывался ли метод Dispose()
        private bool isDisposed = false;
```

```
private void Cleaning(bool disposing)
//вспомогательный метод
{
    //убедиться, что ресурсы ещё не освобождены
    if (!isDisposed) //очищать только один раз
    {
        //если true, то освобождаем все управляемые
        //ресурсы
        if (disposing)
        {
            WriteLine(«Освобождение управляемых
                    ресурсов»);
        }
        WriteLine(«Освобождение неуправляемых
                    ресурсов»);
    }
    isDisposed = true;
}

public void Dispose()
{
    //вызов вспомогательного метода
    //true - очистка инициирована пользователем
    //объекта
    Cleaning(true);
    //запретить сборщику мусора осуществлять
    //финализацию
    GC.SuppressFinalize(this);
}

~DisposeExample()
{
    //false указывает на то, что очистку
    //инициировал сборщик мусора
    Cleaning(false);
}
```

```
public void DoSomething()
{
    WriteLine(«Выполнение определенных операций»);
}
class Program
{
    static void Main(string[] args)
    {
        using (DisposeExample test =
            new DisposeExample())
        {
            test.DoSomething();
        }
    }
}
```

Результат работы программы будет аналогичным (Рисунок 13.1). И если посмотреть скомпилированный CIL-код при помощи утилиты ildasm.exe, то Вы увидите, что синтаксис `using` преобразуется на самом деле в блок `try-finally` (Рисунок 13.2).

Урок №7

```
SimpleProject.Program::Main : void(string[])
{
.method private hidebysig static void Main(string[] args) cil managed
{
    .entrypoint
    // Размер кода:      30 (0x1e)
    .maxstack 1
    .locals init ([0] class SimpleProject.DisposeExample test)
    IL_0000:  nop
    IL_0001:  newobj     instance void SimpleProject.DisposeExample::ctor()
    IL_0006:  stloc.0
    .try
    {
        IL_0007:  nop
        IL_0008:  ldloc.0
        IL_0009:  callvirt   instance void SimpleProject.DisposeExample::DoSomething()
        IL_000e:  nop
        IL_000f:  nop
        IL_0010:  leave.s    IL_001d
    } // end .try
    finally
    {
        IL_0012:  ldloc.0
        IL_0013:  brfalse.s  IL_001c
        IL_0015:  ldloc.0
        IL_0016:  callvirt   instance void [mscorlib]System.IDisposable::Dispose()
        IL_001b:  nop
        IL_001c:  endfinally
    } // end handler
    IL_001d:  ret
} // end of method Program::Main
```

Рисунок 13.2. IL код синтаксиса using

14. Домашнее задание

Написать класс Money, предназначенный для хранения денежной суммы (в гривнах и копейках). Для класса реализовать перегрузку операторов + (сложение денежных сумм), – (вычитание сумм), / (деление суммы на целое число), * (умножение суммы на целое число), ++ (сумма увеличивается на 1 копейку), -- (сумма уменьшается на 1 копейку), <, >, ==, !=.

Класс не может содержать отрицательную сумму. В случае если при исполнении какой-либо операции получается отрицательная сумма денег, то класс генерирует исключительную ситуацию «Банкрот».

Программа должна с помощью меню продемонстрировать все возможности класса Money. Обработка исключительных ситуаций производится в программе.



Урок №7 Обработка исключений. Сборщик мусора

© Юрий Задерей.

© Компьютерная Академия «Шаг»
www.itstep.org.

Все права на охраняемые авторским правом фото, аудио- и видеопроизведения, фрагменты которых использованы в материале, принадлежат их законным владельцам. Фрагменты произведений используются в иллюстративных целях в объёме, оправданном поставленной задачей, в рамках учебного процесса и в учебных целях, в соответствии со ст. 1274 ч. 4 ГК РФ и ст. 21 и 23 Закона Украины «Про авторське право і суміжні права». Объём и способ цитируемых произведений соответствует принятым нормам, не наносит ущерба нормальному использованию объектов авторского права и не ущемляет законные интересы автора и правообладателей. Цитируемые фрагменты произведений на момент использования не могут быть заменены альтернативными, не охраняемыми авторским правом аналогами, и как таковые соответствуют критериям добросовестного использования и честного использования.

Все права защищены. Полное или частичное копирование материалов запрещено. Согласование использования произведений или их фрагментов производится с авторами и правообладателями. Согласованное использование материалов возможно только при указании источника.

Ответственность за несанкционированное копирование и коммерческое использование материалов определяется действующим законодательством Украины.