Л а б о р а т о р н а я р а б о т а № 5

**ОБРАБОТКА ИСКЛЮЧИТЕЛЬНЫХ СИТУАЦИЙ.** [**ВЫЧИСЛЕНИЯ С ИСПОЛЬЗОВАНИЕМ МАТЕМАТИЧЕСКИХ ФОРМУЛ (ГЕОМЕТРИЧЕСКИЕ ЗАДАЧИ)**](#_Toc290318453)

**Цель работы:**

* выработать практические навыки работы с системой С#,
* познакомиться с диагностическими сообщениями компилятора об ошибках при выполнении программ, реализующих линейные алгоритмы;
* научиться правильно использовать математические формулы;
* научиться составлять программы решения задач на разветвляющиеся алгоритмы .

**Краткие теоретические сведения**

**Обработка исключений. Конструкция try..catch..finally**

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

try

{

}

catch

{

}

finally

{

}

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

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

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

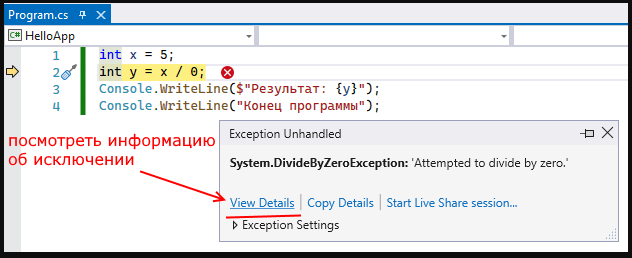
Рассмотрим следующий пример:

int x = 5;

int y = x / 0;

Console.WriteLine($"Результат: {y}");

Console.WriteLine("Конец программы");

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

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

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

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

try

{

int x = 5;

int y = x / 0;

Console.WriteLine($"Результат: {y}");

}

catch

{

Console.WriteLine("Возниклоисключение!");

}

finally

{

Console.WriteLine("Блок finally");

}

Console.WriteLine("Конецпрограммы");

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

int y = x / 0;

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

После блока **catch** будет выполняться блок **finally**.

Возникло исключение!

Блок **finally**

Конец программы

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

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

try

{

int x = 5;

int y = x / 0;

Console.WriteLine($"Результат: {y}");

}

catch

{

Console.WriteLine("Возникло исключение!");

}

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

try

{

int x = 5;

int y = x / 0;

Console.WriteLine($"Результат: {y}");

}

finally

{

Console.WriteLine("Блок finally");

}

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

**Обработка исключений и условные конструкции**

Ряд исключительных ситуаций может быть предвиден разработчиком. Например, пусть в программе есть метод, который принимает строку, конвертирует ее в число и вычисляет квадрат этого числа:

Square("12"); // Квадрат числа 12: 144

Square("ab"); // !Исключение

void Square(string data)

{

int x = int.Parse(data);

Console.WriteLine($"Квадратчисла {x}: {x \* x}");

}

Если пользователь передаст в метод не число, а строку, которая содежит нецифровые символы, то программа выпадет в ошибку. С одной стороны, здесь как раз та ситуация, когда можно применить блок **try..catch**, чтобы обработать возможную ошибку. Однако гораздо оптимальнее было бы проверить допустимость преобразования:

Square("12"); // Квадрат числа 12: 144

Square("ab"); // Некорректный ввод

void Square(string data)

{

if (int.TryParse(data, out var x))

{

Console.WriteLine($"Квадратчисла {x}: {x \* x}");

}

else

{

Console.WriteLine("Некорректныйввод");

}

}

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

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

**Определение блока catch**

За обработку исключения отвечает блок **catch**, который может иметь следующие формы:

catch

{

// выполняемые инструкции

}

Обрабатывает любое исключение, которое возникло в блоке **try**. Выше уже был продемонстрирован пример подобного блока.

catch (тип\_исключения)

{

// выполняемые инструкции

}

Обрабатывает только те исключения, которые соответствуют типу, указаному в скобках после оператора **catch**.

Например, обработаем только исключения типа **DivideByZeroException**:

try

{

int x = 5;

int y = x / 0;

Console.WriteLine($"Результат: {y}");

}

catch(DivideByZeroException)

{

Console.WriteLine("ВозниклоисключениеDivideByZeroException");

}

Однако если в блоке **try** возникнут исключения каких-то других типов, отличных от **DivideByZeroException**, то они не будут обработаны.

catch (тип\_исключенияимя\_переменной)

{

// выполняемые инструкции

}

Обрабатывает только те исключения, которые соответствуют типу, указаному в скобках после оператора **catch**. А вся информация об исключении помещается в переменную данного типа. Например:

try

{

int x = 5;

int y = x / 0;

Console.WriteLine($"Результат: {y}");

}

catch(DivideByZeroException ex)

{

Console.WriteLine($"Возниклоисключение {ex.Message}");

}

Фактически этот случай аналогичен предыдущему за тем исключением, что здесь используется переменная. В данном случае в переменную **ex**, которая представляет тип **DivideByZeroException**, помещается информация о возникшем исключени. И с помощью свойства **Message** мы можем получить сообщение об ошибке.

Если нам не нужна информация об исключении, то переменную можно не использовать как в предыдущем случае.

**Фильтры исключений**

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

catchwhen(условие)

{

}

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

int x = 1;

int y = 0;

try

{

int result1 = x / y;

int result2 = y / x;

}

catch (DivideByZeroException) when (y == 0)

{

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

}

catch(DivideByZeroException ex)

{

Console.WriteLine(ex.Message);

}

В данном случае будет выброшено исключение, так как y=0. Здесь два блока **catch**, и оба они обрабатывают исключения типа **DivideByZeroException**, то есть по сути все исключения, генерируемые при делении на ноль. Но поскольку для первого блока указано условие y == 0, то именно этот блок будет обрабатывать данное исключение - условие, указанное после оператора **when** возвращает **true**.

Противоположная ситуация:

int x = 0;

int y = 1;

try

{

int result1 = x / y;

int result2 = y / x;

}

catch (DivideByZeroException) when (y == 0)

{

Console.WriteLine("y недолженбытьравен 0");

}

catch(DivideByZeroException ex)

{

Console.WriteLine(ex.Message);

}

В данном случае будет выброшено исключение, так как x=0. Условие первого блока **catch** - y == 0 теперь возвращает false. Поэтому CLR будет дальше искать соответствующие блоки **catch** далее и для обработки исключения выберет второй блок **catch**. В итоге если мы уберем второй блок **catch**, то исключение вобще не будет обрабатываться.

**Типы исключений. Класс Exception**

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

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

Например, обработаем исключения типа **Exception**:

try

{

int x = 5;

int y = x / 0;

Console.WriteLine($"Результат: {y}");

}

catch (Exception ex)

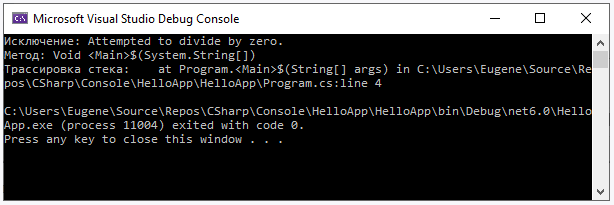
{

Console.WriteLine($"Исключение: {ex.Message}");

Console.WriteLine($"Метод: {ex.TargetSite}");

Console.WriteLine($"Трассировкастека: {ex.StackTrace}");

}



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

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

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

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

static void Main(string[] args)

{

try

{

int[] numbers = new int[4];

numbers[7] = 9; // IndexOutOfRangeException

int x = 5;

int y = x / 0; // DivideByZeroException

Console.WriteLine($"Результат: {y}");

}

catch (DivideByZeroException)

{

Console.WriteLine("ВозниклоисключениеDivideByZeroException");

}

catch (IndexOutOfRangeException ex)

{

Console.WriteLine(ex.Message);

}

Console.Read();

}

Вданномслучаеблоки catch обрабатываютисключениятипов**IndexOutOfRangeException**и**DivideByZeroException**. Когда в блоке **try** возникнет исключение, то CLR будет искать нужный блок **catch** для обработки исключения. Так, в данном случае на строке

numbers[7] = 9;

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

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

Однако рассмотрим другую ситуацию:

try

{

object obj = "you";

intnum = (int)obj; // System.InvalidCastException

Console.WriteLine($"Результат: {num}");

}

catch (DivideByZeroException)

{

Console.WriteLine("ВозниклоисключениеDivideByZeroException");

}

catch (IndexOutOfRangeException)

{

Console.WriteLine("ВозниклоисключениеIndexOutOfRangeException");

}

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

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

try

{

object obj = "you";

intnum = (int)obj; // System.InvalidCastException

Console.WriteLine($"Результат: {num}");

}

catch (DivideByZeroException)

{

Console.WriteLine("ВозниклоисключениеDivideByZeroException");

}

catch (IndexOutOfRangeException)

{

Console.WriteLine("ВозниклоисключениеIndexOutOfRangeException");

}

catch (Exception ex)

{

Console.WriteLine($"Исключение: {ex.Message}");

}

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

**Практическая часть.**

Задание: Напишите оконное приложение для решения геометрической задачи. Примените не менее 2 различных обработчиков исключений. Окно должно иметь заголовок, содержащий № лабораторной работы и варианта и кнопку закрытия окна. Окно должно содержать условие задачи. Ввод оформлять с использованием графического оформления, также ответ должен быть дан в развернутом виде. После решения задачи на форме оставить кнопку очистки элементов, которая приводит окно в первоначальное состояние, заблокировать поля ввода и спрятать кнопку запуска решения.

1. Треугольник задан длинами своих сторон A, Б, C. Написать программу, которая вычисляет площадь треугольника, если он равнобедренный или периметр впротивном случае.
2. Треугольник задан длинами своих сторон A, Б, C. Определить вид треугольника: прямоугольный, равносторонний, равнобедренный, обычный. Найти его периметр. Если треугольник построить не удается, вывести сообщение.
3. Даны координаты 3 вершин прямоугольника. Определите координаты 4 вершины.
4. Даны гипотенуза и катет. Определить радиус окружности, описанный около определяемого ими треугольника и его площадь.
5. Даны 2 катета. Найти площадь треугольника, его периметр и длину гипотенузы.
6. Даны 3 точки со своими координатами. Если они могут быть вершинами треугольника, то вычислить его площадь. В противном случае – сумму длин трех отрезков.
7. Даны 3 точки А, В, С со своими координатами. Определить площадь прямоугольного треугольника АВД, в предположении, что АВСД – прямоугольник. Если прямоугольник по заданным точкам построить невозможно – выдать соответствующее сообщение.
8. Даны 3 точки А, В, С со своими координатами. Точка Д находится водной полуплоскости с точкой А относительно прямой ВС. Определить периметр треугольника ВСД. Если известно, что АВСД – параллелограмм
9. Даны 3 точки А, В, С со своими координатами. Точка Д находится в разных полуплоскостях с точкой А относительно прямой ВС. Определить площадь треугольника АВД, в предположении, что АВСД – параллелограмм. Если параллелограмм по заданным точкам построить невозможно – выдать соответствующее сообщение.
10. Треугольник задан длинами своих сторон: найти: длины высот, длины медиан, длины биссектрис.
11. Вычислить площадь треугольника со сторонами К и прилежащими углами a и b.
12. Найти площадь круга, вписанного в треугольник с заданными сторонами
13. Найти площадь квадрата, вписанного в правильный треугольник, если сторона треугольника равна k.
14. **(х1, у1) и (х2, у2**) — координаты левой верхней и правой нижней вершин прямоугольника. Найдите его площадь.
15. Даны два угла треугольника (в градусах). Определить, существует ли такой треугольник. Если да, то прямоугольный ли он.
16. Заданы координаты (на плоскости) вершин четырехугольника. Определить, является ли он:
17. ромбом;
18. параллелограммом;
19. прямоугольником;
20. квадратом.
21. Для двух треугольных пирамид, заданных ребрами, определить, площадь полной поверхности которой из них больше и на сколько.

**Контрольные вопросы**

1. Какие условные операторы Вам известны?
2. Перечислите операторы ввода\вывода.
3. Какие математические операции Вам известны?
4. Как в C# реализовать обработчик исключительных ситуаций?
5. Перечислите типы исключений используемые в обработчиках.