12. ОБРАБОТКА ИСКЛЮЧЕНИЙ

Оглавление

§12.1 Исключения

Исключение — это событие при выполнении программы, которое приводит к её ненормальному или неправильному поведению. Существует два вида исключений:

- **Аппаратные** (структурные, SE-Structured Exception), которые генерируются процессором. К ним относятся, например,
  - деление на 0;
  - выход за границы массива;
  - обращение к невыделенной памяти;
  - переполнение разрядной сетки.
- **Программные**, генерируемые операционной системой и прикладными программами – возникают тогда, когда программа их явно инициирует. Когда встречается аномальная ситуация, та часть программы, которая ее обнаружила, может сгенерировать, или возбудить, исключение.

Механизм структурной обработки исключений позволяет однотипно обрабатывать как программные, так и аппаратные исключения.

§12.2 Обработка программных исключений

Фундаментальная идея обработки исключительных ситуаций состоит в том, что функция, обнаружившая проблему, но не знающая как её решить, генерирует исключение в надежде, что вызвавшая её (непосредственно или косвенно) функция сможет решить возникшую проблему. Функция, которая может решать проблемы данного типа, указывает, что она перехватывает такие исключения.

Язык C# предоставляет разработчикам возможности для обработки таких ситуаций. Для этого в C# предназначена конструкция `try...catch...finally`.

Листинг 12.1

```
1  try
2  {
3      // Блок кода, проверяемый на наличие ошибок.
4  }
5  catch
6  {
7      // Обработчик исключения.
8  }
9  finally
10 {
11     //Выполняется всегда
12 }
```

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:

Листинг 12.2

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

```
11  }
```

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:

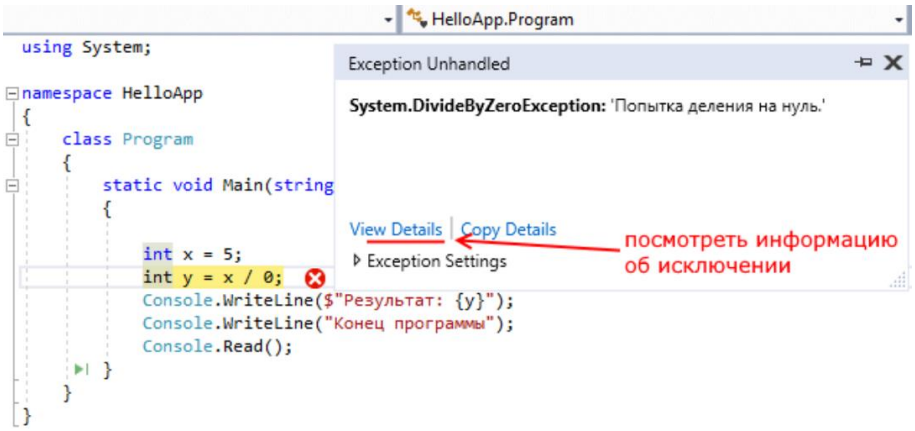

Рисунок 12.1

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

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

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

Листинг 12.3

```
1   class Program
2   {
3       static void Main(string[] args)
4       {
5           try
6           {
7               int x = 5;
8               int y = x / 0;
9               Console.WriteLine($"Результат: {y}");
10          }
11          catch
12          {
13              Console.WriteLine("Возникло исключение!");
14          }
```

```
15          finally
16          {
17              Console.WriteLine("Блок finally");
18          }
19          Console.WriteLine("Конец программы");
20          Console.Read();
21      }
22  }
```

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

```
1  int y = x / 0;
```

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

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

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`.

§12.3 Обработка исключений и условные конструкции

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

Листинг 12.4

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

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

Листинг 12.5

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

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

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

§12.4 Блок catch и фильтры исключений

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, указанному в скобках после оператора catch.

Например, обработаем только исключения типа `DivideByZeroException`:

Листинг 12.6

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

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

`catch (тип_исключения имя_переменной)`
`{`
`// выполняемые инструкции`
`}`
	Обрабатывает только те исключения, которые соответствуют типу, указанному в скобках после оператора `catch`. А вся информация об исключении помещается в переменную данного типа.

Например:

Листинг 12.7

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

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 исключении. 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:

```
1  catch when(условие)
2  {
3
4  }
```

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

Листинг 12.8

```
1  int x = 1;
2  int y = 0;
3  try
4  {
5      int result = x / y;
6  }
7  catch(DivideByZeroException) when (y==0 && x == 0)
8  {
9      Console.WriteLine("y не должен быть равен 0");
10 }
11 catch(DivideByZeroException ex)
12 {
13     Console.WriteLine(ex.Message);
14 }
```

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

§12.5 Класс Exception

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

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

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

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

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

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

Листинг 12.9

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

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

```
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:

Листинг 12.10

```
1   static void Main(string[] args)
```

```
2   {
3       try
4       {
5           object obj = "you";
6           int num = (int)obj;     // InvalidCastException
7           Console.WriteLine($"Результат: {num}");
8       }
9       catch (DivideByZeroException)
10      {
11          Console.WriteLine("Возникло исключение
DivideByZeroException");
12      }
13      catch (IndexOutOfRangeException)
14      {
15          Console.WriteLine("Возникло исключение
IndexOutOfRangeException");
16      }
17      Console.Read();
18  }
```

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

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

Листинг 12.11

```
1   static void Main(string[] args)
2   {
3       try
4       {
5           object obj = "you";
6           int num = (int)obj;     // InvalidCastException
7           Console.WriteLine($"Результат: {num}");
8       }
9       catch (DivideByZeroException)
10      {
```

```
11          Console.WriteLine("Возникло исключение DivideByZeroException");
12      }
13      catch (IndexOutOfRangeException)
14      {
15          Console.WriteLine("Возникло исключение IndexOutOfRangeException");
16      }
17      catch (Exception ex)
18      {
19          Console.WriteLine($"Исключение: {ex.Message}");
20      }
21      Console.Read();
22  }
```

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