ИСКЛЮЧЕНИЯ

Говорят, что в любой программе есть ошибки. Так это, или нет, мы обсуждать не готовы, но в любом случае во время выполнения программы могут возникать ситуации, когда дальнейшее выполнение программы не возможно. Например, система пытается сохранить данные в файл, занятый другой программой, или выполнить некоторые действия, на выполнение которых нет прав. Любая аналогичная ситуация делает дальнейшее выполнение кода бессмысленным и называется *исключительной ситуацией*.

Исключение – это специальный объект, создаваемый в программе, указывающий на возникноверие исключительной ситуации. Поля этого объекта хранят информацию о причине и месте возникновения исключения.

Конечно, кроме создания исключений язык программирования должен предоставлять способы их обработки.

Зачем нужны исключения

Чтобы разобраться, как работают исключения и в чем преимущества работы с ними, рассмотрим несколько простых примеров.

Пример 1

Посмотрите на следующий фрагмент кода:

```
for (int i = 0; i < a1.Length; i++)
{
   for (int j = 0; j < a2.Lenght; j++)
   {
      DoAction(a1[i], a2[j]);
   }
}
```

Предположим, что в методе DoAction выполняются некоторые действия, а сам метод возвращает false, если цикл обработки нужно прервать. Единственный способ сделать это – использовать оператор break:

```
for (int i = 0; i < a1.Length; i++)
{
   for (int j = 0; j < a2.Lenght; j++)
   {
      if (!DoAction(a1[i], a2[j]))
      {
         break;
      }
   }
}
```

Использование оператора break прервет выполнение внутреннего цикла, но не решит проблему с внешним циклом, хотя его работу тоже нужно прервать. Значит, нужен некий флаг, контролируя который мы сможем прервать работу внешнего цикла. Например, это можно сделать следующим образом:

```
bool stop = false;
for (int i = 0; i < a1.Length; i++)
{
   for (int j=0; j< a2.Lenght; j++)
   {
      if (!DoAction(a1[i], a2[j]))
      {
         stop = true;
         break;
      }
   }
   if (stop)
   {
      break;
   }
}
```

Даже в таком простом примере нам пришлось вводить лишнюю переменную и писать код, который останавливает внешний цикл. А представьте, сколько лишнего кода придется написать в большой системе! Кроме того, не всегда можно изменять код программы – вполне может случиться, что метод DoAction будет вызываться из внешней системы, исходного кода которой у нас попросту нет.

Пример 2

Рассмотрим работу с файлами:

```
FileStream f = new FileStream("test.txt", FileMode.Open);
DoAction(f);
f.Close();
```

Предположим, что в результате выполнения метода DoAction что-то случилось, и программа аварийно завершается. Это означает, что команда f.Close() не будет выполнена и файл останется не закрытым. Конечно, его закроет сборщик мусора, но в какой момент времени это случится – не понятно. В результате, файл в течение некоторого времени останется заблокированным.

Обе проблемы, рассмотренные в предыдущих примерах, решаются с помощью механизмов, предоставляемых операционной системой: исключений и блоков обработки исключений. Исключение позволяет прервать работу любого кода. Блок обработки исключений позволяет обработать исключение, т.е., указать код, который выполнится при возникновении исключения. Чуть позже мы покажем, как просто выглядит тот же код, когда используются исключения, а пока рассмотрим операторы языка C# для работы с исключениями.

Операторы throw, try, catch, finally

Как и все в C#, исключения представляются классами. Все классы исключений порождены от базового класса Exception, который определен в пространстве имен System. Для создания исключительной ситуации используется оператор throw, например:

```
throw new Exception();
```

Управление обработкой исключений основывается на использовании операторов *try, catch и finally*. Синтаксис управления обработкой исключений:

```
try  // контролируемый блок
{
   …
}
catch //один или несколько блоков обработки исключений
{
   …
}
finally //блок завершения
{
   …
}
```

Программные инструкции, которые нужно проконтролировать на предмет исключений, помещаются в блок *try*. При возникновении исключения внутри этого блока управление будет передано на тот блок catch, который умеет обрабатывать возникшие исключения. Блок finally будет выполнен в любом случае, не зависимо от того, возникли исключения внутри блока try или нет.

В языках C++, Delphi и многих других существовало два механизма обработки ошибок: исключения и коды ошибок. Многие функции возвращали код ошибки, если ее выполнение завершилось неудачей. Это создавало массу неудобств – часто было очень сложно понять, что же нужно делать – либо обрабатывать исключения, либо код ошибки, либо и то и другое сразу. Разработчики языка C# отказались от кодов ошибок и используют единый механизм – при возникновении ошибки выполнения метода, метод генерирует исключение соответствующего типа.

Рассмотрим фрагмент программы, демонстрирующий, как отследить и перехватить исключение.

```
static void Main()
{
   Console.Write("x=");
   int x = int.Parse(Console.ReadLine());
   int y = 1 / x;
   Console.WriteLine("y={0}", y);
}
```

Перечислим, какие исключительные ситуации могут возникнуть.

1. Пользователь может ввести нечисловое значение и выполнение оператора Parse завершится неудачей.
2. Пользователь может ввести значение 0, и произойдет деление на 0.

Создайте указанные исключительные ситуации и посмотрите, как отреагирует на них система.

Задание

Переменные x и y объявлены целочисленными. Объясните, что будет выведено на экран, если замените их тип на double и ввести с клавиатуры значение x равное 0, и почему.

Теперь попробуем обработать эти ситуации. Для этого изменим ("обернем") выполняемый код блоком try-catch, а для полноты добавим еще и блок finally:

```
static void Main()
{
   try
   {
      Console.Write("x=");
      int x = int.Parse(Console.ReadLine());
      int y = 1 / x;
      Console.WriteLine("y={0}", y);
      Console.WriteLine("блок try выполнился успешно");
   }
   catch
   {
      Console.WriteLine("возникла какая-то ошибка");
   }
   finally
   {
      Console.WriteLine("блок finally выполнился успешно");
   }
   Console.WriteLine("конец программы");
}
```

Результат работы программы:

```
x=5
y(5)=0
блок try выполнился успешно
блок finally выполнился успешно
конец программы
```

```
x=0
возникла какая-то ошибка
блок finally выполнился успешно
конец программы
```

```
x=пять
возникла какая-то ошибка
блок finally выполнился успешно
конец программы
```


Рассмотрим, как шло выполнение программы в этих случаях. В первом случае все прошло успешно, и выполнились все операторы блока try, затем операторы блока finally, а затем операторы, идущие вслед за блоком try. Во втором и третьем случаях выполнение кода было прервано возникшей исключительной ситуацией, и управление перешло на код блока catch, после чего выполнился блок finally и операторы после блока него.

Следует отметить, что в блок finally помещаются такие действия, которые необходимо обязательно выполнить для корректного завершения работы программы, например, закрыть файл, разорвать сетевое соединение, записать данные в базу. Если таких обязательных действий делать не надо, то блок finally можно не использовать.

Обработчик исключений позволяет не только обработать исключение, но и вывести полную информацию о нем. Для демонстрации сказанного заменим блок catch следующим фрагментом:

```
catch (Exception error)
{
   Console.WriteLine("Возникла ошибка {0}", error);
}
```

Теперь, если возникнет исключительная ситуация, переменная error будет содержать подробную информацию о ней. Класс Exception содержит поле Message, хранящее сообщение об исключительной ситуации. Данное поле можно использовать для получения информации об исключительной ситуации, например, следующим образом:

```
catch (Exception error)
{
   Console.WriteLine(error.Message);
}
```

Теперь при возникновении исключения на экран будует выведено короткое сообщение, содержащее описание ошибки.

В таблице ниже приведены несколько стандартных типов исключений.

Имя	**Описание**
ArithmeticException	Ошибка в арифметических операциях, или преобразованиях.
ArrayTypeMismatchException	Попытка сохранения в массиве элемента несовместимого типа.
DivideByZeroException	Попытка деления на ноль.
FormatException	Попытка передать в метод аргумент неверного формата.
IndexOutOfRangeException	Индекс массива выходит за границу диапазона.
InvalidCastException	Ошибка преобразования типа.
OutOfMemoryException	Недостаточно памяти для нового объекта.
OverflowException	Перевыполнение при выполнении арифметических операций.
StackOverflowException	Переполнение стека.

Блок catch позволяет "фильтровать" исключения, которые он "ловит", что дает возможность, во-первых, обработать только нужные исключения, а во-вторых, обработать их по-разному.

Внесем изменения в программу.

```
static void Main()
{
   try
   {
      int x = int.Parse(Console.ReadLine());    // 1 ситуация
      int y = 1 / x;                            // 2 ситуация
      Console.WriteLine("y={0}", y);
      Console.WriteLine("блок try выполнилсь успешно");
   }
```

```
        catch (FormatException error)        // обработка 1 ситуации
        {
           Console.WriteLine(error.Message);
        }
        catch (DivideByZeroException error)  //обработка 2 ситуации
        {
           Console.WriteLine(error.Message);
        }
        Console.WriteLine("конец программы");
     }
```

Теперь операторы первого блока catch будет выполнены в случае ввода строки вместо числа, а операторы второго блока – при вводе нуля.

Одно из основных достоинств обработки исключений состоит в том, что она позволяет программе отреагировать на ошибку и продолжить выполнение. Рассмотрим фрагмент программы, которая строит таблицу значений для функции вида $y(x)=\frac{100}{x^2-1}$.

```
static void Main()
{
   Console.Write("a=");
   int a = int.Parse( Console.ReadLine());
   Console.Write("b=");
   int b = int.Parse(Console.ReadLine());
   for (int i = a; i <= b; i++)
   {
      try
      {
         Console.WriteLine("y({0})={1}", i, 100 / (i * i - 1));
      }
      catch (DivideByZeroException error)
      {
         Console.WriteLine("y({0})={1}", i, error.Message);
      }
   }
}
```

Результат работы программы:

```
a=-3
b=1
y(-3)=12
y(-2)=33
y(-1)=Попытка деления на нуль.
y(0)=-100
y(1)= Попытка деления на нуль.
```

Если встречается деление на нуль, генерируется исключение типа DivideByZeroException. В программе это исключение обрабатывается выдачей сообщения об ошибке, после чего выполнение программы продолжается. При этом попытка разделить на нуль не вызывает внезапную динамическую ошибку (блок обработки прерываний помещен внутрь цикла for). Вместо этого исключение позволяет красиво выйти из ошибочной ситуации и продолжить выполнение программы.

Использование исключений

Теперь вернемся к примерам, с которых мы начали этот раздел. Проблема с прерыванием вложенных циклов (пример 1) и вообще любого другого кода, решается очень просто:

```
for (int i = 0; i < a1.Length; i++)
{
   for (int j = 0; j < a2.Lenght; j++)
   {
      if (!DoAction(a1[i], a2[j]))
      {
         throw new Exception("Ошибка в DoAction!");
      }
   }
}
```

Проблема с закрытием файла (пример 2) будет решаться с помощью блока finally:

```
FileStream f = new FileStream("test.txt", FileMode.Open);
try
{
   DoAction(f);
}
finally
{
   f.Close();
}
```

Но следует помнить, что генерация исключения – очень долгая операция и в реальной программе она может существенно затормозить ее выполнение, поэтому на практике не стоит использовать исключения там, где их можно не использовать. Так в примере, приведенном с построением таблицы значений функции, ситуацию с делением на ноль вполне можно было бы обойти, проверив значение i перед выполнением операции деления.

Аналогично, если нужно проверить является ли строка числом или нет, то вместо следующего кода:

```
bool isNumber = false;
try
{
   int i = int.Parse(s);
   isNumber = true;
}
catch
{ … }
```

Лучше использовать метод int.TryParse(s, out i), который вернет true, если удалось преобразовать строку s в число и записать ее в парамер i. Использовать исключения здесь не нужно, т.к. проверка ввода пользователя на корректность – вполне нормальная практика.

Операторы checked и unchecked

В C# предусмотрено специальное средство, которое связано с генерированием исключений, вызванных переполнением результата в арифметических вычислениях. Например, когда значение арифметического выражения выходит за пределы диапазона, определенного для

типа данных выражения, желательно сообщить о переполнении. Рассмотрим небольшой фрагмент программы:

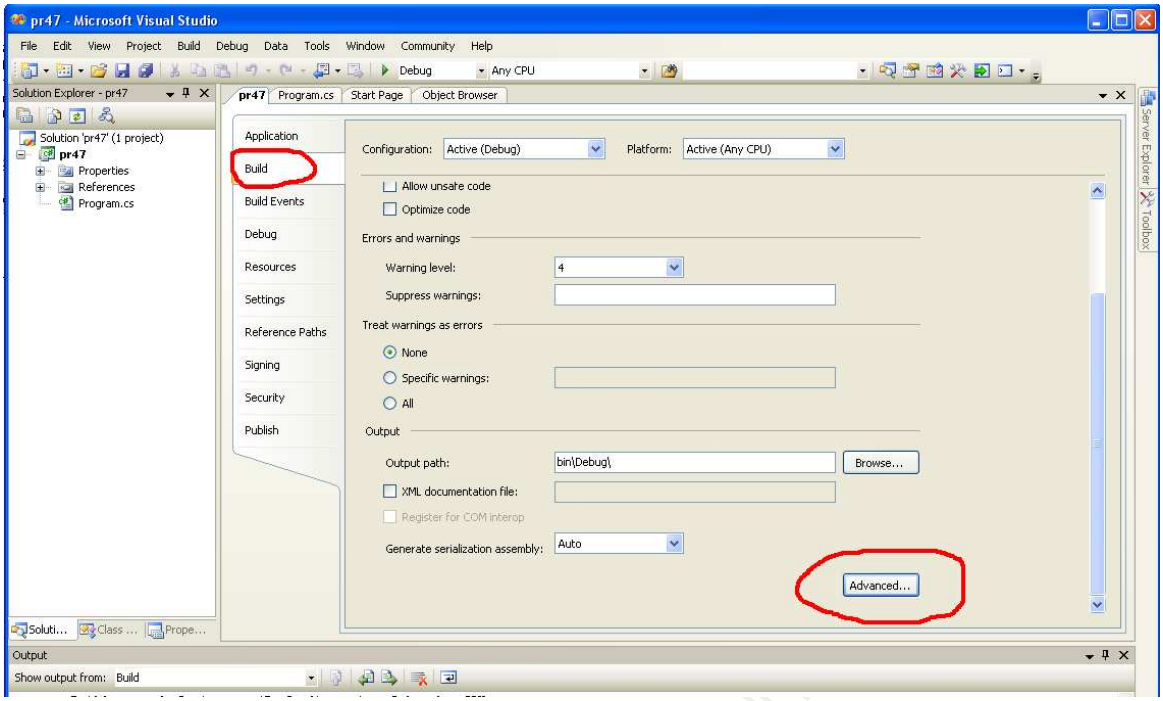
```
static void Main()
{
   byte x = 200;
   byte y = 200;
   byte result = (byte) (x + y);
   Console.WriteLine(result);
}
```

Здесь сумма значений a и b превышает диапазон представления значений типа byte. Следовательно, результат данного выражения не может быть записан в переменную result, которой также задан тип byte.

Для управления подобными исключительными ситуациями в C# используются операторы checked и unchecked. Чтобы указать, что некоторое выражение должно быть проконтролировано на предмет переполнения, используйте ключевое слово checked, а чтобы проигнорировать переполнение – ключевое слово unchecked. В последнем случае результат будет усечен так, чтобы его тип соответствовал типу результата выражения.

По умолчанию проверка переполнения отключена. В результате код выполняется быстро, но программист должен быть уверен, что переполнения не случится или сам предусмотреть его возникновение. Как мы уже упоминали, можно включить проверку переполнения для всего проекта, однако она не всегда нужна. С помощью использования операторов checked и unchecked в C# реализуется механизм гибкого управления проверкой. Включить (или отключить) проверку переполнения можно сразу для всего проекта. Для этого необходимо выполнить следующие действия:

1) щелкнуть правой кнопкой мыши на имени проекта;
2) в выпадающем меню выбрать Properties;
3) в появившемся окне (см. рис.) выбрать слева страницу Build;
4) щелкнуть на кнопке Advanced;
5) в появившемся окошке поставить или убрать галочку напротив Check for arithmetic overflow/underflow property.



Оператор *checked* имеет две формы

1. Операторная форма, предназначенная для проверки конкретного выражения:

```
checked ((тип-выражения) expr)
```

   Здесь *expr* – выражение, значение которого необходимо контролировать. Если значение контролируемого выражения переполнилось, генерируется исключение типа OverflowException.

2. Проверка блока инструкций:

```
checked
{
   // Инструкции, подлежащие проверке.
}
```

Оператор *unchecked* также имеет две формы.

1. Операторная форма, которая позволяет игнорировать переполнение для заданного выражения:

```
unchecked ((тип-выражения) expr)
```

   где *expr* – выражение, которое не проверяется на предмет переполнения. В случае переполнения это выражение усекается.

2. Блочная форма, которая позволяет проигнорировать возможное переполнение в блоке инструкций:

```
unchecked
{
   // Инструкции, для которых переполнение игнорируется.
}
```

Рассмотрим фрагмент программы, которая демонстрирует использование checked и unchecked.

```
static void Main()
{
   byte x = 200;
   byte y = 200;
   try
   {
      byte result = unchecked((byte)(x + y));
      Console.WriteLine("1: {0}", result);
      result = checked((byte)(x + y));
      Console.WriteLine("2: ", result);
   }
   catch (OverflowException error)
   {
      Console.WriteLine(error.Message);
   }
}
```

Результат выполнения программы:

```
1: 144
Переполнение в результате выполнения арифметической операции.
```

Задание

Уберите блок unchecked. Посмотрите, что изменится в выполнении программы и дайте этому объяснение.

В данном примере мы рассмотрели, как использовать checked и unchecked для проверки выражения. Теперь посмотрим, как использовать их для контроля за блоком инструкций.

```
static void Main()
{
   byte n;
   byte i;
   try
   {
      unchecked    //блок без проверки
      {
         n = 1;
         for (i = 1; i < 10; i++)
         {
            n *= i;
         }
         Console.WriteLine("1: {0}", n);
      }
      checked      //блок с проверкой
      {
         n = 1;
         for (i = 1; i < 10; i++)
         {
            n *= i;
         }
         Console.WriteLine("2: ", n);
      }
   }
```

```
        catch (OverflowException error)
        {
           Console.WriteLine(error.Message);
        }
     }
```

Результат выполнения программы:

```
1: 128
Переполнение в результате выполнения арифметической операции.
```

Задание

Поменяйте местами блоки unchecked и checked. Посмотрите, что изменится в выполнении программы и дайте этому объяснение.

Примеры использования исключений

Рассмотрим несколько полезных примеров использования обработчиков исключений.

Пример 1

Как упоминалось выше, для того, чтобы исключение было перехвачено, его тип должен совпадать с типом, заданным в catch-инструкции. В противном случае это исключение не будет перехвачено. Можно перехватывать все исключения, используя catch-инструкцию без параметров. Кроме того, с try-блоком можно связать не одну, а несколько catch-инструкций. В этом случае все catch-инструкции должны перехватывать исключения различного типа. Если вы все же не уверены, что предусмотрели все ситуации, то последней можно добавить catch-инструкцию без параметров.

```
static void Main()
{
   for (int i = 0; i < 5; i++)
   {
      try
      {
         Console.WriteLine("Введите два числа");
         int a = int.Parse(Console.ReadLine());
         int b = int.Parse(Console.ReadLine());
         Console.WriteLine("{0}/{1}={2}", a, b, a/b);
      }
      catch (FormatException)
      {
         Console.WriteLine("Нужно ввести число!");
      }
      catch (DivideByZeroException)
      {
         Console.WriteLine("Делить на нуль нельзя!")
      }
      catch
      {
         Console.WriteLine("Какая-то ошибка");
      }
```

```
        finally
        {
           Console.WriteLine("после try-блока");
        }
     }
  }
```

Задание

Протестируйте данную программу, вводя поочередно следующие значения:

```
a=4, b=2
a=3, b=g
a=d, b=1
a=2, b=0,
a=123456789987654321, b=1
```

Пример 2

Один try-блок можно вложить в другой. Исключение, сгенерированное во внутреннем try-блоке и не перехваченное catch-инструкцией, которая связана с этим try-блоком, передается во внешний try-блок. Например:

```
static void Main()
{
   Console.Write("a=");
   byte a = byte.Parse(Console.ReadLine());
   Console.Write("b=");
   byte b = byte.Parse(Console.ReadLine());
   int f = 1;
   try  //внешний блок-try
   {
      for (byte i = a; i <= b; i++)
      {
         try  //внутренний блок-try
         {
            f = checked((int)(f*i*i));
            Console.WriteLine("y({0})={1}", i, 100000 / (f - 1));
         }
         catch (DivideByZeroException err)
         {
            Console.WriteLine("y({0})={1}", i, err.Message);
         }
      }
   }
   catch (Exception err)
   {
      Console.WriteLine(err.Message);
   }
}
```

Результат работы программы:

```
a=1
b=10
y(1)=Попытка деления на нуль.
y(2)=3333
```

```
y(3)=2857
y(4)=173
y(5)=6
y(6)=0
y(7)=0
y(8)=0
Переполнение в результате выполнения арифметических операций.
```

В данном случае, внутренним try-блоком было предусмотрено обработка только деления на нуль, поэтому возникшее исключение OverflowException() было перехвачено и обработано внешним try-блоком.

Пример 3

Исключение, перехваченное одной catch-инструкцией, можно сгенерировать повторно, чтобы обеспечить возможность его перехвата другой (внешней) catch-инструкцией. Это позволяет нескольким обработчикам получить доступ к исключению.

```
using System;
namespace Example
{
   class Program
   {
      static void GenException ()
      {
         try
         {
            Console.Write("a=");
            int a = int.Parse(Console.ReadLine());
            if (a < 0)
            {
               throw new ArgumentException(); //генерируем исключение
            }
         }
         catch (Exception er)
         {
            Console.WriteLine("genException: {0}", er.Message);
            throw;  //повторно генерируем исключение
         }
      }
      static void Main()
      {
         try
         {
            GenException();
         }
         catch (Exception er)
         {
            Console.WriteLine("Main: {0}",er.Message);
         }
      }
   }
}
```

Результат работы программы:

```
a = -5
genException: Значение не попадает в ожидаемый диапазон.
Main: Значение не попадает в ожидаемый диапазон.

a = пять
genException: Входная строка имела неверный формат.
Main: Входная строка имела неверный формат.
```

Полезные советы

Наиболее хорошо преимущества механизма исключений видны в сложных системах, поэтому полезные советы по работе с исключениями мы дадим на примере некой системы, которая состоит их трех блоков: блок чтения данных, блок обработки данных и блок отображения данных.

В обязанности блока чтения данных входит выполнение всех операций по взаимодействию с внешними источниками данных: базами, дисковыми файлами, сервисами и т.д. Данный блок никак не взаимодействует с пользователем.

Блок обработки данных содержит в себе классы и методы, отвечающие за логику работы программы. Именно здесь происходят все вычисления, выборки и анализ данных. По сути, этот блок является ядром программы, освобожденным от проблем взаимодействия с любыми источниками данных.

Блок отображения данных предназначен для вывода результатов работы программы на экран пользователя, а также для обработки поступающих от пользователя команд.

Подобная организация работы приложения существенно упрощает процедуру его проверки. Например, можно автоматически протестировать логику работы приложения с помощью специальной программы вместо того, чтобы вручную вводить данные для проведения каждого теста. Кроме этого, при необходимости перевести приложение, например, с консольного интерфейса на Winforms точно известно, что нужно менять только один блок.

Для системы, организованной подобным образом, справедливы следующие правила обработки исключений.

Правило 1

Исключения должны обрабатываться в том месте кода, где они могут быть обработаны.

В принципе, исключения, возникающие в блоке чтения данных, могут быть обработаны в любом из блоков, но обрабатывать их в блоке чтения не стоит, так как нужно как-то сообщить пользователю о проблеме, а такой возможности в этом блоке нет.

В блоке обработки данных могут быть обработаны исключения, связанные с получением некорректных данных. Например, если у нас есть договоренность заменять все не числовые данные на -1, то исключение, возникающее при попытке преобразования прочитанных данных в число, можно обработать во втором блоке. Именно там мы можем корректно отреагировать на проблему и создать корректные данные вместо "испорченных". Блок чтения данных такие исключения ловить не может, т.к. он не знает какие данные корректны, а какие нет. В блоке отображения данных ошибку данных ловить уже поздно.

В блоке отображения данных мы можем обрабатывать все другие исключения, такие как отсутствие источника данных, отсутвие прав на его чтение и т.д. Здесь мы вольны обработать все оставшиеся исключения, т.к. можем показать пользователю информацию о возникшей проблеме.

Правило 2

Не глотайте исключения без обработки.

Очень часто программисты обрабатывают исключения просто добавляя блок пустой catch, из-за чего абсолютно все исключения будут пойманы и скрыты. Казалось бы, что тут плохого – программа работает, пользователь доволен. Но представьте себе ситуацию, когда конечный пользователь сообщает, что система перестала отображать данные. Найти причину этой проблемы фактически будет не реально, т.к. все исключения скрыты. Правильный подход заключается в том, чтобы, как минимум, записывать исключения в файл, или в специльные хранилища.

Правило 3

Обходитесь без исключений, если это возможно.

Создание исключений очень ресурсоемкая операция. Поэтому, используйте исключения только там, где они действительно нужны. Используйте методы TryParse(), проверку на ноль и другие способы, если это позволит обойтись без исключений.

Правило 4

Сообщайте информацию о коде с помощью исключений

В больших системах над кодом работают не один и, часто, даже не десять программистов, а значительно больше. То, что знает про код один разработчик, должны знать все другие. Механизм исключений позволяет сообщить дополнительную информацию о способах использования кода и сократить время отладки. Например, есть некоторый метод:

```
public void DoAction(int param)
{  …  }
```

Разработчик этого кода знает, что вызов этого метода с параметром -1 не допустим, т.к. он приведет к зацикливанию. Конечно, он может написать комментарий к этому методу, где укажет эту особенность. Но где гарантия, что другой разработчик, который использует этот код, прочитает это сообщение? Вероятнее всего, в случае зацикливания, он будет отлаживать код, найдет причину зацикливания, найдет, что это происходит именно в этом методе и только потом прочитает комментарий. Время будет потеряно. А ведь ничего не стоило сделать так:

```
public void DoAction(int param)
{
   if (param == -1)
      throw new AgrumentException("Нельзя вызывать DoAction с -1");
   …
}
```

Теперь, если кто-то попытается вызвать этот метод с параметром -1, он точно узнает, что делать этого нельзя. Две строчки кода и экономия нескольких часов времени!

Мы рекомендуем всегда проверять значения параметров всех public методов. Это позволит избежать затрат времени ваших коллег.

Приведем еще несколько примеров таких проверок.

```
public void DoAction(object param)
{
   if (param == null)
      throw new AgrumentNullException("param");
   …
}

public void DoAction(string param)
{
   if (string.IsNullOrEmpty(param))
      throw new AgrumentNullException("param");
   …
}
```

Правило 5

Используйте исключения, а не коды ошибок

Посмотрим, как работают коды ошибок. Предположим, что метод загрузки данных из файла в блоке чтения данных возвращает коды ошибок, описанные в виде специального перечисления:

```
ReadCodeError Read(out int data)
{ … }
```

Само перечисление может быть таким:

```
enum ReadCodeError
{
   NoData = -1, // нет данных
   NoPermission = -2, // нет прав на чтение
   ConnectionError = -3, // проблема подключения к источнику данных
   NotIntValue = -4, // прочитанные данные не число
}
```

В блоке обработки данных этот метод вызывается и, соответственно, возвращаемый результат придется анализировать именно там. Например, для кода NotIntValue значение нужно заменить на -1, или провести вычисления другим путем, отличным от стандартного. А вот все другие коды придется обрабатывать по-другому: критические коды ошибок должны прервать все вычисления и передать управление блоку отображения данных, который должен уведомить пользователя о возникшей проблеме. Такая архитектура имеет множество недостатков.

1. Прервать выполнение методов расчета бывает не просто; мы уже обсуждали этот вопрос в самом начале главы.
2. Необходимо делать ветвление по некоторым кодам ошибок, а по другим не делать. Более того, если блок чтения когда-либо изменится и появятся новые коды, нам придется менять и блок обработки данных тоже.
3. Что сообщать пользователю? Значение кодов определены в блоке чтения данных, а отображать сообщение нужно в блоке отображения данных. Сообщить пользователю только код ошибки – это не очень хорошо. Пользователю придется искать документацию, искать, что означает этот код и т.д. Можно сделать метод, который по коду ошибки вернет строку, расшифровывающую проблему, но тогда окажется что либо эту строку придется передавать прямо с блок чтения данных, либо метод преобразования кода в

сообщение придется делать в блоке отображения данных, что опять снижает возможность расширения блока чтения данных.

4. Нет возможности передать дополнительную информацию о возникшей проблеме. Все, что мы имеем в блоке отображения – это код.

Исключения решают эти проблемы очень легко. Блок чтения данных при возникновении проблем генерирует исключение, содержащее и сообщение, и код ошибки, и любую другую информацию. Блок обработки данных обрабатывает только те исключения, которые ему нужны и не обрабатывает те, которые он обрабатывать не умеет. Необработанные исключения прерывают работу этого блока. Блок отображения данных отображает пользователю подробное сообщение об ошибке.

Проблему расширяемости системы мы рассмотрим в следующем совете.

Правило 6

Используйте иерархию исключений

Предположим, что первый модуль генерирует несколько исключений:

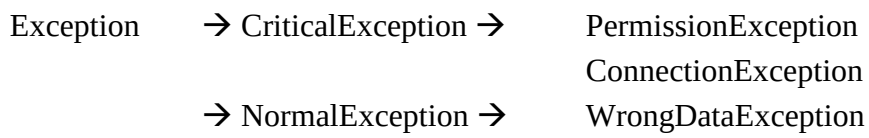
- DataException – нет данных в источнике данных;
- PermissionException – нет прав на чтение данных;
- ConnectionException – ошибка подключения к источнику данных;
- WrongDataException – ошибка в данных.

Мы знаем, что часть этих исключений проводит к остановке работы блока обработки данных и передает управление блоку отображения, который сообщает пользователю об остановке работы системы. Кроме того, часть ошибок не так критична, и блок обработки данных вполне может с ними справится, скорректировав прочитанные данные. Может быть, третья часть исключений вообще не страшна и вполне достаточно просто записать в лог-файл об их возникновении.

С учетом сказанного, код блока отображения может иметь примерно такой вид:

```
try
{
   DoAction();
}
catch (DataException dataException)
{ ... }
catch (PermissionException permissionException)
{ ... }
catch (WrongDataException wrongDataException)
{ ... }
```

Проблема здесь в том, что по мере развития блока чтения, вполне могут появиться новые исключения, о которых блок обработки не знает. Поэтому придется менять код и блока чтения и блока обработки, что не очень хорошо. Исправить ситуацию можно, используя технологии ООП. Как мы знаем, все исключения являются наследниками базового класса Exception. Мы можем создать два собственных класса исключений, а уже от них наследовать все другие:



Тогда код блока обработки можно сделать таким:

```
try
{
   DoAction();
}
catch (CriticalException critical)
{
   throw;
}
catch (NormalException normal)
{
}
```

Теперь при добавлении новых исключений в блоке чтения, он сам сможет решить нужно ли прерывать работу блока обработки, или нет. Конечно этот рецепт не на все случаи и могут появиться исключения, которые потребуют специальной обработки в блоке обработки данных, но минимизировать часть затрат это позволяет.