|  |
| --- |
| , RD Dep. |
| Конспект лекций и раздаточный материал  NET.C#.07 Обработка исключений |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| REVISION HISTORY | | | | | |
| Ver. | Description of Change | Author | Date | Approved | |
| Name | Effective Date |
| 1.0 | Initial version | Анжелика Кравчук |  |  |  |
| 1.1 | Review and corrections. | Владимир Тихон |  |  |  |

Contents

[1. Урок 1. Обработка исключений 3](#_Toc301367320)

[1.1. Что такое исключение? 3](#_Toc301367321)

[1.2. Использование блока try/catch 5](#_Toc301367322)

[1.3. Использование свойств исключений 8](#_Toc301367323)

[1.4. Использование блока finally 9](#_Toc301367324)

[1.5. Использование ключевых слов checked и unchecked 11](#_Toc301367325)

[2. Урок 2. Возникновение исключений 13](#_Toc301367326)

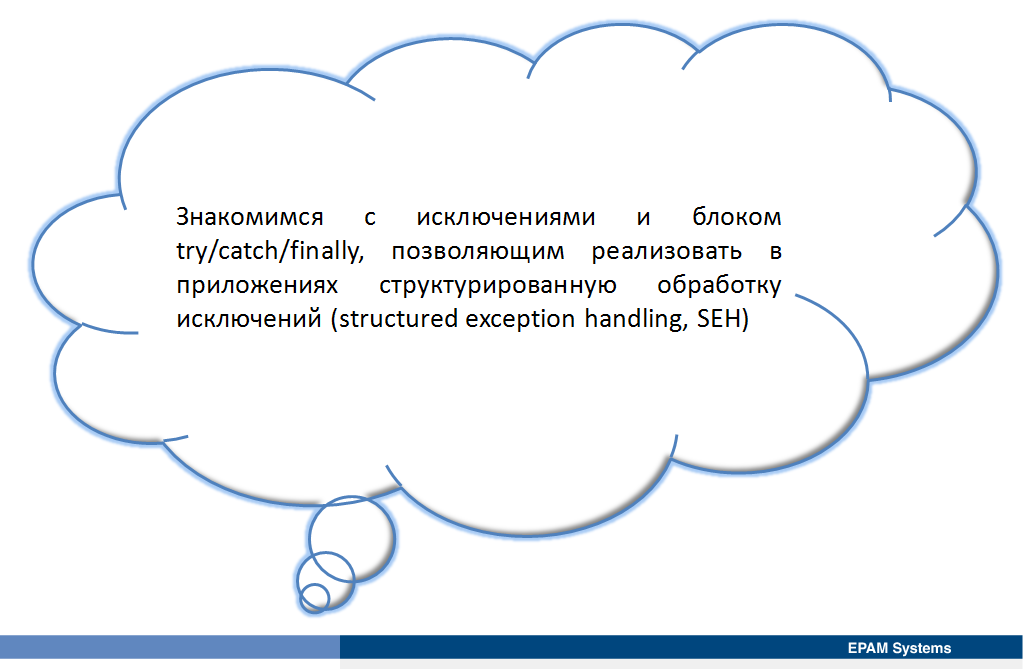
[2.1. Создание объекта исключения 13](#_Toc301367327)

[2.2. Генерация исключений 15](#_Toc301367328)

[2.3. Рекомендации по обработке и генерации исключений 17](#_Toc301367329)

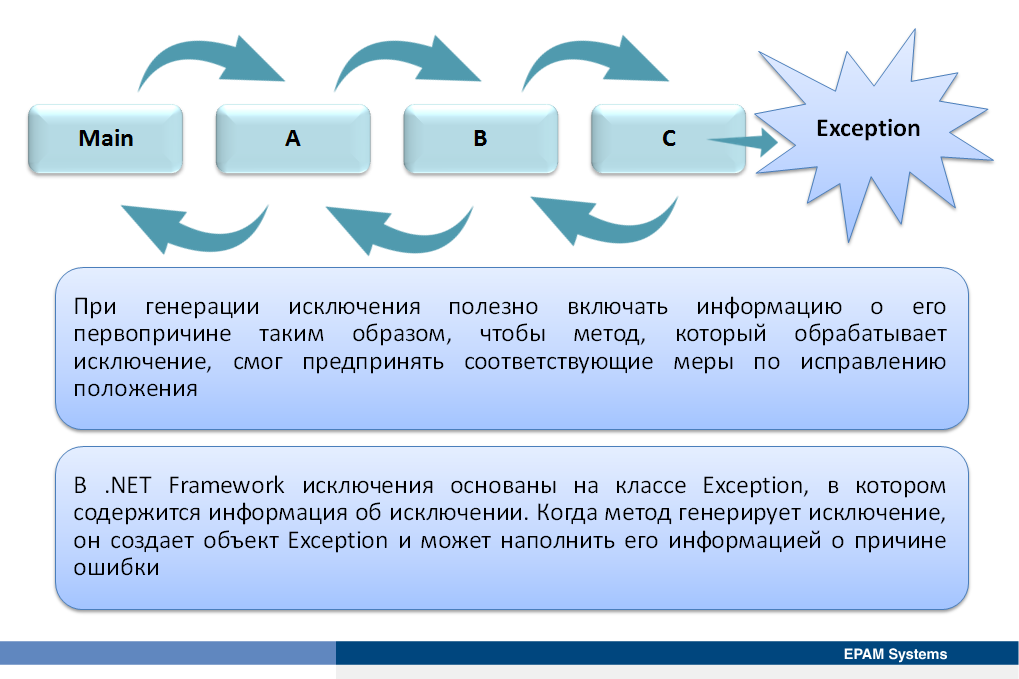
[2.4. Демонстрация: Возникновение и обработка исключений 18](#_Toc301367330)

# Урок 1. Обработка исключений



Приложения могут функционировать ожидаемо во время разработки с ограниченным использованием и контролируемым вводом. Однако, когда приложение разворачивается в реальной среде с учетом ограничений таких, например, как динамически изменяющаяся в больших объемах информация, могут появляться ошибки. Для управления пользователями, а также чтобы обеспечить возможность работы приложения при возникновении исключений, их нужно обрабатывать. Этот урок знакомит c блоком try/catch/finally, позволяющим реализовать в приложениях структурированную обработку исключений (structured exception handling, SEH).

## Что такое исключение?



Во время работы приложения многие вещи могут пойти не так, как ожидалось, не только из-за ошибок в логике, но и в силу того, что приложение обычно зависит от многих факторов вне области видимости приложения, например, таких как наличие файлов в файловой системе или подключения к базам данных. При разработке приложения следует предусмотреть возможность восстановления приложения при возникновении таких проблем.

При написании методов для обспечения их правильной работы часто используется проверка возвращаемых из методов значений. Однако, существуют проблемы, связанные с таким подходом, поскольку, во-первых, не все методы возвращают значение. Во-вторых, часто недостаточно знать о сбое в методе, нужно знать о причине его сбоя. И, наконец, такой подход не распространяется на непредвиденные ошибки, такие, например, как нехватка памяти.

Многие прежние системы использовали концепции глобального объекта ошибки. Согласно такому подходу, когда кусок кода вызвал ошибку, надо было занести данные, указывающие на причину ошибки, в этот объект, а затем вернуться к вызвавшему ее коду. Вызывающий код должен был изучить объект ошибки и определить, как с ним обращаться. Указанный подход не являлся эффективным, поскольку можно было просто забыть про обработку ошибки.

В .NET Framework оказались полезными исключения. Исключением является показателем ошибки или исключительного состояния приложения. Метод может сгенерировать исключение, когда обнаруживает, что случилось что-то неожиданное, например, осуществляется попытка открыть несуществующий файл. Когда метод генерирует исключение, вызывающий код должен быть готов перехватить и обработать это исключение. Если вызывающий код не в состоянии обработать исключение, он прерывается и исключение передается по стеку вызовов до раздела кода, берущего на себя ответственность за обработку исключений. После завершения логики обработки исключения выполнение продолжается в этом разделе кода.

В качестве примера, можно расмотреть следющую цепочку вызовов методов: метод A вызывает метод B, в рамках выполнения которого вызывается метод C (Рис. 1.). При выполнении метода C генерируется исключение, которое приводит к прерыванию метода C. Метод C не готов обработать сгенерированное исключение и оно передается вверх по стеку вызовов в метод B. Если метод B не готов обработать исключение, он также прерывается, и то же исключение передается в метод A. Если метод A обрабатывает исключение, выполнение продолжается в методе A после выполнения логики обработки исключений. Если метод A не готов обработать исключение, исключение будет распространяться вверх по стеку вызовов в метод, который вызвал метод A. Если существует метод Main, который также не готов обработать исключение, приложение сообщает о необработанных исключениях пользователю, а затем завершается.

**Exception**

**Main**

**C**

**B**

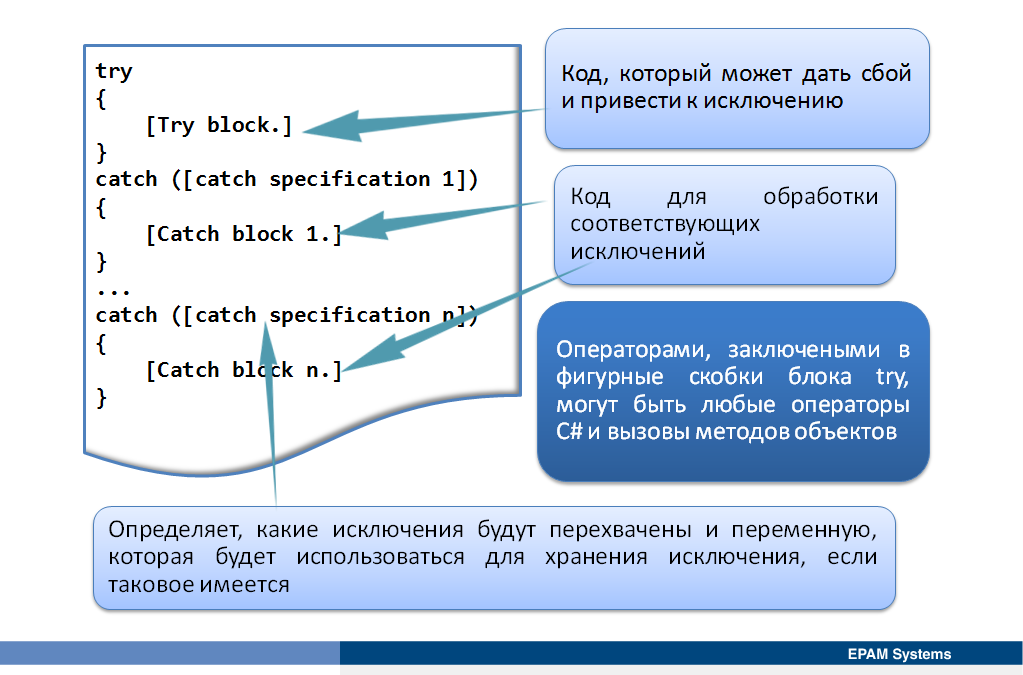
**A**

Рис. 1.

В предыдущем сценарии, метод A «не знает», что метод B вызывает метод C. Следовательно, при прерывании метода B, в силу того, что он не может обработать исключение, для метода A, методом, вызвавшим исключение, будет метод B. Поэтому, при генерации исключения полезно включать информацию о его первопричине таким образом, чтобы метод, который обрабатывает исключение, смог предпринять соответствующие меры по исправлению положения.

В .NET Framework исключения основаны на классе Exception, в котором содержится информация об исключении. Когда метод генерирует исключение, он создает объект Exception и может наполнить его информацией о причине ошибки. Этот объект передается обабытывающему исключение коду, который может его использовать, чтобы определить лучший способ справиться с исключением.

## Использование блока try/catch



Блок try/catch является ключем программной конструкции для структурированной обработки исключений. При его использовании следует обернуть код, который может дать сбой и привести к исключению, блоком try и добавить один или несколько блоков catch для обработки любых исключений, которые могут возникнуть. Блок try/catch имеет следующий интаксис:

**try**

**{**

**[Try block.]**

**}**

**catch ([catch specification 1])**

**{**

**[Catch block 1.]**

**}**

**...**

**catch ([catch specification n])**

**{**

**[Catch block n.]**

**}**

Операторами, заключеными в фигурные скобки блока try, могут быть любые операторы C# и вызовы методов объектов. Если любой из этих операторов становится причиной исключения, среда выполнения проверяет текущий оператор и устанавливает, находится ли он в блоке try. Если это так, то последовательно проверяются любые блоки catch, связанные с данным блоком try, и устанавливается возможность перехвата ими исключения с последующим выполнением операторов соответствующего блока catch. После завершения кода блока catch продолжается выполнение первого после блока try/catch оператора.

Если оператор, сгенерировавший исключение, расположен вне блока try или если блок try, в котором он расположен, не имеет соответствующего блока catch, среда выполнения проверяет вызывающий метод на наличие оператора try и блоков catch. Выполнение продолжается по стеку вызовов в поисках совместимого блока catch. После того, как блок catch найден и выполнен, управление передается следующему после этого блока catch оператору. Не перехваченные исключения обрабатываются универсальным обработчиком исключений, предоставляемым системой, который открывает диалоговое окно с соответствующим сообщением.

Спецификация каждого блока catch [catch specification X] определяет, какие исключения будут перехвачены и также переменную, которая будет использоваться для хранения исключения, если таковое имеется. Указать блоки catch можно для различных типов исключений. .NET Framework определяет различные типы исключений для многих общих исключений, которые могут возникнуть. Например, некоторые методы в пространстве имен System.IO при работе с файлами генерируют исключение FileNotFoundException, когда приложение пытается получить доступ к несуществующему файлу. Кроме того, общеязыковая среда выполнения (CLR) сама выбрасывает исключение DivideByZeroException при попытке выполнить числовое деление на ноль.

Наиболее общим видом блока catch является блок, не имеющий спецификации catch, а, следовательно, перехватывающий исключение любого типа, как показано в следующем примере

try

{

// Try block.

}

catch

{

// Catch block.

}

В этом примере кода, исключение, которое генерируется в блоке try, перехватывается блоком catch. Однако, причина возникновения исключения в блоке catch остается неизвестной. Для доступа к этой информации, необходимо в спецификации catch использовать переменную, как показано ниже.

try

{

// Try block.

}

catch (Exception ex)

{

// Catch block, can access exception in ex.

}

Информация о сгенерированном кодом исключении передается этой переменной. Приведенный выше код будет перехватывать исключения любого типа. Тип Exception часто используется для того, чтобы поймать все исключения, которые не были обработаны иным образом. В следующем примере, если код в блоке try является причиной исключения DivideByZeroException, управление передается соответствующему блоку catch. Если возникает какой-либо другой тип исключения, работает код в блоке catch для типа Exception.

try

{

// Try block.

}

catch (DivideByZeroException ex)

{

// Catch block, can access DivideByZeroException exception in ex.

}

catch (Exception ex)

{

// Catch block, can access exception in ex.

}

Блоки catch следует располагать в правильном порядке. Выполнение нескольких блоков catch осуществляется сверху вниз, при этом, для каждого сгенерированного в блоке try исключения выполняется только один блок catch. Выполняется первый блок catch, соответствующий точному типу или типу базового класса сгенерированного исключения. При этом, если блок catch с соответствующей спецификацией отсутствует, будет выполнен блок catch без спецификации (если таковой имеется). Например, .NET Framework предлагает исключение типа ArithmeticException, которое можно использовать, чтобы указать ошибку при оценке арифметического выражения. Тип DivideByZeroException является спецификацией ArithmeticException (в .NET Framework также определены два типа OverflowException и NotFiniteNumberException исключения ArithmeticException). Если перехватывать исключение DivideByZeroException, то лишь оно и будет поймано. Однако, если перехватывать исключение ArithmeticException, блок catch будет перехватывать исключения DivideByZeroException, OverflowException и NotFiniteNumberException. Поэтому, если сущетвует несколько блоков catch, следует убедиться, что блоки с более конкретной спецификацией размещены перед менее специфичными блоками. Если используется тип Exception, он должен быть завершающим блоком catch.

В следующем примере блок try/catch содержит код, который пытается прочитать данные из файла. Если генерируется исключение типа IOException, выполняется соответствующий блок catch, если генерируется какой-либо другой тип исключения, выполняется общий блок catch.

StreamReader reader = null;

try

{

string fileName = GetFileName();

reader = new StreamReader(fileName);

string savedData = reader.ReadToEnd();

}

catch (IOException ioex)

{

// Handle the IO exception.

}

catch (Exception ex)

{

// Handle all other types of exceptions.

}

Блок try/catch является программной конструкцией, как и любой другой оператор C#. Следовательно, можно вкладывать блоки try/catch друг в друга, и блок try может содержать блок try/catch, как показано в следующем примере.

...

try

{

// Outer try block.

...

try

{

// Nested try block

}

catch (FileNotFoundException ex)

{

// Catch block for nested try block

}

...

// Outer try block continued

}

catch (DivideByZeroException ex)

{

// Catch block, can access DivideByZeroException exception in ex.

}

catch (Exception ex)

{

// Catch block, can access exception in ex.

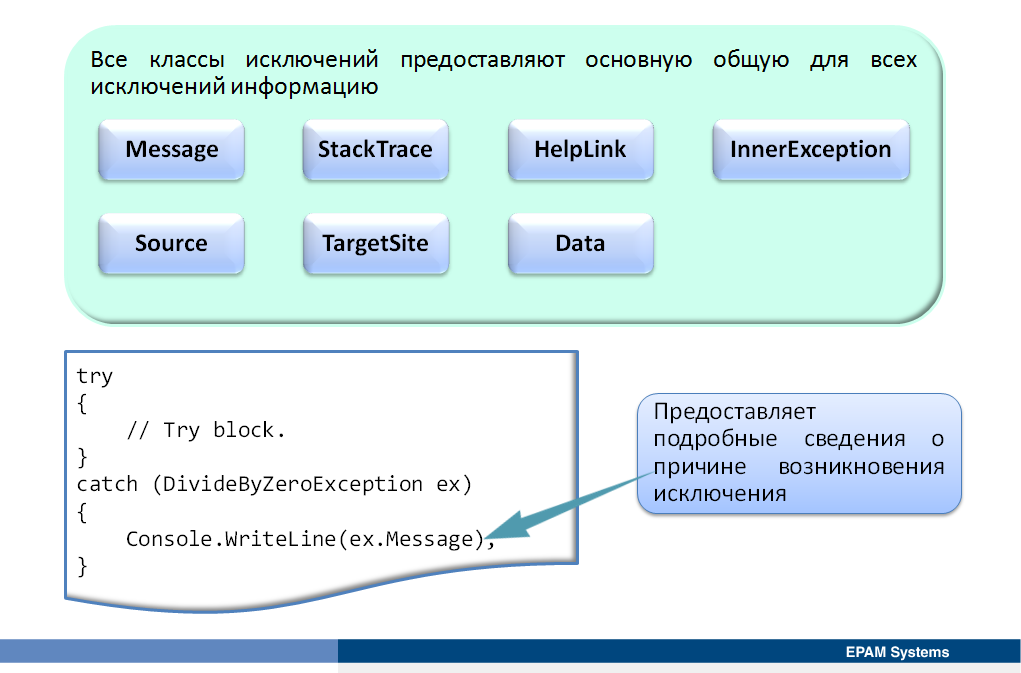
}

...

В приведенном выше примере, если во вложенном блоке try генерируется исключение FileNotFoundException, то работает вложенный блок catch, после завершения которого, продолжается выполнение первого оператора после вложенного блока catch. Если во вложенном блоке try происходит любой другой тип исключения, включая DivideByZeroException, исключение распространяется на внешний блок try, где оно перехватывается либо блоком catch для типа DivideByZeroException, либо блоком catch для типа Exception. Затем продолжается выполнение первого оператора после внешнего блока try/catch.

http://go.microsoft.com/fwlink/?LinkId=192911

## Использование свойств исключений



Все классы исключений предоставляют основную общую для всех исключений информацию, однако они могут также предоставлять дополнительную информацию, которая специфична для указанного типа исключения. Свойства, являющиеся общими для всех исключений приведены в следующей таблице.

|  |  |
| --- | --- |
| **Свойство** | **Описание** |
| **Message** | Предоставляет подробные сведения о причине возникновения исключения. Свойство Message выводится на том языке, который указан в свойстве [Thread.CurrentUICulture](http://msdn.microsoft.com/ru-ru/library/system.threading.thread.currentuiculture.aspx) для потока, который создает исключение. |
| **Source** | Содержит строку, которая указывает на объект или приложение, которое вызвало ошибку. Если свойство не задано, возвращается имя сборки, в которой возникло исключение. |
| **StackTrace** | Содержит трассировку стека, которую можно использовать для определения места возникновения ошибки. Трассировка стека включает имя файла источника и, при наличии отладочной информации, номер программной строки. |
| **TargetSite** | Свойство является строкой, содержащей имя метода, сгенерировавшего исключение. Если метод, выбросивший исключение, недоступен и трассировка стека не является указателем NULL, свойство TargetSite извлекает метод из трассировки стека. Если трассировка стека является нулевой ссылкой, то TargetSite возвращает нулевую ссылку. |
| **InnerException** | Свойство может использоваться для создания и сохранения серии исключений во время обработки исключений. Это свойство можно использовать для создания нового исключения, содержащего ранее перехваченные исключения. Исходное исключение может быть захвачено вторым исключением в свойстве **InnerException**, что позволяет коду, обрабатывающему второе исключение, проверять дополнительные данные. Например, существует метод, выполняющий чтение файла и форматирование данных, который пытается выполнить чтение из файла, но при этом гененрируется исключение FileException. Данный метод перехватывает FileException и создает исключение BadFormatException. В этом случае исключение FileException может быть сохранено в свойстве **InnerException** исключения BadFormatException. |
| **HelpLink** | Свойство может содержать URL к файлу справки, предоставляющему подробные сведения о причине возникновения исключения. |
| **Data** | Свойство является объектом типа IDictionary и может содержать произвольные данные в парах "ключ-значение", которые можно использовать для хранения дополнительной информации об ошибке. |

В следующем примере показан вывод на экран сообщения при генерации исключения DivideByZeroException.

try

{

// Try block.

}

catch (DivideByZeroException ex)

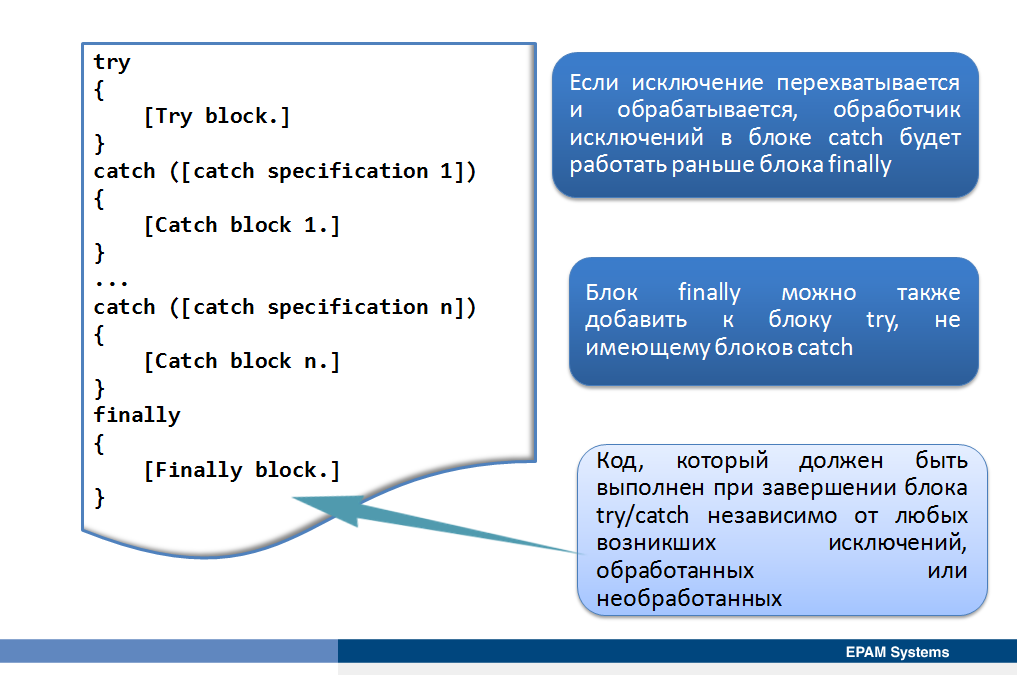
{

Console.WriteLine(ex.Message);

}

<http://go.microsoft.com/fwlink/?LinkId=192912>

## Использование блока finally



Некоторые методы могут содержать крайне необходимый код, который всегда должен быть выполнен, даже если возникает необработанное исключение. Например, методу до его завершения может потребоваться обеспечить закрытие файла, в который осуществлялась запись, или осободить некоторые другие ресурсы. Указанную проблему позволяет решить блок finally. Он указывается после любого блока catch в блоке try/catch. В него помещается код, который должен быть выполнен при завершении блока try/catch независимо от любых возникших исключений, обработанных или необработанных. При этом, если исключение перехватывается и обрабатывается, обработчик исключений в блоке catch будет работать раньше блока finally. Блок finally можно также добавить к блоку try, не имеющему блоков catch. В этом случае, все исключения являются необработанными, при этом блок finally будет работать всегда. Синтаксис использования блока finally показан в следующем примере кода.

**try**

**{**

**[Try block.]**

**}**

**catch ([catch specification 1])**

**{**

**[Catch block 1.]**

**}**

**...**

**catch ([catch specification n])**

**{**

**[Catch block n.]**

**}**

**finally**

**{**

**[Finally block.]**

**}**

При использовании блоков finally, передача управления является более сложной, чем в блоках try/catch, и осуществляется следующим образом:

1. Выполняется блок **try**.
2. Если сгенерировано исключение:

* Если есть соответствующий исключению блок **catch**:

1. Выполняется блок **catch**, который соответствует исключению.
2. Выполняется блок finally.

* Если есть соответствующий исключению блок catch, и этот блок catch сам является причиной исключения:

1. Выполняется блок catch, который соответствует оригиналу исключения.
2. Выполняется блок finally.
3. Исключение, вызванное обработчиком **catch** распространяется на любой внешний блок try/catch, или вызов метода, если такого блока не существует.

* Если нет соответствующего исключению блока catch:

1. Выполняется блок finally.
2. Исключение распространяется на любой внешний блок try/catch, или вызов метода, если такого блока не существует.
3. Если исключение не сгенерировано, выполняется блокfinally.

Таким образом, следует вывод, что блок finally выполняется всегда. Блок finally позволяет все в кодe привести в порядок после сгенерированого исключения и до любого кода с исключением. В следующем примере показана реализация простого блока try/catch/finally.

try

{

OpenFile("MyFile"); // Open a file

WriteToFile(...); // Write some data to the file

}

catch (IOException ex)

{

Console.WriteLine(ex.Message);

}

finally

{

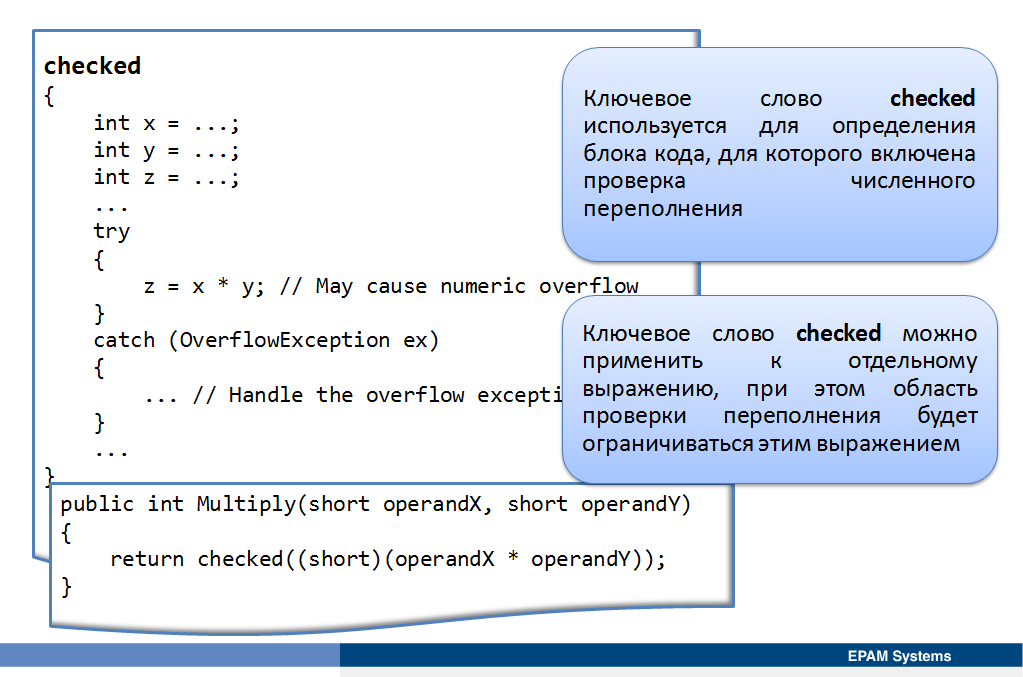
CloseFile("MyFile"); // Close the file

}

Код в блоке try приведенного выше примера вызывает методы, которые открывают файл и записывают в него некоторые данные. Если в блоке try генерируется исключение IOException, блок catch отображает его детали. Наконец блок finally вызывает метод CloseFile для закрытия файла. При этом, код блока finally будет работать всегда, а файл всегда будет закрыт, независимо от того, возникнут исключения или нет.

http://go.microsoft.com/fwlink/?LinkId=192913.

## Использование ключевых слов checked и unchecked



Целочисленная арифметика является соствной частью большинства приложений, в которых часто можно встретить следующий код.

for (int i = 0; i < 10; i++)

{

...

}

Все числовые переменные, в том числе целочисленные, имеют максимальное значение. Если увеличить целое число, которое уже имеет максимальное значение, результатом будет числовое переполнение. Тем не менее, целочисленная арифметика настолько распространена, что проверка численного переполнение после каждой целочисленной операции может серьезно повлиять на производительность приложения. Поэтому в приложениях Microsoft Visual C# при работе с целыми числами проверка переполнения по умолчанию отключена[[1]](#footnote-1). В этих приложениях, существует риск того, что числовое переполнение может привести к неправильным результатам. Если существует фрагмент кода, который может привести к числовому переполнению, можно восстановить контроль над ним с помощью ключевого слова checked. Включить и отключить проверку переполнения можно также для всего приложения с помощью ключевых слов checked и unchecked соответственно.

Включить и отключить проверку переполнения в Microsoft Visual Studio 2010 можно, установив свойства проекта. Для этого в обозревателе решений нужно нажать YourProject (где YourProject является именем проекта). В меню Project выбрать YourProjectProperties. В диалоговом окне Project Properties выбрать вкладку Build. Нажать кнопку Advanced в правом нижнем углу страницы. В диалоговом окне Advanced Build Settings установитить или снять флажок Check for arithmetic overflow/underflow (

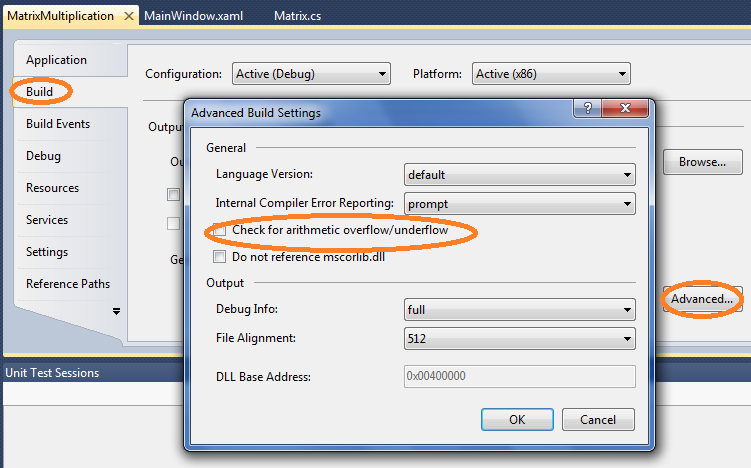


Рис. 2.

Ключевое слово checked используется для определения блока кода, заключенного в фигурные скобки, для которого включена проверка численного переполнения. Если в этом блоке происходит переполнение, операторы его вызвавшие приведут к возникновению исключения OverflowException. В следующем примере кода показано, как включить проверку переполнения и перехватить исключение OverflowException.

checked

{

int x = ...;

int y = ...;

int z = ...;

...

try

{

z = x \* y; // May cause numeric overflow

}

catch (OverflowException ex)

{

... // Handle the overflow exception

}

...

}

Ключевое слово checked можно также применить к отдельному выражению, при этом область проверки переполнения будет ограничиваться этим выражением.

...

public int Multiply(short operandX, short operandY)

{

return checked((short)(operandX \* operandY));

}

...

Если проверка переполнения включена для всего приложения, можно использовать ключевое слово unchecked, чтобы подавить проверку переполнения в блоках или отдельных выражениях. В этом случае синтаксис использования ключевого слова unchecked одинаков как для блока, так и для выражения.

unchecked

{

int1 = 2147483647 + 10;

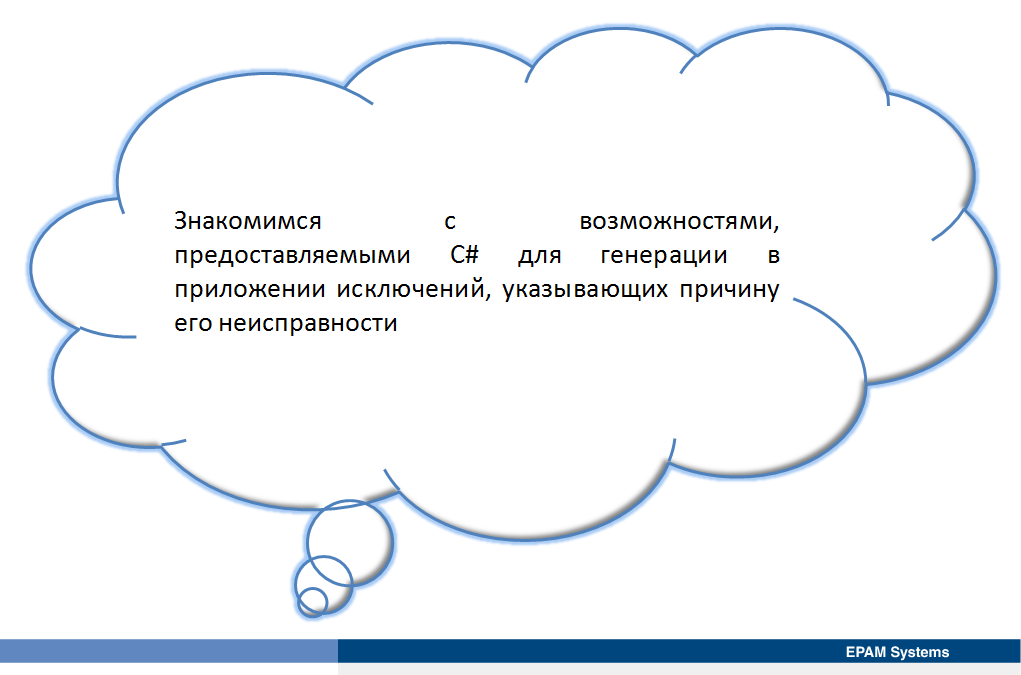
}

...

int1 = unchecked(ConstantMax + 10);

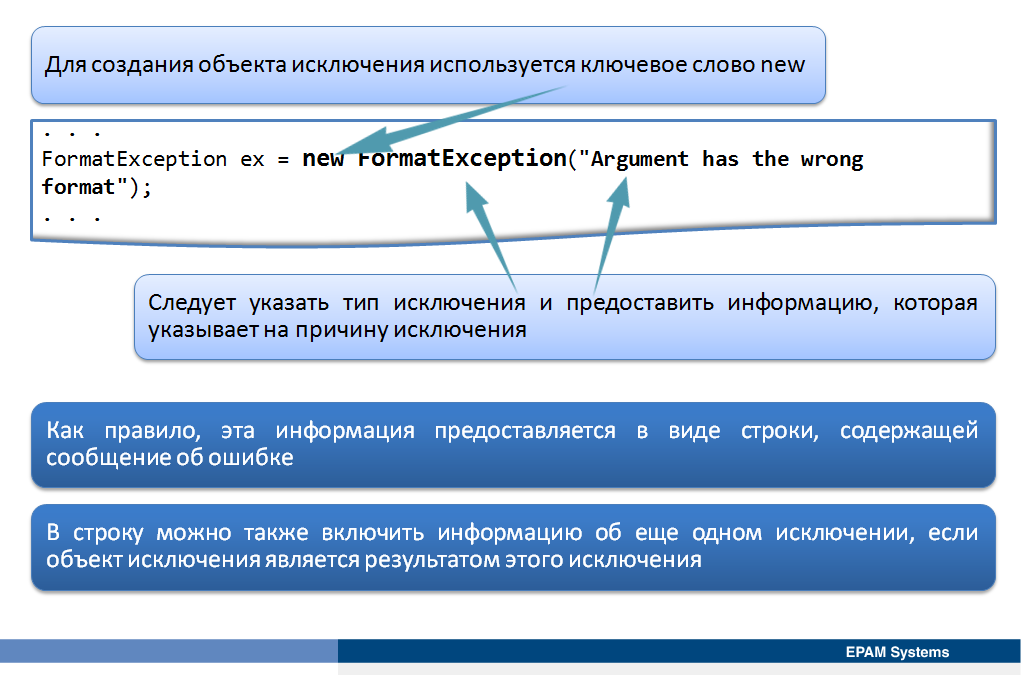
http://go.microsoft.com/fwlink/?LinkId=192914

# Урок 2. Возникновение исключений



Использование блоков try/catch позволяет приложению перехватывать и обрабатывать исключения. Эти исключения могут быть сгенерированы CLR, если приложение пытается выполнить недопустимые операции, такие как попытка деления на ноль или попытка получения доступа к файлу, на который пользователь, запустивший приложение, не имеет разрешения. Однако, приложение может также определять свои ошибочные условия, например, такие, как недопустимое сочетание аргументов, переданных в качестве параметров в метод. В этом случае, для самого приложения полезно генерировать исключение, указывающее причину его неисправности.

## Создание объекта исключения



В методы можно добавить код, который обнаруживает неправильные условия, а затем генерирует соответствующее исключение, чтобы указать характер ошибки вызывающему метод коду. Тогда вызывающий код сможет перехватить и обработать исключение.

.NET Framework предоставляет широкий спектр встроенных типов исключений, которые наследуются от класса Exception[[2]](#footnote-2). Каждый тип исключения предназначен для указания конкретной классификации исключения. Например, тип исключения FileNotFoundException указывает, что была предпринята попытка открыть несуществующий файл, а DivideByZeroException используется для обозначения попытки деления на ноль в математическом выражении. Не существует ограничений, которые не позволяли бы генерировать в методе произвольный тип исключения, однако рекомендуется генерировать исключения типа, подходящего обнаруженному ошибочному условию.

В следующей таблице перечислены некоторые из наиболее часто используемых типов исключений.

|  |  |
| --- | --- |
| **Исключение** | **Описание** |
| **ArgumentException** | Исключение создается, если вызывающий код указывает аргумент для метода, не соответствующий требованиям метода. Этот тип исключения можно использовать, чтобы указать обобщенные ошибки с аргументами (можно использовать типы **ArgumentOutOfRangeException** и **ArgumentNullException**, чтобы указать более конкретные ошибки). |
| **FormatException** | Исключение создается, если вызывающий код специфицирует аргумент, который содержит данные, не имеющие требуемого формата. |
| **NotImplementedException** | Исключение создается, чтобы указать, что код еще не имплементирован в методе. Это исключение, в основном, полезно во время разработки кода, когда есть определенный метод, но не написан код тела метода. |
| **NotSupportedException** | Исключение создается, если вызывающий код пытается выполнить неподдерживаемые операции с помощью вызываемого метода, например, такие как указание аргументов, определяющих, что абонент хочет писать в файл только для чтения. |
| **ArithmeticException** | Основной класс исключений, происходящих при выполнении арифметических операций, таких как DivideByZeroException и OverflowException. |
| **ArrayTypeMismatchException** | Исключение создается, когда массив не может хранить данный элемент, поскольку фактический тип элемента несовместим с фактическим типом массива. |
| **DivideByZeroException** | Исключение создается при попытке разделить целое число на ноль. |
| **IndexOutOfRangeException** | Исключение создается при попытке индексирования массива, когда индекс меньше нуля или выходит за границы массива. |
| **InvalidCastException** | Исключение создается, когда во время выполнения происходит сбой явного преобразования из основного типа в интерфейс или производный тип. |
| **NullReferenceException** | Исключение создается при попытке ссылки на объект, значение которого равно null. |
| **OutOfMemoryException** | Исключение создается при неудачной попытке выделения памяти с помощью оператора new. Это означает, что память, доступная для среды выполнения, уже исчерпана. |
| **OverflowException** | Исключение создается при переполнении арифметической операции в контексте checked. |
| **StackOverflowException** | Исключение создается, когда стек выполнения переполнен за счет слишком большого количества вызовов отложенных методов; обычно является признаком очень глубокой или бесконечной рекурсии. |
| **TypeInitializationException** | Исключение создается, когда статический конструктор создает исключение, и не существует ни одного совместимого предложения catch для его захвата. |

Для создания объекта исключения используется ключевое слово new. При этом следует указать тип исключения и предоставить информацию, которая указывает на причину исключения. Как правило, эта информация предоставляется в виде строки, содержащей сообщение об ошибке; в строку можно также включить информацию об еще одном исключении, если объект исключения является результатом этого исключения. Текст сообщения об ошибке доступен в свойстве исключения Message в блоке catch, обрабатывающем исключение. Если в исключение включить еще один объект исключения, подробную информацию о нем можно получить в блоке catch, обрабатывающем исключение, в свойстве InnerException. В следующем примере показаны два примера создания объекта FormatException.

// Example 1

// Create a FormatException containing an error message.

FormatException ex = new FormatException("Argument has the wrong format");

...

// Example 2

try

{

...// Statements that might cause an exception if data is in the wrong format

}

catch (Exception e)

{

// Create a FormatException containing an error message

// and a reference to the original exception.

FormatException ex = new FormatException("Argument has the wrong format", e);

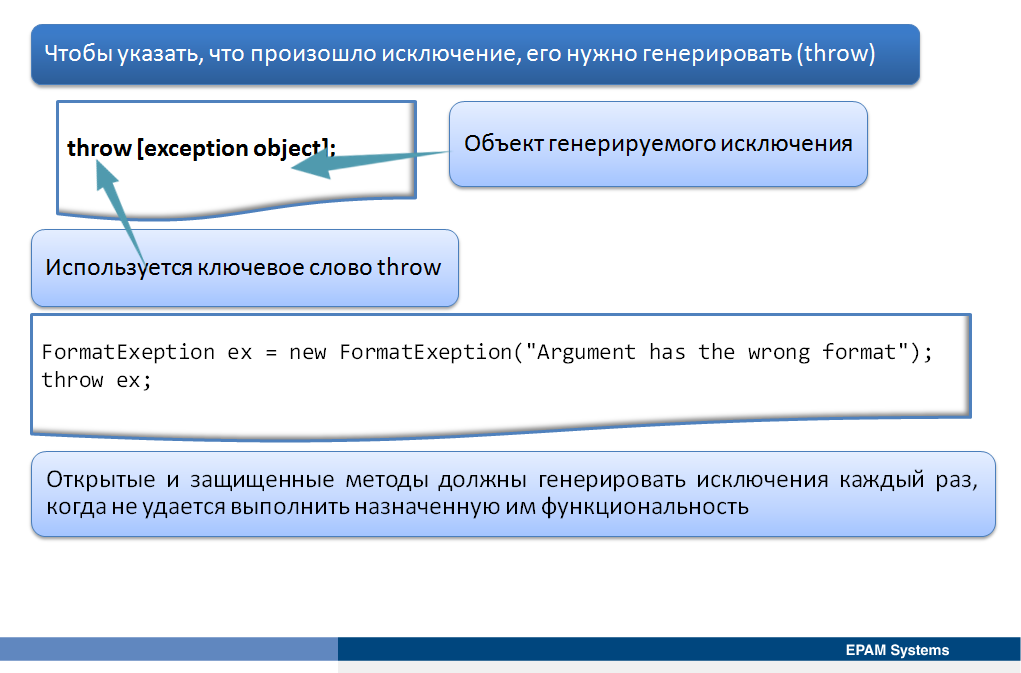
...

}

Различные классы исключений обеспечены конструкторами, которые принимают дополнительные параметры. В следующем примере кода показано исключение ArgumentOutOfRangeException. Этот тип исключения имеет конструктор, который может принимать два строковых параметра. Первым параметром является имя параметра, находящегося за пределами диапазона, а вторым – текст сообщения об ошибке.

ArgumentException argEx = new ArgumentOutOfRangeException ("param1", "Parameter param1 too large.");

## Генерация исключений



После создания объекта исключения, его можно генерировать (throw), чтобы указать, что произошло исключение. Когда генерируется исключение, исполнение текущего блока кода прекращается, и CLR передает управление первому доступному обработчику исключений, который перехватывает исключение так, как это было описано ранее. Генерация исключения является дорогостоящей операцией с точки зрения циклов процессора (CPU), поэтому ее следует использовать с осторожностью.

Для генерации исключения используется ключевое слово throw, после которого указывается объект исключения, которое генерируется:

**throw [exception object];**

Например, создать и сгенерировать исключение типа **FormatException можно следующим образом**

FormatExeption ex = new FormatExeption("Argument has the wrong format");

throw ex;

Открытые и защищенные методы должны генерировать исключения каждый раз, когда не удается выполнить назначенную им функциональность. Генерируемый класс исключения должен быть самым определенным доступным исключением, удовлетворяющим условиям ошибки. Эти исключения должны документироваться в составе функций класса, а производные классы или обновления исходного класса должны сохранять то же поведение для обеспечения обратной совместимости.

Общей стратегией для метода или блока кода является перехват любого исключения и попытка справиться с ними. Если блок catch для исключения не может устранить ошибку, он может перебросить исключение и распространить его на вызывающий код. Для этого нужно указать ключевое слово throw, как показано в следующем примере кода.

try

{

... // Statements that might cause an exception

}

catch(Exception e)

{

// Attempt to handle the exception

...

// If this catch handler cannot resolve the exception,

// throw it to the calling code

throw;

}

Генерировать исключения необходимо при выполнении одного или нескольких из следующих условий:

* *Метод не способен выполнить свои определенные функции.* Например, если значение параметра метода является недопустимым:

static void CopyObject(SampleClass original)

{

**if** (original == null)

{

throw new ArgumentException("Parameter cannot be null", "original");

}

}

* *На основе состояния объекта выполнен неправильный вызов объекта.* В качестве примера можно привести попытку записи в файл, доступный только для чтения. В случаях, когда состояние объекта не допускает выполнения операции, генерируется экземпляр InvalidOperationException или объекта на основе наследования этого класса. Ниже показан пример метода, генерирующий объект InvalidOperationException:

class ProgramLog

{

FileStream logFile = null;

void OpenLog(FileInfo fileName, FileMode mode) { }

void WriteLog()

{

if (!this.logFile.CanWrite)

{

throw new InvalidOperationException("Logfile cannot be read-only");

}

// Else write data to the log and return.

}

}

* *Аргумент метода вызывает исключение*. В этом случае, должно быть перехвачено исходное исключение и создан экземпляр ArgumentException. Исходное исключение должно быть передано конструктору ArgumentException в качестве параметра InnerException:

static int GetValueFromArray(int[] array, int index)

{

try

{

return array[index];

}

catch (IndexOutOfRangeException ex)

{

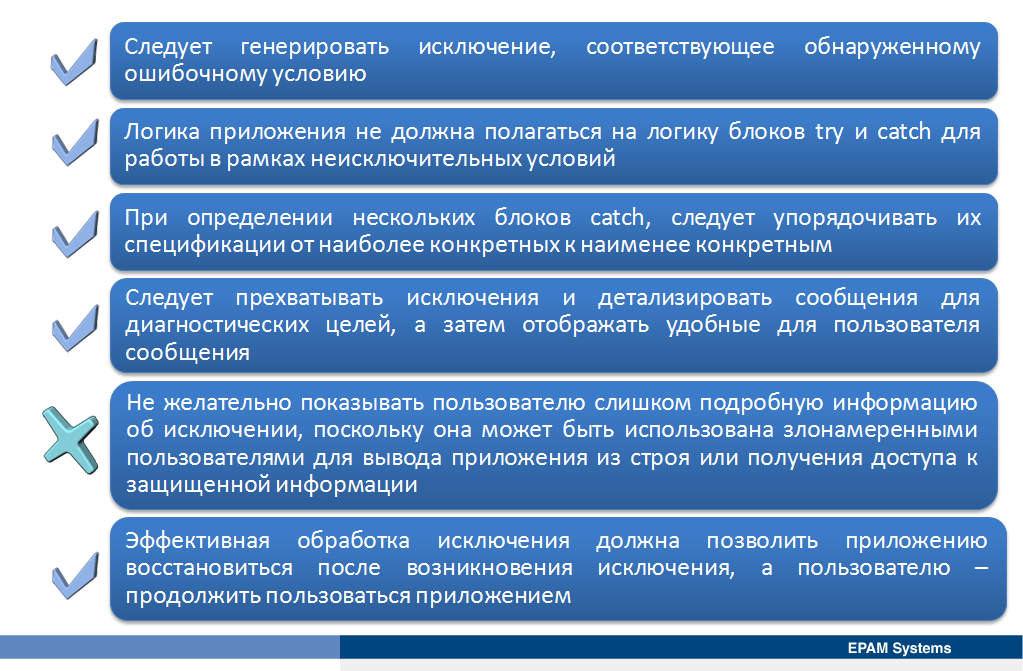
ArgumentException argEx = new ArgumentException("Index is out of range", "index", ex);

throw argEx;

}

}

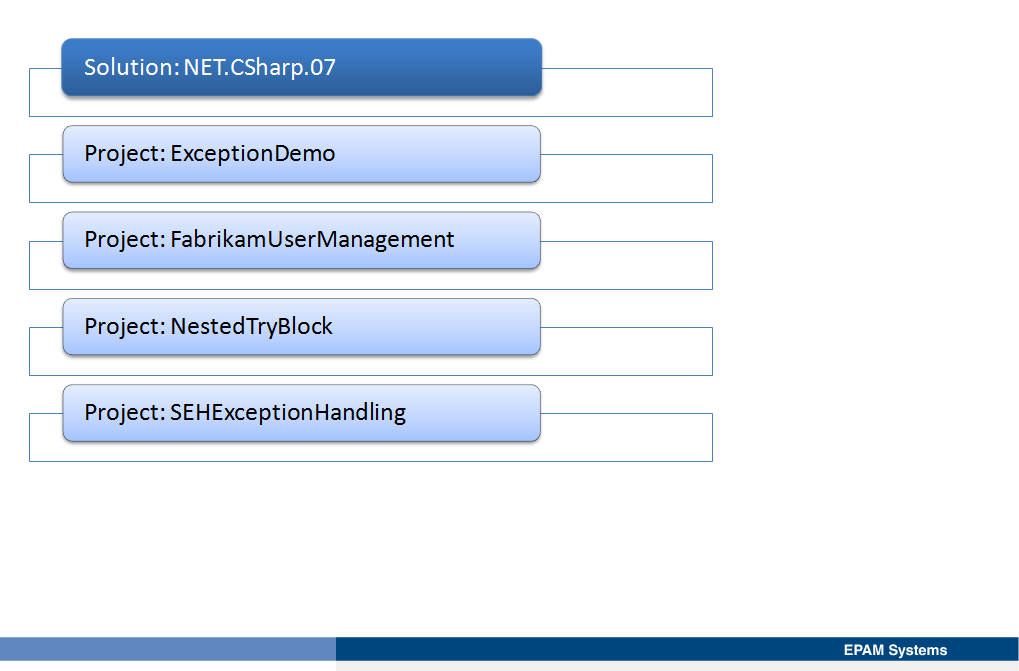
## Рекомендации по обработке и генерации исключений



Конструкции для реализации обработки исключений в приложениях просты в использовании, но для них, как и для любой программной конструкции, важно соблюдать хороший дизайн. В следующем списке описаны некоторые рекомендации по обработке исключений:

* Следует генерировать исключение, соответствующее обнаруженному ошибочному условию.
* Логика приложения не должна полагаться на логику блоков try и catch для работы в рамках неисключительных условий. Следует разрабатывать методы так, чтобы при нормальных обстоятельствах они не выбрасывали исключений. Создавать и генерировать исключения нужно для условий, которые находятся вне ожидаемого логического потока приложения.
* При определении нескольких блоков catch, следует упорядочивать их спецификации от наиболее конкретных к наименее конкретным. Если перехватывается тип исключения Exception, то он должен быть последним обработчиком во множестве блоков catch.
* Следует прехватывать исключения и детализировать сообщения для диагностических целей, а затем отображать удобные для пользователя сообщения. Любой текст, который отображается для пользователя должен быть локализуемым, текст должен извлекаться из файлов ресурсов.
* Не желательно показывать пользователю слишком подробную информацию об исключении, поскольку она может быть использована злонамеренными пользователями для вывода приложения из строя или получения доступа к защищенной информации. Например, распространенной ошибкой, выполняемой в слоях доступа к данным, является представление подробной информации об ошибке в результате неправильного запроса к базе данных. Такая информация может позволить злоумышленнику понять логику приложения и использовать эти знания для атаки на систему.
* Эффективная обработка исключения должна позволить приложению восстановиться после возникновения исключения, а пользователю – продолжить пользоваться приложением. В случае возникновения исключения пользователь не должен потерять данные, а приложение не должно потерпеть крах.

## Демонстрация: Возникновение и обработка исключений



1. По умолчанию выражение, содержащее только константные значения, приводит к ошибке компиляции в том случае, если результат его вычисления выходит за допустимые пределы значений конечного типа. Если выражение содержит одно или несколько неконстантных значений, компилятор не выполняет проверку переполнения. [↑](#footnote-ref-1)
2. Можно создавать свои собственные типы исключений, наследуя от класса System.Exception. [↑](#footnote-ref-2)