



EXCEPTION HANDLING. LOGGING. NLOG

.NET & JS LAB, RD BELARUS

Anzhelika Kravchuk

Содержание модуля

- Выбрасывание исключений
- Обработка исключений

Что такое исключение?

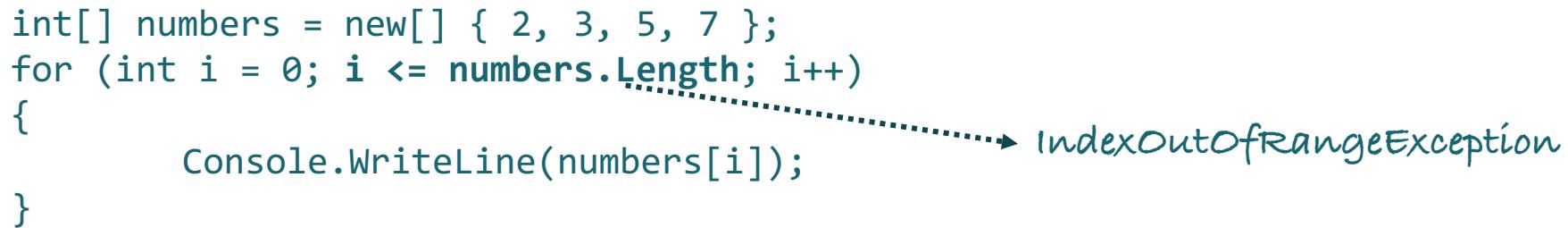
Исключение - это событие, которое происходит во время выполнения программы, нарушая нормальный поток инструкций программы.

Когда API-интерфейс сообщает об ошибке с помощью исключения, это прерывает обычный поток выполнения с переходом прямо к ближайшему подходящему коду обработки ошибок, что позволяет, в некоторой степени, отделить логику обработки ошибок кода от кода, предназначенного для выполнения поставленной задачи.

Причины исключений

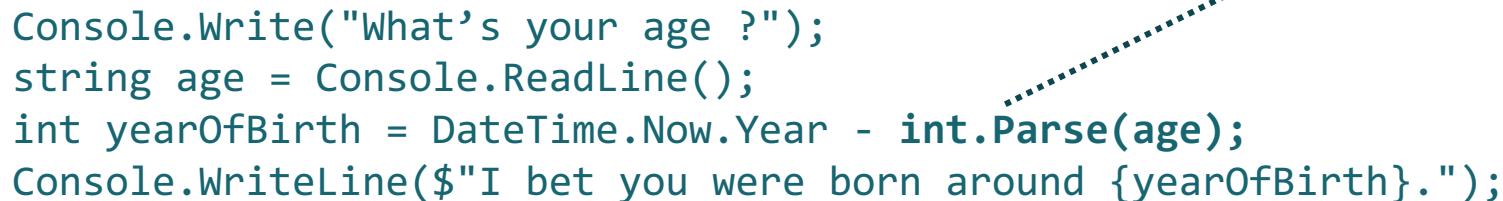
Code defects (дефекты кода), такие как отсутствие проверки нулевых ссылок, вызывающих NullReferenceExceptions, или передача недопустимых аргументов для метода, который сигнализирует об ошибке с использованием исключения ArgumentException и др.

```
int[] numbers = new[] { 2, 3, 5, 7 };
for (int i = 0; i <= numbers.Length; i++)
{
    Console.WriteLine(numbers[i]);
}
```



IndexOutOfRangeException

```
Console.Write("What's your age ?");
string age = Console.ReadLine();
int yearOfBirth = DateTime.Now.Year - int.Parse(age);
Console.WriteLine($"I bet you were born around {yearOfBirth}.");
```



FormatException

Причины исключений

```
Console.Write("What's your age?");  
string age = Console.ReadLine();  
try  
{  
    int yearOfBirth = DateTime.Now.Year - int.Parse(age);  
    Console.WriteLine($"I bet you were born around {yearOfBirth}.");  
}  
catch (FormatException)  
{  
    Console.WriteLine($"Invalid {nameof(age)} specified.");  
}
```

versus

```
Console.Write("What's your age?");  
string ageString = Console.ReadLine();  
int age;  
if (int.TryParse(ageString, out age))  
{  
    int yearOfBirth = DateTime.Now.Year - age;  
    Console.WriteLine($"I bet you were born around {yearOfBirth}.");  
}
```

Причины исключений

Runtime disasters, проблемы которые препятствуют продолжению выполнения механизма выполнения CLR; например, из-за недостаточной памяти (`OutOfMemoryException`) или переполнения стека.

```
static void Main()
{
    Main();
```

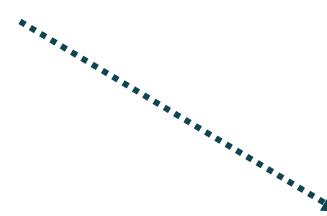


StackOverflowException

Причины исключений

Error conditions в среде, где работает код, например, отсутствующий файл конфигурации, невозможность подключения к веб-службе или базе данных или отказ внешнего компонента.

```
static void PrintFile(string path)
{
    foreach (string line in File.ReadAllLines(path))
    {
        Console.WriteLine(line);
    }
}
```



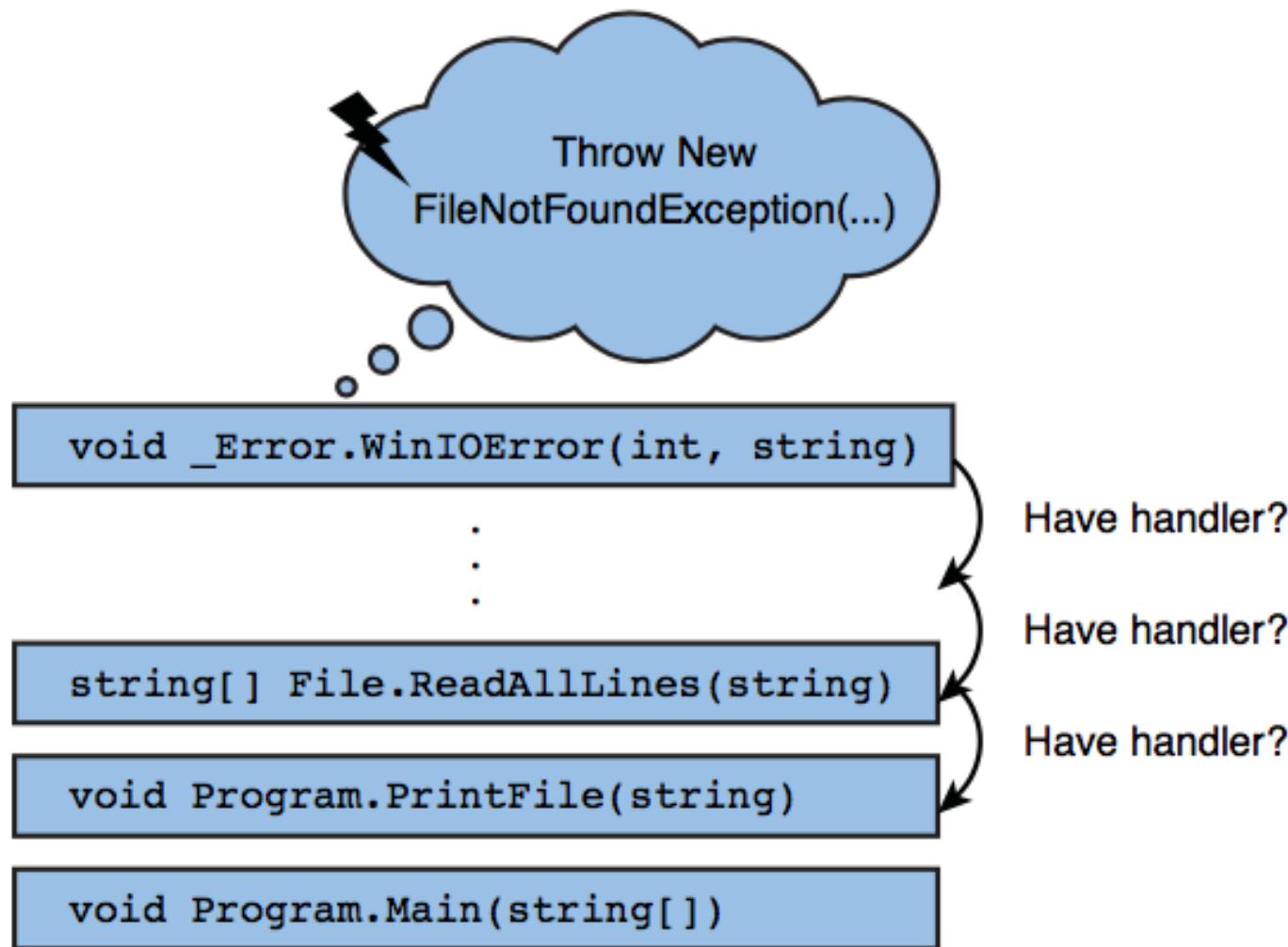
FileNotFoundException

```
if (!File.Exists(path))
{
    // Handle the problematic situation.
}
```

Источники исключений

- Программа использует API-интерфейс библиотеки классов, который выявляет проблему (ссылка null, где она недопустима, или пустая строка, где недопустима, например как имя файла; ошибки из разряда следствия ошибок в коде, немного иная категория ошибок, аргументы не вызывают нареканий, просто невозможно из-за текущего состояния окружения приложения);
- Собственный код выявляет проблему;
- Среда выполнения выявляет неудачное завершение операции (StackOverflowException, DivideByZeroException)
- среда выполнения распознает ситуацию, находящуюся вне контроля и влияет на код (например, выполнение потока прерывается в результате завершения работы приложения)

Поиск стека вызовов для обработчика исключений



Класс System.Exception

Свойство	Описание
Data	Предоставляет возможность добавлять конструкции "ключ-значение" к исключениям, которые могут быть использованы для снабжения исключений некоторой дополнительной информацией
HelpLink	Связь со справочным файлом, в котором представлена более подробная информация об исключении
InnerException	Если исключение было возбуждено внутри блока catch, то InnerException содержит объект исключения, который был передан в этот catch-блок
Message	Текст, описывающий условие ошибки
Source	Имя приложения или объекта, вызвавшего исключение
StackTrace	Информация о вызовах методов в стеке (для того, чтобы помочь в поиске метода, возбудившего исключение)
Targetsite	Объект рефлексии .NET, который описывает метод, возбудивший исключение

Классы исключений в C#

Exception Class	Description
System.IO.IOException	Handles I/O errors.
System.IndexOutOfRangeException	Handles errors generated when a method refers to an array index out of range.
System.ArrayTypeMismatchException	Handles errors generated when type is mismatched with the array type.
System.NullReferenceException	Handles errors generated from referencing a null object.
System.DivideByZeroException	Handles errors generated from dividing a dividend with zero.
System.InvalidCastException	Handles errors generated during typecasting.
System.OutOfMemoryException	Handles errors generated from insufficient free memory.
System.StackOverflowException	Handles errors generated from stack overflow.

Генерация исключения

```
void DisplayUserProfile(string user)
{
    if (user == null)
        throw new ArgumentNullException(nameof(user));
    if (string.IsNullOrEmpty(user))
        throw new ArgumentException("exception");
    if (!Profiles.Exists(user))
        throw new UserNotFoundException(nameof(user));
    // Do real work here.
}
```

A program throws an exception when a problem shows up. This is done using a `throw` keyword

Обработка исключений

```
try
{
    [Try block.]
}
catch ([catch specification 1])
{
    [Catch block 1.]
}
...
catch ([catch specification n])
{
    [Catch block n.]
}
```

A try block identifies a block of code for which particular exceptions is activated

A program catches an exception with an exception handler at the place in a program where you want to handle the problem. The catch keyword indicates the catching of an exception

Обработка исключений

```
try
{
    // Try block.
}
catch
{
    // Catch block.
}
```

The most common kind of catch block that does not have a catch specification, and therefore intercepts any type of exception

```
try
{
    // Try block.
}
catch (Exception ex)
{
    // Catch block, can access exception in ex.
}
```

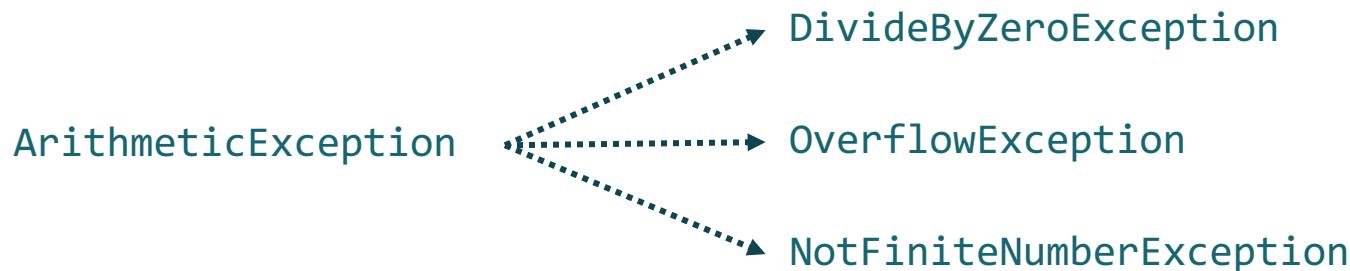
The Exception type is often used to catch all exceptions that have not been otherwise processed

Обработка исключений

```
try
{
    // Try block.
}
catch (DivideByZeroException ex) ← Logic that might cause an exception
{
    // Catch block, can access DivideByZeroException
    // exception in ex.
}
catch (Exception ex) ← Logic to run in the event of
{
    // Catch block, can access exception in ex.
}
```

Logic to run in the event of a DivideByZeroException exception

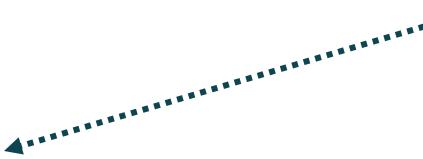
Logic to run in the event of any other exceptions



Обработка исключений

```
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.
}
```

You can also nest
try/catch blocks



Обработка исключений

```
try
{
    [Try block.]
}
catch ([catch specification 1])
{
    [Catch block 1.]
}
...
catch ([catch specification n])
{
    [Catch block n.]
}
finally <---->
{
    [Finally block.]
}
```

The `finally` block is used to execute a given set of statements, whether an exception is thrown or not thrown. For example, if you open a file, it must be closed whether an exception is raised or not.

Обработка исключений

Successful Execution

```
try
{
    // Do something.
    ThisCouldFail();
    // Do something else.
}
catch (SomeException ex)
{
    // Handle exception.
}
finally
{
    // clean-up.
}
```

Exceptional Execution

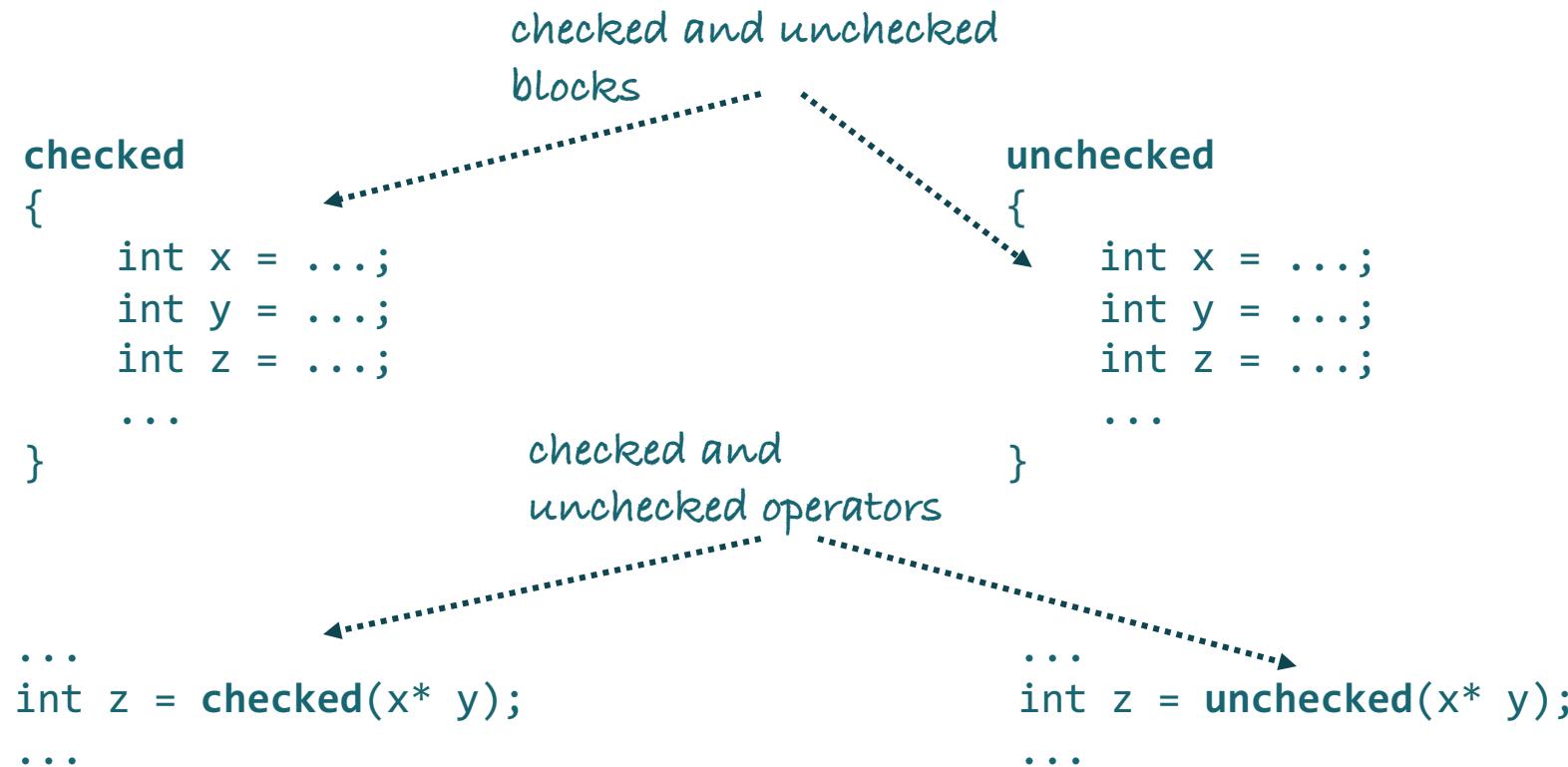
```
Throws
SomeException

try
{
    // Do something.
    ThisCouldFail();
    // Do something else.
}
catch (SomeException ex)
{
    // Handle exception.
}
finally
{
    // clean-up.
}
```

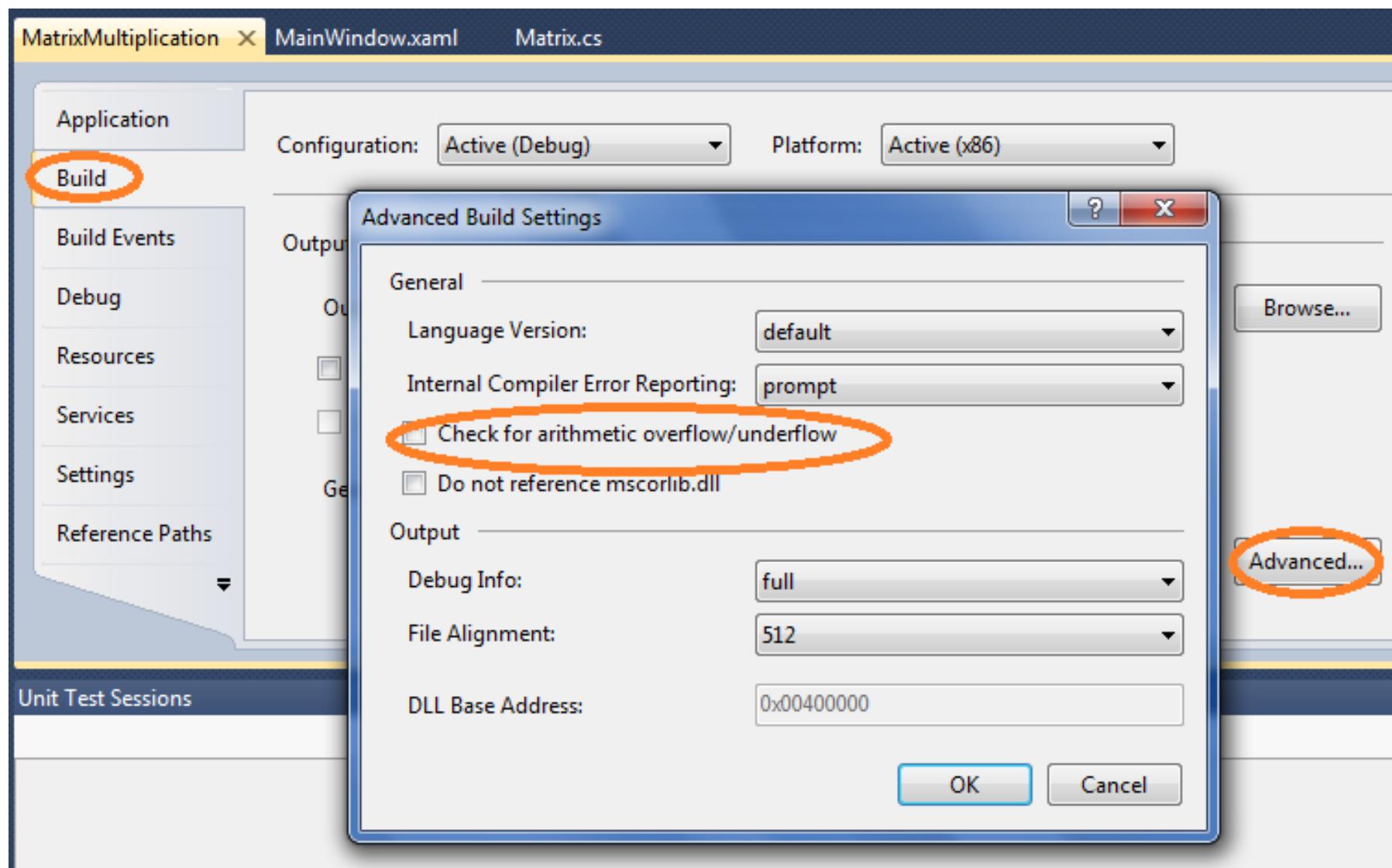
Использование ключевых слов checked и unchecked

Приложения C# запускаются с отключенной проверкой целочисленного числового переполнения по умолчанию. Эту настройку можно изменить

Можно контролировать проверку переполнения определенного кода с помощью ключевых слов checked и unchecked



Использование ключевых слов checked и unchecked



Guidelines and Best Practices



«Принцип самурая» это принцип разработки, призванный описать «контракт» между функцией и вызывающим ее кодом и заключается в следующем. Любая функция, реализующая некоторую единицу работы должна следовать тому же кодексу чести «бусидо», по которому живет любой самурай. Так, самурай не будет выполнять никаких заданий, противоречащих его «кодексу чести» и если к нему подойти с «непристойным» предложением, то он снесет вам башку раньше, чем вы успеете глазом моргнуть. Но если уж самурай возьмется за дело, то можно быть уверенным в том, что он доведет его до конца «сделай или умри».

Guidelines and Best Practices

Проверяйте аргументы методов

```
static void CopyObject(SampleClass original)
{
    if (original == null)
    {
        throw new ArgumentNullException(nameof(original));
    }
}
```

`System.ArgumentException`

- `System.ArgumentNullException`
- `System.ArgumentOutOfRangeException`
- `System.DuplicateWaitObjectException`

Guidelines and Best Practices

Блоков finally не может быть слишком много

```
public void SomeMethod()
{
    FileStream fs = new FileStream(@"C:\Data.bin ", FileMode.Open);
    try
    {
        // Outputting the quotient from the division of 100 by the first byte
        // of the file
        Console.WriteLine(100 / fs.ReadByte());
    }
    finally
    {
        // In finally block clearing code is situated, guaranteeing
        // closure of the file independently from the occurrence of the
        // exception (for example if the first byte is 0) or not.
        if (fs != null) fs.Dispose();
    }
}
```

Guidelines and Best Practices

Язык C# автоматически генерирует блоки try/finally всякий раз, когда используются **lock**, **using**, and **foreach**. Компилятор C# также генерирует блоки try/finally всякий раз, когда переопределяется финализатор класса (метод Finalize). При использовании этих конструкций компилятор помещает код, который написан внутри блока try, и автоматически помещает код очистки внутри блока finally. А именно:

- При использовании оператора lock, блокировка освобождается внутри блока finally.
- При использовании оператор using, у объекта есть метод Dispose, который вызывается внутри блока finally.
- При использовании оператора foreach, объект IEnumerator имеет свой метод Dispose, который вызывается внутри блока finally.
- При использовании метода деструктора, метод Finalize базового класса вызывается внутри блока finally.

Guidelines and Best Practices

```
public void SomeMethod()
{
    using (FileStream fs = new FileStream(@"C:\Data.bin", FileMode.Open))
    {
        // Display 100 divided by the first byte in the file.
        Console.WriteLine(100 / fs.ReadByte());
    }
}
```

Guidelines and Best Practices

Не следует перехватывать все исключения

```
try
{
    // try to execute code that the programmer knows might fail...
}
catch (Exception)
{
    // rethrow!
}
```

Guidelines and Best Practices

Корректное восстановление после исключения

```
public String CalculateSpreadsheetCell(int row, int column)
{
    string result;
    try
    {
        result = /* Code for calculating the value of a spreadsheet cell */
    }
    catch (DivideByZeroException)
    {
        result = "Can't show value: Divide by zero";
    }
    catch (OverflowException)
    {
        result = "Can't show value: Too big";
    }
    return result;
}
```

Guidelines and Best Practices

Отмена незавершенных операций при невосстановимых исключениях

```
public void SerializeObjectGraph(FileStream fs, IFormatter formatter, object rootObj)
{
    // Save current position to file
    long beforeSerialization = fs.Position;
    try
    {
        // Try to serialize an object graph and write it to file.
        formatter.Serialize(fs, rootObj);
    }
    catch
    {
        // Catching every single exception.
        // With ANY deviation bring the file back to normal state.
        fs.Position = beforeSerialization;
        // Truncate the file.
        fs.SetLength(fs.Position);
        // SIDENOTE: previous code isn't placed into the finally block,
        // thread flush is needed only if serialization fails.
        // Notify the calling code about what happened,
        // By generating the SAME EXCEPTION again.
        throw;
    }
}
```

Guidelines and Best Practices

Сокрытие деталей реализации для сохранения контракта или соглашения метода

```
internal sealed class PhoneBook
{
    private string pathname; // Path to the file with telephone directory.

    // The other methods are here.

    public string GetPhoneNumber(string name)
    {
        string phone;
        FileStream fs = null;
        try
        {
            fs = new FileStream(pathname, FileMode.Open);
            // Code for reading from fs, while the number won't be found
            phone = /* phone number is found */
        }
    }
}
```

Guidelines and Best Practices

```
catch (FileNotFoundException e)
{
    // Generate another exception with the name of the subscriber
    // setting the initial exception as the internal exception of
    // the new one.
    throw new NameNotFoundException(name, e);
}
catch (IOException e)
{
    // Generate another exception with the name of the subscriber
    // setting the initial exception as the internal exception of
    // the new one.
    throw new NameNotFoundException(name, e);
}
finally
{
    if (fs != null) fs.Close();
}
return phone;
}
```

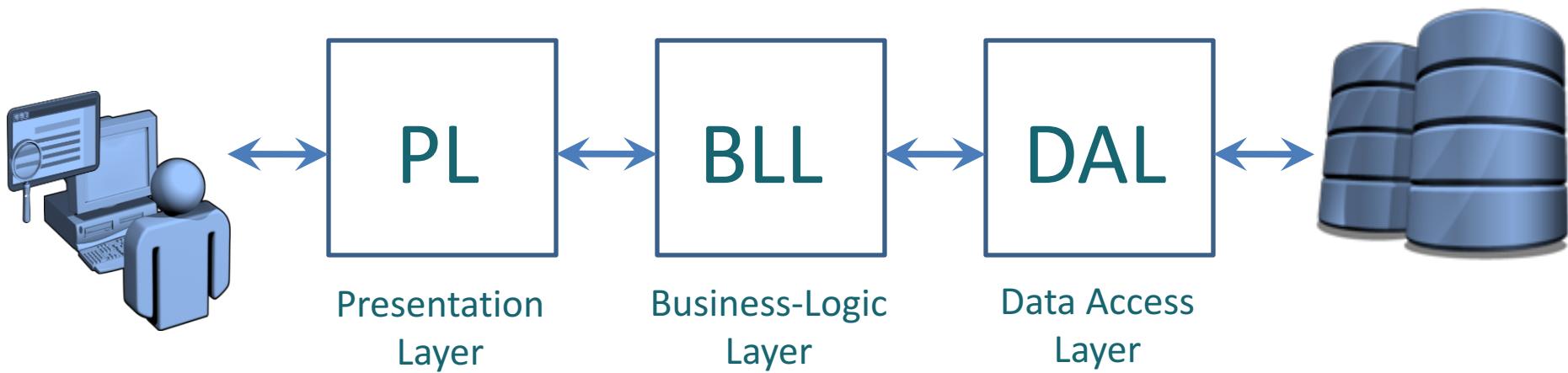
Guidelines and Best Practices

Исключения и наследование имеют общее свойство: используемые разумно, они могут уменьшить сложность. Используемые чрезмерно, они могут сделать код абсолютно нечитаемым

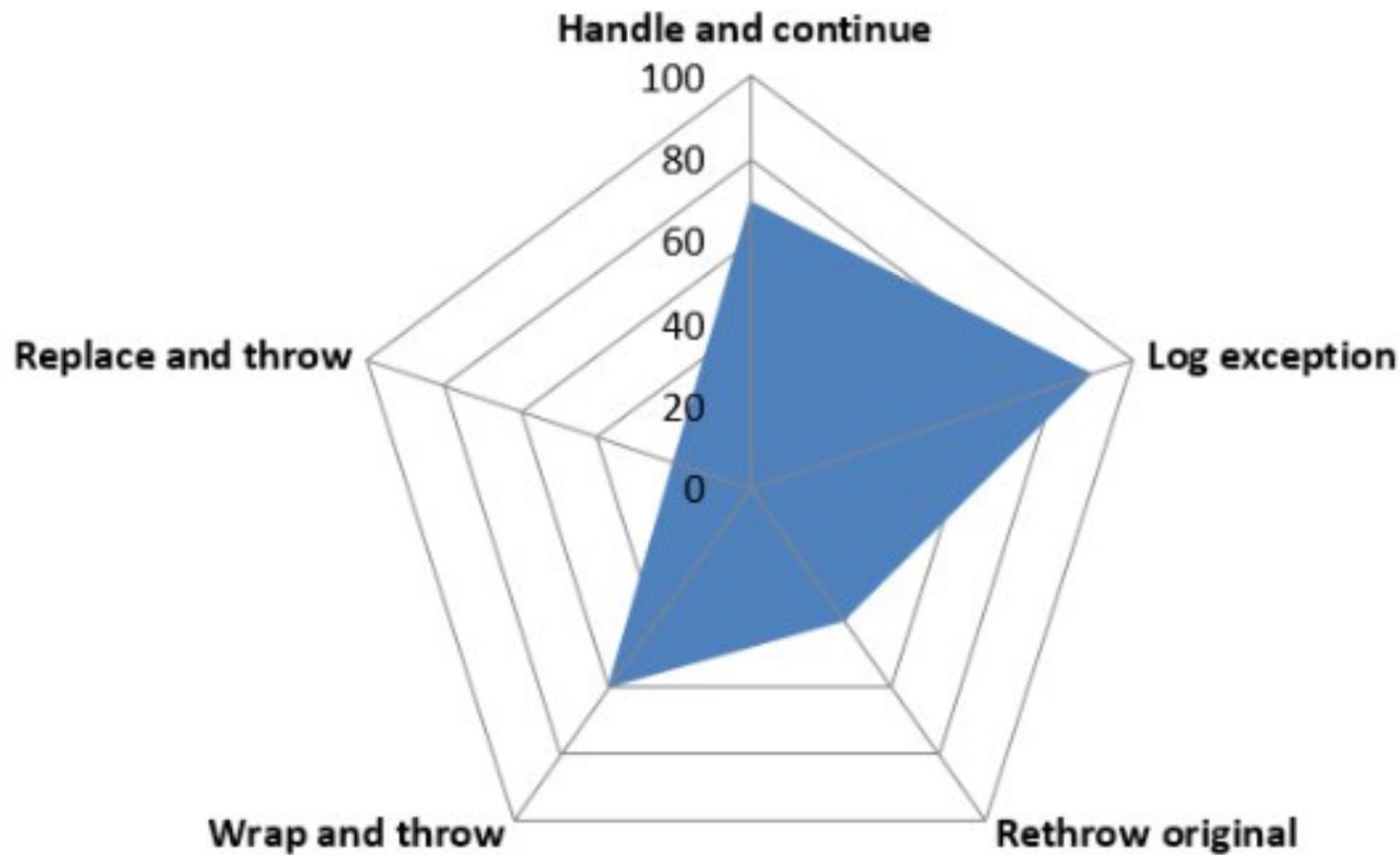
Стив Макконнелли, «Совершенный код»

1. Используйте исключения для оповещения других частей программы об ошибке, которую нельзя игнорировать
2. Генерируйте исключения только для действительно исключительных ситуаций
3. Не используйте исключения по мелочам
4. Вносите в описание исключения всю информацию о его причинах
5. Выясните, какие исключения генерирует используемая библиотека
6. Стандартизируйте использование исключений в вашем проекте
7. Рассмотрите альтернативы исключениям
8. Генерируйте исключения на правильном уровне абстракции

Guidelines and Best Practices



Guidelines and Best Practices



Логирование. Примитивный подход

Основные цели, ради которых существуют лог-файлы

- сказать, что же делает система прямо сейчас, не прибегая к помощи отладчика
- провести «расследование» обстоятельств, которые привели к определенному состоянию системы (например, падению или багу)
- проанализировать, на что тратится больше времени/ресурсов (профилирование)

```
public static void Log(string message)
{
    File.AppendAllText("log.txt", message);
}
```

Логирование

- **Уровни логирования и фильтрация сообщений**

помогают определить критичность сообщения и приемлемое время реакции на него

- **Ротация лог-файлов**

возможность подменять активный файл при наступлении определенных условий

- **Возможность записи сообщений не только в файлы**

поддержка отправки сообщений по протоколу UDP, запись в базу, взаимодействие с очередями сообщений

- **Потокобезопасность**

Плохой логгер может:

- пропустить часть сообщений;
- выбросить исключение
- отрицательно повлиять на производительность

- **Асинхронное логирование**

настраиваемый размер буфера, возможность писать debug-сообщения по 100 штук,

а error – немедленно после возникновения

- **Формат и конфигурация логов**

Формат должен быть настраиваемый, с возможностью указать то, что писать и куда писать

Логирование

Debug: сообщения отладки, профилирования

Info: обычные сообщения, информирующие о действиях системы. Реагировать на такие сообщения вообще не надо, но они могут помочь, например, при поиске багов, расследовании интересных ситуаций и т.д.

Warn: записывая такое сообщение, система пытается привлечь внимание обслуживающего персонала. Произошло что-то странное. Возможно, это новый тип ситуации, еще не известный системе. Следует разобраться в том, что произошло, что это означает, и отнести ситуацию либо к инфо-сообщению, либо к ошибке. Соответственно, придется доработать код обработки таких ситуаций

Error: ошибка в работе системы, требующая вмешательства. Что-то не сохранилось, что-то отвалилось. Необходимо принимать меры довольно быстро! Ошибки этого уровня и выше требуют немедленной записи в лог, чтобы ускорить реакцию на них

Fatal: это особый класс ошибок. Такие ошибки приводят к неработоспособности системы в целом, или неработоспособности одной из подсистем. Чаще всего случаются фатальные ошибки из-за неверной конфигурации или отказов оборудования. Требуют срочной, немедленной реакции

Логирование

Разрабатываемая система – сотрудник почты, который принимает посылки.
Принесли посылку.

Debug: Получена посылка 1. Проверяю размер...

Debug: Размер посылки 1: 40x100

Debug: Взвешиваю посылку...

Debug: Вес посылки 1: 1кг

Debug: Проверяю соответствие нормам...

Info (не Error!): Посылка 1 размером 40x100, весом 1кг, отклонена: превышен максимальный размер

...

Info: Посылка 2 размером 20x60, весом 0.5 кг передана на обработку оператору 1

...

Warn: Отказано в обработке для посылки 3: дата на посылке относится к будущему:
2050-01-01

...

Error: Не удалось отдать посылку оператору: оператор не отвечает: таймаут ожидания ответа оператора

...

Fatal: Произошёл отказ весов. Посылки не будут приниматься до восстановления работоспособности.

Логирование с использование фреймворка NLog

Фреймворки для логирования (logging framework): log4net, **NLog** (<http://nlog-project.org>, <http://www.codeproject.com>), Microsoft Logging Application Block

Преимущества готовых logging framework-ов:

- Гибкая настройка лог-файла
- Широкие возможности
- Наличие конфигурационных xml-файлов с xsd-схемами
- Готовое стабильное решение (экономия времени на отладке)
- Наличие документации (форумов, блогов)

<https://github.com/NLog/NLog/wiki/Tutorial>

Логирование с использование фреймворка NLog

Каждое сообщение имеет уровень логирования (log level), в лог-файл заноситься только то сообщение, уровень логирования которого ниже указанного в правиле логирования

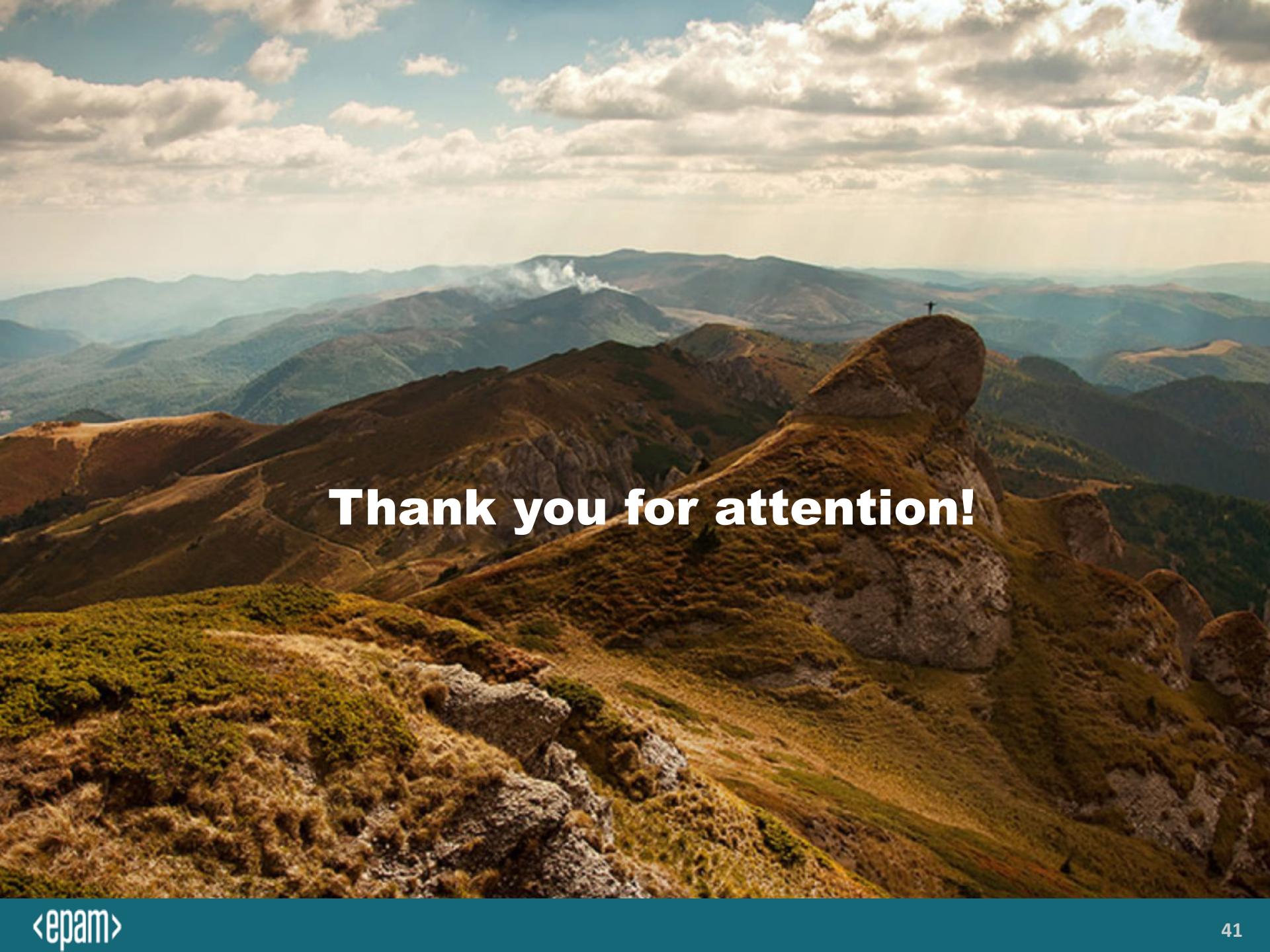
NLog поддерживает следующие уровни (service level agreement):

- Trace - детальное логирование (все сообщения) если Debug не позволяет локализовать ошибку. В нем полезно отмечать вызовы разнообразных блокирующих и асинхронных операций
- Debug - отладочное логирование, менее детально чем Trace
- Info - логирование информационных сообщений
- Warn - логирование сообщений о предупреждениях
- Error - логирование сообщений об ошибках
- Fatal - логирование сообщений о критических ошибках(только критические сообщения)
- Off - логирование сообщений не производиться

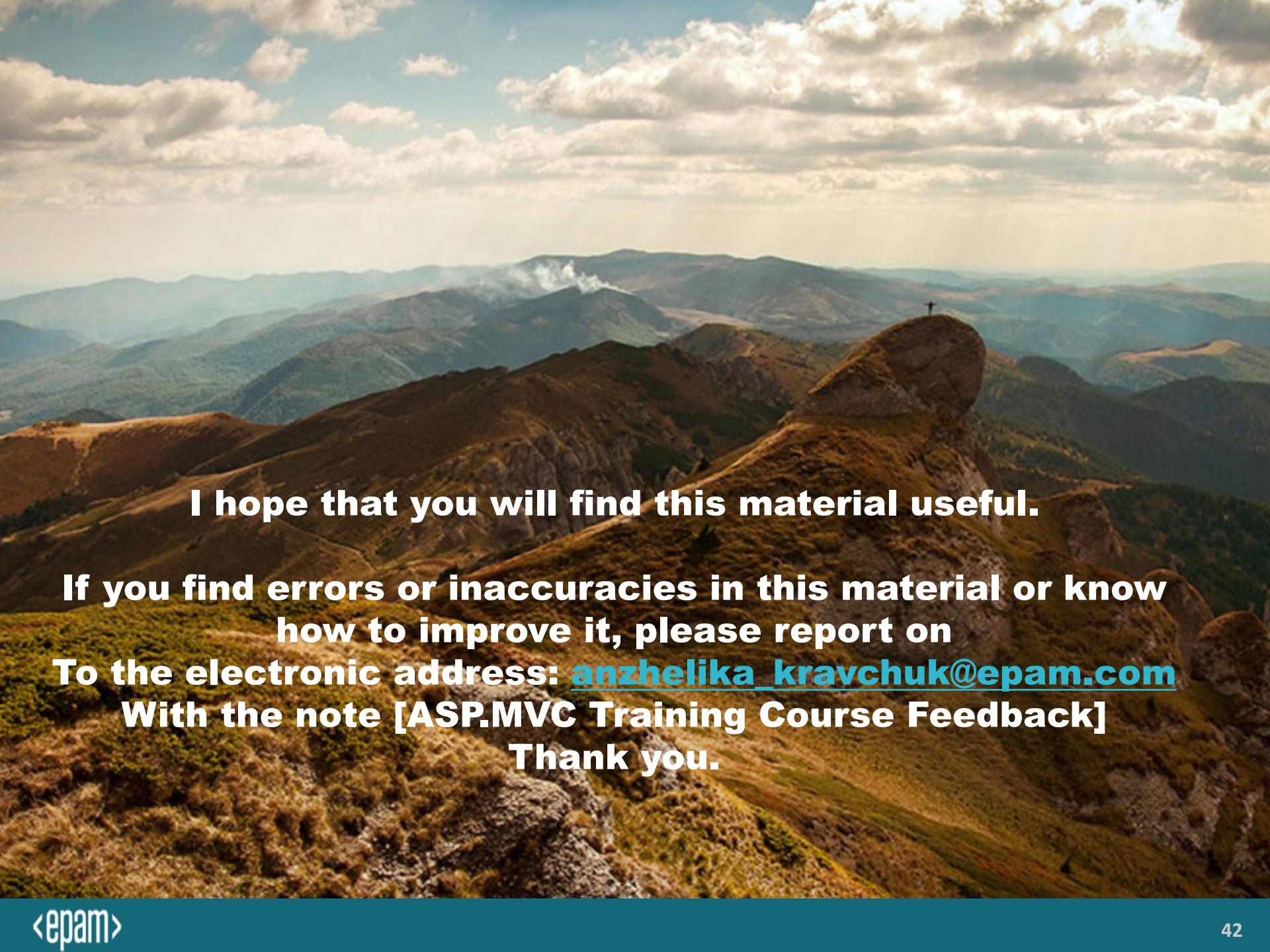
Логирование с использование фреймворка NLog

Правила записи исключений в лог-файлы

- ✓ если исключение обработано
 - исключение считается обработанным и не прорывается выше по стеку - в этом случае исключение записывается с подробным стеком в лог
 - исключение прорывается выше по стеку в той же подсистеме - такое исключение не логируется, однако убедитесь, что выше по стеку оно будет записано
 - исключение прорывается выше по стеку в другую подсистему (например, на другую машину или в другой процесс) - такое исключение логируется, или записывается диагностическое сообщение об исключении;
- ✓ если исключение не обработано – оно не логируется, однако следует убедится, что выше это исключение будет обязательно залогировано

A wide-angle photograph of a mountain range under a dramatic sky. In the foreground, rocky terrain and green slopes are visible. In the middle ground, a person stands on a prominent, rounded rock formation on a ridge. The background features multiple layers of mountains, with one emitting a small plume of white smoke or steam from its peak. The sky is filled with large, fluffy clouds.

Thank you for attention!

A wide-angle photograph of a mountainous landscape. In the foreground, there are rocky, grassy slopes. In the middle ground, several mountain ridges are visible, with one prominent peak on the right side where a small figure of a person stands. The background shows more distant mountain ranges under a sky filled with scattered, fluffy clouds.

I hope that you will find this material useful.

**If you find errors or inaccuracies in this material or know
how to improve it, please report on
To the electronic address: anzhelika_kravchuk@epam.com
With the note [ASP.MVC Training Course Feedback]
Thank you.**