

# **Лекция 6.**

## **Исключения**

# Исключительные ситуации

- При исполнении кода могут возникать такие ситуации, когда исполнение нашего кода не пойдет по предполагаемому сценарию
- Например, при попытке прочитать данные из файла, во время исполнения программы, обнаружим что файла не существует
- Или, при чтении данных из сети, обрывается соединение
- Или кончается свободное место на диске
- Такие ситуации называют **исключительными ситуациями** (или просто **исключениями**)

# Реакция на исключения

- По умолчанию, при возникновении исключения, программа падает
- Но можно зарегистрировать свой код-перехватчик исключений, и тогда программа падать не будет

# Как реагировать на исключения

- На исключительные ситуации можно реагировать разным образом:
  - Можно завершить программу, выдав сообщение об ошибке
  - Можно просто продолжить исполнение со следующей команды
  - Можно попытаться выполнить действие ещё раз
  - И т.д.
- В каждом конкретном случае, реагировать на исключительные ситуации нужно по-своему

# Исключения

- В С# и многих других языках, реализован механизм обработки исключительных ситуаций
- Для этого используется конструкция `try-catch-finally`
- А **бросить исключение** (т.е. инициировать его) можно при помощи оператора `throw`
- [http://professorweb.ru/my/csharp/charp\\_theory/level8/8\\_1.php](http://professorweb.ru/my/csharp/charp_theory/level8/8_1.php)

# Терминология

- Информация об исключительной ситуации передается при помощи специальных классов-**исключений (exceptions)**
- Они содержат в себе сообщение об ошибке и стек вызовов, который очень полезен при отладке
- Если в функции может возникнуть исключительная ситуация, то говорят, что функция **бросает исключение (throws exception)**
- **Поймать исключение** – сделать так, чтобы выполнялся ваш обработчик исключения
- **Обработать исключение** – обработчик исключения успешно отработал до конца, не вызвав новое исключение

# Выброс исключения

- ```
public static void F()  
{  
    // этот код выполнится  
    throw new IOException("Message");  
    // этот код никогда не выполнится  
}
```
- Можно самим выбросить исключение, например, если мы сами проверили аргументы метода, и видим, что передали недопустимое значение
- Для этого нужно бросить экземпляр исключения при помощи слова **throw** (с англ. - бросить)
- В данном случае мы создаем новый объект исключения **IOException**

# Что происходит при бросании исключения

1. Делается проверка, находимся ли мы внутри конструкции `try-catch`, при этом там есть `catch`, который соответствует типу нашего исключения
  2. Если да, то попадаем в `catch`. Распространение исключения прекращается
  3. Если нет, то метод немедленно завершается, мы вместе с брошенным исключением попадаем в то место, откуда метод был вызван
  4. В этом месте выполняются действия из п.1 и т.д.
- Так исключение может дойти до **Main**. Если `try-catch` нет и там, то программа падает и завершается



# Раскрутка стека вызовов

- При возникновении исключения происходит **раскрутка стека вызовов**
- Т.е. стек «**разматывается**» в обратном порядке, исполнение возвращается назад, в вызвавшую функцию и т.д.
- ```
public static void F() {  
    // этот код выполнится  
    throw new IOException("Message");  
    // этот код никогда не выполнится  
}  
public static void Main() {  
    F();  
    Console.WriteLine("OK");  
}
```

# Обработка исключения

- Рассмотрим как обработать исключение
- Есть два блока – `try` и `catch`

- `try`

```
{  
    // тут идет код для благополучного хода событий  
    // например, когда файл есть, нет никаких ошибок  
}  
catch (IOException e)  
{  
    // этот код выполнится, при возникновении исключения  
    // IOException в любой из команд, которая выполняется внутри  
    // блока try  
}
```

Блок `catch` ловит исключение указанного типа. Если исключение поймано, то оно считается обработанным и не распространяется дальше по стеку вызовов

# Вариант – исключения не было

- `try`  
{  
    // тут идет код для благополучного хода событий  
    // например, когда файл есть, нет никаких ошибок  
}  
`catch (IOException e)`  
{  
    // этот код выполнится, при возникновении исключения  
    // IOException в любой из команд, которая выполняется внутри  
    // блока try  
}
- Просто выполняется блок `try`, а затем выполняется код, который идет ниже, чем `try-catch`

# Вариант – произошло IOException

- ```
try {  
    // тут идет код для благополучного хода событий  
    // например, когда файл есть, нет никаких ошибок  
} catch (IOException e) {  
    // этот код выполнится, при возникновении исключения  
    // IOException в любой из команд, которая выполняется внутри  
    // блока try  
}
```
- Выполняется блок `try`, в какой-то его команде возникает исключение `IOException`
- Тогда произойдет размотка стека до этой команды. Остальные команды блока `try` не выполнятся
- Исполнение переходит в блок `catch`, он исполняется. Если в нем возникло новое исключение, то тогда стек разматывается дальше
- Если `catch` успешно отработал, то исполняется код после `try-catch`

# Вариант – произошло другое искл-е

- `try {`  
    // тут идет код для благополучного хода событий  
    // например, когда файл есть, нет никаких ошибок  
`} catch (IOException e) {`  
    // этот код выполнится, при возникновении исключения  
    // IOException в любой из команд, которая выполняется внутри  
    // блока `try`  
`}`
- Блок `catch` не поймает исключение, поэтому стек вызовов будет разматываться дальше, туда, откуда вызван метод, содержащий этот `try-catch`

# Несколько блоков catch

- ```
try {  
    // код  
} catch (FileNotFoundException e) {  
    // обработка исключения FileNotFoundException  
} catch (IOException e) {  
    // обработка исключения IOException  
}
```
- Блоков catch может быть много
- Классы исключений могут наследоваться друг от друга, например, `FileNotFoundException` наследуется от `IOException`
- И вот тут интересный момент – как выбирается нужный блок catch для выброшенного исключения

# Несколько блоков catch

- ```
try {  
    // код  
} catch (FileNotFoundException e) {  
    // обработка исключения FileNotFoundException  
} catch (IOException e) {  
    // обработка исключения IOException  
}
```
- Допустим, брошено исключение
- Смотрятся блоки `catch` в порядке их объявления
- Если указанный тип исключения совместим с типом брошенного исключения, т.е. проверка `is` дает `true`, то выбирается этот блок `catch`
- Компилятор следит чтобы `catch` с наследником шел раньше

# Условия на catch в C# 6

- ```
try
{
    // код
}
catch (MyException e) when (e.Code == 42)
{
    // обработка исключения если выполняется условие
}
```
- В новом C# можно добавлять условия на обработчики `catch`
- `catch` сработает только если выражение внутри `when` даст `true`, в нём можно использовать объект исключения



# Повторный выброс исключения

- ```
try {  
    // код  
} catch (IOException e) {  
    Console.WriteLine("Error occurred: " + e);  
    throw; // обратим внимание, что бросаем тот же  
           // объект исключения  
}
```
- Поймав исключение блоком `catch`, его можно выбросить повторно при помощи `throw`;
- Это полезно, если мы при помощи блока `catch` хотим либо частично обработать исключение, либо просто записать в лог информацию о том, что произошла ошибка
- Заметим, что в C# `ToString` у исключения даёт полную информацию об исключении вместе со стеком

# Блок finally

- ```
try {  
    // код  
} catch (IOException e) {  
    // обработка исключения IOException  
} finally {  
    // обычно, освобождение ресурсов  
}
```
- **finally** – это блок кода, который выполняется всегда после блока **try** или блока **catch**
- Обычно он используется для освобождения ресурсов
- Например, мы хотим поработать с файлом, после работы с ним, нужно его закрыть в любом случае – если все прошло хорошо, и если произошло исключение

# Блок `finally` – нет исключения

- ```
try {  
    // код  
} catch (IOException e) {  
    // обработка исключения IOException  
} finally {  
    // обычно, освобождение ресурсов  
}
```
- Сначала выполняется блок `try`, потом – блок `finally`
- Причем блок `finally` исполнится даже если внутри `try` будет `return`, `break` или `continue`

# Блок `finally` – брошен `IOException`

- ```
try {  
    // код  
} catch (IOException e) {  
    // обработка исключения IOException  
} finally {  
    // обычно, освобождение ресурсов  
}
```
- Частично выполняется блок `try`, потом `catch`, потом блок `finally`
- Причем блок `finally` исполнится даже если внутри `catch` будет `return`, `break` или `continue`, или будет выброшено новое исключение

# Блок `finally` – брошен не `IOException`

- ```
try {  
    // код  
} catch (IOException e) {  
    // обработка исключения IOException  
} finally {  
    // обычно, освобождение ресурсов  
}
```
- Частично выполняется блок `try`, потом блок `finally`, потом исключение распространяется дальше по стеку вызовов
- В общем, блок `finally` выполняется реально **всегда**
- И это как раз очень полезно для освобождения ресурсов – то что мы не забудем их освободить

# Return из finally

- В C# `return` из `finally` запрещен

# Блока catch может не быть

- ```
try
{
    // код
}
finally
{
    // обычно, освобождение ресурсов
}
```
- Блока `catch` может не быть, но тогда должен быть `finally`
- Этот вариант полезен, если при возникновении исключения мы хотим выполнить код, обычно очистку ресурсов, но само исключение ловить и обрабатывать не хотим

# Правильное закрытие ресурсов в C#

- ```
public static void Main()
{
    using (StreamReader reader = new StreamReader("input.txt"))
    {
        // ... много строк кода, чтение из файла
    }
}
```
- Эта конструкция автоматически вызывает метод `Dispose` своего аргумента после завершения блока любым образом
- По сути, это и есть реализация, которая указана слайдом ранее
- Все классы, которые реализуют интерфейс `IDisposable` могут использовать этот синтаксис



# Иерархия исключений в C#

- Все исключения наследуются от `System.Exception`
- Свои исключения нужно наследовать от этого класса

# Как поймать все исключения

- ```
try {  
    // код  
} catch (Exception e) {  
    // обработка исключения  
}
```
- Такой блок `catch` будет ловить все исключения
- Есть ещё такой вариант, но он плох тем, что объект исключения недоступен в `catch`
- ```
try {  
    // код  
} catch {  
    // обработка исключения  
}
```

# Создание своих исключений

- Лучше создавать свои исключения если:
  - В C# нет такого стандартного исключения
  - Свой тип исключения поможет отличить их от других исключений
- При создании своих классов исключений рекомендуется заканчивать их имена на Exception
- Пример:
- ```
public class MyException : Exception {  
    public MyException(string message) : base(message) {  
    }  
}
```

# InnerException

- Допустим, мы пишем класс, который грузит конфигурацию нашего приложения
- По принципу инкапсуляции внешний код не знает как это все реализовано
- Но допустим, данные читаются из файла
- Если файла не окажется, то вылетит `FileNotFoundException`
- Но мы хотим скрыть детали реализации, а тип исключения все раскрывает
- Чтобы это исправить, мы поймаем `FileNotFoundException`, и выкинем некоторый наш тип исключения `LoadConfigurationException`

# InnerException

- ```
try {  
    // читаем из файла  
} catch (FileNotFoundException e) {  
    throw new LoadConfigurationException("Не удалось  
загрузить конфигурацию");  
}
```
- В этом коде плохо, что мы теряем изначальную информацию – стек вызовов исходного исключения, сообщение и т.д.
- Поэтому прямо так никто не делает

# InnerException

- Есть возможность указать связь между исключениями – что одно исключение является причиной другого
- У класса `Exception` есть конструктор, который принимает другое исключение
- И во всех подобных ситуациях надо использовать этот конструктор
- Своим исключениям надо тоже делать такой конструктор
- ```
try {  
    // читаем из файла  
} catch (FileNotFoundException e) {  
    throw new LoadConfigurationException("Не удалось  
загрузить конфигурацию", e);  
}
```

# InnerException

- Получить исключение-причину можно через свойство **InnerException**
- InnerException – это тоже очень важная информация об ошибке, надо обязательно логировать

# Как правильно логировать исключения

- Правильно просто вызвать метод `ToString` у исключения:
- `string text = e.ToString();`
- Этот метод выдаст вам тип и сообщение исключения, и эту же информацию по всем `InnerException`'ам
- Ведь у `InnerException`'а может быть свой `InnerException` и т.д.
- Не нужно самим получать `Message`, циклом проходить по `InnerException`'ам и т.д., просто используйте `ToString`



# Альтернатива исключениям

- Возможная альтернатива исключениям – возвращение специальных значений из метода
- Например, в некоторых случаях, если произошла ошибка, можно возвращать из метода `null`
- Либо, если метод должен возвращать неотрицательное целое число, можно возвращать отрицательные числа в качестве кодов ошибок, а затем сравнивать результат с этими кодами
- Достоинством данного подхода является более высокая производительность
- Но читаемость кода при этом существенно страдает

# Язык С – работа с файлами

```
FILE* file = fopen("input.txt", "r"); // открытие файла для чтения
int a;
float b;
if (file != NULL) { // проверка что нет ошибки
    fscanf(file, "%d", &a); // считываем из файла целое число в переменную a
    if (ferror(file)) {
        printf("Ошибка при чтении файла");
        fclose(file);
    } else {
        fscanf(file, "%f", &b ); // считываем вещ. число в переменную b
        if (ferror(file)) {
            printf("Ошибка при чтении файла");
            fclose(file);
        }
    }
} else {
    printf("Ошибка при чтении файла");
}
```

# Язык С – если бы были исключения

```
try (FILE* file = fopen("input.txt", "r")) {  
    int a;  
    float b;  
    fscanf(file, "%d", &a); // считываем из файла целое число в переменную a  
    fscanf(file, "%f", &b); // считываем вещ. число в переменную b  
} catch (FileNotFoundException e) {  
    printf("Ошибка при чтении файла");  
} catch (IOException e) {  
    printf("Ошибка при чтении файла");  
}
```

## Достоинства:

- Четко виден основной вариант алгоритма, когда нет ошибок
- Обработка ошибок, которые хотим обрабатывать одинаково, происходит централизованно
- Если мы не можем обработать ошибку здесь, то брошенное исключение может быть обработано «выше» по стеку вызовов