Тема 12. Обработка исключительных ситуаций. Примеры использования. Типы исключений. Блок try-catch-finally. Пользовательские исключения.

Учебные вопросы:

1. Введение.
2. Типы исключений
3. Обработка исключений. Блок try-catch-finally.
4. Создание пользовательских исключений.

1. Введение.

Исключения в C# – это механизм, позволяющий программе реагировать на нестандартные ситуации и ошибки во время выполнения. Они представляют собой события, которые прерывают нормальный поток выполнения программы и позволяют передать информацию о произошедшей проблеме.

Зачем нужны исключения?

- **Структурированная обработка ошибок:** Исключения позволяют отделить основной поток логики программы от обработки ошибок. Это делает код более читаемым и поддерживаемым.
- **Улучшение надежности:** Правильное использование исключений помогает предотвратить неожиданные сбои программы и обеспечивает более стабильную работу.
- **Сообщения об ошибках:** Исключения содержат информацию о том, что пошло не так, что позволяет разработчику быстро локализовать и исправить проблему.

Разница между ошибками и исключениями

- **Ошибки** – это более широкое понятие, которое включает в себя как исключения, так и другие виды проблем, например, синтаксические ошибки, которые обнаруживаются компилятором.
- **Исключения** (или exceptions) — это события, которые возникают при выполнении программы и могут быть **обработаны** внутри программы. Исключения обычно связаны с логическими ошибками или неправильным использованием ресурсов, которые можно исправить или обойти.

Ошибки (или errors) — это обычно более серьезные проблемы, которые часто вызваны проблемами на уровне системы, и они могут быть **непреодолимыми**. Например, ошибка переполнения памяти (OutOfMemoryError) или ошибка нехватки ресурсов (StackOverflowError).

Ошибки указывают на состояние программы или системы, которое **невозможно** или очень сложно исправить на уровне прикладного кода.

Причины возникновения исключений

1.Ошибки в коде:

- **Неправильные операции с данными**: Например, попытка деления на ноль или обращение к элементу коллекции по несуществующему индексу.
- **Неверное использование объектов**: Попытка доступа к объекту, который не был инициализирован (NullReferenceException).
- **Нарушение контрактов и соглашений**: Например, вызов метода с недопустимыми параметрами.

2.Проблемы с ресурсами:

- **Недоступность ресурсов**: Например, файл, к которому пытается получить доступ программа, отсутствует или заблокирован другим процессом.
- **Исчерпание ресурсов**: Например, недостаток оперативной памяти или места на диске.
- **Ошибки ввода-вывода**: Например, разрыв соединения с сетью или потеря связи с базой данных.

3.Внешние факторы:

- **Проблемы с аппаратным обеспечением**: Например, сбой оборудования или временная потеря связи с устройствами ввода-вывода.
- **Пользовательские ошибки**: Например, ввод неверных данных пользователем, таких как недопустимый формат даты или пустое поле, где ожидалось значение.
- **Взаимодействие с внешними системами**: Например, получение некорректного ответа от удаленного сервера или неожиданное изменение формата данных, передаваемых по сети.

Таким образом, **ошибки** часто означают, что программа не может продолжить выполнение и должна быть **завершена**, в то время как **исключения** можно и нужно **обрабатывать**, чтобы программа могла **продолжить** выполнение.

2. Типы исключений

В C# существует множество различных типов исключений, каждый из которых предназначен для обозначения определенного вида ошибки.

Все исключения наследуются от базового класса **Exception**.

Основные категории исключений:

SystemException: Базовый класс для всех исключений, связанных с системными ошибками.

- ArithmeticException: Ошибки арифметических операций (например, деление на ноль).
- OutOfMemoryException: Недостаток памяти.
- IndexOutOfRangeException: Выход за границы массива.
- NullReferenceException: Попытка обратиться к нулевой ссылке.
- InvalidOperationException: Операция не может быть выполнена в текущем состоянии.

ApplicationException: Базовый класс для пользовательских исключений, создаваемых разработчиком.

Другие важные типы:

ArgumentException: Передан некорректный аргумент в метод.

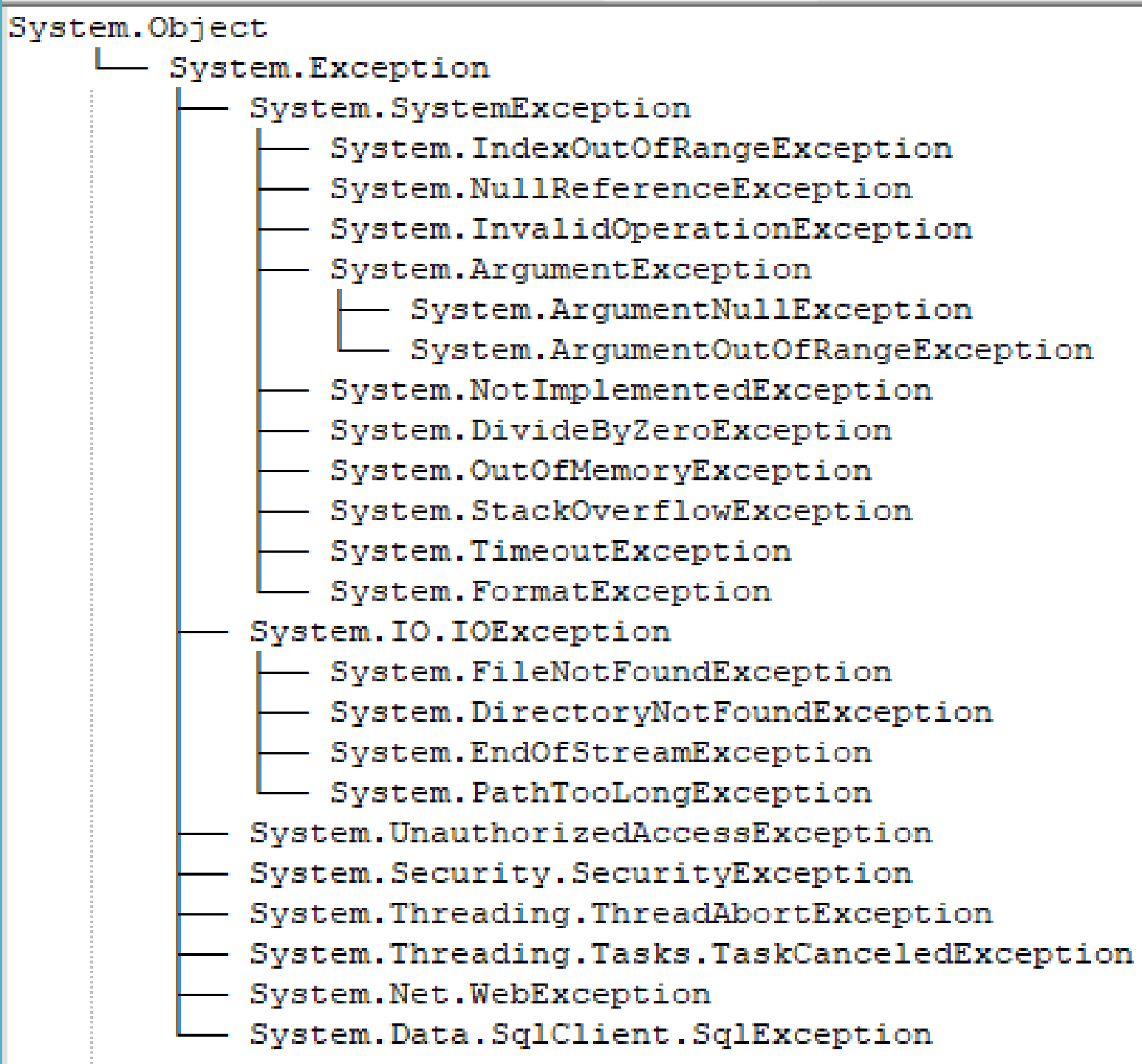
- ArgumentNullException: Передан нулевой аргумент.
- ArgumentOutOfRangeException: Аргумент находится вне допустимого диапазона.

IOException: Ошибки ввода-вывода.

TimeoutException: Операция не была завершена в течение заданного времени.

UnauthorizedAccessException: Отсутствуют необходимые права доступа.

Иерархия исключений



3. Обработка исключений. Блок try-catch-finally.

Обработка исключений в C# позволяет управлять непредвиденными ситуациями, которые могут возникнуть во время выполнения программы.

Это делается с помощью конструкции **try-catch-finally**, которая предоставляет механизм для перехвата, обработки и завершения выполнения блока кода в случае возникновения исключения.

Основные конструкции для обработки исключений

- **try**: Определяет блок кода, в котором могут возникнуть исключения.
- **catch**: Определяет блок кода, который выполняется, если в блоке **try** возникло исключение определенного типа.
- **finally**: Определяет блок кода, который выполняется всегда, независимо от того, произошло исключение или нет.

Как работает обработка исключений

- Выполнение блока **try**: Исполняется код внутри блока try.
- Возникновение исключения: Если в блоке **try** происходит ошибка, генерируется исключение.
- Поиск подходящего блока **catch**: Исполнение переходит к первому подходящему блоку **catch**, тип исключения которого соответствует типу сгенерированного исключения.
- Выполнение блока **catch**: Выполняется код внутри найденного блока **catch**.
- Выполнение блока **finally**: Если блок **finally** определен, то он выполняется всегда, независимо от того, было ли перехвачено исключение.

```
try
{
    int result = 10 / 0;
}
catch (DivideByZeroException ex)
{
    Console.WriteLine("Ошибка: Деление на ноль!");
}
catch (Exception ex)
{
    // Обработка всех остальных исключений
    Console.WriteLine($"Общее исключение: {ex.Message}");
}
```

ex - это переменная, которая содержит объект исключения. Этот объект содержит информацию о произошедшей ошибке, такую как:

- Сообщение об ошибке: ex.Message
- Тип исключения: ex.GetType()
- Дополнительная информация: другие свойства и методы, зависящие от конкретного типа исключения

Использование:

- Для вывода сообщения об ошибке пользователю: Console.WriteLine(ex.Message);
- Для логирования ошибки: Запись информации об ошибке в лог-файл.
- Выполнения других действий: Например, отправки уведомления администратору, отката транзакции и т.д.

```
try
{
    int result = 10 / 2;
}
catch (Exception ex)
{
    Console.WriteLine($"Исключение: {ex.Message}");
}
finally
{
    Console.WriteLine("Этот код выполняется всегда.");
}
```

В некоторых случаях необходимо передать управление вызывающему коду, **повторно** выбросив исключение. Это можно сделать с помощью оператора **throw**.

```
try
{
    // Код, который может вызвать исключение
}
catch (Exception ex)
{
    // Логирование исключения
    Console.WriteLine("Логирование исключения.");
    throw; // Повторное выбрасывание исключения
}
```

Ключевое слово **throw** используется в C# для явного создания и бросания исключения. Это позволяет программисту инициировать исключительную ситуацию в коде, когда определенные условия не выполняются или возникают ошибки, которые необходимо обработать.

```
Ссылок: 0
public static int Divide(int a, int b)
{
    if (b == 0)
    {
        throw new DivideByZeroException("Деление на ноль недопустимо");
    }
    return a / b;
}
```

Когда использовать throw:

- Проверка входных данных: Проверять корректность входных данных перед выполнением операции и бросать исключение в случае некорректных данных.
- Создание кастомных логических ошибок: Описывать ошибки, специфичные для вашего приложения, с помощью пользовательских исключений.
- Сигнализация о недопустимых состояниях: Сообщать о ситуациях, когда объект находится в недопустимом состоянии.

4. Создание пользовательских исключений.

В C# разработчики могут определять собственные пользовательские исключения, наследуя их от базового класса **System.Exception** или его производных. Это позволяет создавать более специфичные и осмысленные типы исключений, соответствующие предметной области приложения.

Чтобы определить пользовательское исключение, необходимо:

- Создать новый класс, наследуемый от System.Exception или его производного класса.
- Добавить конструкторы, которые позволят создавать экземпляры этого исключения.
- Добавить свойства или методы, которые могут предоставлять дополнительную информацию об исключении.

Например, необходимо создать пользовательское исключение, которое будет срабатывать, если пользователь введет отрицательный возраст или возраст больше 100.

```
namespace ConsoleApp
{
    Ссылок: 1
    public class InvalidAgeException : Exception
    {
        Ссылок: 0
        public InvalidAgeException(string message) : base(message)
        {
        }
    }
```

Разъяснение кода:

- **public class InvalidAgeException : Exception:** Мы создаем новый класс InvalidAgeException, который наследуется от базового класса Exception. Это означает, что наш новый класс также является исключением и может быть перехвачен в блоках try-catch.
- **public InvalidAgeException(string message) : base(message):** Конструктор класса принимает строку message, которая будет содержать сообщение об ошибке. Это сообщение передается базовому классу Exception для дальнейшей обработки.

Разъяснение кода:

- **public class InvalidAgeException : Exception:** Мы создаем новый класс InvalidAgeException, который наследуется от базового класса Exception. Это означает, что наш новый класс также является исключением и может быть перехвачен в блоках try-catch.
- **public InvalidAgeException(string message) : base(message):** Конструктор класса принимает строку message, которая будет содержать сообщение об ошибке. Это сообщение передается базовому классу Exception для дальнейшей обработки.

Заключение.

Лучшие практики обработки исключений

- Избегайте избыточного использования блоков try-catch: Обработка исключений не должна использоваться для управления логикой программы.
- Специфичность обработки: Перехватывайте только те исключения, которые вы можете корректно обработать.
- Логирование исключений: Важно сохранять информацию об исключениях для анализа и отладки.
- Использование finally: Используйте finally для освобождения ресурсов, таких как файлы, соединения с базой данных и т.д.
- Повторное выбрасывание исключений: Повторно выбрасывайте исключения, если их обработка невозможна на данном уровне.

Контрольные вопросы:

- Что такое исключения в C# и зачем они нужны?
- Чем отличаются ошибки от исключений?
- Приведите примеры ситуаций, когда могут возникнуть исключения в программе.
- Какие существуют основные категории исключений в C#?
- Какие типы исключений относятся к SystemException? Приведите примеры.
- Опишите, как работает блок try-catch-finally.
- Для чего используется блок finally?
- Как происходит поиск подходящего блока catch при возникновении исключения?
- Как можно повторно выбросить исключение? В каких случаях это может быть полезно?
- Как создать пользовательское исключение в C#?

Материалы лекций:

https://github.com/ShViktor72/Education2025