Исключения

1. Введение

Ошибки — неотъемлемая часть любого программирования. Даже если код компилируется без проблем, это **ещё не означает, что он будет работать корректно**. В реальных программах ошибки могут возникать по множеству причин — от неверного ввода данных пользователем до отказа оборудования или сети.

Существует три основных типа ошибок:

1. **Синтаксические** — возникают при нарушении правил языка программирования (например, забыли точку с запятой). Такие ошибки обнаруживаются **на этапе компиляции**, и программа просто не запустится.
2. **Логические** — когда код написан корректно, но выполняет не то, что задумывал программист (например, неверная формула). Эти ошибки **не вызывают исключений**, но приводят к неправильным результатам.
3. **Ошибки времени выполнения (runtime errors)** — возникают уже **во время работы программы**, когда происходят непредвиденные ситуации: деление на ноль, попытка обратиться к несуществующему элементу массива, открытие несуществующего файла и т. д.

Для обработки ошибок времени выполнения в C# существует специальный механизм — **исключения (exceptions)**. Исключения позволяют “поймать” ошибку, корректно её обработать и предотвратить аварийное завершение программы. Вместо того чтобы программа “падала”, мы можем аккуратно обработать проблему и, например, показать пользователю понятное сообщение.

Примеры типичных ситуаций, когда возникают исключения:

- Деление на ноль (`DivideByZeroException`);
- Обращение к объекту, равному `null` (`NullReferenceException`);
- Попытка выйти за границы массива (`IndexOutOfRangeException`);
- Ошибка при чтении или записи файла (`IOException`).

Механизм исключений — это **централизованный способ обработки ошибок**, который делает программы надёжнее и безопаснее. В этой лекции мы разберём, как работают исключения в C#, как их правильно использовать, а также как создавать собственные типы исключений для своих задач.

2. Обработка исключений

Когда во время выполнения программы возникает непредвиденная ситуация — например, деление на ноль, выход за границы массива или обращение к `null` — **в C# создаётся объект исключения** (`Exception`). Этот объект содержит всю информацию о произошедшей ошибке: её тип, сообщение и стек вызовов.

После возникновения исключения **нормальное выполнение программы прерывается**, и система начинает искать подходящий блок `catch`, который сможет обработать эту ошибку.

- Если подходящий обработчик найден — программа переходит к его выполнению.
- Если обработчик отсутствует — выполнение программы завершается с сообщением об ошибке.

Пример простейшей ситуации:

```
int a = 10;
int b = 0;
int c = a / b; // DivideByZeroException
Console.WriteLine("Эта строка не выполнится");
```

В этом случае программа прервётся с ошибкой “Attempted to divide by zero”. Чтобы программа не завершала работу, нужно **обернуть потенциально опасный код в блок** `try-catch`.

Конструкция `try-catch-finally`

```
try
{
    // Код, который может вызвать исключение
}
catch (Exception ex)
{
    // Обработка ошибки
    Console.WriteLine($"Произошла ошибка: {ex.Message}");
}
finally
{
    // Код, который выполнится в любом случае
    Console.WriteLine("Операция завершена");
}
```

Разберём назначение блоков:

- `try` — содержит “опасный” код, где может произойти ошибка.
- `catch` — перехватывает исключение и выполняет нужные действия (логирование, уведомление пользователя, завершение операции).
- `finally` — выполняется **всегда**, независимо от того, произошло исключение или нет. Обычно используется для освобождения ресурсов: закрытия файлов, сетевых соединений и т. д.

Пример: чтение файла с обработкой ошибок

```
try
{
    string text = File.ReadAllText("data.txt");
    Console.WriteLine(text);
}
catch (FileNotFoundException)
{
    Console.WriteLine("Файл не найден!");
}
catch (IOException ex)
{
    Console.WriteLine($"Ошибка при работе с файлом: {ex.Message}");
}
finally
{
    Console.WriteLine("Попытка чтения файла завершена.");
}
```

В этом примере, если файла `data.txt` нет, программа не “упадёт”, а выведет понятное сообщение пользователю и продолжит работу.

Таким образом, `try-catch-finally` — это **основной инструмент безопасного выполнения кода** в C#. Он позволяет изолировать потенциально опасные участки программы, корректно реагировать на ошибки и гарантировать освобождение ресурсов.

3. Иерархия исключений

В C# **все исключения являются объектами**, которые наследуются от базового класса `System.Exception`. Этот класс содержит общие свойства, доступные для всех ошибок:

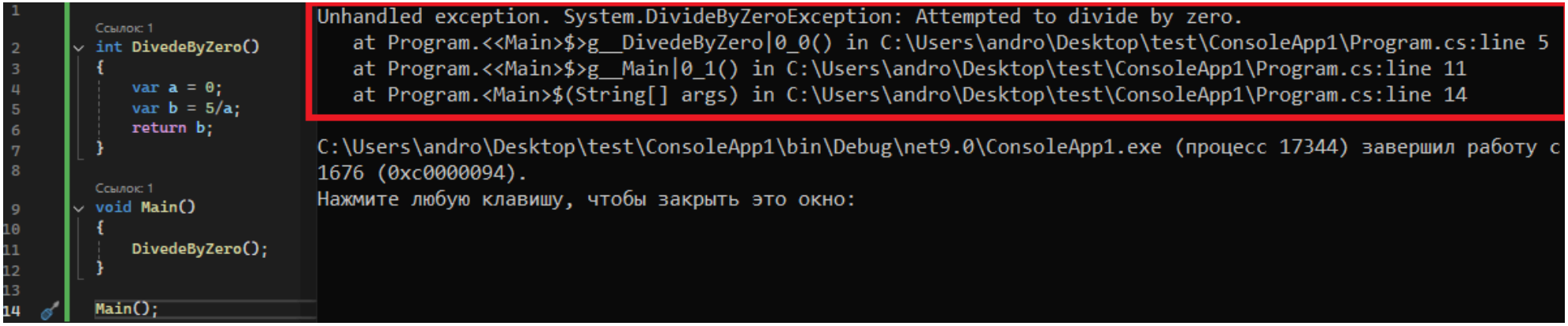
- `Message` — текстовое описание ошибки;
- `StackTrace` — стек вызовов, показывающий, где именно она произошла;
- `InnerException` — вложенная ошибка (если одна ошибка вызвала другую).

Базовая структура

```
System.Exception
│
├── System.SystemException
│    ├── System.NullReferenceException
│    ├── System.InvalidOperationException
│    ├── System.IndexOutOfRangeException
│    ├── System.DivideByZeroException
│    ├── System.IO.IOException
│    └── и т.д.
│
└── System.ApplicationException
     └── (пользовательские исключения)
```

Примеры часто используемых исключений

Исключение	Когда возникает	
`System.NullReferenceException`	Попытка обращения к объекту через `null`.	
`System.IndexOutOfRangeException`	Индекс массива или списка выходит за пределы.	
`System.InvalidOperationException`	Некорректное состояние объекта для выполнения операции.	
`System.IO.IOException`	Ошибка при работе с файлами или потоками ввода/вывода.	
`System.ArgumentException`	Метод получил некорректный аргумент.	
`System.ArgumentNullException`	Аргумент оказался `null`, когда этого быть не должно.	
`System.FormatException`	Ошибка преобразования формата (например, строки в число).	
`System.TimeoutException`	Операция заняла слишком много времени.	



Пример перехвата конкретного типа ошибки

```
try
{
    var text = File.ReadAllText("file.txt");
    int number = int.Parse(text);
}
catch (FileNotFoundException)
{
    Console.WriteLine("Файл не найден!");
}
catch (FormatException)
{
    Console.WriteLine("Ошибка формата данных в файле!");
}
catch (Exception ex)
{
    Console.WriteLine($"Неизвестная ошибка: {ex.Message}");
}
```

В этом примере программа **по-разному реагирует** на разные типы исключений.
Если файла нет — выводится одно сообщение, если ошибка формата — другое.
Все остальные ошибки перехватываются в общем `catch (Exception)`.

4. Пробрасывание исключений

Иногда задача программы — **не обработать ошибку полностью**, а **сообщить о ней выше по стеку вызовов**. В таких случаях используется оператор `throw`, который **пробрасывает исключение дальше** — в вызывающий метод, где его можно перехватить или завершить выполнение программы.

Пример пробрасывания исключения

```
void ProcessData()
{
    try
    {
        ReadFile();
    }
    catch (Exception ex)
    {
        Console.WriteLine("Ошибка при обработке данных");
        throw; // пробрасывает исключение дальше
    }
}

void ReadFile() => throw new FileNotFoundException("Файл не найден!");

try
{
    ProcessData();
}
catch (Exception ex)
{
    Console.WriteLine($"Главный обработчик: {ex.Message}");
}
```

Что произойдет:

1. В методе `ReadFile()` выбрасывается `FileNotFoundException`.
2. В `ProcessData()` исключение перехватывается, выводится сообщение, затем оно **повторно пробрасывается** (`throw;`).
3. Главный обработчик ловит ту же ошибку и выводит сообщение.

Разница между `throw;` и `throw ex;`

Запись	Что делает
`throw;`	Пробрасывает **то же исключение**, сохраняя оригинальный стек вызовов (где реально произошла ошибка).
`throw ex;`	Создаёт новое исключение, теряя исходный стек вызовов (будет казаться, что ошибка произошла в текущем месте).

Пример для наглядности

```
try
{
    Divide(10, 0);
}
catch (Exception ex)
{
    Console.WriteLine(ex.StackTrace);
}

void Divide(int a, int b)
{
    try
    {
        var result = a / b;
    }
    catch (Exception ex)
    {
        throw ex; // ❌ теряет исходный стек вызовов
    }
}
```

В этом случае `StackTrace` покажет только место, где выполнен `throw ex;`, а **не то, где произошло деление на ноль**. Если же использовать просто `throw;`, стек сохранится полностью, и будет видно реальное место возникновения ошибки.

Когда использовать

- Когда метод **не может сам обработать ошибку**, но должен **сообщить о ней выше**.
- Когда нужно **добавить контекст**, но не терять исходное исключение.

Пример добавления контекста:

```
try
{
    SaveUserData();
}
catch (IOException ex)
{
    throw new Exception("Ошибка при сохранении профиля пользователя", ex);
}
```

Здесь новое исключение содержит исходное (`InnerException`), что помогает при диагностике и логировании.

5. Создание собственных исключений

Иногда стандартных исключений недостаточно — особенно когда нужно описать **ошибку в бизнес-логике** приложения.

Например, если вы разрабатываете систему заказов, логично, что при попытке обработать несуществующего пользователя или оформить некорректный заказ должны возникать **понятные и осмысленные ошибки**, такие как `UserNotFoundException` или `InvalidOrderException`.

Создавайте пользовательские исключения, когда:

- Ошибка имеет **специфическое значение** для вашей предметной области.
- Стандартные типы исключений не отражают **суть проблемы**.
- Вы хотите сделать код более читаемым и поддерживаемым.

Например:

- `UserNotFoundException` — если пользователь не найден;
- `InvalidOrderException` — если заказ содержит некорректные данные;
- `InsufficientFundsException` — если на счёте недостаточно средств.

Как создавать собственное исключение

Для этого нужно сделать наследование от базового класса `System.Exception`.
Обычно достаточно переопределить конструктор и передать сообщение в базовый класс через `base(message)`.

```
class UserNotFoundException : Exception
{
    public UserNotFoundException(string username)
        : base($"Пользователь '{username}' не найден.") { }
}
class UserService
{
    private List<string> _users = new List<string> { "Alice", "Bob" };

    public void FindUser(string username)
    {
        if (!_users.Contains(username))
            throw new UserNotFoundException(username);

        Console.WriteLine($"Пользователь {username} найден!");
    }
}
try
{
    var service = new UserService();
    service.FindUser("Charlie");
}
catch (UserNotFoundException ex)
{
    Console.WriteLine($"Ошибка: {ex.Message}");
}
```

Рекомендации по созданию собственных исключений

1. **Наследуйтесь от** `Exception`, а не от других исключений, если только не создаёте их семантическое расширение.
2. Используйте **осмысленные имена**, оканчивающиеся на `Exception`.
3. При необходимости добавляйте **дополнительные свойства** для контекста ошибки, например:

```
class InvalidOrderException : Exception
{
    public int OrderId { get; }

    public InvalidOrderException(int orderId, string message)
        : base(message)
    {
        OrderId = orderId;
    }
}
```

4. Не злоупотребляйте пользовательскими исключениями — создавайте их только тогда,
   когда это действительно улучшает читаемость и структуру кода.

Таким образом, **собственные исключения делают код самодокументируемым** — по имени исключения сразу видно, что пошло не так, без необходимости читать внутреннюю логику программы.

6. Практическое использование

В реальных программах исключения чаще всего возникают при **работе с внешними ресурсами** — файлами, сетью, базами данных, пользовательским вводом и т. д.
Такие операции часто выходят из-под контроля программы (например, файл может отсутствовать, сервер быть недоступен или пользователь ввести некорректные данные).

Пример: обработка исключений при работе с файлами

```
try
{
    string path = "data.txt";
    string text = File.ReadAllText(path);
    Console.WriteLine($"Содержимое файла: {text}");
}
catch (FileNotFoundException ex)
{
    Console.WriteLine($"Файл не найден: {ex.FileName}");
}
catch (IOException ex)
{
    Console.WriteLine($"Ошибка ввода-вывода: {ex.Message}");
}
catch (Exception ex)
{
    Console.WriteLine($"Неизвестная ошибка: {ex.Message}");
}
finally
{
    Console.WriteLine("Попытка чтения файла завершена.");
}
```

Здесь используется несколько `catch`-блоков:

- `FileNotFoundException` — если файл отсутствует;
- `IOException` — если произошла общая ошибка ввода-вывода;
- `Exception` — “страховка” на случай непредвиденных ошибок.

Блок `finally` выполнится **всегда**, даже если в `try` произошла ошибка — например, чтобы закрыть файл или освободить ресурсы.

Пример: ввод данных пользователя

```
try
{
    Console.Write("Введите число: ");
    int number = int.Parse(Console.ReadLine());
    Console.WriteLine($"Квадрат числа: {number * number}");
}
catch (FormatException)
{
    Console.WriteLine("Ошибка: введено не число!");
}
catch (OverflowException)
{
    Console.WriteLine("Ошибка: число слишком большое!");
}
```

Если пользователь введёт строку `"abc"`, программа не упадёт, а аккуратно сообщит об ошибке.

Логирование ошибок

Важно не просто “поймать” исключение, но и **зафиксировать детали** для диагностики.
Для этого можно вывести.

- `ex.Message` — сообщение об ошибке;
- `ex.StackTrace` — стек вызовов (где именно произошла ошибка);
- `ex.InnerException` — вложенное исключение, если ошибка была результатом другой ошибки.

```
try
{
    ProcessFile("report.csv");
}
catch (Exception ex)
{
    Console.WriteLine("Произошла ошибка:");
    Console.WriteLine($"Сообщение: {ex.Message}");
    Console.WriteLine($"Стек вызовов: {ex.StackTrace}");
    if (ex.InnerException != null)
        Console.WriteLine($"Внутреннее исключение: {ex.InnerException.Message}");
}
```

Также можно использовать `Environment.StackTrace`, чтобы получить текущий стек вызовов даже без исключения.

7. Исключения и производительность

Исключения — это **инструмент обработки ошибок**, а не способ управлять логикой программы. Их следует использовать **только в действительно исключительных ситуациях**, а не вместо обычных проверок условий. Каждое исключение в .NET — это полноценный объект, который создаётся и заполняется большим количеством служебной информации (включая стек вызовов, типы, сообщения, внутренние исключения). Создание и обработка исключения — **дорогая операция по времени и памяти**. Если выбрасывать и ловить исключения часто (например, в цикле), это серьёзно снизит производительность программы.

Пример неправильного использования

```
int sum = 0;
string[] numbers = { "1", "2", "abc", "4" };

foreach (var n in numbers)
{
    try
    {
        sum += int.Parse(n); // выбросит FormatException на "abc"
    }
    catch
    {
        // Ошибку просто игнорируем
    }
}

Console.WriteLine($"Сумма: {sum}");
```

Здесь программа **использует исключение как способ проверки**, можно ли строку преобразовать в число. Для каждой ошибки создаётся объект `FormatException`, что очень затратно.

Правильный подход — использовать TryParse

```
int sum = 0;
string[] numbers = { "1", "2", "abc", "4" };

foreach (var n in numbers)
{
    if (int.TryParse(n, out int value))
        sum += value;
}

Console.WriteLine($"Сумма: {sum}");
```

Этот код делает то же самое, но **без создания исключений**. `TryParse()` просто возвращает `true` или `false`, что в десятки раз быстрее, чем `try-catch`.

Общие рекомендации

- Не используйте исключения для **управления потоком программы** (например, как “альтернативу if”).
- В циклах и часто вызываемых методах **проверяйте данные заранее**.
- Исключения должны сигнализировать о **непредсказуемых ошибках** — сбое сети, отсутствии файла, повреждённых данных и т.п.
- Если ошибка может быть ожидаема, используйте `Try...`-методы (`TryParse`, `TryGetValue`, `TryRead` и т.д.).

Практическое задание

Что нужно сделать:

1. Обработать все потенциальные ошибки ввода
   Проверить и обезопасить:

   - ввод индексов задач (`delete` , `update` , `status` , `read`);
   - ввод числовых значений (`top` , `BirthYear`);
   - ввод дат (`search --from` , `--to`);
   - ввод пустых строк;
   - обращение к несуществующему профилю;
   - повторную регистрацию с уже существующим логином.

   Использовать:

   - `int.TryParse`
   - `DateTime.TryParse`
   - проверки `null`
   - проверки диапазонов индексов

   Программа не должна завершаться с `Unhandled Exception` .

2. Добавить глобальный try-catch в основной цикл

   В `Program.cs` основной цикл обработки команд должен быть обёрнут:

```
try
{
    var command = CommandParser.Parse(input);
    command.Execute();
}
catch (Exception ex)
{
    Console.WriteLine($"Ошибка: {ex.Message}");
}
```

   Запрещено оставлять пустые `catch`.

3. Создать собственные классы исключений

   Создать отдельную папку `Exceptions` и реализовать собственные типы ошибок:

   - `InvalidCommandException`
   - `InvalidArgumentException`
   - `TaskNotFoundException`
   - `ProfileNotFoundException`
   - `AuthenticationException`
   - `DuplicateLoginException`

   Пример:

```
public class TaskNotFoundException : Exception
{
    public TaskNotFoundException(string message) : base(message) { }
}
```

4. Использовать пользовательские исключения в логике программы
   Вместо:

```
Console.WriteLine("Неверный индекс");
```

   Нужно:

```
throw new TaskNotFoundException("Задача с таким индексом не существует.");
```

   И перехватывать её в основном цикле.
5. Добавить проверку бизнес-логики
   Программа должна выбрасывать исключения если:
   - пользователь не авторизован, но пытается работать с задачами;
   - выполняется `undo`, когда стек пуст;
   - выполняется `redo`, когда стек пуст;
   - команда `search` получает некорректный формат даты;
   - используется неизвестный флаг;
   - вызывается команда, не зарегистрированная в словаре обработчиков.

6. Разделить обработку исключений
   Добавить отдельные блоки:

```
catch (TaskNotFoundException ex)
{
    Console.WriteLine($"Ошибка задачи: {ex.Message}");
}
catch (AuthenticationException ex)
{
    Console.WriteLine($"Ошибка авторизации: {ex.Message}");
}
catch (InvalidCommandException ex)
{
    Console.WriteLine($"Ошибка команды: {ex.Message}");
}
```

И только в конце:

```
catch (Exception ex)
{
    Console.WriteLine("Неожиданная ошибка.");
}
```