Лекция №12. Обработка исключений. Конструкция try … catch … finally.

ОГЛАВЛЕНИЕ

§12.1 Понятие исключения

Исключение — это событие при выполнении программы, которое приводит к её ненормальному или неправильному поведению.

Существует два вида исключений:

1. **Аппаратные** (*структурные, SE-Structured Exception*), которые генерируются процессором. К ним относятся, например:
   - Деление на 0.
   - Выход за границы массива.
   - Обращение к невыделенной памяти.
   - Переполнение разрядной сетки.
2. **Программные**, генерируемые операционной системой и прикладными программами – возникают тогда, когда программа их явно инициирует. Когда встречается аномальная ситуация, та часть программы, которая ее обнаружила, может сгенерировать, или возбудить, исключение.

Механизм структурной обработки исключений позволяет однотипно обрабатывать как программные, так и аппаратные исключения.

§12.2 Иерархия исключений в .NET

В языке **C#** и платформе **.NET** обработка ошибок на этапе выполнения

программы осуществляется с помощью механизма **исключений** — специальных объектов, представляющих собой сигналы о возникновении аварийных или нештатных ситуаций. Все исключения в **.NET** основаны на единой иерархии классов, что обеспечивает единообразие в их обработке и позволяет строить гибкие и надёжные системы управления ошибками.

Корнем всей иерархии исключений является абстрактный класс **System.Exception** — базовый класс, от которого напрямую или косвенно наследуются все типы исключений в **.NET**. Этот класс определяет общие свойства и методы, присущие любому исключению:

- **Message** — текстовое описание ошибки;
- **StackTrace** — трассировка стека вызовов, где произошло исключение;
- **InnerException** — ссылка на вложенное исключение (если текущее исключение было вызвано другим);
- **Source** — имя приложения или объекта, вызвавшего исключение;
- **HelpLink** — ссылка на документацию по ошибке (опционально).

Создавать экземпляры **Exception** напрямую не рекомендуется — вместо этого используются более специализированные производные классы, отражающие конкретную природу ошибки.

В **.NET** принято деление исключений на две основные ветви:

1. **System.SystemException** — базовый класс для всех системных исключений, генерируемых самой средой выполнения **.NET** при возникновении фундаментальных ошибок (например, *переполнение стека, ошибки памяти, проблемы с типами* и т.д.). Эти исключения, как правило, указывают на серьёзные проблемы в работе среды или нарушения контрактов языка.

2. **System.ApplicationException** — изначально предназначался для пользовательских или прикладных исключений, создаваемых разработчиками. Однако, начиная с **.NET Framework 2.0**, рекомендация **Microsoft** заключается в том, чтобы не использовать **ApplicationException** как базовый класс для

своих исключений. Вместо этого рекомендуется наследоваться непосредственно от **Exception** или от более подходящих стандартных классов. Таким образом, **ApplicationException** считается устаревшим в современных приложениях.

Рассмотрим наиболее распространённые классы исключений, унаследованные от **Exception**, и их семантическое значение:

1. System.ArgumentException — выбрасывается при передаче недопустимого аргумента в метод.

```
1. if (age < 0)
2.     throw new ArgumentException("Ошибка возраста");
```

От него наследуются:

- **ArgumentNullException** — если аргумент равен **null**.
- **ArgumentOutOfRangeException** — если значение аргумента выходит за допустимые границы (например, индекс за пределами массива).

2. System.NullReferenceException — возникает при попытке доступа к члену объекта, когда ссылка равна **null**.

```
1. string str = null;
2. // Исключение:
3. // Object reference not set to an instance of an object.
4. int len = str.Length;
```

3. System.IndexOutOfRangeException — выбрасывается при попытке доступа к элементу массива или коллекции по индексу, выходящему за границы допустимого диапазона.

```
1. int[] arr = new int[3];
2. Console.WriteLine(arr[5]); // Исключение
```

4. System.InvalidOperationException — указывает, что операция недопустима в текущем состоянии объекта. Например, попытка вызвать *MoveNext()* у завершённого **IEnumerator**.

5. System.FormatException — возникает при попытке преобразования строки в числовой тип, если формат строки некорректен.

```
1. int number = int.Parse("abc"); // Исключение
```

6. System.IO.FileNotFoundException — частный случай

System.IO.IOException, возникает при попытке открыть файл, который не существует.

```
using (var file = File.OpenRead("nonexistent.txt"))
{
    // ...
}
// Исключение: Could not find file ...
```

7. **System.DivideByZeroException** — выбрасывается при попытке целочисленного деления на ноль.

```
int result = 5 / 0; // Исключение
```

Важно помнить, что для операций с **double** деление на ноль не вызывает исключение, а возвращает **Infinity** или **NaN**.

Иерархия исключений в **.NET** представляет собой древовидную структуру, где каждый более специфичный класс уточняет тип ошибки. Пример упрощённого дерева:

```
System.Exception
├── System.SystemException
│   ├── System.ArgumentException
│   │   ├── ArgumentNullException
│   │   └── ArgumentOutOfRangeException
│   ├── System.NullReferenceException
│   ├── System.IndexOutOfRangeException
│   ├── System.DivideByZeroException
│   ├── System.InvalidOperationException
│   └── System.IO.IOException
│       └── System.IO.FileNotFoundException
└── (пользовательские исключения)
```

Эта иерархия позволяет использовать полиморфизм при обработке исключений. Например, можно перехватить **ArgumentException**, и он будет ловить не только базовый тип, но и все его производные — **ArgumentNullException**, **ArgumentOutOfRangeException** и т.д.

Понимание иерархии исключений критически важно при проектировании **catch-блоков**. Порядок обработки должен быть от более специфичных типов к более общим, иначе компилятор выдаст ошибку о недостижимом кодом.

Пример корректной обработки:

```
try
{
    int[] arr = new int[3];
    arr[5] = 42;
}
catch (IndexOutOfRangeException ex)
{
    Console.WriteLine($"Ошибка: {ex.Message}");
}
catch (Exception ex)
{
    Console.WriteLine($"Ошибка: {ex.Message}");
}
```

Если поменять местами блоки catch, компилятор сообщит: *A previous catch clause already catches all exceptions of this type* — потому что **Exception** «*перекрывает*» все остальные.

§12.3 Конструкция try ... catch ... finally

Одним из ключевых механизмов обеспечения устойчивости и надёжности программ в языке **C#** является система обработки исключений, реализованная с помощью конструкции **try ... catch ... finally**. Эта конструкция позволяет программе перехватывать и обрабатывать ошибки.

Программа может столкнуться с множеством проблем во время выполнения: деление на ноль и т.п. Без обработки такие ошибки приводят к аварийному завершению программы с выводом трассировки стека (*stack trace*).

Конструкция **try-catch-finally** позволяет:

- Выполнять потенциально опасный код в защищённой среде.
- Перехватывать возникающие исключения.
- Обрабатывать ошибки (логировать, сообщать пользователю, пытаться исправить ситуацию).
- Гарантированно выполнять критически важные действия, например, освобождение ресурсов.

Базовая структура выглядит следующим образом:

```
try
{
    // Код, который может выбросить исключение
}
catch (ExceptionType ex)
{
    // Обработчик исключения определённого типа
}
finally
{
    // Код, который выполнится в любом случае
}
```

- Блок **try** — обязательный, содержит код, который потенциально может вызвать исключение.
- Блок **catch** — опциональный, может быть один или несколько. Перехватывает исключения указанного типа.
- Блок **finally** — опциональный, но крайне полезный. Выполняется всегда, независимо от того, было ли исключение и было ли оно обработано.

При использовании конструкции **try-catch-finally** важно помнить:

- **try** должен сопровождаться хотя бы одним **catch** или **finally**.
- Нельзя использовать **catch** или **finally** без **try**.
- **catch** и **finally** могут использоваться вместе или по отдельности.

Рассмотрим простой пример, где пользователь вводит делитель:

```
Console.Write("Введите делитель: ");
string input = Console.ReadLine();

try
{
    int divisor = int.Parse(input);
    int result = 100 / divisor;
    Console.WriteLine($"Результат: {result}");
}
catch (FormatException)
{
    Console.WriteLine("Ошибка: введено не число.");
}
catch (DivideByZeroException)
{
    Console.WriteLine("Ошибка: деление на ноль!");
}
```

Здесь:

- Если пользователь ввёл не число — выбросится **FormatException**, который будет перехвачен первым **catch**.
- Если ввёл **0** — сработает **DivideByZeroException** и второй **catch**.
- Если всё в порядке — исключений не будет, и выполнится только **try**.

Блок **finally** — это гарантированный участок кода, который выполняется в любом случае:

- Было ли исключение.
- Было ли оно обработано.
- Даже если в **try** или **catch** был оператор **return**, **break** или **goto**.

Пример: освобождение ресурсов

```
 1. FileStream file = null;
 2. try
 3. {
 4.     file = File.OpenRead("data.txt");
 5.     byte[] b = new byte[1024];
 6.     int bRead = file.Read(buffer, 0, b.Length);
 7.     Console.WriteLine($"Прочитано байт: {bRead}");
 8. }
 9. catch (FileNotFoundException)
10. {
11.     Console.WriteLine("Файл не найден.");
12. }
13. catch (UnauthorizedAccessException)
14. {
15.     Console.WriteLine("Нет доступа к файлу.");
16. }
17. finally
18. {
19.     file?.Close(); // Закрываем файл в любом случае
20.     Console.WriteLine("Файл закрыт.");
21. }
```

Даже если **FileStream** не был создан (например, файл не найден), **finally** всё равно выполнится, и *file?.Close()* безопасно обработает **null**.

Рассмотрим, как ведёт себя **finally**, если в **try** или **catch** есть **return** при вызове *Divide(10, 0)*:

```
 1. static int Divide(int a, int b)
 2. {
 3.     try
 4.     {
 5.         return a / b;
 6.     }
 7.     catch (DivideByZeroException)
 8.     {
 9.         return -1;
10.     }
11.     finally
12.     {
13.         Console.WriteLine("Блок finally!!");
14.     }
15. }
```

Сначала возникает исключение **DivideByZeroException** и возвращение значения «*-1*», но перед тем, как значение вернется, выполнится блок **finally**.

Иногда достаточно просто знать, что исключение произошло, без анализа его деталей:

```
1. try
2. {
3.     riskyOperation();
4. }
5. catch (IOException)  // Без параметра
6. {
7.     Console.WriteLine("Ошибка ввода-вывода.");
8. }
```

Но если нужен доступ к **Message**, **StackTrace** и т.д. — параметр обязателен:

```
1. catch (IOException ex)
2. {
3.     Console.WriteLine($"Ошибка: {ex.Message}");
4. }
```

Так же важно понимать, что использование пустого **catch** в разработке является **антипаттерном**.

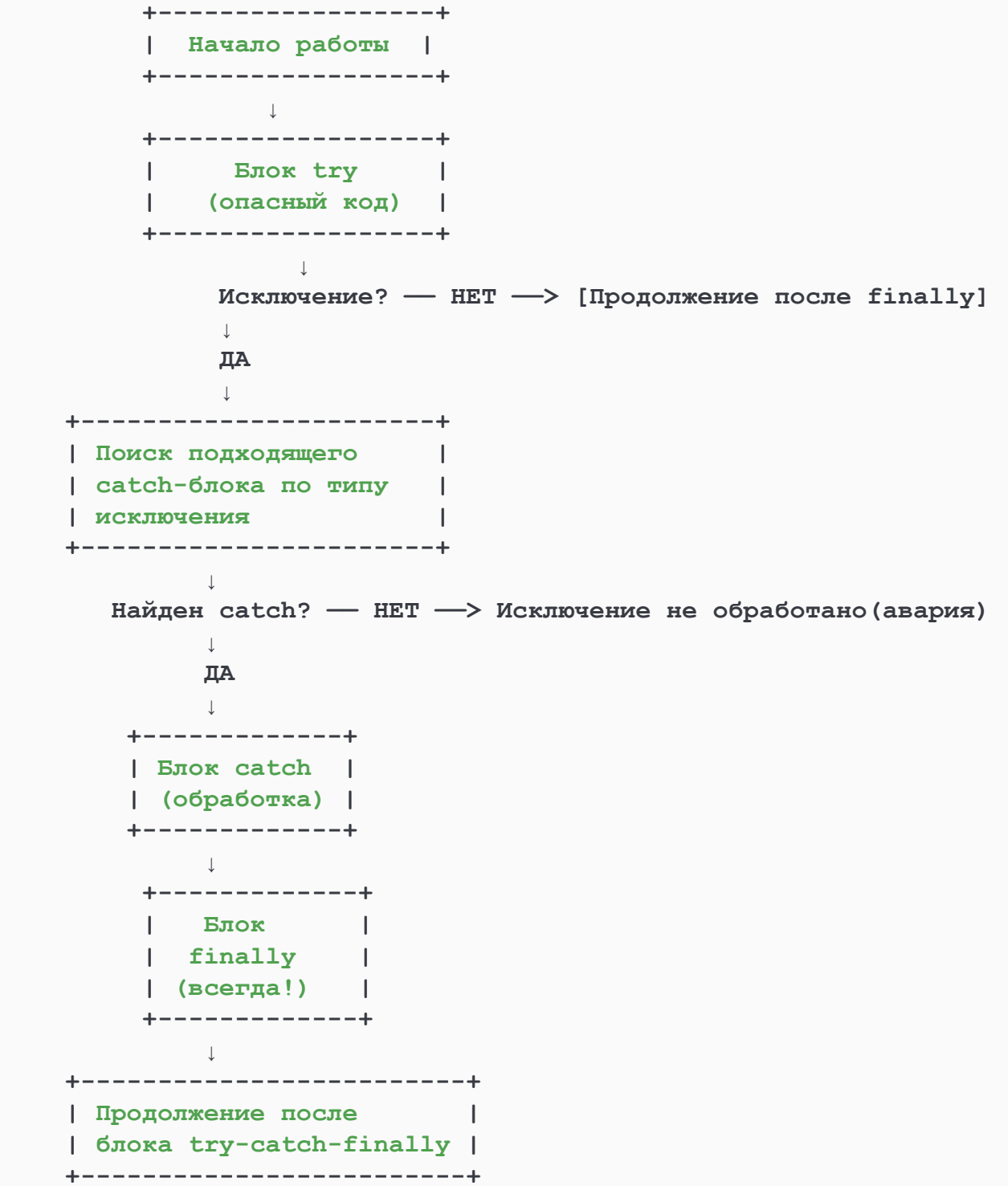
```
1. try { ... }
2. // "проглатывание" исключения — плохо!
3. catch (Exception) { }
```

Такой код скрывает ошибки, мешает отладке и нарушает принципы

надёжности. Если вы не можете обработать исключение — лучше не ловить его, либо записать в лог:

```
1. catch (Exception ex)
2. {
3.     Debug.WriteLine($"Ошибка: {ex}");
4.     // или: Logger.LogError(ex);
5. }
```

Схема поведения конструкции **try-catch-finally** представлена ниже:



§12.4 Множественные блоки catch

Хотя многие ошибки могут быть обработаны универсальным способом, на практике разные типы исключений часто требуют различной реакции. Например, ошибка ввода-вывода (например, файл не найден) — это не то же самое, что попытка преобразовать строку «abc» в число или деление на ноль. Чтобы обеспечить точечную и адекватную реакцию на разные виды сбоев, в **C#** предусмотрена возможность использования нескольких блоков **catch** в одной конструкции **try-catch**.

Такая структура позволяет программе анализировать тип исключения и выполнять специализированную обработку для каждого из них.

Конструкция выглядит следующим образом:

```
try
{
    // Потенциально опасный код
}
catch (Type1 ex)
{
    // Обработка исключения типа Type1
}
catch (Type2 ex)
{
    // Обработка исключения типа Type2
}
catch (Exception ex)
{
    // Обработка всех остальных исключений
}
finally
{
    // Код, выполняющийся всегда (опционально)
}
```

Каждый **catch**-блок указывает:

- Тип исключения, которое он может перехватить;
- Необязательное имя переменной (обычно **ex**), через которую можно получить доступ к деталям исключения: сообщению, стеку вызовов, внутреннему исключению и т.д.

Рассмотрим программу, которая запрашивает у пользователя возраст и

пытается преобразовать его в число:

```
 1. Console.Write("Введите ваш возраст: ");
 2. string input = Console.ReadLine();
 3.
 4. try
 5. {
 6.     int age = int.Parse(input);
 7.
 8.     if (age < 0)
 9.         throw new ArgumentException("Ошибка возраста!");
10.
11.     Console.WriteLine($"Ваш возраст: {age}");
12. }
13. catch (FormatException)
14. {
15.     Console.WriteLine("Ошибка: вы ввели не число.");
16. }
17. catch (OverflowException)
18. {
19.   Console.WriteLine("Ошибка: число выходит за рамки int.");
20. }
21. catch (ArgumentException ex)
22. {
23.     Console.WriteLine($"Ошибка: {ex.Message}");
24. }
25. catch (Exception ex)
26. {
27.     Console.WriteLine($"Ошибка: {ex.Message}");
28. }
```

Разберём возможные сценарии:

Ввод пользователя	Исключение	Обработчик
"abc"	FormatException	Первый catch
"9999999999"	OverflowException	Второй catch
"-5"	ArgumentException	Третий catch
Любая другая ошибка	Exception	Четвертый catch

Исключение **ArgumentException** выбрасывается явно с помощью оператора **throw**, что демонстрирует, как программист может инициировать исключение вручную при нарушении логики.

Поскольку все исключения наследуются от **System.Exception**, а многие

— от более узких базовых классов (например, *SystemException, IOException*), механизм **catch** использует полиморфизм.

Например:

1. **FileNotFoundException** наследуется от **IOException**, а тот — от **Exception**.

2. Если вы напишете **catch** (*IOException ex*), он перехватит все исключения, унаследованные от **IOException**, включая:

- DirectoryNotFoundException.
- DriveNotFoundException.
- EndOfStreamException.
- PathTooLongException.

Это позволяет группировать обработку связанных по смыслу ошибок.

§12.5 Блок finally и его назначение

Блок **finally** — это часть конструкции **try-catch-finally**, который выполняется всегда, независимо от того, было ли исключение, было ли оно перехвачено или программа завершила **try** или **catch** штатно, с оператором **return**, **break**, **goto** или даже при возникновении необработанного исключения.

Основное назначение **finally** — гарантированное освобождение ресурсов и выполнение критически важных действий, даже если произошла ошибка.

К таким действиям относятся:

- Закрытие файлов.
- Закрытие соединений с базой данных.
- Освобождение неуправляемых ресурсов (память, дескрипторы).
- Сброс состояния объектов.
- Логирование завершения операции.

Для наглядности работы блока **finally** рассмотрим код, который работает с файловой системой.

Данный код представлен ниже:

```
 1. FileStream file = null;
 2. try
 3. {
 4.     file = File.OpenRead("data.txt");
 5.     byte[] buffer = new byte[1024];
 6.     file.Read(buffer, 0, buffer.Length);
 7. }
 8. catch (FileNotFoundException)
 9. {
10.     Console.WriteLine("Файл не найден.");
11. }
12. finally
13. {
14.     file?.Close();  // Выполнится всегда
15.     Console.WriteLine("Ресурсы освобождены.");
16. }
```

Даже если файл не найден и сработал **catch**, fiinally всё равно выполнится — файл будет закрыт, и программа не оставит «утечек» ресурсов.

Для объектов, реализующих интерфейс **IDisposable** (например, *FileStream, StreamReader, SqlConnection*), вместо ручного написания **finally** лучше использовать оператор **using**:

```
1. using (var reader = new StreamReader("data.txt"))
2. {
3.     Console.WriteLine(reader.ReadToEnd());
4. }
5. // reader.Dispose()- автоматически — как в finally
```

§12.6 Фильтры исключений (Exception Filters)

Несмотря на мощь и гибкость конструкции **try-catch**, в ряде ситуаций стандартной проверки по типу исключения оказывается недостаточно. Иногда требуется принять решение о том, следует ли обрабатывать исключение, не только на основе его типа, но и по дополнительным условиям — например, по значению поля, коду ошибки, контексту выполнения или сообщению.

Именно для таких случаев в **C#** с версии **6.0** был введён механизм **фильтров исключений** (*exception filters*), позволяющий условно активировать блок **catch** с помощью логического выражения.

Фильтры исключений расширяют стандартную семантику **catch**, позволяя более тонко управлять потоком обработки ошибок без

необходимости повторного выбрасывания исключений и обеспечивая сохранение оригинальной трассировки стека — важное преимущество по сравнению с альтернативными подходами.

Фильтр добавляется к блоку **catch** с помощью ключевого слова **when**, за которым следует логическое выражение, возвращающее значение типа **bool**:

```
1. try
2. {
3.     // Опасный код
4. }
5. catch (ExceptionType ex) when (условие)
6. {
7.     // Обработчик выполнится ТОЛЬКО если условие истинно
8. }
```

Выражение в **when** может включать:

- Поля исключения (ex.Message, ex.HResult).
- Внешние переменные.
- Вызовы методов.
- Сложные логические конструкции.

Допустим, мы работаем с **API**, которое возвращает разные ошибки по одному и тому же типу исключения, но с разными сообщениями:

```
 1. try
 2. {
 3.     SimulateApiCall();
 4. }
 5. catch (HttpRequestException ex)
 6.    when (ex.Message.Contains("404"))
 7. {
 8.     Console.WriteLine("Ресурс не найден.");
 9. }
10. catch (HttpRequestException ex)
11.    when (ex.Message.Contains("500"))
12. {
13.     Console.WriteLine("Внутренняя ошибка сервера.");
14. }
15. catch (HttpRequestException ex)
16. {
17.     Console.WriteLine($"Ошибка запроса: {ex.Message}");
18. }
```

Здесь:

- Первый catch сработает только при HTTP 404.
- Второй — при 500.
- Третий — перехватит все остальные HttpRequestException, не отвечающие условиям.

Без фильтров пришлось бы использовать **if** внутри **catch** и, возможно, **throw** — что привело бы к потере стека.

Фильтры могут использовать внешние переменные, что позволяет адаптировать поведение в зависимости от режима работы приложения:

```
 1. bool isDebugMode = true;
 2.
 3. try
 4. {
 5.     CriticalOperation();
 6. }
 7. catch (Exception ex) when (!isDebugMode)
 8. {
 9.     Console.WriteLine("Ошибка в production.");
10. }
11. catch (Exception ex)
12. {
13.     Console.WriteLine($"Подробности ошибки: {ex}");
14. }
```

В данном случае в режиме отладки исключение не будет перехвачено первым **catch**, так как условие «*!isDebugMode*» ложно, и управление перейдёт ко второму блоку, где будет выведена подробная информация. В *production* же — ошибка будет обработана тихо.

§12.7 Генерация исключений: оператор throw

Механизм обработки исключений в **C#** не ограничивается лишь перехватом ошибок, возникающих автоматически во время выполнения программы. Программист также может явно инициировать исключение с помощью оператора **throw**. Это важный инструмент для обеспечения целостности данных, валидации входных параметров, сигнализации о нарушении контрактов и передачи управления вверх по стеку вызовов при

обнаружении недопустимого состояния.

Оператор **throw** позволяет выбросить экземпляр любого класса, унаследованного от **System.Exception**, и тем самым прервать нормальный ход выполнения, активировав механизм **catch** на соответствующем уровне стека.

Оператор **throw** имеет простой синтаксис:

```
throw new ArgumentException("Некорректный аргумент.");
```

Он может использоваться в любом месте, где допустимо выполнение кода, за исключением выражений. Обычно **throw** применяется:

- В начале метода для проверки аргументов.
- При обнаружении недопустимого состояния объекта.
- Для сигнализации о невозможности выполнить операцию.

Пример — валидация параметра:

```
1. public void SetAge(int age)
2. {
3.     if (age < 0 || age > 150)
4.         throw new ArgumentOutOfRangeException("age",
5.                                 "Возраст от 0 до 150.");
6.
7.     _age = age;
8. }
```

Здесь выбрасывается **ArgumentOutOfRangeException**, указывающий не только на ошибку, но и на имя параметра, что крайне полезно для отладки.

Одна из распространённых ошибок при генерации исключений — жёсткая привязка к строковым литералам, представляющим имена параметров:

```
throw new ArgumentException("Значение в минусе?", "age");
```

Если в будущем имя параметра изменится, а строка останется прежней, диагностическая информация станет некорректной. Чтобы избежать этого, в **C#** рекомендуется использовать оператор **nameof**.

Оператор **nameof** возвращает имя символа (переменной, параметра, поля) в виде строки во время компиляции, обеспечивая типобезопасность и поддержку рефакторинга:

```
1.public void ProcessValue(int count)
2.{
3. if (count <= 0)
4.  throw new ArgumentException("Количество в минусе!",
5.                              nameof(count));
6.}
```

Если позже параметр **count** будет переименован, например, в **itemCount**, среда разработки (например, *Visual Studio*) автоматически обновит *nameof(count)* → *nameof(itemCount)*, и сообщение об ошибке останется корректным.

В блоке **catch** иногда возникает необходимость обработать исключение, но затем передать его дальше по стеку. Здесь критически важно использовать правильный синтаксис.

Рассмотрим два подхода:

```
 1. catch (FormatException ex)
 2. {
 3.     Logger.Log("Ошибка формата: " + ex.Message);
 4.     throw; // ☑ Правильно
 5. }
 6.
 7. catch (FormatException ex)
 8. {
 9.     Logger.Log("Ошибка формата: " + ex.Message);
10.     throw ex; // ✖ Опасно!
11. }
```

Разница принципиальна:

- **throw**; — повторно выбрасывает текущее исключение с сохранением оригинальной трассировки стека.
- **throw** ex; — создаёт видимость повторного выброса, но обнуляет стек вызовов, начиная с текущего метода. Это затрудняет отладку, так как теряется информация о месте первоначального возникновения ошибки.

§12.8 Обработка исключений и управление потоком программы

Хотя механизм исключений в **C#** предоставляет мощные средства для реагирования на ошибки времени выполнения, его нельзя использовать как замену стандартным конструкциям управления потоком программы, таким

как **if**, **switch**, **while** или **try**.

Использование исключений для организации логики выполнения — это серьёзное нарушение принципов проектирования, которое приводит к снижению производительности, ухудшению читаемости кода и затруднению отладки. В этом разделе мы рассмотрим, в чём принципиальная разница между обработкой исключительных ситуаций и управлением нормальным ходом программы, и почему смешение этих понятий является **антипаттерном**.

Исключения предназначены для обработки исключительных, аварийных или непредвиденных ситуаций, таких как:

- Попытка открыть несуществующий файл;
- Деление на ноль в целочисленной арифметике;
- Нарушение контракта метода (например, передача null туда, где ожидается объект).

Они не должны использоваться для управления ожидаемым поведением программы, даже если это поведение связано с ошибкой. Например, попытка преобразовать строку в число — это нормальная операция, и возможность ввода некорректных данных — ожидаемый сценарий, а не исключительная ситуация.

Рассмотрим типичный **антипаттерн**:

```
1. int number;
2. try
3. {
4.     number = int.Parse(input);
5. }
6. catch (FormatException)
7. {
8.     number = 0;
9. }
```

Здесь исключение используется как средство ветвления: «*если не получилось распарсить — установить значение по умолчанию*». Но *int.Parse* выбрасывает **FormatException** всегда, когда строка не является корректным числом — это не ошибка, а нормальный результат валидации. Поэтому использование **try-catch** здесь — избыточное и неэффективное.

Правильный подход — использовать метод, предназначенный для безопасного преобразования:

```
1. if (int.TryParse(input, out int number))
2. {
3.     Console.WriteLine($"Число: {number}");
4. }
5. else
6. {
7.     Console.WriteLine("Некорректный ввод.");
8. }
```

Метод *int.TryParse()* возвращает **bool**, сигнализируя об успехе или неудаче, и не выбрасывает исключений. Это делает код быстрее, читаемее, производительнее.

Исключения оправданы, когда:

- Происходит настоящая ошибка, которую нельзя предсказать или проверить заранее.
- Операция не может быть выполнена по объективным причинам (например, сбой диска, потеря сети).
- Нарушается контракт метода (например, передача **null** в метод, не принимающий **null**).

Пример:

```
 1. public void SaveData(string path, string data)
 2. {
 3.     if (path == null)
 4.         throw new ArgumentNullException(nameof(path));
 5.
 6.     try
 7.     {
 8.         File.WriteAllText(path, data);
 9.     }
10.     catch (UnauthorizedAccessException)
11.     {
12.       throw new InvalidOperationException("Нет прав.", ex);
13.     }
14.     catch (IOException ex)
15.     {
16.        throw new InvalidOperationException("Ошибка.", ex);
17.     }
18. }
```

Здесь исключения используются по назначению: для сигнализации о

сбоях, которые невозможно предсказать до выполнения операции.

§12.9 Пользовательские исключения

Хотя платформа **.NET** предоставляет богатую иерархию стандартных исключений, покрывающих большинство типовых ошибок, в сложных приложениях зачастую возникает необходимость в более семантически точных и предметно-ориентированных сообщениях об ошибках. В таких случаях стандартных типов, таких как **ArgumentException** или **InvalidOperationException**, может оказаться недостаточно, поскольку они не отражают специфику предметной области. Для решения этой задачи в C# разрешается создавать пользовательские исключения — собственные классы, наследующие от **System.Exception** или его производных, которые позволяют выразить ошибку на языке самой предметной области.

Пользовательские исключения — это не просто обёртки над стандартными, а осознанный способ проектирования контрактов между компонентами, повышающий читаемость, сопровождаемость и надёжность кода.

Чтобы определить собственное исключение, необходимо создать класс, унаследованный от одного из базовых классов исключений. Наиболее типичный выбор — **Exception** или, при необходимости, более специфичный предок, например **ArgumentException** или **InvalidOperationException**.

Простейший пример — исключение для недопустимого возраста:

```
 1. public class InvalidAgeException : Exception
 2. {
 3.     public InvalidAgeException() { }
 4.
 5.     public InvalidAgeException(string message)
 6.         : base(message) { }
 7.
 8.     public InvalidAgeException(string message,
 9.                                Exception innerException)
10.         : base(message, innerException) { }
11. }
```

Здесь реализованы три стандартных конструктора, соответствующих шаблону проектирования для исключений:

- Конструктор по умолчанию.

- Конструктор с сообщением.
- Конструктор с сообщением и внутренним исключением (для инкапсуляции причины).

Такой класс может быть использован в методе, проверяющем возраст:

```
1. public void SetAge(int age)
2. {
3.     if (age < 0 || age > 150)
4.         throw new InvalidAgeException($"Ошибка: {age}.");
5.
6.     _age = age;
7. }
```

При обработке такое исключение можно перехватить отдельно, не смешивая с другими ошибками:

```
1. catch (InvalidAgeException ex)
2. {
3.     Console.WriteLine($"Ошибка: {ex.Message}");
4. }
```

Пользовательские исключения могут не только унаследовать поведение базового класса, но и расширить его, добавив собственные свойства, отражающие контекст ошибки.

Например, исключение при работе с банком может содержать информацию о счёте и недостаточной сумме:

```
 1. public class InsufficientFundsException : Exception
 2. {
 3.     public decimal b { get; }
 4.     public decimal amount { get; }
 5.
 6.     public InsufficientFundsException(decimal b, decimal a)
 7.         : base($"Недостаточно средств при {b} и {a}.")
 8.     {
 9.         Balance = balance;
10.         WithdrawalAmount = amount;
11.     }
12.
13.     public InsufficientFundsException(decimal b, decimal a,
14.                                       Exception iE)
15.         : base($"Недостаточно средств при {b} и {a}.", iE)
16.     {
17.         b = balance;
18.         this.amount = amount;
19.     }
20. }
```

Теперь обработчик может не только вывести сообщение, но и проанализировать детали:

```
1. catch (InsufficientFundsException ex)
2. {
3.     Console.WriteLine($"Ошибка: {ex.Message}");
4.     // Предложить частичный вывод или пополнение
5. }
```

Для крупных приложений рекомендуется строить иерархию пользовательских исключений, группируя их по смыслу. Например:

```
1. public class BankingException
2.     : Exception { /* ... */ }
3. public class InsufficientFundsException
4.     : BankingException { /* ... */ }
5. public class AccountLockedException
6.     : BankingException { /* ... */ }
7. public class InvalidTransactionException
8.     : BankingException { /* ... */ }
```

Такой подход позволяет:

- Обрабатывать все банковские ошибки единообразно через **catch** (*BankingException ex*).
- При необходимости — реагировать на конкретные случаи.
- Поддерживать модульность и расширяемость.

Создание собственного исключения оправдано, когда:

- Стандартные типы не передают семантику ошибки.
- Ошибка относится к предметной области (например, «*недопустимый статус заказа*», «*лицензия истекла*»).
- Требуется дополнительная информация, полезная для обработки.
- Исключение используется в публичном API, и важно сделать контракт понятным.

Однако не стоит создавать пользовательские исключения для каждой мелкой ошибки. Чрезмерное дробление иерархии усложняет код. Как правило, достаточно одного-двух уровней специализации.