# Подробная лекция по ключевым концепциям C#: LINQ, асинхронность, исключения, коллекции и работа с файлами

### Введение: Фундаментальные Концепции.NET 8-9

Современная разработка на C# в среде.NET требует глубокого понимания ряда фундаментальных концепций. Инструменты и подходы, такие как интегрированные запросы к данным (LINQ), асинхронное программирование, надежная обработка ошибок, эффективное управление коллекциями и взаимодействие с файловой системой, являются основой для создания высокопроизводительных, масштабируемых и отказоустойчивых приложений. Последние версии.NET 8 и.NET 9 приносят множество оптимизаций и улучшений в этих областях, что делает их изучение еще более актуальным.

Данная лекция предназначена для систематизации и углубления знаний по этим ключевым темам. Она объединяет теоретические основы с подробными примерами кода, демонстрируя практическое применение каждой концепции. Цель состоит в том, чтобы не только объяснить "как" использовать эти инструменты, но и дать понимание "почему" они работают именно так, помогая разработчикам принимать более обоснованные архитектурные решения.

### Глава 1: LINQ — Язык Запросов

LINQ, или Language Integrated Query, представляет собой мощную и гибкую функцию языка C#, которая позволяет писать запросы к различным источникам данных непосредственно в коде. Это обеспечивает единый синтаксис для работы с коллекциями в памяти, базами данных (через Entity Framework), XML-документами и другими источниками.

#### 1.1. Основы LINQ: Синтаксис запросов и методов

В LINQ существуют два основных синтаксиса для написания запросов, которые семантически эквивалентны:

**Синтаксис запросов (Query Syntax):** Этот синтаксис напоминает SQL и часто используется для упрощения сложных запросов. Он начинается с оператора from и заканчивается оператором select. Например, для выборки четных чисел из массива можно использовать следующий код:

int numbers = ;  
  
// Синтаксис запросов  
IEnumerable<int> evenNumbersQuery = from num in numbers  
 where num % 2 == 0  
 orderby num  
 select num;  
  
foreach (int number in evenNumbersQuery)  
{  
 Console.Write(number + " ");  
}  
// Вывод: 6 8 10 12

**Синтаксис методов (Method Syntax):** Этот подход использует методы расширения, которые вызываются последовательно с помощью точечной нотации. Он считается более гибким и часто используется в профессиональной разработке. Код, эквивалентный приведенному выше запросу, будет выглядеть так:

int numbers = ;  
  
// Синтаксис методов  
IEnumerable<int> evenNumbersMethod = numbers  
 .Where(num => num % 2 == 0)  
 .OrderBy(num => num);  
  
foreach (int number in evenNumbersMethod)  
{  
 Console.Write(number + " ");  
}  
// Вывод: 6 8 10 12

В основе синтаксиса методов лежат **методы расширения**, которые "расширяют" интерфейс IEnumerable<T>, добавляя такие методы, как Where, Select, OrderBy и другие. Это позволяет вызывать их так, как если бы они были встроенными методами самого типа. **Лямбда-выражения** (num => num % 2 == 0) обеспечивают удобный способ передачи логики фильтрации или трансформации в эти методы, делая код лаконичным и выразительным.

#### 1.2. Базовые операторы LINQ

Ключевая мощь LINQ заключается в его стандартных операторах запросов. Вот наиболее часто используемые из них:

* **Фильтрация (Where)**: Оператор Where отбирает элементы из последовательности, которые удовлетворяют заданному условию. Например, чтобы получить всех студентов мужского пола из коллекции \_db.Students, можно использовать следующий код:

public IEnumerable<Student> GetMaleStudents(DbSet<Student> students)  
{  
 return students.Where(s => s.Gender == "Male");  
}

**Проекция (Select)**: Select используется для трансформации элементов в новую форму или для выбора только определенных свойств объектов. Это позволяет создать новую коллекцию, содержащую только необходимую информацию, что особенно важно для оптимизации запросов к базам данных. Пример получения полных имен студентов:

public IEnumerable<string> GetStudentFullNames(IEnumerable<Student> students)  
{  
 return students.Select(student => $"{student.FirstName} {student.LastName}");  
}

* **Сортировка (OrderBy, OrderByDescending, ThenBy)**: OrderBy сортирует элементы в порядке возрастания, а OrderByDescending — в порядке убывания. Оператор ThenBy используется для вторичной сортировки, когда несколько элементов имеют одинаковое значение по первому критерию. Например, чтобы сначала отсортировать студентов по имени, а затем по возрасту, если имена совпадают, используется такая цепочка:

public IEnumerable<Student> GetStudentsOrderedByNameAndThenByAge(IEnumerable<Student> students)  
{  
 return students.OrderBy(s => s.FirstName).ThenBy(s => s.Age);  
}

#### 1.3. Ключевая концепция: Отложенное и немедленное выполнение

Одним из самых важных аспектов LINQ является концепция **отложенного выполнения (deferred execution)**, также известная как "ленивая оценка" (lazy evaluation). Это означает, что LINQ-запрос не выполняется в момент его создания; вместо этого он формирует "план" или "инструкцию" для будущего выполнения.3 Фактическое выполнение происходит только тогда, когда результат запроса действительно необходим, например, при перечислении с помощью цикла

foreach, или при вызове методов материализации, таких как ToList(), ToArray(), Count() или FirstOrDefault().4

Механизм отложенного выполнения обеспечивается ключевым словом yield. Компилятор преобразует методы-итераторы, содержащие yield return, в конечный автомат, который позволяет возвращать элементы по одному, по мере их запроса.4

Этот подход имеет два важных следствия:

* **Преимущество:** Отложенное выполнение значительно повышает производительность, особенно при работе с большими коллекциями и цепочками запросов. Это позволяет пройти по исходной коллекции всего один раз, выполняя все операции поочередно. Например, в бенчмарках было показано, что метод, использующий отложенное выполнение без ToList(), может быть в 100 раз быстрее, поскольку он обрабатывает только те элементы, которые ему нужны, и не выделяет память под полный список.4
* **Антипаттерн:** Повторное перечисление одного и того же "ленивого" запроса без его материализации может привести к многократному выполнению дорогих операций. Если метод IsRelevant в запросе, например, выполняет долгий вызов к базе данных, то многократное использование переменной relevantCustomers без ToList() приведет к многократному выполнению этого дорогого вызова.4 В таких случаях явная материализация запроса с помощью  
  ToList() после фильтрации может значительно повысить производительность, предотвратив повторные вызовы.

#### 1.4. LINQ и базы данных: Оптимизация запросов

Наиболее критичным аспектом LINQ является его взаимодействие с базами данных через ORM, такие как Entity Framework (EF). В этом контексте LINQ-запросы транслируются в SQL. Способ, которым вы пишете LINQ-запрос, напрямую влияет на генерируемый SQL и, следовательно, на производительность приложения.

Если вы сначала материализуете всю коллекцию, а затем фильтруете ее в памяти приложения (ctx.Things.ToList().Where(...)), EF сначала выполнит SQL-запрос, который выберет *все* записи из таблицы Things, а затем приложение самостоятельно выполнит фильтрацию. Этот подход является крайне неэффективным и опасным, так как приводит к избыточной загрузке данных и может вызвать ошибки OutOfMemoryException на больших объемах.5

Правильный подход заключается в том, чтобы использовать отложенное выполнение LINQ, позволяя EF самостоятельно перевести Where и другие операторы в SQL-запрос с условием WHERE (ctx.Things.Where(...).ToList()). В этом случае EF отправляет в базу данных оптимизированный запрос, и она возвращает только те записи, которые удовлетворяют условию.5 Это фундаментальное различие между "фильтровать на клиенте" и "фильтровать на сервере", и именно оно делает LINQ to Entities столь мощным.

Связанные с этим лучшие практики включают:

* Использование .Select() для проекции данных в DTO (Data Transfer Object), что позволяет извлекать из базы данных только те столбцы, которые действительно нужны.5 Это предотвращает утечку моделей базы данных в другие слои приложения и снижает объем передаваемых данных.
* Избегание «ленивой загрузки» (Lazy Loading) и осторожное использование .Include(). Явная загрузка всех связанных данных может привести к «картезианскому взрыву», когда выборка небольшого количества сущностей приводит к загрузке тысяч строк данных из связанных таблиц.5

**Таблица 1: Сравнение синтаксиса LINQ**

|  |  |  |
| --- | --- | --- |
| Операция | Синтаксис запросов (Query Syntax) | Синтаксис методов (Method Syntax) |
| **Фильтрация** | from s in students where s.Age > 20 select s; | students.Where(s => s.Age > 20); |
| **Проекция** | from s in students select s.Name; | students.Select(s => s.Name); |
| **Сортировка** | from s in students orderby s.Name select s; | students.OrderBy(s => s.Name); |
| **Вторичная сортировка** | from s in students orderby s.Name, s.Age select s; | students.OrderBy(s => s.Name).ThenBy(s => s.Age); |

### Глава 2: Асинхронное Программирование (async/await)

Асинхронное программирование позволяет создавать отзывчивые и высокопроизводительные приложения, которые не блокируют текущий поток выполнения во время ожидания завершения операций.

#### 2.1. Проблемы синхронного кода и преимущества асинхронности

В традиционном синхронном коде операции выполняются последовательно: каждая операция должна завершиться, прежде чем начнется следующая. Если одна из них является «связанной с вводом-выводом» (I/O-bound) — например, чтение файла, запрос к базе данных или API — поток, на котором она выполняется, простаивает в ожидании ответа.7 Это приводит к двум проблемам:

* В десктопных и мобильных приложениях блокируется поток пользовательского интерфейса (UI), что приводит к «зависанию» приложения.
* В веб-приложениях и сервисах это снижает масштабируемость, поскольку поток из пула потоков не может обслуживать другие запросы, пока ждет ответа.

Асинхронное программирование решает эту проблему, позволяя потоку освободиться для выполнения другой работы во время ожидания.7

#### 2.2. Механизм async и await

Ключевые слова async и await — это синтаксический сахар, который значительно упрощает написание неблокирующего кода.9

* Слово async указывает компилятору, что метод является асинхронным и может содержать оператор await.
* Слово await используется перед вызовом асинхронной операции (которая возвращает Task или Task<T>).7 При достижении оператора  
  await выполнение метода приостанавливается, и управление возвращается вызывающему коду. Это позволяет потоку продолжить другую работу, не дожидаясь завершения операции.7 Когда асинхронная операция завершится, выполнение метода будет возобновлено с того места, где оно было приостановлено.8

Под капотом компилятор C# преобразует асинхронный метод в **конечный автомат**, который отслеживает состояние выполнения, включая место, где метод должен быть возобновлен после завершения ожидаемой задачи.7

#### 2.3. Различия между I/O-bound и CPU-bound задачами

Применение async/await зависит от характера задачи:

* **I/O-bound (связанные с вводом-выводом):** Эти операции тратят большую часть времени на ожидание (например, ответа от сети или чтения с диска), а не на вычисления. Это идеальный сценарий для async/await. Использование await перед таким вызовом, как HttpClient.GetStringAsync, освобождает поток, и он не ждет впустую.7
* **CPU-bound (связанные с вычислениями):** Эти операции интенсивно используют процессор, например, сложные математические расчеты или обработка данных. Простое использование await для таких задач не поможет, так как поток все равно будет занят вычислениями. Для того чтобы не блокировать UI, такие задачи необходимо запускать в фоновом потоке, используя Task.Run.7  
  Task.Run перемещает задачу в пул потоков, а await затем ожидает ее завершения, не блокируя основной поток.7

Эта разница является критически важной. Неверное применение Task.Run для I/O-bound задач может привести к ненужной трате ресурсов пула потоков, а отсутствие Task.Run для CPU-bound задач — к блокировке UI.

#### 2.4. Task vs Task<T>

Асинхронные методы должны возвращать один из трех типов, за исключением обработчиков событий:

* **Task:** Используется, когда асинхронный метод не возвращает никакого значения. Он просто сигнализирует о своем завершении.10
* **Task<T>:** Используется, когда асинхронный метод должен вернуть значение типа T. Когда оператор await применяется к Task<T>, он «распаковывает» результат, и вы получаете значение T.10
* **void:** Допустим только для асинхронных обработчиков событий, так как их нельзя «ожидать». Использование void в других случаях является антипаттерном, поскольку исключения, возникающие в таком методе, не могут быть перехвачены вызывающим кодом.11

#### 2.5. Параллельное выполнение и Task.WhenAll/Task.WhenAny

Если у вас есть несколько независимых асинхронных операций, последовательное ожидание их с помощью await неэффективно, поскольку каждая следующая операция начнется только после завершения предыдущей.

// Неэффективно: задачи выполняются последовательно  
await GetFirstThingAsync();  
await GetSecondThingAsync();  
await GetThirdThingAsync();

Для повышения производительности можно запустить все задачи одновременно и затем ожидать их завершения вместе, используя Task.WhenAll.

* **Task.WhenAll:** Ожидает завершения *всех* переданных ему задач.12 Он полезен, когда для продолжения работы нужны результаты всех операций.

var task1 = GetFirstThingAsync();  
var task2 = GetSecondThingAsync();  
var task3 = GetThirdThingAsync();  
  
// Задачи выполняются параллельно, ожидаем их завершения.  
await Task.WhenAll(task1, task2, task3);

* **Task.WhenAny:** Завершается, как только *любая одна* из задач завершится.12 Этот метод полезен в сценариях, где вам нужен первый доступный результат.

**Таблица 2: Сравнение I/O-bound и CPU-bound задач**

|  |  |  |
| --- | --- | --- |
| Характеристика | I/O-Bound (Ввод-вывод) | CPU-Bound (Вычисления) |
| **Определение** | Простаивает в ожидании внешних операций (диск, сеть, база данных). | Активно использует процессор для выполнения расчетов. |
| **Типичные примеры** | Загрузка файла, запрос к API, чтение из сокета. | Сложные математические вычисления, сжатие данных, парсинг больших объемов текста. |
| **Рекомендуемый подход** | await без Task.Run для неблокирующего ожидания. | Task.Run() для переноса работы в фоновый поток. |

### Глава 3: Обработка Исключений

Обработка исключений — это механизм, который позволяет приложению корректно реагировать на ошибки во время выполнения, предотвращая его неожиданное завершение.

#### 3.1. Основы: Блоки try, catch и finally

* **try:** Этот блок содержит код, который потенциально может сгенерировать исключение.14
* **catch:** Если в блоке try возникает исключение, управление передается в соответствующий блок catch. Можно указать конкретный тип исключения, чтобы обрабатывать только его. Если исключение не соответствует типу, оно будет передано в следующий блок catch или далее по стеку вызовов.14
* **finally:** Код в этом блоке гарантированно выполняется, независимо от того, было ли сгенерировано исключение, поймано оно или нет.14 Это делает  
  finally идеальным местом для освобождения ресурсов, например, закрытия файлов или сетевых соединений.

try  
{  
 // Код, который может вызвать ошибку  
 int result = 10 / 0;  
}  
catch (DivideByZeroException ex)  
{  
 // Обработка конкретного исключения  
 Console.WriteLine($"Ошибка: {ex.Message}");  
}  
finally  
{  
 // Код, который всегда выполняется  
 Console.WriteLine("Операция завершена.");  
}

#### 3.2. Лучшие практики обработки исключений

* **Избегайте использования исключений для управления потоком.** Исключения должны быть зарезервированы для по-настоящему «исключительных» или неожиданных ситуаций. Для рутинных проверок, таких как деление на ноль или проверка существования файла, лучше использовать условные операторы (if).14
* **Будьте специфичны.** Всегда старайтесь ловить конкретные типы исключений, а не общий Exception. Это позволяет создавать более точный и надежный код. Если вы все-таки используете общий catch, размещайте его последним, чтобы он не перехватил более специфичные исключения.14
* **Повторно выбрасывайте исключения с помощью throw;**. Если вы перехватили исключение для логирования, а затем хотите передать его дальше, используйте throw;. Это сохраняет исходный трассировочный стек (StackTrace) и указывает на истинную причину ошибки. Использование throw ex; перезаписывает трассировку и затрудняет отладку.14
* **Используйте фильтры исключений (when)**. Эта функция позволяет добавлять дополнительные условия к блокам catch, не разворачивая стек вызовов. Это дает возможность провести логирование или выполнить другие проверки, не прерывая выполнение.

try  
{  
 // Потенциально опасный код  
}  
catch (Exception ex) when (ex is ArgumentException |

| ex is DivideByZeroException)

{

Console.WriteLine($"Обработка ошибки: {ex.Message}");

}

#### 3.3. Пользовательские исключения

Создание собственных классов исключений позволяет сделать код более выразительным, предоставляя специфическую для вашего приложения информацию об ошибках.18 Для этого необходимо наследовать новый класс от

Exception или одного из его производных. Рекомендуется предоставлять как минимум три стандартных конструктора: без параметров, с сообщением об ошибке и с внутренним исключением (innerException).18

public class InvalidNameException : Exception  
{  
 public InvalidNameException() { }  
 public InvalidNameException(string message) : base(message) { }  
 public InvalidNameException(string message, Exception inner) : base(message, inner) { }  
}  
  
public void ValidateName(string name)  
{  
 if (string.IsNullOrWhiteSpace(name))  
 throw new InvalidNameException("Имя не может быть пустым.");  
}

#### 3.4. Особенности обработки исключений в асинхронном коде

Исключения, возникающие в асинхронном методе, который возвращает Task или Task<T>, не выбрасываются немедленно. Вместо этого они «хранятся» в возвращаемой задаче и «распаковываются» и выбрасываются заново только при вызове await.11

Ключевой момент в том, что если несколько асинхронных операций, запущенных одновременно, вызывают исключения (например, при использовании Task.WhenAll), все они будут обернуты в одно исключение AggregateException.20 Оператор

await автоматически распакует только первое из них, и остальные исключения могут быть потеряны, если их не обработать явно. Чтобы получить все исключения, необходимо получить доступ к свойству Task.Exception, которое возвращает AggregateException с коллекцией InnerExceptions.20

public static async Task ThrowMultipleExceptionsAsync()  
{  
 Task allTasks = null;  
 try  
 {  
 var firstTask = Task.Run(() => throw new IndexOutOfRangeException("Первое исключение"));  
 var secondTask = Task.Run(() => throw new InvalidOperationException("Второе исключение"));  
 allTasks = Task.WhenAll(firstTask, secondTask);  
 await allTasks;  
 }  
 catch (AggregateException ex)  
 {  
 Console.WriteLine("Возникли следующие исключения:");  
 foreach (var innerEx in ex.InnerExceptions)  
 {  
 Console.WriteLine($" - {innerEx.Message} ({innerEx.GetType().Name})");  
 }  
 }  
}

Такой подход гарантирует, что ни одно исключение не будет проигнорировано.

### Глава 4: Коллекции

Коллекции в C# — это классы, предназначенные для хранения, управления и манипулирования группами объектов. Правильный выбор коллекции имеет решающее значение для производительности и читаемости кода.

**4.1. Введение: дженерики vs. не-дженерики**

Современный .NET настоятельно рекомендует использовать дженерик-коллекции, такие как List<T>, Dictionary<TKey, TValue> и HashSet<T>, которые находятся в пространстве имён System.Collections.Generic. Они обеспечивают строгую типизацию на этапе компиляции, что исключает ошибки приведения типов во время выполнения и значительно повышает производительность.

Не-дженерик коллекции (ArrayList, Hashtable), которые могут хранить элементы любого типа (как object), считаются устаревшими и их следует избегать. При использовании таких коллекций происходит упаковка (boxing) для типов значений (например, int, bool), что снижает производительность и увеличивает потребление памяти.

**4.2. Работа с типами значений и ссылочными типами: по значению или по ссылке?**

Важно понимать, какие данные хранятся в коллекциях и как они передаются, поскольку это влияет на поведение программы.

**● Общее правило:**

* Типы значений (int, double, struct, enum) хранятся по значению.
* Ссылочные типы (class, string, массивы, делегаты) хранятся по ссылке.

Однако внутри коллекций это работает следующим образом:

**Коллекции хранят** копии значений или ссылок**, а не сами объекты напрямую.**

|  |  |  |
| --- | --- | --- |
|  |  |  |
| Тип значения(int,DateTime,struct) | Копия значения | Изменение элемента в коллекции не влияет на внешнюю переменную. |
| Ссылочный тип(class,string,object) | Копия ссылкина объект | Несколько элементов могут ссылаться на один и тот же объект. Изменения в объекте видны через все ссылки. |

#### Наиболее используемые коллекции

* **List<T>:** List<T> — это динамически расширяемый массив.23 Он идеально подходит для хранения упорядоченной последовательности элементов.
  + **Производительность:** Доступ к элементу по индексу O(1). Добавление в конец списка занимает в среднем O(1), но может стать O(n) при изменении размера внутреннего массива. Поиск элемента по значению занимает O(n) (линейное время).25

// Создание и инициализация List<string>  
List<string> salmons = ["chinook", "coho", "pink", "sockeye"];  
  
// Добавление элемента  
salmons.Add("coho");  
  
// Удаление элемента  
salmons.Remove("coho");  
  
// Доступ по индексу  
Console.WriteLine(salmons); // chinook

**Dictionary<TKey, TValue>:** Dictionary<TKey, TValue> хранит данные в виде пар «ключ-значение».23 Он основан на хеш-таблице, что обеспечивает очень высокую производительность поиска по ключу.

* + **Производительность:** Поиск, добавление и удаление элемента по ключу занимают O(1) в среднем (константное время). Это делает его идеальным для сценариев, где требуется быстрый доступ к данным по уникальному идентификатору.25

// Создание Dictionary<int, string>  
Dictionary<int, string> foods = new Dictionary<int, string>();  
foods.Add(1, "Soda");  
foods.Add(2, "Burger");  
  
// Доступ к элементу по ключу  
string item = foods; // "Soda"  
  
// Проверка наличия ключа  
if (foods.ContainsKey(2))  
{  
 Console.WriteLine(foods); // Burger  
}

* **HashSet<T>:** HashSet<T> — это коллекция, которая содержит только уникальные элементы.25 Она основана на хеш-таблице, как и  
  Dictionary, но не хранит значения, а только ключи.25
  + **Производительность:** Добавление, удаление и проверка наличия элемента (Contains) занимают в среднем O(1).25
  + **Ключевые особенности:** HashSet также предоставляет высокопроизводительные операции над множествами, такие как объединение (UnionWith), пересечение (IntersectWith) и разность (ExceptWith).27

HashSet<int> setA = new HashSet<int> { 1, 2, 3 };  
HashSet<int> setB = new HashSet<int> { 3, 4, 5 };  
  
// Добавление уникального элемента  
bool added = setA.Add(1); // false, так как 1 уже есть  
  
// Объединение множеств  
setA.UnionWith(setB); // setA теперь {1, 2, 3, 4, 5}  
  
// Проверка на уникальность  
bool exists = setA.Contains(4); // true

#### 4.3. Сравнительный анализ и когда что использовать

Выбор коллекции должен быть обусловлен не только производительностью, но и семантикой. List<T> является универсальной коллекцией, но для специфических задач существуют более подходящие структуры данных. Если требуется хранить уникальные элементы, HashSet<T> является идеальным выбором, так как его семантика и производительность для операций проверки на уникальность (O(1)) превосходят List<T> (O(n)).25 Если нужен быстрый доступ к элементам по ключу,

Dictionary<TKey, TValue> — правильный инструмент.23

**Таблица 3: Сравнительная характеристика коллекций**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Характеристика | List<T> | Dictionary<TKey, TValue> | HashSet<T> | Queue<T> | Stack<T> |
| **Уникальность** | Нет | Уникальные ключи | Уникальные элементы | Нет | Нет |
| **Порядок** | Индексный | Нет (по хешу) | Нет (по хешу) | FIFO | LIFO |
| **Доступ** | По индексу | По ключу | Непосредственный доступ отсутствует | Dequeue | Pop |
| **Добавление** | O(1) (в среднем) | O(1) (в среднем) | O(1) (в среднем) | O(1) | O(1) |
| **Поиск (Contains)** | O(n) | O(1) | O(1) | O(n) | O(n) |
| **Типичный сценарий** | Упорядоченная последовательность, частые операции добавления/удаления. | Быстрый поиск по ключу. | Хранение уникальных элементов, операции над множествами. | Очереди задач, обработка событий. | Стек вызовов, отмена операций. |

#### 4.4. Другие полезные коллекции

● Queue<T>: Реализует структуру данных "очередь", работающую по принципу FIFO (First-In, First-Out). Элементы добавляются в конец очереди методом Enqueue, а извлекаются из начала — методом Dequeue. Также доступны методы Peek (просмотр первого элемента без удаления) и TryDequeue (безопасное извлечение с проверкой на наличие элементов).

Пример использования Queue<T>:

Queue<string> printerQueue = new Queue<string>();

printerQueue.Enqueue("Документ 1");

printerQueue.Enqueue("Документ 2");

printerQueue.Enqueue("Документ 3");

while (printerQueue.Count > 0)

{

string currentDoc = printerQueue.Dequeue();

Console.WriteLine($"Печатается: {currentDoc}");

}

// Вывод:

// Печатается: Документ 1

// Печатается: Документ 2

// Печатается: Документ 3

● Stack<T>: Реализует структуру данных "стек", работающую по принципу LIFO (Last-In, First-Out). Элементы добавляются методом Push и извлекаются методом Pop с вершины стека. Доступны также Peek (просмотр верхнего элемента без извлечения) и TryPop.

Пример использования Stack<T>:

Stack<int> history = new Stack<int>();

history.Push(10);

history.Push(20);

history.Push(30);

while (history.Count > 0)

{

int lastValue = history.Pop();

Console.WriteLine($"Восстановлено: {lastValue}");

}

// Вывод:

// Восстановлено: 30

// Восстановлено: 20

// Восстановлено: 10

*Часто используется в алгоритмах, например, при разборе выражений, отмене действий (undo), обходе графов (DFS).*

● HashSet<T>: Представляет коллекцию уникальных элементов без дубликатов. Обеспечивает высокую производительность операций добавления, удаления и поиска (в среднем O(1)). Подходит для задач, где важна уникальность и быстрый доступ.

Пример использования HashSet<T>:

HashSet<string> tags = new HashSet<string>();

tags.Add("C#");

tags.Add("LINQ");

tags.Add("C#"); // Дубликат — будет проигнорирован

Console.WriteLine(string.Join(", ", tags));

// Вывод: C#, LINQ

Также полезны методы:

* UnionWith — объединение множеств,
* IntersectWith — пересечение,
* ExceptWith — разность.

● SortedSet<T>: Аналог HashSet<T>, но элементы хранятся в отсортированном порядке. Основан на сбалансированном дереве (красно-чёрное дерево), операции выполняются за O(log n).

Пример:

SortedSet<int> numbers = new SortedSet<int> { 5, 1, 9, 3 };

foreach (int num in numbers)

{

Console.Write(num + " ");

}

// Вывод: 1 3 5 9

Полезен, когда нужна сортировка "из коробки" и уникальность.

● LinkedList<T>: Двусвязный список. Каждый элемент (узел) содержит ссылку на предыдущий и следующий узлы. Быстрое добавление и удаление в любом месте (если есть ссылка на узел), но медленный произвольный доступ (O(n)).

Пример:

LinkedList<string> tasks = new LinkedList<string>();

var node1 = tasks.AddFirst("Задача 1");

var node2 = tasks.AddAfter(node1, "Задача 2");

tasks.AddBefore(node2, "Задача 1.5");

foreach (string task in tasks)

{

Console.WriteLine(task);

}

// Вывод:

// Задача 1

// Задача 1.5

// Задача 2

Уместен при частых вставках/удалениях в середине списка.

Резюме:

|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  |  |
| Queue<T> | FIFO | Быстрое добавление/удаление с концов | Очереди задач, обработка запросов |
| Stack<T> | LIFO | Быстрое добавление/удаление с вершины | Отмена действий, парсинг, DFS |
| HashSet<T> | Без порядка | Уникальность, быстрый поиск | Фильтрация дубликатов, проверка вхождения |
| SortedSet<T> | Сортировка | Уникальность + порядок | Ранжированные списки, словари |
| LinkedList<T> | Связный список | Гибкость при вставках/удалениях | Реализация сложных структур данных |

Эти коллекции расширяют возможности стандартных List<T> и Dictionary<TKey, TValue>, позволяя эффективно решать специфические задачи.

### Глава 5: Работа с Файлами и Потоками Данных

Эффективная работа с файловой системой и потоками данных является неотъемлемой частью разработки на C#..NET предоставляет богатый набор классов для выполнения этих задач.

#### 5.1. Обзор классов файловой системы

* **File:** Это статический класс, предоставляющий удобные методы для простых, одноразовых операций с файлами, таких как чтение, запись или удаление (ReadAllText, WriteAllText, Delete).30 Он удобен для быстрых действий, но не рекомендуется для множественных операций с одним файлом из-за накладных расходов, связанных с проверками безопасности.30
* **FileInfo:** Класс-экземпляр, который предоставляет те же методы, что и File, но лучше подходит для многократных операций с одним файлом.30 Проверки безопасности выполняются только один раз при создании экземпляра, что повышает производительность.
* **Directory и DirectoryInfo:** Аналогичны File и FileInfo, но предназначены для работы с каталогами (создание, удаление, перемещение).30
* **Path:** Статический класс для удобной работы с путями к файлам и папкам (например, для объединения путей или получения расширения).30

#### 5.2. Потоки данных: сущность Stream

В основе всех операций ввода-вывода в.NET лежит абстрактный класс Stream.33 Поток представляет собой последовательность байтов и может быть связан с файлом, памятью, сетевым соединением и т.д. Классы, производные от

Stream, предназначены для работы с данными на низком уровне (байты), а специализированные классы, такие как StreamReader/StreamWriter, надстраиваются над ними для работы с более сложными типами данных.33

Крайне важно правильно управлять жизненным циклом потоков, которые реализуют интерфейс IDisposable, используя оператор using. Это гарантирует, что ресурсы будут корректно освобождены, даже если возникнет исключение.34

#### 5.3. Работа с текстовыми файлами: StreamReader и StreamWriter

* **StreamReader:** Класс, предназначенный для чтения символьных данных из потока. Он идеально подходит для построчного чтения текстовых файлов с помощью метода ReadLine().35
* **StreamWriter:** Используется для записи символьных данных в поток. Он предоставляет удобные методы, такие как Write() и WriteLine(), и по умолчанию использует кодировку UTF-8.34

Пример синхронной работы с текстовым файлом:

string filePath = "notes.txt";  
// Запись в файл  
using (StreamWriter writer = new StreamWriter(filePath))  
{  
 writer.WriteLine("Первая строка.");  
 writer.WriteLine("Вторая строка.");  
}  
  
// Чтение из файла  
using (StreamReader reader = new StreamReader(filePath))  
{  
 string? line;  
 while ((line = reader.ReadLine())!= null)  
 {  
 Console.WriteLine(line);  
 }  
}

#### 5.4. Работа с двоичными файлами: FileStream

FileStream — это специализированный класс для работы с файлами на диске. Он напрямую читает и записывает байты, что делает его незаменимым для работы с двоичными данными, такими как изображения, видео или исполняемые файлы.36

FileStream также поддерживает асинхронные операции (ReadAsync, WriteAsync), что является лучшим подходом при работе с большими файлами.

string imagePath = "image.jpg";  
// Пример чтения двоичного файла в массив байтов  
byte imageBytes;  
using (FileStream fs = new FileStream(imagePath, FileMode.Open, FileAccess.Read))  
{  
 imageBytes = new byte[fs.Length];  
 fs.Read(imageBytes, 0, (int)fs.Length);  
}  
  
// Пример записи массива байтов в новый файл  
using (FileStream fs = new FileStream("new\_image.jpg", FileMode.Create, FileAccess.Write))  
{  
 fs.Write(imageBytes, 0, imageBytes.Length);  
}

Для упрощения работы с примитивными типами данных в двоичном формате существуют классы-обертки BinaryReader и BinaryWriter, которые надстраиваются над любым потоком.38

#### 5.5. Синхронные и асинхронные операции с файлами

Работа с файлами является классическим примером I/O-bound операции. При чтении или записи больших объемов данных синхронные методы, такие как File.ReadAllText, будут блокировать поток, что приведет к «зависанию» приложения.35 Использование асинхронных методов (

ReadAllTextAsync, WriteAllTextAsync, ReadAsync, WriteAsync) позволяет выполнять эти операции в фоновом режиме, не блокируя основной поток и сохраняя отзывчивость приложения.35

**Таблица 4: Обзор классов для работы с файлами и потоками**

|  |  |  |
| --- | --- | --- |
| Класс | Назначение | Когда использовать |
| **File** | Статический класс для простых операций с одним файлом. | Для быстрых, одноразовых операций (проверить наличие, прочитать весь текст). |
| **FileInfo** | Экземплярный класс для операций с файлом. | Для многократных операций с одним и тем же файлом (например, в цикле). |
| **Path** | Статический класс для работы с путями. | Для форматирования, объединения или получения частей пути без взаимодействия с файловой системой. |
| **Stream** | Абстракция для последовательности байтов. | Как базовый класс для других потоков. |
| **StreamReader** | Чтение текстовых данных из потока. | Построчное чтение текстовых файлов. |
| **StreamWriter** | Запись текстовых данных в поток. | Пошаговая запись текстовых файлов. |
| **FileStream** | Чтение и запись двоичных данных в файл. | Работа с изображениями, видео, или любыми двоичными файлами. |
| **BinaryReader** | Чтение примитивных типов из потока в двоичном формате. | Чтение данных из двоичного файла. |

### Глава 6: Создание Собственных Проектов

Теоретические знания приобретают ценность только тогда, когда они применяются на практике. Создание собственных проектов — это лучший способ закрепить изученные концепции и научиться их интегрировать.

#### 6.1. Идеи для проектов

* **Простые консольные приложения:**
  + **"ToDo List Manager"**: Идеальный проект для начинающих, так как он охватывает базовые CRUD-операции (Create, Read, Update, Delete).39 Можно использовать  
    List<T> для хранения задач в памяти, а затем добавить работу с файлами для сохранения данных между запусками.
  + **"Адресная книга"**: Отличная возможность попрактиковаться с коллекциями. Контакты можно хранить в Dictionary<TKey, TValue>, где ключом будет имя, что обеспечит быстрый доступ.39
  + **"Игра 'Крестики-нолики'"**: Помогает развить логическое мышление и навыки работы с пользовательским вводом.39
* **Более сложные проекты:**
  + **"Система управления библиотекой"**: Включает в себя более сложные модели данных (книги, читатели), что требует использования коллекций и, возможно, базы данных.
  + **"Приложение для отслеживания личных финансов"**: Позволяет попрактиковаться с более сложной бизнес-логикой и отчетами, что требует использования LINQ для агрегации и фильтрации данных.40

#### 6.2. Детальный план для учебного проекта: "Менеджер задач"

Для демонстрации синергии всех изученных тем, предлагается реализовать консольное приложение **"Менеджер задач"**. Этот проект является идеальным, поскольку он позволяет последовательно применить все концепции, создавая при этом полезный и функциональный инструмент.

Модель данных и коллекции:

Создадим простой класс Task с такими свойствами, как Id, Title, Description и IsCompleted. Для хранения задач в памяти будем использовать List<Task>.

LINQ для манипуляции данными:

LINQ сделает код для управления задачами чистым и лаконичным.

* **Фильтрация:** tasks.Where(t =>!t.IsCompleted) — для получения списка незавершенных задач.
* **Поиск:** tasks.FirstOrDefault(t => t.Id == taskId) — для быстрого поиска задачи по идентификатору.
* **Сортировка:** tasks.OrderBy(t => t.Title) — для сортировки задач по названию.

Асинхронная работа с файлами:

Для сохранения и загрузки данных будем использовать асинхронные операции с файлами, что позволит избежать блокировки UI при работе с большим объемом задач.

* **Сохранение:** await File.WriteAllTextAsync("tasks.json", jsonString);
* **Загрузка:** string jsonString = await File.ReadAllTextAsync("tasks.json");

Обработка исключений:

Необходимо добавить блоки try-catch для всех файловых операций, чтобы приложение не падало при возникновении ошибок.

* **Обработка FileNotFoundException:** Если файл с задачами не найден, можно создать пустой список и уведомить пользователя.
* **Обработка IOException:** Если нет прав на запись, можно сообщить об ошибке и предложить пользователю проверить разрешения.

#### 6.3. Пошаговая реализация ключевых аспектов

**Шаг 1: Инициализация и работа с коллекцией**

using System.Text.Json;  
  
public class TodoApp  
{  
 private List<TodoTask> \_tasks = new List<TodoTask>();  
 private string \_filePath = "tasks.json";  
  
 public async Task RunAsync()  
 {  
 await LoadTasksAsync();  
 //... основная логика приложения...  
 }  
  
 public void AddTask(string title, string description)  
 {  
 int newId = \_tasks.Any()? \_tasks.Max(t => t.Id) + 1 : 1;  
 \_tasks.Add(new TodoTask(newId, title, description));  
 Console.WriteLine("Задача добавлена.");  
 }  
   
 public void DisplayTasks()  
 {  
 var activeTasks = \_tasks.Where(t =>!t.IsCompleted).ToList();  
 if (activeTasks.Count == 0)  
 {  
 Console.WriteLine("Нет активных задач.");  
 return;  
 }  
   
 foreach (var task in activeTasks.OrderBy(t => t.Title))  
 {  
 Console.WriteLine($"- [ ] {task.Title}");  
 }  
 }  
}

**Шаг 2: Асинхронное сохранение и загрузка**

private async Task LoadTasksAsync()  
{  
 try  
 {  
 if (File.Exists(\_filePath))  
 {  
 string jsonString = await File.ReadAllTextAsync(\_filePath);  
 \_tasks = JsonSerializer.Deserialize<List<TodoTask>>(jsonString)?? new List<TodoTask>();  
 Console.WriteLine("Задачи загружены.");  
 }  
 else  
 {  
 Console.WriteLine("Файл с задачами не найден. Создан новый список.");  
 }  
 }  
 catch (Exception ex)  
 {  
 Console.WriteLine($"Ошибка при загрузке: {ex.Message}");  
 }  
}  
  
public async Task SaveTasksAsync()  
{  
 try  
 {  
 string jsonString = JsonSerializer.Serialize(\_tasks, new JsonSerializerOptions { WriteIndented = true });  
 await File.WriteAllTextAsync(\_filePath, jsonString);  
 Console.WriteLine("Задачи сохранены.");  
 }  
 catch (Exception ex)  
 {  
 Console.WriteLine($"Ошибка при сохранении: {ex.Message}");  
 }  
}  
  
public class TodoTask  
{  
 public int Id { get; set; }  
 public string Title { get; set; }  
 public string Description { get; set; }  
 public bool IsCompleted { get; set; }  
   
 public TodoTask(int id, string title, string description)  
 {  
 Id = id;  
 Title = title;  
 Description = description;  
 IsCompleted = false;  
 }  
}

Этот проект демонстрирует, как разные концепции работают вместе. Использование List<T> и LINQ для управления данными, асинхронные файловые операции для сохранения и загрузки, а также блоки try-catch для надежной обработки ошибок создают единую и профессиональную архитектуру, которая не только функциональна, но и эффективна, и отказоустойчива.

### Заключение

Изученные в этой лекции темы — LINQ, асинхронное программирование, обработка исключений, коллекции и работа с файлами — являются столпами современной разработки на C#. Глубокое понимание этих концепций, включая их внутренние механизмы и лучшие практики, позволяет создавать высокопроизводительные, надежные и легко поддерживаемые приложения.

* **LINQ** делает код для работы с данными выразительным и лаконичным, а понимание отложенного выполнения позволяет оптимизировать запросы к любым источникам.
* **Асинхронность** (async/await) является ключом к созданию отзывчивых приложений, которые не блокируют потоки, особенно при выполнении I/O-bound операций.
* **Обработка исключений** с использованием специфических типов и фильтров обеспечивает отказоустойчивость, а правильное повторное выбрасывание (throw;) критически важно для эффективной отладки.
* **Коллекции** представляют собой основу для эффективного управления данными в памяти. Выбор подходящего типа коллекции (например, HashSet<T> для уникальности или Dictionary<TKey, TValue> для быстрого поиска) должен исходить из семантики задачи.
* **Работа с файлами** с использованием потоков и асинхронных методов позволяет создавать приложения, которые могут работать с большими объемами данных без потери производительности.

Все эти концепции тесно связаны и образуют единый каркас для разработки. Использование LINQ для фильтрации коллекции, асинхронной записи ее в файл с корректной обработкой исключений — это не просто набор разрозненных техник, а демонстрация синергии, которая является отличительной чертой опытного разработчика. Для дальнейшего развития рекомендуется изучить ASP.NET Core для создания веб-приложений и Entity Framework Core для профессиональной работы с базами данных, где все эти темы проявляются в полной мере.

#### Источники

1. Deferred execution and lazy evaluation - LINQ to XML - .NET | Microsoft Learn, дата последнего обращения: августа 22, 2025, <https://learn.microsoft.com/en-us/dotnet/standard/linq/deferred-execution-lazy-evaluation>
2. Understanding C# LINQ's deferred execution | by Ahiya Elster ..., дата последнего обращения: августа 22, 2025, <https://engineering.payoneer.com/understanding-c-linqs-deferred-execution-ba388a3ce995>
3. How to become more optimal with LINQ ? : r/csharp - Reddit, дата последнего обращения: августа 22, 2025, <https://www.reddit.com/r/csharp/comments/1lej9sk/how_to_become_more_optimal_with_linq/>
4. LINQ lambda optimization .NET 8 - Stack Overflow, дата последнего обращения: августа 22, 2025, <https://stackoverflow.com/questions/77997285/linq-lambda-optimization-net-8>
5. Asynchronous programming scenarios - C# | Microsoft Learn, дата последнего обращения: августа 22, 2025, <https://learn.microsoft.com/en-us/dotnet/csharp/asynchronous-programming/async-scenarios>
6. C# Async/Await Explained: Complete Guide with Examples [2025] - NDepend Blog, дата последнего обращения: августа 22, 2025, <https://blog.ndepend.com/c-async-await-explained/>
7. Asynchronous programming - C# | Microsoft Learn, дата последнего обращения: августа 22, 2025, <https://learn.microsoft.com/en-us/dotnet/csharp/asynchronous-programming/>
8. what is a Task? how is it different from async? : r/csharp - Reddit, дата последнего обращения: августа 22, 2025, <https://www.reddit.com/r/csharp/comments/z3ajmk/what_is_a_task_how_is_it_different_from_async/>
9. Async return types - C# | Microsoft Learn, дата последнего обращения: августа 22, 2025, <https://learn.microsoft.com/en-us/dotnet/csharp/asynchronous-programming/async-return-types>
10. Think Beyond Synchronous: The Ultimate Guide to Tasks in C# - Chris Woody Woodruff, дата последнего обращения: августа 22, 2025, <https://www.woodruff.dev/think-beyond-synchronous-the-ultimate-guide-to-tasks-in-c/>
11. For Mid Developers. Do you use Task.WhenAll() ? : r/csharp - Reddit, дата последнего обращения: августа 22, 2025, <https://www.reddit.com/r/csharp/comments/1l46d58/for_mid_developers_do_you_use_taskwhenall/>
12. 7 Best Practices to Handle Exceptions in .NET 8 | by Nouman ..., дата последнего обращения: августа 22, 2025, <https://medium.com/@noumanbaloch/7-best-practices-to-handle-exceptions-in-net-8-99a47eeb5e07>
13. Exception-handling statements - throw and try, catch, finally - C# ..., дата последнего обращения: августа 22, 2025, <https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/statements/exception-handling-statements>
14. C# try catch finally (How It Works For Developers) - IronPDF, дата последнего обращения: августа 22, 2025, <https://ironpdf.com/blog/net-help/csharp-try-catch-finally/>
15. How using try catch for exception handling is best practice - Stack Overflow, дата последнего обращения: августа 22, 2025, <https://stackoverflow.com/questions/14973642/how-using-try-catch-for-exception-handling-is-best-practice>
16. Creating and Using Custom Exceptions in .NET C# | Reintech media, дата последнего обращения: августа 22, 2025, <https://reintech.io/blog/creating-and-using-custom-exceptions-in-net-c-sharp>
17. Creating and Throwing Exceptions - C# | Microsoft Learn, дата последнего обращения: августа 22, 2025, <https://learn.microsoft.com/en-us/dotnet/csharp/fundamentals/exceptions/creating-and-throwing-exceptions>
18. Handling Exceptions in Asynchronous Methods in C# | CodeGuru.com, дата последнего обращения: августа 22, 2025, <https://www.codeguru.com/csharp/async-methods-exception-handling/>
19. Handling Exceptions in Async Code - C# — James Millar, дата последнего обращения: августа 22, 2025, <https://www.james-millar.co.uk/blog/handling-exceptions-in-async-code-c>
20. Selecting a Collection Class - .NET | Microsoft Learn, дата последнего обращения: августа 22, 2025, <https://learn.microsoft.com/en-us/dotnet/standard/collections/selecting-a-collection-class>
21. Essential Collections in C#. When programming in C#, one often needs… | by Roman Fairushyn | Medium, дата последнего обращения: августа 22, 2025, <https://medium.com/@fairushyn/essential-collections-in-c-4ec7e90598ff>
22. Collections in C# Programming | Simplilearn - Simplilearn.com, дата последнего обращения: августа 22, 2025, <https://www.simplilearn.com/tutorials/c-sharp-tutorial/collections-in-c-sharp>
23. What's the difference between a dictionary, a list and a HashSet? : r/Unity3D - Reddit, дата последнего обращения: августа 22, 2025, <https://www.reddit.com/r/Unity3D/comments/48bv22/whats_the_difference_between_a_dictionary_a_list/>
24. HashSet vs List vs Dictionary | theburningmonk.com, дата последнего обращения: августа 22, 2025, <https://theburningmonk.com/2011/03/hashset-vs-list-vs-dictionary/>
25. Understanding HashSet in C#, дата последнего обращения: августа 22, 2025, <https://www.c-sharpcorner.com/article/understanding-hashset-in-c-sharp/>
26. What is Hashset in C#?. In this article we will explore what is… | by Rupen Anjaria | Medium, дата последнего обращения: августа 22, 2025, <https://rupen-anjaria.medium.com/what-is-hashset-in-c-646df468ced6>
27. HashSet
28. C# Files & Directories - Tutorials Teacher, дата последнего обращения: августа 22, 2025, <https://www.tutorialsteacher.com/csharp/csharp-file>
29. File Class (System.IO) | Microsoft Learn, дата последнего обращения: августа 22, 2025, <https://learn.microsoft.com/en-us/dotnet/api/system.io.file?view=net-9.0>
30. Directory Class (System.IO) | Microsoft Learn, дата последнего обращения: августа 22, 2025, <https://learn.microsoft.com/en-us/dotnet/api/system.io.directory?view=net-9.0>
31. FileStream vs/differences StreamWriter? - Stack Overflow, дата последнего обращения: августа 22, 2025, <https://stackoverflow.com/questions/4963667/filestream-vs-differences-streamwriter>
32. StreamWriter Class (System.IO) | Microsoft Learn, дата последнего обращения: августа 22, 2025, <https://learn.microsoft.com/en-us/dotnet/api/system.io.streamwriter?view=net-9.0>
33. StreamReader Class (System.IO) | Microsoft Learn, дата последнего обращения: августа 22, 2025, <https://learn.microsoft.com/en-us/dotnet/api/system.io.streamreader?view=net-9.0>
34. FileStream C# (How It Works For Developers) - IronPDF, дата последнего обращения: августа 22, 2025, <https://ironpdf.com/blog/net-help/filestream-csharp/>
35. FileStream.Read Method (System.IO) | Microsoft Learn, дата последнего обращения: августа 22, 2025, <https://learn.microsoft.com/en-us/dotnet/api/system.io.filestream.read?view=net-9.0>
36. BinaryReader Class (System.IO) | Microsoft Learn, дата последнего обращения: августа 22, 2025, <https://learn.microsoft.com/en-us/dotnet/api/system.io.binaryreader?view=net-9.0>
37. Top 10 C# Project Ideas for Beginners in 2025 - GeeksforGeeks, дата последнего обращения: августа 22, 2025, <https://www.geeksforgeeks.org/blogs/c-project-ideas-for-beginners/>
38. 13 Best C Sharp Project Ideas (C#) [With Source Code] - GUVI, дата последнего обращения: августа 22, 2025, <https://www.guvi.in/blog/best-c-sharp-project-ideas/>