LINQ

Когда мы работаем с коллекциями, код часто получается громоздким: приходится писать циклы, временные списки, условия, вручную фильтровать, сортировать и преобразовывать данные.

LINQ (Language Integrated Query) — это встроенный в C# механизм для удобной работы с коллекциями, решает эту проблему. Он позволяет описывать операции с коллекциями **декларативно** — то есть указывать **что нужно сделать**, а не **как именно это реализовать**. Вместо пошагового алгоритма мы формулируем **запрос к данным**, похожий на SQL, только прямо внутри языка C#. Он может работать не только с обычными коллекциями (`List` , `Array`), но и с другими источниками данных — XML-файлами, базами данных и т.д.

1. Основные операции LINQ

LINQ предоставляет множество встроенных методов для работы с коллекциями.
Все они реализованы как **методы расширения** для `IEnumerable<T>` , поэтому применяются к любому списку, массиву или другому перечислимому типу.

Рассмотрим самые важные и часто используемые операции на примере списка студентов:

```
var students = new List<Student>
{
    new Student("Анна", 19, 1),
    new Student("Борис", 21, 2),
    new Student("Виктор", 20, 2),
    new Student("Дарья", 22, 3),
    new Student("Елена", 19, 1)
};
```

Where — фильтрация элементов

Позволяет выбрать только те элементы, которые соответствуют условию:

```
var adults = students.Where(s => s.Age >= 20);

foreach (var s in adults)
    Console.WriteLine(s.Name);
// Выведет: Борис, Виктор, Дарья
```

Select — преобразование данных

Позволяет преобразовать элементы коллекции в другой тип — например, взять только имена:

```
var names = students.Select(s => s.Name);

foreach (var n in names)
    Console.WriteLine(n);
// Анна, Борис, Виктор, Дарья, Елена
```

Можно также преобразовать данные в новый тип:

```
var shortInfo = students.Select(s => new { s.Name, s.Age });
```

OrderBy / ThenBy — сортировка

Сортировка коллекции по одному или нескольким критериям:

```
var sorted = students
    .OrderBy(s => s.Course)
    .ThenBy(s => s.Name);

foreach (var s in sorted)
    Console.WriteLine($"{s.Course}: {s.Name}");
```

Take / Skip — выбор подмножеств

Позволяют пропустить или взять заданное количество элементов:

```
var firstTwo = students.Take(2); // взять первых двух
var skipTwo = students.Skip(2);  // пропустить первых двух
```

Distinct — удаление дубликатов

Удаляет повторяющиеся элементы из коллекции:

```
var courses = students
    .Select(s => s.Course)
    .Distinct(); // только уникальные курсы
```

Any / All / Contains — проверки условий

Проверяют, есть ли в коллекции элементы, удовлетворяющие условию:

```
bool hasThirdCourse = students.Any(s => s.Course == 3); // хотя бы один
bool allAdults = students.All(s => s.Age >= 18);        // все
bool hasAnna = students.Select(s => s.Name).Contains("Анна"); // содержит
```

First / FirstOrDefault / Single — выбор элемента

Позволяют выбрать конкретный элемент из коллекции:

```
var first = students.First();                    // первый элемент
var firstAdult = students.First(s => s.Age > 20);
var maybe = students.FirstOrDefault(s => s.Age > 25); // null, если нет
var thirdYearStudent = students.Single(s => s.Course == 3) // вернет элемент, если он единственный, иначе будет
ошибка
```

Count / Sum / Max / Min / Average — агрегатные операции

Вычисляют сводные значения по коллекции:

```
int count = students.Count();
double avgAge = students.Average(s => s.Age);
double avgAge2 = students.Sum(s => s.Age) / students.Count()
int maxCourse = students.Max(s => s.Course);
int minCourse = students.Min(s => s.Course);
```

2. Проекция и анонимные типы

Метод `Select` — позволяет **создавать проекцию данных**, то есть выбирать из исходной коллекции только нужные поля или преобразовывать объекты в другую форму.

Пример: выбор отдельных полей

Допустим, у нас есть список людей:

```
var people = new List<Person>
{
    new Person("Андрей", 25, "Программист"),
    new Person("Елена", 30, "Дизайнер"),
    new Person("Сергей", 28, "Маркетолог")
};
```

Мы хотим получить только имена и возраст — без профессии:

```
var result = people.Select(p => new { p.Name, p.Age });
```

Теперь `result` содержит коллекцию **анонимных объектов**, у которых есть только два свойства — `Name` и `Age`.
Это временные структуры данных, не требующие создания отдельного класса.

Что такое анонимный тип

Анонимный тип создаётся с помощью конструкции `new { ... }` без явного имени класса.
Компилятор автоматически создаёт под капотом тип с нужными свойствами.

```
var item = new { Name = "Анна", Age = 22 };
Console.WriteLine($"{item.Name}, {item.Age} лет");
```

Пример использования в отчётах

```
var report = people
    .Where(p => p.Age > 25)
    .Select(p => new { p.Name, Info = $"{p.Age} лет, {p.Job}" });

foreach (var r in report)
    Console.WriteLine($"{r.Name}: {r.Info}");
```

Результат:

```
Елена: 30 лет, Дизайнер
Сергей: 28 лет, Маркетолог
```

3. Группировка и объединение данных

Работа с данными часто требует не только фильтрации и сортировки, но и **группировки** или **объединения** нескольких источников.

GroupBy — группировка по ключу

Метод `GroupBy` позволяет объединить элементы коллекции по определённому признаку (ключу). Например, сгруппируем студентов по номеру группы:

```
var students = new List<Student>
{
    new Student("Андрей", "A1"),
    new Student("Елена", "A2"),
    new Student("Сергей", "A1"),
    new Student("Мария", "A2"),
    new Student("Олег", "A3")
};

var grouped = students.GroupBy(s => s.Group);

foreach (var group in grouped)
{
    Console.WriteLine($"Группа {group.Key}:");

    foreach (var student in group)
        Console.WriteLine($"  {student.Name}");
}
```

Результат:

```
Группа A1:
  Андрей
  Сергей
Группа A2:
  Елена
  Мария
Группа A3:
  Олег
```

Каждая группа (`group`) — это объект `IGrouping<TKey, TElement>`, где `Key` — значение, по которому группировали, а `TElement` — элементы, входящие в эту группу.

Join — соединение двух коллекций

Если нужно объединить данные из двух разных источников (например, студентов и их групп), можно использовать `Join`.

```
var groups = new List<Group>
{
    new Group("A1", "Программирование"),
    new Group("A2", "Дизайн"),
    new Group("A3", "Маркетинг")
};

var joined = students.Join(
    groups,                    // коллекция для объединения
    s => s.Group,              // ключ из первой коллекции
    g => g.Code,               // ключ из второй коллекции
    (s, g) => new { s.Name, g.Specialization } // результат объединения
);

foreach (var item in joined)
    Console.WriteLine($"{item.Name} - {item.Specialization}");
```

Результат:

```
Андрей - Программирование
Елена - Дизайн
Сергей - Программирование
Мария - Дизайн
Олег - Маркетинг
```

GroupJoin — группировка с объединением

Метод `GroupJoin` используется, когда одной записи из первой коллекции соответствует несколько записей из второй. Например, получим список групп с их студентами:

```
var groupJoin = groups.GroupJoin(
    students,
    g => g.Code,
    s => s.Group,
    (g, studs) => new { g.Specialization, Students = studs }
);

foreach (var g in groupJoin)
{
    Console.WriteLine($"{g.Specialization}:");

    foreach (var s in g.Students)
        Console.WriteLine($"  {s.Name}");
}
```

Результат:

```
Программирование:
  Андрей
  Сергей
Дизайн:
  Елена
  Мария
Маркетинг:
  Олег
```

SelectMany — разворачивание вложенных коллекций

Иногда коллекции содержат **списки внутри списков**, например, у каждой группы есть список студентов.
`SelectMany` помогает развернуть такие вложенные структуры в одну последовательность.

```
var allStudents = groups.SelectMany(
    g => g.Students.Select(s => $"{s.Name} ({g.Specialization})")
);

foreach (var s in allStudents)
    Console.WriteLine(s);
```

Результат:

```
Андрей (Программирование)
Сергей (Программирование)
Елена (Дизайн)
Мария (Дизайн)
Олег (Маркетинг)
```

Таким образом:

- `GroupBy` — группирует элементы по признаку;
- `Join` — объединяет две коллекции по ключам;
- `GroupJoin` — объединяет коллекции с сохранением структуры групп;
- `SelectMany` — разворачивает вложенные коллекции.

4. Синтаксис LINQ

LINQ поддерживает **два синтаксиса** написания запросов — **синтаксис запросов (query syntax)** и **синтаксис методов (method syntax)**. Оба делают одно и то же, а компилятор C# автоматически преобразует первый вариант во второй.

Query Syntax

Этот вариант похож на SQL и читается как запрос к данным.
Он начинается с ключевого слова `from` и заканчивается `select`.

```
var numbers = new List<int> { 1, 2, 3, 4, 5, 6 };

// Запросный синтаксис
var evenNumbers =
    from n in numbers
    where n % 2 == 0
    select n;

foreach (var n in evenNumbers)
    Console.WriteLine(n);
```

Результат:

```
2
4
6
```

Method Syntax

Это тот стиль, который мы использовали в первых разделах этой лекции, в нем используются **методы расширения LINQ** (например, `Where`, `Select`, `OrderBy`), которые вызываются цепочкой, такой подход называется **Fluent API** (цепочка вызовов).

```
var evenNumbers = numbers
    .Where(n => n % 2 == 0)
    .Select(n => n);
```

Результат будет тем же — `2, 4, 6`.

Оба варианта абсолютно эквивалентны — компилятор просто превращает **query syntax** в **method syntax**. Например, выражение:

```
from n in numbers
where n % 2 == 0
select n
```

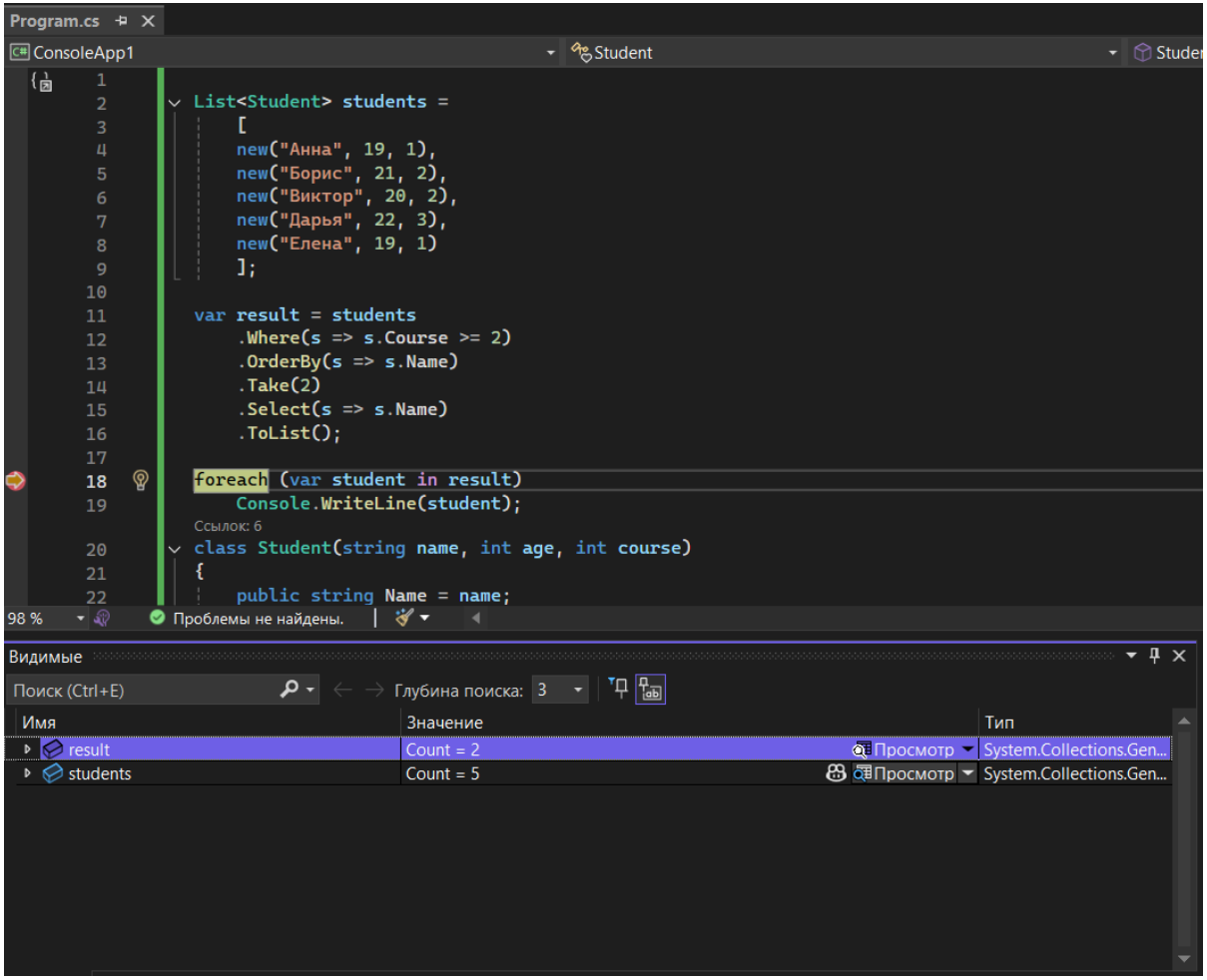
компилятор превращает в:

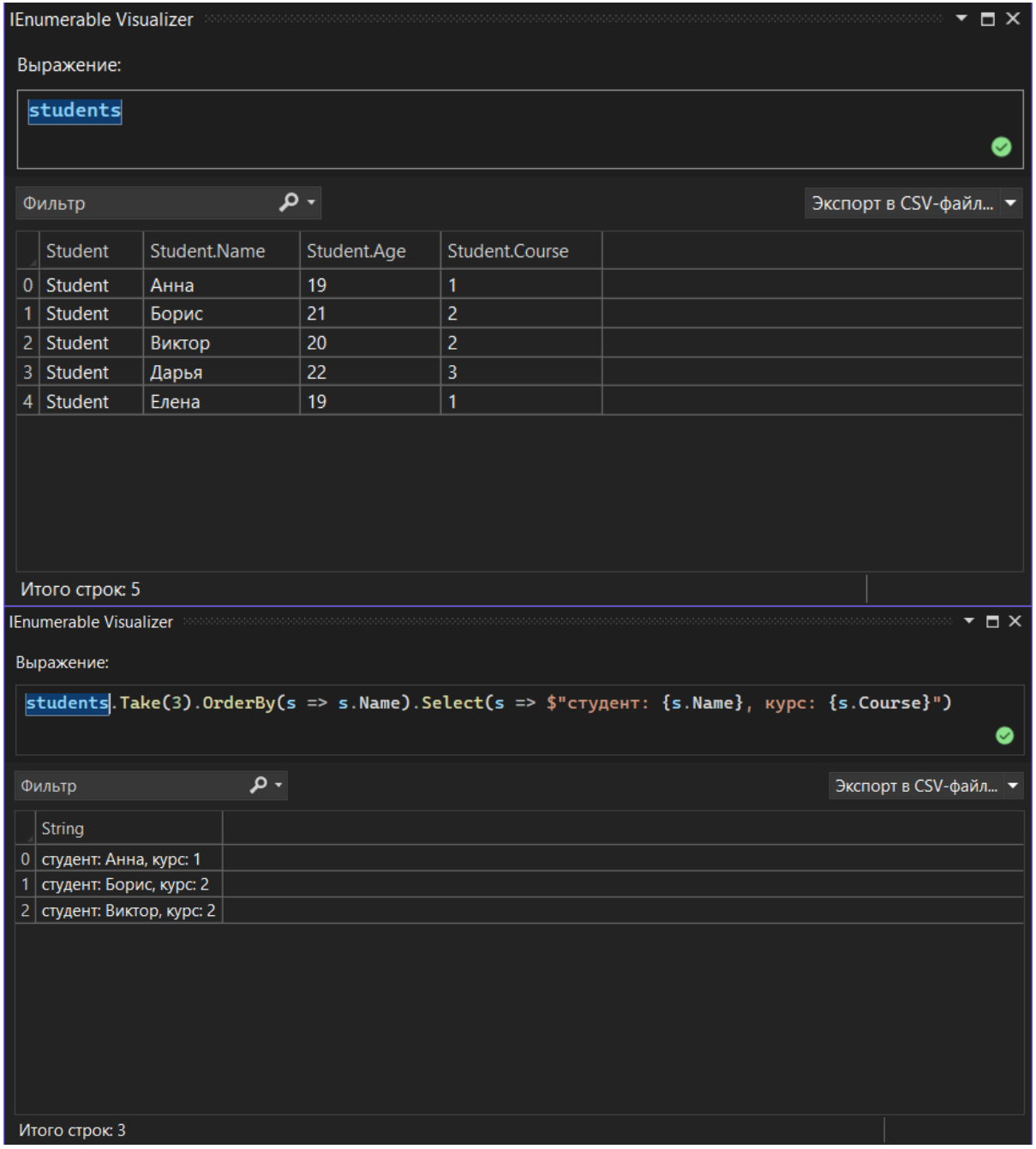
```
numbers.Where(n => n % 2 == 0).Select(n => n);
```

5. Интерактивная работа с LINQ

Работа с LINQ особенно удобна благодаря встроенным инструментам Visual Studio, которые позволяют интерактивно исследовать и делать отладку запросов.

Во время отладки кода, в списке переменных, рядом с переменными типа `IEnumerable` находится кнопка, которая позволяет посмотреть содержимое этой коллекции. Если нажать на нее, то откроется **IEnumerable Visualizer**. С помощью него можно не только смотреть данные, но так же создавать LINQ-запросы и сразу же увидеть, как при этом изменяются данные.




IEnumerable Visualizer
Выражение:
students
Фильтр
Экспорт в CSV-файл...
Student
Student.Name
Student.Age
Student.Course
0 Student Анна 19 1
1 Student Борис 21 2
2 Student Виктор 20 2
3 Student Дарья 22 3
4 Student Елена 19 1
Итого строк: 5
IEnumerable Visualizer
Выражение:
students.Take(3).OrderBy(s => s.Name).Select(s => $"студент: {s.Name}, курс: {s.Course}")
Фильтр
Экспорт в CSV-файл...
String
0 студент: Анна, курс: 1
1 студент: Борис, курс: 2
2 студент: Виктор, курс: 2
Итого строк: 3

6. Основная идея LINQ и Fluent-подход

В своей основе **LINQ** — это набор методов расширения для интерфейса `IEnumerable<T>`.
Каждая операция (`Where`, `Select`, `OrderBy`, `GroupBy` и т. д.) — это обычный метод,
который возвращает новую последовательность, над которой можно продолжать вызывать другие методы.

Такой стиль называется **Fluent-интерфейсом** (*Fluent API*),
поскольку вызовы методов **связываются в цепочку** (“текут” один за другим).

Рассмотрим пример фильтрации и сортировки списка студентов:

```
var students = new List<Student>
{
    new Student("Анна", 19),
    new Student("Борис", 22),
    new Student("Виктор", 17),
    new Student("Дарья", 21)
};

// Fluent-цепочка
var result = students
    .Where(s => s.Age >= 18)    // фильтрация
    .OrderBy(s => s.Name)       // сортировка
    .Select(s => s.Name);       // проекция
```

Каждый метод возвращает `IEnumerable<T>`, поэтому следующий метод можно вызывать сразу после предыдущего.

Как это устроено

LINQ реализует **композицию функций** — каждый вызов не изменяет коллекцию, а создаёт **новую последовательность**, которая будет обработана только при необходимости.

Например, цепочка:

```
students.Where(...).Select(...).OrderBy(...)
```

эквивалентна последовательному применению функций:

```
OrderBy(Select(Where(students)))
```

Благодаря этому:

- Код становится **декларативным**: вы описываете, *что нужно получить*, а не *как это сделать*.
- Можно **гибко комбинировать** методы без промежуточных циклов и временных коллекций.
- Все коллекции, реализующие `IEnumerable<T>`, автоматически поддерживают LINQ.

7. Реализация базовых LINQ-операций вручную

Чтобы лучше понять, как работает LINQ под капотом, давайте реализуем некоторые его базовые операции самостоятельно — `Where` и `Select`.

Реализация метода Where

Метод `Where` фильтрует коллекцию, возвращая только те элементы, которые удовлетворяют заданному условию (предикату).

```
public static class MyLinqExtensions
{
    // Расширение для IEnumerable<T>
    public static IEnumerable<T> WhereMy<T>(
        this IEnumerable<T> source,
        Func<T, bool> predicate)
    {
        foreach (var item in source)
        {
            if (predicate(item))
                yield return item; // возвращаем только подходящие элементы
        }
    }
}
```

Здесь `yield return` позволяет возвращать элементы постепенно, без создания промежуточных списков. То есть элементы фильтруются “на лету” — это называется **отложенное выполнение**.

Реализация метода Select

Метод `Select` преобразует элементы коллекции — например, выбирает одно поле из объекта или изменяет тип данных.

```
public static class MyLinqExtensions
{
    public static IEnumerable<TResult> SelectMy<TSource, TResult>(
        this IEnumerable<TSource> source,
        Func<TSource, TResult> selector)
    {
        foreach (var item in source)
        {
            yield return selector(item); // применяем функцию-преобразователь
        }
    }
}
```

Пример использования

```
var numbers = new List<int> { 1, 2, 3, 4, 5 };

// Цепочка вызовов
var result = numbers
    .WhereMy(n => n % 2 == 0)     // выбираем чётные
    .SelectMy(n => n * n);        // возводим в квадрат

foreach (var n in result)
    Console.WriteLine(n);
```

8. Отложенное выполнение

Одно из ключевых свойств LINQ — это **отложенное выполнение**. Это означает, что запрос выполняется не сразу, когда вы его объявляете, а лишь тогда, когда начинается перебор его результатов — например, через `foreach`, `ToList()` или другие операции, которые реально обращаются к данным.

Пример

```
var numbers = new List<int> { 1, 2, 3, 4, 5 };

// создаём запрос, но он ещё не выполняется!
var query = numbers.Where(n => n > 2);

numbers.Add(6); // изменяем коллекцию после создания запроса

foreach (var n in query)
    Console.WriteLine(n);
```

Результат:

```
3
4
5
6
```

Как видно, элемент `6` тоже попал в результат, потому что LINQ-запрос вычисляется только в момент выполнения `foreach`, а не при его объявлении.

Когда это удобно

- Можно описывать сложные цепочки запросов, не создавая промежуточных коллекций.
- LINQ сам обрабатывает элементы по мере необходимости, что экономит память и ускоряет работу с большими данными.
- Отлично подходит для потоковой обработки данных или чтения из файлов, БД и API.

Когда это может быть проблемой

Иногда нам нужно “зафиксировать” состояние данных на конкретный момент времени.
Если коллекция изменится — результат запроса изменится тоже.

В таких случаях нужно преобразовать результат в конкретную коллекцию с помощью:

```
var result = query.ToList();  // выполняет запрос и сохраняет результат
```

Теперь, даже если исходный список изменится, `result` останется прежним.

9. Производительность и лучшие практики

LINQ — это мощный инструмент, который делает код короче, выразительнее и понятнее,
но за удобство иногда приходится платить — в первую очередь производительностью.
Важно понимать, как именно LINQ работает “под капотом”, чтобы использовать его эффективно.

LINQ-запросы добавляют дополнительный уровень абстракции: каждая операция (`Where`, `Select`, `OrderBy` и т.д.) создаёт новый итератор, поэтому при больших объёмах данных или в часто вызываемых циклах это может приводить к ненужным накладным расходам.

Оптимизация LINQ

1. Не вызывайте `ToList()` без необходимости

Многие добавляют `ToList()` в конце цепочки на всякий случай, но это приводит к немедленному выполнению запроса и созданию новой коллекции в памяти.

Используйте `ToList()` только тогда, когда нужно зафиксировать результат
или повторно использовать его несколько раз.

```
// Плохо — создаётся лишний список
var data = numbers.Where(n => n > 0).ToList();

// Хорошо — запрос выполняется по мере обхода
foreach (var n in numbers.Where(n => n > 0))
    Console.WriteLine(n);
```

2. Избегайте LINQ внутри циклов

Вызов LINQ внутри вложенных циклов, может привести к многократному пересозданию итераторов.

```
// Плохо — запрос выполняется на каждой итерации
foreach (var item in list1)
{
    if (list2.Any(x => x.Id == item.Id)) { ... }
}
```

Лучше вынести результат запроса заранее или использовать коллекцию для быстрого поиска:

```
var lookup = list2.Select(x => x.Id).ToHashSet();

foreach (var item in list1)
{
    if (lookup.Contains(item.Id)) { ... }
}
```

3. Понимайте, где запрос выполняется один раз, а где — при каждом проходе

Помните об отложенном выполнении: если вы итерируете один и тот же запрос несколько раз, он будет выполняться заново каждый раз.

```
var query = numbers.Where(n => n > 10);

Console.WriteLine(query.Count());  // выполняется первый раз
Console.WriteLine(query.Count());  // выполняется второй раз
```

Если данные не меняются, лучше зафиксировать результат:

```
var cached = query.ToList();
Console.WriteLine(cached.Count);
Console.WriteLine(cached.Count);
```

4. Иногда обычный цикл быстрее

Для простых фильтраций или подсчётов `foreach` может быть эффективнее.
LINQ — это инструмент для **удобства и читаемости**,
а не для критически важных участков кода.

Практическое задание

Что нужно сделать:

1. **Добавьте новую консольную команду `search`:**
   - Команда не изменяет данные, а только выполняет поиск и вывод результата.
   - Возвращает список задач, удовлетворяющих условиям поиска.
2. **Реализуйте поддержку флагов команды `search`:**
   Флаги для работы с текстом задачи (текст в кавычках):

   - `--contains <text>` — задачи, содержащие указанный текст.
   - `--starts-with <text>` — задачи, начинающиеся с указанного текста.
   - `--ends-with <text>` — задачи, заканчивающиеся указанным текстом.
     Флаги для работы с датами:
   - `--from <date>` — задачи, у которых дата последнего изменения **не раньше** указанной.
   - `--to <date>` — задачи, у которых дата последнего изменения **не позже** указанной.
     Формат даты:

```
yyyy-MM-dd
```

   Флаги для статуса:

   - `--status <status>`
     Флаги сортировки:
   - `--sort text` — сортировка по тексту задачи.
   - `--sort date` — сортировка по дате последнего изменения.
   - `--desc` — сортировка по убыванию (по умолчанию — по возрастанию).
     Ограничение результата:
   - `--top <n>` — вывести только первые `n` задач после фильтрации и сортировки.

3. **Реализация через LINQ:**
   - Все операции поиска, фильтрации, сортировки и ограничения должны быть реализованы **через LINQ**:
     - `Where`
     - `OrderBy` / `OrderByDescending`
     - `Take`
     - `ThenBy`
   - Запрещено использовать циклы `for` / `foreach` для фильтрации.
4. **Формирование результата:**
   - Результаты команды `search` должны выводиться **в виде таблицы**, аналогично команде `view`.
   - Колонки:

     ```
     Index | Text | Status | LastUpdate
     ```

   - Текст задачи выводить укороченным (например, первые 30 символов + `...`).
5. **Интеграция с существующей архитектурой:**
   - Команда `search` должна быть реализована как отдельный класс, реализующий `ICommand`.
   - Создание команды должно происходить через `CommandParser`:
     - регистрация обработчика в словаре делегатов

       ```
       Dictionary<string, Func<string, ICommand>>
       ```

   - Использовать `CommandParser.Todos` для доступа к задачам текущего пользователя.
6. **Обработка ошибок и проверок:**
   - Проверять корректность дат (`DateTime.TryParse`).
   - Проверять числовые параметры (`top`).
   - Если не найдено ни одной задачи — выводить сообщение:

     ```
     Ничего не найдено
     ```

Примеры использования команды:

```
search --contains "work at"
search --starts-with "fix" --sort date --desc
search --from 2024-01-01 --to 2024-02-01
search --status in-progress --top 5
search --contains "test" --sort text
```