

## Лабораторная работа №10. Современные подходы: LINQ, лямбды и асинхронность

**Цель:** Освоить современные технологии и синтаксические возможности C#, включая лямбда-выражения, анонимные типы, интерфейсы и основы асинхронного программирования.

### Задачи:

- Использовать анонимные типы и лямбда-выражения для краткой записи логики.
- Определить и реализовать интерфейсы, применяя принципы SOLID.
- Ознакомиться с асинхронным программированием (async, await) и корутинами.
- Решать задачи с использованием LINQ для обработки коллекций.

### Шаг 1. Анонимные типы

Анонимные типы — это **"одноразовые"** классы, которые создаются на лету **без явного объявления**. Они полезны, когда нужно быстро сгруппировать данные для временного использования.

### Основные свойства:

- **Не имеют имени** (отсюда "анонимные").
- **Только для чтения** (свойства нельзя изменить после создания).
- **Используются внутри метода** (нельзя вернуть из метода как обычный класс).
- **Компилятор сам генерирует для них класс**.

Объявите анонимный тип и выведите его на консоль:

```
static void Main(string[] args)
{
    var monster = new
    {
        Name = "Дракон",
        Damage = 50,
        IsBoss = true,
        Description = "Древний огнедышащий змей"
    };

    Console.WriteLine(monster.Description); // Древний огнедышащий змей
}
```

Другой простой пример:

```
static void Main(string[] args)
{
    var player = new { Name = "Артур", Health = 100, Level = 5 };
    Console.WriteLine($"Игрок: {player.Name}, Уровень: {player.Level}");
}
```

Таким образом, анонимные типы — это удобный способ хранения данных, без создания классов.

Давайте рассмотрим ещё один пример где мы создадим список товаров с ценами и категориями.

В начале создадим список анонимных типов:

```
// Создание списка анонимных объектов (товаров)
var products = new[]
{
    new { Name = "Молоко", Price = 59.99, Category = "Продукты" },
    new { Name = "Ноутбук", Price = 49990.00, Category = "Электроника" },
    new { Name = "Щётка для зубов", Price = 120.50, Category = "Гигиена" }
};
```

Затем выведем их через цикл **foreach**:

```
// Вывод данных
Console.WriteLine("Список товаров:");
foreach (var product in products)
{
    Console.WriteLine($"{product.Name} ({product.Category}) - {product.Price} P");
}
```

С делегатами тесно связаны **анонимные методы**. Анонимные методы используются для создания экземпляров делегатов.

Определение анонимных методов начинается с ключевого слова **delegate**, после которого идет в скобках список параметров и тело метода в фигурных скобках:

```
delegate(параметры)
{
    // инструкции
}
```

После создадим делегат и вызовем его:

```

delegate void MessageHandler(string message);
0 references
static void Main(string[] args)
{
    MessageHandler handler = delegate (string mes)
    {
        Console.WriteLine(mes);
    };
    handler("hello world!");
}

```

Анонимный метод не может существовать сам по себе, он используется для инициализации экземпляра делегата, как в данном случае переменная handler представляет анонимный метод. И через эту переменную делегата можно вызвать данный анонимный метод.

Другой пример анонимных методов - передача в качестве аргумента для параметра, который представляет делегат:

```

delegate void MessageHandler(string message);
0 references
static void Main(string[] args)
{
    ShowMessage("hello!", delegate (string mes)
    {
        Console.WriteLine(mes);
    });

    static void ShowMessage(string message, MessageHandler handler)
    {
        handler(message);
    }
}

```

## Шаг 2. Лямбды

Лямбда-выражения — это короткий способ записи анонимных функций. Они часто используются для:

- **Функций высшего порядка** (методы, принимающие функции как аргументы).
- **LINQ-запросов** (Where, Select, OrderBy).
- **Событий и делегатов.**

Лямбда-выражения бывают двух видов:

1. **Однострочные** (с => и возвратом значения):

(параметры) => выражение

## 2. Блочные (с фигурными скобками и return):

(параметры) => {

*// код...*

return результат;

}

Рассмотрим их на примере:

```
static void Main(string[] args)
{
    // 1. Через метод
    int Square(int x)
    {
        return x * x;
    }
    Console.WriteLine(Square(5));
    // 2. То же самое через лямбду
    Func<int, int> square = x => x * x;
    Console.WriteLine(square(5)); // 25
}
```

Также можем использовать лямбды в событиях.

Создаём условный класс кнопки:

```
// Создаём кнопку (условно)
2 references
public class Button
{
    public event Action OnClick;

    1 reference
    public void Click()
    {
        OnClick?.Invoke();
    }
}
```

И после имитируем нажатие на неё:

```
static void Main(string[] args)
{
    // Подписываемся через лямбду
    Button button = new Button();
    button.OnClick += () => Console.WriteLine("Кнопка нажата!");
    button.Click(); // Выведет: "Кнопка нажата!"
}
```

Можем использовать лямбду с несколькими параметрами:

```
static void Main(string[] args)
{
    // Сумма двух чисел
    Func<int, int, int> sum = (a, b) => a + b;
    Console.WriteLine(sum(3, 7)); // 10
}
```

Мы также можем вызывать несколько операций (блочная лямбда):

```
static void Main(string[] args)
{
    Func<int, string> numberToWord = num =>
    {
        switch (num)
        {
            case 1: return "один";
            case 2: return "два";
            default: return "много";
        }
    };

    Console.WriteLine(numberToWord(2)); // "два"
}
```

Обратите внимание, что лямбда — это анонимная функция, а делегат — тип, который может хранить ссылку на метод.

```
// Делегат (явное объявление)
delegate int MathOperation(int a, int b);
0 references
static void Main(string[] args)
{
    MathOperation add = (a, b) => a + b;
    Console.WriteLine(add(2, 3)); // 5
}
```

Зачем это нужно?

- **Сокращение кода:** Вместо методов можно писать короткие лямбды.
- **Гибкость:** Можно передавать логику как параметр.
- **Удобство:** Часто используются в асинхронном программировании и событиях.

Где применяются лямбды?

- **LINQ** (Where, Select, OrderBy).
- **События** (button.Click += () => ...).

- Асинхронные задачи (Task.Run(() => { ... })).
- Методы, принимающие функции (Find, ForEach).

### Шаг 3. Интерфейсы

Интерфейсы в C# — это контракты, которые определяют «что может делать класс», но не реализуют конкретное поведение. Они нужны для:

- Единого API для разных классов.
- Множественного наследования (в отличие от классов).
- Внедрения зависимостей (Dependency Injection).

Давайте рассмотрим простой пример. Допустим, у нас есть объекты, которые могут «звучать»: кошка, телефон, гитара.

За пределами метода **Main** объявим интерфейс:

```
public interface ISoundMaker
{
    3 references
    void MakeSound(); // Метод без реализации
}
```

После реализуем интерфейс в классах:

```
public class Cat : ISoundMaker
{
    2 references
    public void MakeSound()
    {
        Console.WriteLine("Мяу!");
    }
}
```

```
public class Phone : ISoundMaker
{
    2 references
    public void MakeSound()
    {
        Console.WriteLine("Дзынь-дзынь!");
    }
}
```

И затем используем в ключевом методе **Main**:

```
static void Main(string[] args)
{
    List<ISoundMaker> soundMakers = new List<ISoundMaker> { new Cat(), new Phone() };
    foreach (var item in soundMakers)
    {
        item.MakeSound(); // Вызовется своя реализация для каждого объект
    }
}
```

Интерфейсы могут содержать не только методы, но и свойства. Давайте реализуем ещё один пример.

```
public interface INamed
{
    1 reference
    string Name { get; set; } // Свойство
}

0 references
public class Book : INamed
{
    1 reference
    public string Name { get; set; }
}
```

И используем его в методе **Main**:

```
static void Main(string[] args)
{
    Book book = new Book { Name = "Война и мир" };
    Console.WriteLine(book.Name); // "Война и мир"
}
```

Ещё одним из плюсов интерфейсов, является возможность классов реализовывать их множество.

Создадим новые интерфейсы:

```
public interface IFlier
{
    2 references
    void Fly();
}
```

```
public interface ISwimmer
{
    2 references
    void Swim();
}
```

```
public class Duck : IFlier, ISwimmer
{
    2 references
    public void Fly() => Console.WriteLine("Утка летит!");
    2 references
    public void Swim() => Console.WriteLine("Утка плавёт!");
}
```

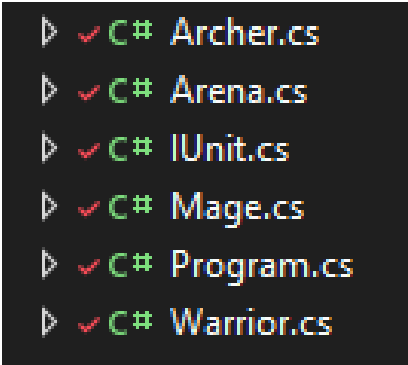
И вызовем в методе Main:

```
static void Main(string[] args)
{
    Duck duck = new Duck();
    ((IFlier)duck).Fly(); // Утка летит!
    ((ISwimmer)duck).Swim(); // Утка плавёт!
}
```

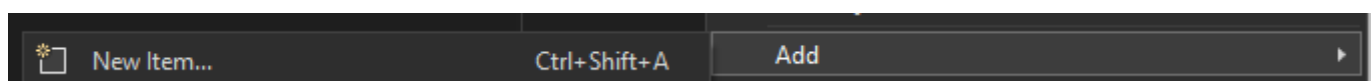
Давайте создадим с вами мини-проект: Боевая Арена с Интерфейсами.

В папке с проектом создайте следующие классы:

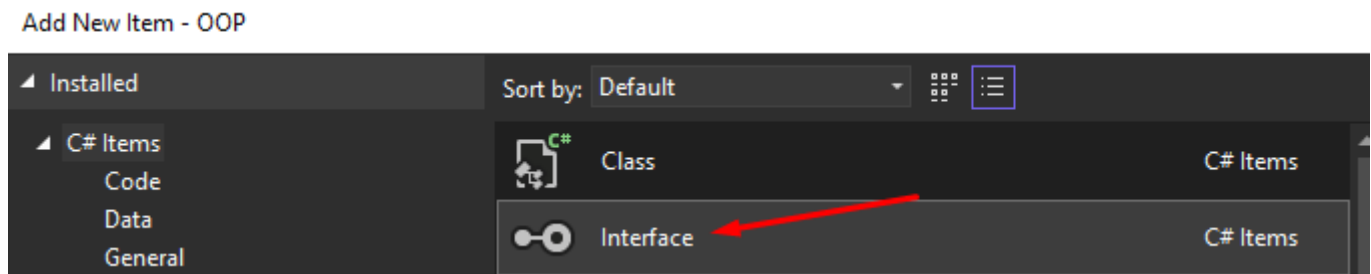
**Структура:**

	<ul style="list-style-type: none"> <li>• <b>IUnit.cs</b> — интерфейс</li> <li>• <b>Warrior.cs</b>, <b>Mage.cs</b>, <b>Archer.cs</b> — разные классы, реализующие интерфейс</li> <li>• <b>Arena.cs</b> — логика симуляции боя</li> <li>• <b>Program.cs</b> — точка входа, создающая арену и запускающая бой</li> </ul>
--	---

При создании мы можем выбирать пункт **Add – New Item**:



И там выбрать интерфейс:



В интерфейсе давайте с вами определим имя, здоровье и атаку:



```

public interface IUnit
{
    12 references
    string Name { get; }
    6 references
    int Health { get; }
    4 references
    void Attack(IUnit target);
}

```

Для воина можете реализовать по следующему примеру класс, который будет наследоваться от интерфейса IUnit:

```

public class Warrior : IUnit
{
    6 references
    public string Name { get; private set; }
    2 references
    public int Health { get; private set; }

    1 reference
    public Warrior(string name)
    {
        Name = name;
        Health = 150;
    }

    2 references
    public void Attack(IUnit target)
    {
        Console.WriteLine($"{Name} рубит мечом {target.Name}!");
    }
}

```

Для мага и лучника реализуйте сами классы.

Затем реализуйте класс арены:

```

public class Arena
{
    private List<IUnit> _units = new();

    3 references
    public void AddUnit(IUnit unit)
    {
        _units.Add(unit);
    }

    1 reference
    public void StartBattle()
    {
        Console.WriteLine("Битва началась!");
        foreach (var attacker in _units)
        {
            foreach (var defender in _units)
            {
                if (attacker != defender)
                    attacker.Attack(defender);
            }
        }
    }
}

```

И в конце в методе **Main** класса **Program**, создайте объект арены, и запустите туда бойцов:

```

static void Main(string[] args)
{
    Arena arena = new();
    arena.AddUnit(new Warrior("Рагнар"));
    arena.AddUnit(new Mage("Гендальф"));
    arena.AddUnit(new Archer("Леголас"));
    arena.StartBattle();
}

```

## Шаг 4. Асинхронное программирование

Асинхронное программирование — это способ **не блокировать главный поток выполнения** (обычно интерфейс пользователя или игровой цикл), пока ты, например:

- загружаешь файлы;
- отправляешь запрос в интернет;
- ждёшь завершения длительной операции.

Это как: ты поставил чайник и пока он кипит — продолжаешь читать книгу, а не просто стоишь и смотришь на него.

Ещё один пример. Представьте, что вы — повар в ресторане. У вас есть две задачи:

- варить суп (долгая задача, нужно ждать);
- резать салат (быстрая задача).

Если делать это синхронно (последовательно), вы потратите много времени на ожидание, пока суп сварится.

Но если работать асинхронно, вы сможете начать варить суп, переключиться на нарезку салата, а потом вернуться к супу, когда он будет готов.

В C#:

- **async**: указывает, что метод **асинхронный**.
- **await**: говорит программе **подожди здесь**, пока операция не завершится — но не блокируй остальной код.

Рассмотрим пример. Создаём асинхронный метод **BoilWaterAsync**. Он начинает "закипание" воды и ждёт 3 секунды асинхронно:

```
0 references
static void Main(string[] args)
{
}
0 references
static async Task BoilWaterAsync()
{
    Console.WriteLine("2. Кипятим воду (ждём 3 секунды)...");
    await Task.Delay(3000); // Асинхронная пауза
    Console.WriteLine("...вода закипела!");
}
```

Чтобы использовать `await`, метод `Main` должен быть асинхронным, и для этого нужно указать `static async Task Main(...)`.

```
0 references
static async Task Main()
{
}
0 references
```

Внутри мы вызовем `await BoilWaterAsync()`, и это даёт программе возможность не блокировать основной поток во время асинхронной операции (в нашем случае — `Task.Delay(3000)`):

```
static async Task Main()
{
    Console.WriteLine("1. Включили чайник...");
    await BoilWaterAsync();
    Console.WriteLine("3. Завариваем чай!");
}
```

Теперь давайте сделаем наш пример более жизненным и параллельным — пока вода кипит, мы будем резать лимон.

Создадим новый асинхронный метод:

```
static async Task SliceLemonAsync()
{
    Console.WriteLine("3. Режем лимон (ждём 2 секунды)...");
    await Task.Delay(2000);
    Console.WriteLine("...лимон порезан!");
}
```

Обновим наш метода **Main**:

```
static async Task Main()
{
    Console.WriteLine("1. Включили чайник...");

    // Запускаем оба действия параллельно
    Task boilTask = BoilWaterAsync();
    Task sliceTask = SliceLemonAsync();

    // Ждём, пока обе задачи завершатся
    await Task.WhenAll(boilTask, sliceTask);

    Console.WriteLine("4. Завариваем чай с лимоном!");
}
```

- Task boilTask = BoilWaterAsync(); — начинает кипятить воду **асинхронно**.
- Task sliceTask = SliceLemonAsync(); — одновременно режем лимон.
- await Task.WhenAll(...) — ждём обе задачи одновременно, не по очереди.

Для закрепления реализуем ещё один пример варки ужина. Создаём два метода приготовления супа и салата:

```

1 reference
public static async Task<string> MakeSoupAsync()
{
    Console.WriteLine("Начали варить суп...");
    await Task.Delay(3000);
    return "Суп";
}

1 reference
public static async Task<string> MakeSaladAsync()
{
    Console.WriteLine("Начали резать салат...");
    await Task.Delay(1000);
    return "Салат";
}

```

Затем создадим метод `CookDinnerAsync`

```

1 reference
public static async Task CookDinnerAsync()
{
    // Запускаем задачи параллельно
    Task<string> soupTask = MakeSoupAsync();
    Task<string> saladTask = MakeSaladAsync();
    // Ожидаем завершения всех задач
    string soup = await soupTask;
    string salad = await saladTask;

    Console.WriteLine($"Готово: {soup} и {salad}");
}

```

Здесь начинается настоящая магия:

- Оба метода вызываются и **начинаются параллельно**.
  - `MakeSoupAsync()` запустит задачу с 3-секундной задержкой.
  - `MakeSaladAsync()` — с 1-секундной.
- Ожидаем завершения обоих задач.
- Важно: салат будет готов через 1 секунду, и программа **не будет простаивать** всё это время — она продолжит ждать, пока суп тоже сварится.

И после в методе `Main` выведем сообщение о готовке ужина, затем вызываем метод `CookDinnerAsync()` и ждём, пока он завершится. После этого выводим "Ужин готов!":

```

static async Task Main()
{
    Console.WriteLine("Начинаем готовить ужин...");
    // Вызываем асинхронный метод и ждём его завершения
    await CookDinnerAsync();
    Console.WriteLine("Ужин готов!");
}

```

## Шаг 5. Корутины

**Корутина** — это метод, который может **приостанавливать свою работу**, подождать сколько нужно, а потом **продолжить с того же места**. В С# такую механику можно реализовать с помощью ключевого слова `yield`.

То есть корутина как бы говорит: > «Я сделаю немного, потом отдохну, потом снова продолжу». В Unity активно используются данные методы, поэтому разберём их.

Обратите внимание что в старых версиях С# нужно подключать коллекции:

```
using System.Collections.Generic;
```

Итак, сперва создадим корутину:

```

static IEnumerable<int> GenerateNumbers()
{
    Console.WriteLine("Начинаем генерацию...");
    yield return 1;

    Console.WriteLine("Пауза...");
    yield return 2;

    Console.WriteLine("И ещё раз...");
    yield return 3;

    Console.WriteLine("Готово!");
}

```

И затем в методе `Main` пройдемся через цикл:

```

static void Main(string[] args)
{
    foreach (int number in GenerateNumbers())
    {
        Console.WriteLine($"Получено число: {number}");
    }
}

```

- `yield return` возвращает значение, **приостанавливая** выполнение метода.
- При следующем обращении цикл `foreach` снова «будит» метод, и тот продолжает с того места, где остановился.

В Unity корутины используют `IEnumerator`, как и в примере выше, но вместо чисел часто «возвращают» время ожидания.

Корутины и `async/await` в C# действительно похожи: оба позволяют выполнять действия со временем без блокировки. Но у них *разная механика, цели и области применения*. Вот как можно их сравнить:

	Корутины ( <code>IEnumerator</code> )	Асинхронность ( <code>async/await</code> )
Ключевое слово	<code>yield return</code>	<code>await</code>
Возвращаемый тип	<code>IEnumerator</code> (или <code>IEnumerator&lt;T&gt;</code> )	<code>Task</code> / <code>Task&lt;T&gt;</code>
Где часто используется	Unity	.NET (WPF, консоль, веб, и др.)
Можно приостанавливаться и продолжать	✓	✓
Управляется кем	Игровой движок (например, Unity через <code>StartCoroutine</code> )	Планировщик задач CLR
Поддержка параллелизма (выполнение в фоне)	✗ (не по умолчанию)	✓

- **Корутины** больше похожи на **последовательный сценарий поведения**:
  - подождать 2 секунды,
  - проиграть анимацию,
  - затем воспроизвести звук,
  - затем активировать объект.

Используются чаще в Unity, потому что легко интегрируются с игровым циклом:

```
yield return new WaitForSeconds(1f);
```

- **Async/await** — это способ **работать с внешними асинхронными источниками**:
  - сетевыми запросами,
  - файлами,
  - базами данных,
  - запускать что-то в фоне без блокировки.

Например: `await DownloadDataAsync();`

**Проще говоря...**

- **Корутина** = "Поставь действие в очередь, дождись — и продолжай".
- **Async/await** = "Делай задачу в фоне, сообщи мне, когда будет готово".

Давайте рассмотрим это на примере. Сперва создадим вариант с корутиной:

```
static void Main(string[] args)
{
    foreach (var step in WaitCoroutine())
    {
        Console.WriteLine(step);
        Thread.Sleep(1000); // имитируем "ожидание" как в Unity
    }

    Console.WriteLine("Готово (корутина)!");
}
1 reference
static IEnumerable<string> WaitCoroutine()
{
    yield return "Ждём... 1 секунда";
    yield return "Ждём... 2 секунда";
}
```

Здесь yield return возвращает шаги по одному. Мы вручную делаем паузу в Main, как будто Unity сама управляла временем ожидания.

Теперь решим эту задачу вторым вариантом (асинхронно):

```
static async Task Main()
{
    Console.WriteLine("Ждём... 2 секунды");
    await Task.Delay(2000); // реальная асинхронная пауза
    Console.WriteLine("🕒 Готово (async/await)!");
}
```

Тут Task.Delay приостанавливает **только текущую задачу**, но **не блокирует поток**, то есть в WPF, WinForms или ASP.NET интерфейс остаётся отзывчивым.



	Корутина (yield)	Async/await
Основано на	<code>IEnumerator</code>	<code>Task</code>
Останавливает выполнение	вручную, по шагам	автоматически с <code>await</code>
Подходит для	сценариев "пошагово"	фоновых задач, таймеров, веб-запросов
Блокирует поток?	Да (если <code>Thread.Sleep()</code> )	Нет
Уровень гибкости	Просто	Более мощно и гибко

## Шаг 6. LINQ

**LINQ (Language Integrated Query)** — это встроенный в C# язык запросов, который позволяет удобно фильтровать, сортировать и преобразовывать данные. Давайте сравним, как одна и та же задача решается **без LINQ** и **с LINQ**, чтобы увидеть разницу.

### Задача: Обработка списка чисел

Допустим, у нас есть список чисел, и нужно:

1. Выбрать только чётные.
2. Умножить их на 10.
3. Отсортировать по убыванию.
4. Вывести результат.

Сперва решим стандартным способом.

Создадим список чисел:

```
List<int> numbers = new List<int> { 5, 2, 9, 4, 7, 3, 8, 23, 45, 12, 34, 13, 98, 67, 56, 99 };
```

1. Затем сделаем фильтрацию чётных чисел:

```
// 1. Фильтрация чётных чисел
List<int> evenNumbers = new List<int>();
foreach (var num in numbers)
{
    if (num % 2 == 0)
        evenNumbers.Add(num);
}
```

2. Затем умножим числа на 10:

```
// 2. Умножение на 10
List<int> multiplied = new List<int>();
foreach (var num in evenNumbers)
{
    multiplied.Add(num * 10);
}
```

3. Сделаем сортировку по убыванию:

```
// 3. Сортировка по убыванию
multiplied.Sort((a, b) => b.CompareTo(a));
```

4. Выведем наши числа:

```
// 4. Вывод
Console.WriteLine("Результат (без LINQ):");
foreach (var num in multiplied)
{
    Console.WriteLine(num);
}
```

Теперь перепишем решение с помощью **LINQ**. Для этого нам нужно в начале программы подключить пространство имён для работы:

```
using System.Linq;
```

И заменим пункты с 1 по 4:

```
var result = numbers
    .Where(n => n % 2 == 0) // 1. Фильтрация чётных
    .Select(n => n * 10) // 2. Умножение на 10
    .OrderByDescending(n => n); // 3. Сортировка по убыванию

// 4. Вывод
Console.WriteLine("Результат (с LINQ):");
foreach (var num in result)
{
    Console.WriteLine(num);
}
```

Рассмотрим ещё пару примеров.

Используем фильтрацию строк для вывода длинных имён (больше 4 символов):

```

List<string> names = new List<string> { "Анна",
    "Иван", "Мария", "Пётр" };

// Без LINQ
List<string> longNames = new List<string>();
foreach (var name in names)
{
    if (name.Length > 4)
        longNames.Add(name);
}
foreach (var name in longNames)
{
    Console.WriteLine(name);
}

```

Теперь посмотрим, как это выглядит с LINQ:

```

// С LINQ
var linqLongNames = names.Where(name => name.Length > 4);

foreach (var name in linqLongNames)
{
    Console.WriteLine(name);
}

```

Следующий пример – поиск минимального числа.

```

List<int> nums = new List<int> { 10, 5, 20, 3 };

// Без LINQ
int min = nums[0];
foreach (var num in nums)
{
    if (num < min)
        min = num;
}

Console.WriteLine($"Минимальное число (без LINQ): {min}");

```

С LINQ:

```
List<int> nums = new List<int> { 10, 5, 20, 3 };

// с LINQ
int linqMin = nums.Min();
Console.WriteLine($"Минимальное число (с LINQ): {linqMin}");
```

Теперь решим задачу нахождения квадратов чисел:

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

// Без LINQ
List<int> squares = new List<int>();
foreach (var num in numbers)
{
    squares.Add(num * num);
}

Console.WriteLine("\nКвадраты чисел (без LINQ):");
Console.WriteLine(string.Join(", ", squares));
```

```
// с LINQ
var linqSquares = numbers.Select(n => n * n);
Console.WriteLine("Квадраты чисел (с LINQ):");
Console.WriteLine(string.Join(", ", linqSquares));
```

Теперь давайте следующий сценарий. У нас есть список студентов и их успеваемости. Мы хотим:

- Найти всех, у кого средний балл выше 80.
- Отсортировать их по имени.
- Посчитать средний балл по всем студентам.

Создадим класс студента, у которого есть имя и средний балл:

```
class Student
{
    public string Name;
    public int Score;
}
```

Затем в методе Main создадим список студентов:

```
List<Student> students = new()
{
    new Student { Name = "Андрей", Score = 90 },
    new Student { Name = "Вика", Score = 78 },
    new Student { Name = "Денис", Score = 84 },
    new Student { Name = "Лена", Score = 95 },
    new Student { Name = "Олег", Score = 65 }
};
```

После создаём пустой список, куда будем сохранять результат выбора и переменную для счёта:

```
List<Student> goodStudents = new List<Student>();
int total = 0;
```

Пробегаемся по циклу, чтобы сохранить в пустой список нужных нам студентов:

```
foreach (var s in students)
{
    total += s.Score;
    if (s.Score > 80)
        goodStudents.Add(s);
}
```

Делаем сортировку воспользовавшись встроенными методами:

```
goodStudents.Sort((a, b) => a.Name.CompareTo(b.Name));
```

И дальше выводим наших студентов:

```
Console.WriteLine("Студенты с баллом выше 80:");
foreach (var s in goodStudents)
{
    Console.WriteLine($"{s.Name} - {s.Score}");
}

Console.WriteLine($"Средний балл: {(double)total / students.Count:F1}");
```

Теперь перепишем наш код, используя LINQ. Оставляем только наш список:

```
static void Main(string[] args)
{
    List<Student> students = new()
    {
        new Student { Name = "Андрей", Score = 90 },
        new Student { Name = "Вика", Score = 78 },
        new Student { Name = "Денис", Score = 84 },
        new Student { Name = "Лена", Score = 95 },
        new Student { Name = "Олег", Score = 65 }
    };
}
```

И напишем код:

```
var goodStudents = students
    .Where(s => s.Score > 80)
    .OrderBy(s => s.Name);

var averageScore = students.Average(s => s.Score);

Console.WriteLine("Студенты с баллом > 80:");
foreach (var s in goodStudents)
    Console.WriteLine($"{s.Name} - {s.Score}");

Console.WriteLine($"Средний балл: {averageScore:F1}");
```

## Самостоятельные задания

### Задание 1 — Анонимные типы

1. Создайте анонимный объект student с полями Name, Age, Group.
2. Выведите значения всех свойств в консоль.
3. Создайте список таких объектов (5 шт.) и выведите в формате таблицы.

### Задание 2 — Лямбда-выражения

1. Создайте список чисел от 1 до 20.
2. Используя List<int>.FindAll, получите:
  - Только чётные числа.
  - Только числа, кратные 3.
3. Для каждого результата — выведите список.

### Задание 3 — Интерфейсы

1. Создайте интерфейс `IPrintable` с методом `void PrintInfo()`;
2. Реализуйте интерфейс в классах `Book` и `Magazine`.
3. Создайте список `List<IPrintable>`, заполните его объектами `Book` и `Magazine`, вызовите `PrintInfo()` у каждого.

### Задание 4 — LINQ-запросы

1. Создайте массив строк с названиями городов.
2. Используя LINQ:
  - Найдите города, начинающиеся на букву «К».
  - Отсортируйте по длине названия.
  - Получите список городов длиной более 6 символов.

### Задание 5 — Асинхронное программирование

1. Создайте метод `Task<int> GetDataAsync()`, который возвращает число после задержки в 2 секунды.
2. В `Main` вызовите метод асинхронно и выведите результат.
3. Добавьте сообщение "Ждём данные..." до вызова, и "Данные получены!" — после.

### Задание 6 — Мини-проект "Университет"

Создайте систему:

1. Интерфейс `IPerson` с методом `GetDescription()`.
2. Классы `Student`, `Teacher`, реализующие `IPerson`.
3. Коллекцию людей, LINQ-запросы:
  - Все студенты старше 20 лет.
  - Учителя с определённой кафедры.
4. Асинхронный метод `LoadStudentsAsync()` — имитация загрузки данных (через `Task.Delay`).
5. Используйте лямбду для сортировки по имени.