

Лабораторная работа №9. Модулярность и события: библиотеки и делегаты

Цель: Научиться структурировать проект с помощью библиотек классов, подключать внешние пакеты и использовать делегаты и события для построения реактивных систем.

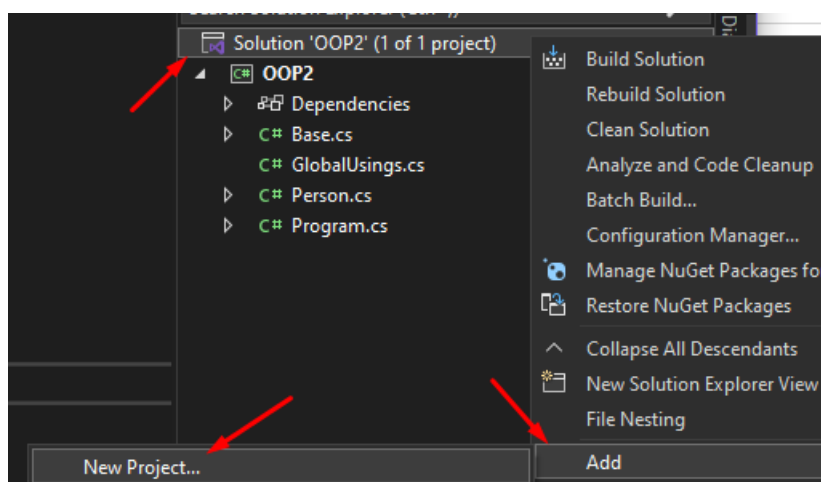
Задачи:

- Создать и подключить библиотеку классов (Class Library).
- Установить NuGet-пакет и применить его в проекте.
- Разобраться с понятием null в контексте ссылочных и значимых типов.
- Изучить делегаты и события — как основную модель событийного программирования.
- Реализовать мини-проект «Умный дом» с использованием событий и делегатов.

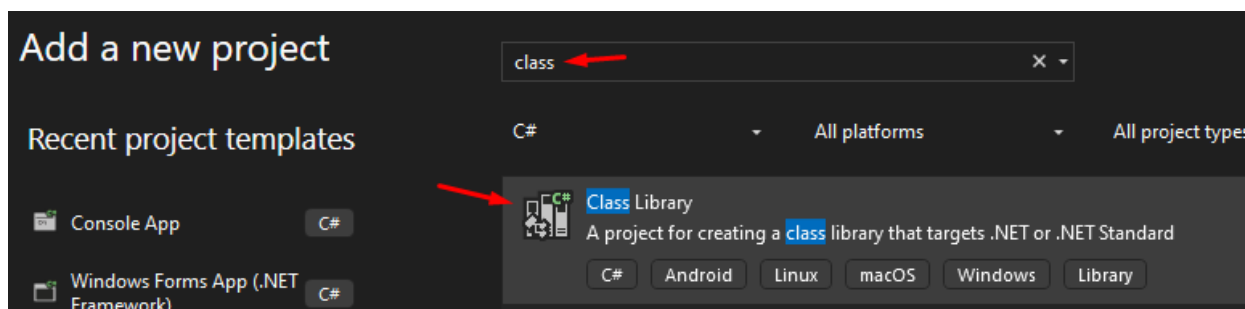
Шаг 1. Создание библиотеки классов в Visual Studio 2022

Нередко различные классы и структуры оформляются в виде отдельных библиотек, которые компилируются в файлы dll и затем могут подключаться в другие проекты. Благодаря этому мы можем определить один и тот же функционал в виде библиотеки классов и подключать в различные проекты или передавать на использование другим разработчикам.

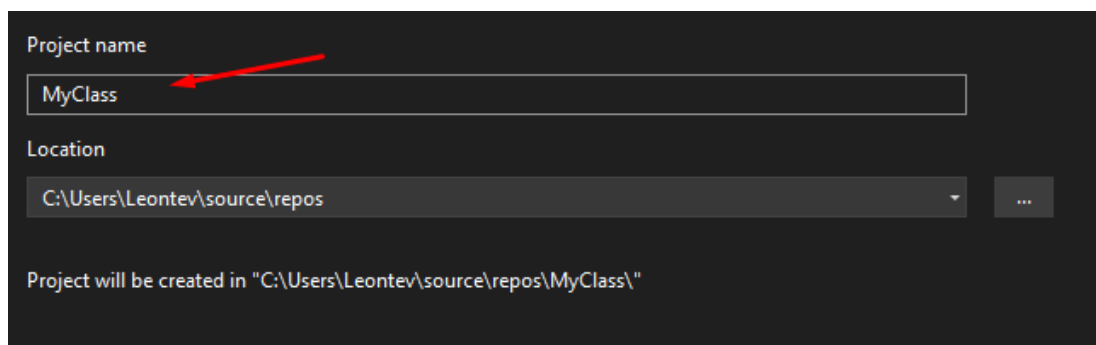
Создадим и подключим библиотеку классов. Создадим в текущем проекте новый – **Add – New Project:**



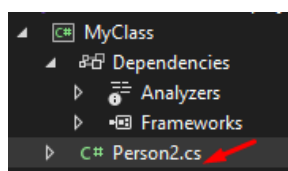
Ищем **Class Library**:



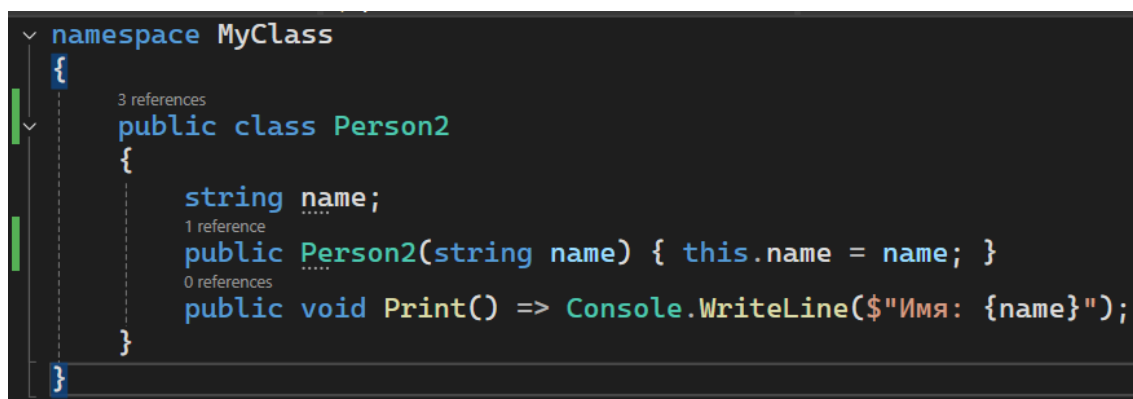
Называем его **MyClass**:



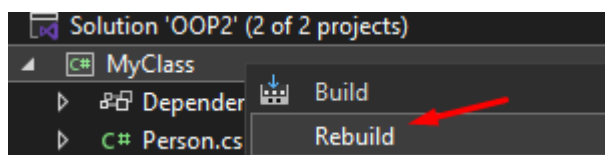
Переименуем файл **Class1.cs** в **Person2.cs**:



И определите простейший код:



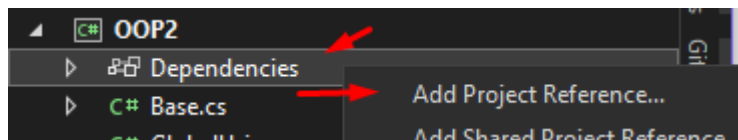
Теперь скомпилируем библиотеку классов. Для этого нажмем правой кнопкой на проект библиотеки классов и в контекстном меню выберем пункт **Rebuild**:



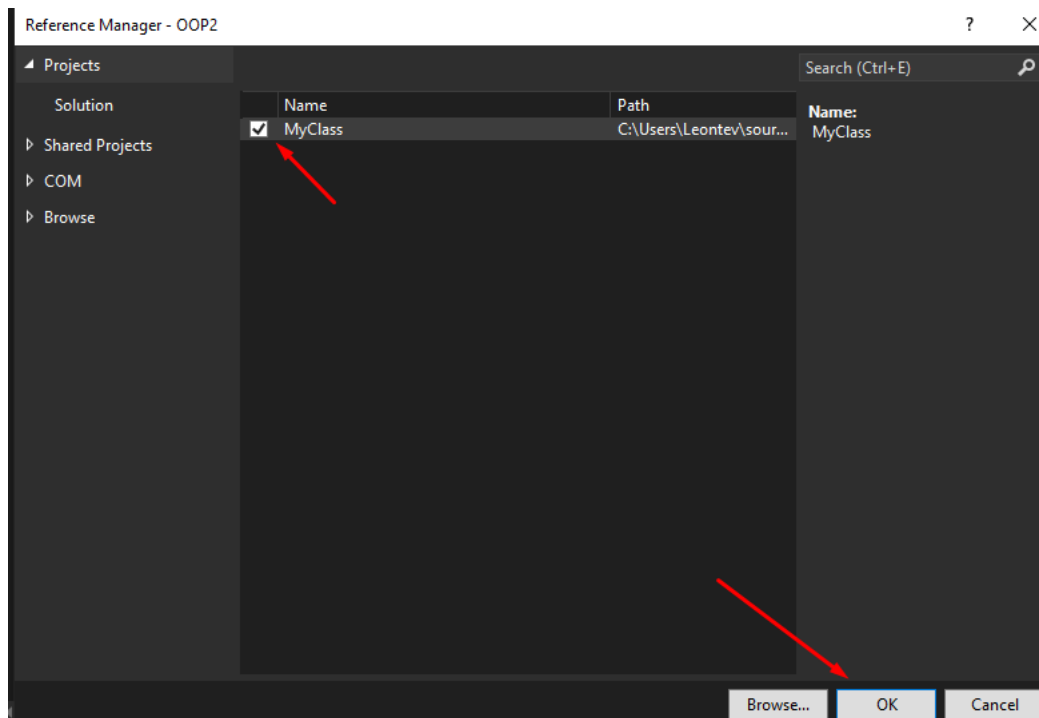
После компиляции библиотеки классов в папке проекта в каталоге **bin/Debug/net8.0** мы сможем найти скомпилированный файл **dll (MyClass.dll)**.

C:\Users\Leontev\source\repos\MyClass\bin\Debug\net8.0				
Имя	Дата изменения	Тип	Размер	
MyClass.deps.json	19.06.2025 16:24	Исходный файл J...	1 КБ	
MyClass.dll	19.06.2025 16:24	Расширение при...	4 КБ	
MyClass.pdb	19.06.2025 16:24	Program Debug D...	11 КБ	

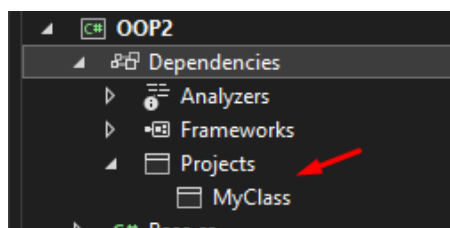
Подключим его в основной проект. Для этого в основном проекте нажмем правой кнопкой на узел **Dependencies** и в контекстном меню выберем пункт **Add Project Reference**:



Далее нам откроется окно для добавления библиотек. В этом окне выберем пункт **Solution**, который позволяет увидеть все библиотеки классов из проектов текущего решения, поставим отметку рядом с нашей библиотекой и нажмем на кнопку **OK**:



Развернув зависимости, вы можете увидеть, что они появились в проекте:



После успешного подключения библиотеки в главном проекте изменим файл **Program.cs**, чтобы он использовал класс **Person** из библиотеки классов:

```
1 using MyClass;  
2 Person2 anton = new("Anton");
```

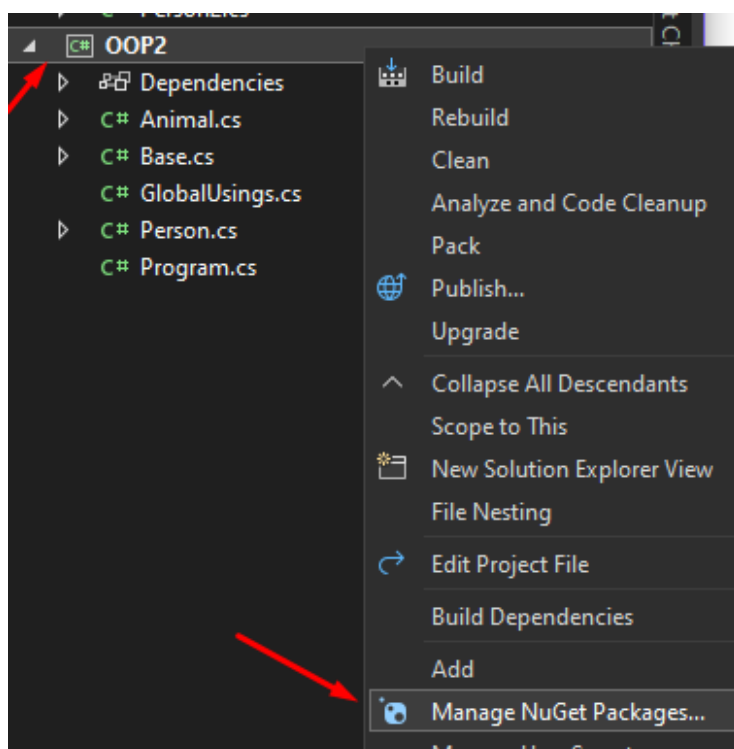
Шаг 2. Установка пакетов Nuget

NuGet — это **менеджер пакетов** для C# (как npm для JavaScript или pip для Python). С помощью NuGet можно добавить:

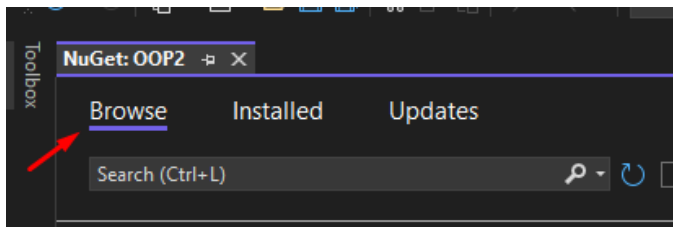
- библиотеки для работы с JSON,
- базы данных,
- логирование,
- парсеры, графики, и многое другое.

Рассмотрим на примере **Newtonsoft.Json**. Это один из самых популярных пакетов. Он позволяет удобно работать с JSON-файлами: преобразовывать объекты в JSON и наоборот.

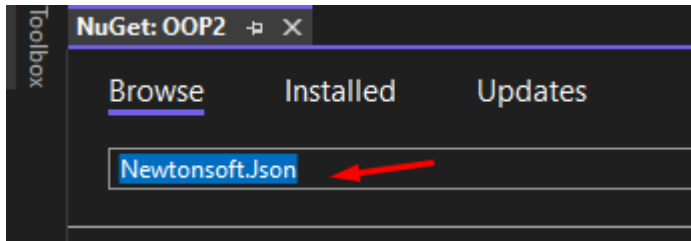
В новом или текущем проекте Щёлкните **ПКМ** по проекту в **Solution Explorer** → **Manage NuGet Packages**:



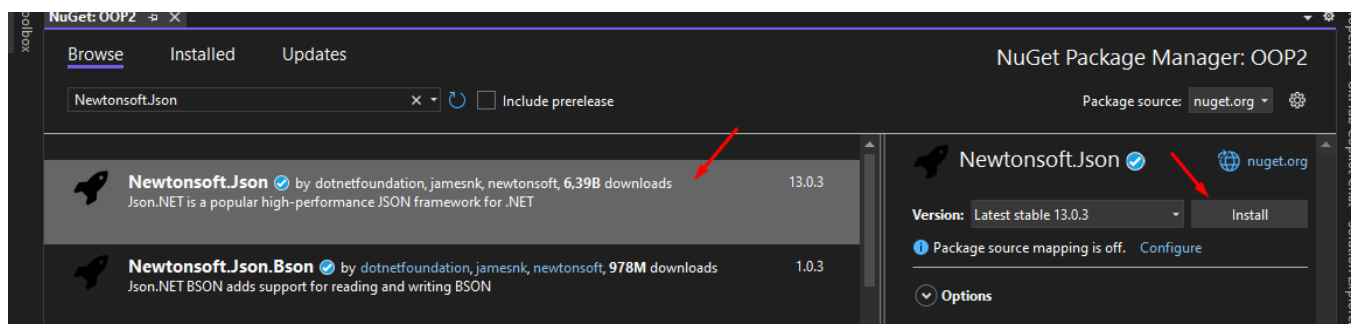
Перейдите во вкладку **Browse**:



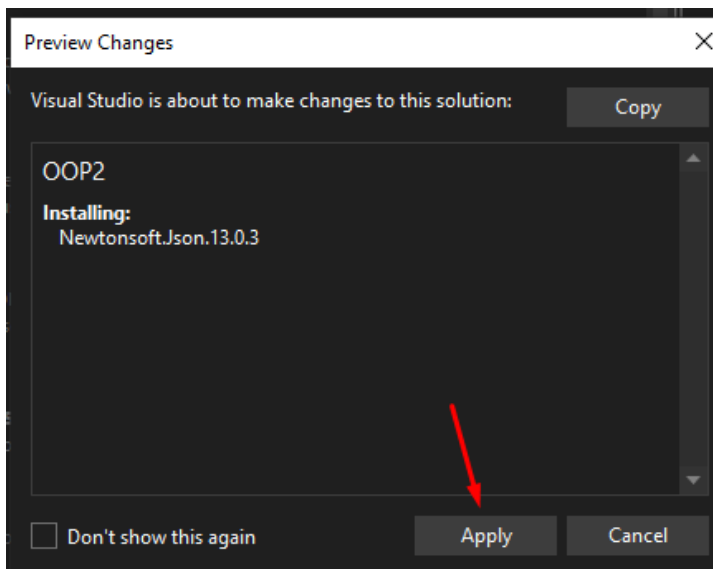
Введите: **Newtonsoft.Json**



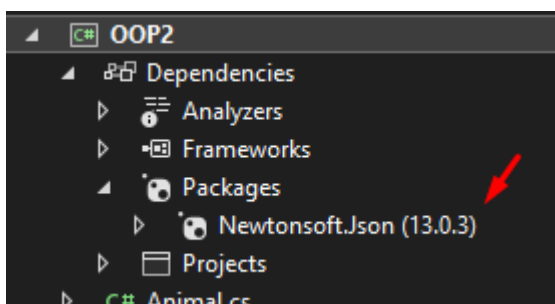
Выбери пакет, нажми **Install**:



Нажмите **Apply**:



После установки в проекте в зависимостях появится ссылка на библиотеку:



В начале подключите библиотеку:

```
using Newtonsoft.Json;
```

Дальше создайте класс фруктов с двумя свойствами (имя и количество):

```
public class Fruit
{
    2 references
    public string? Name { get; set; }
    2 references
    public int Quantity { get; set; }
}
```

Затем в методе **Main()** создадим объект **apple**:

```
static void Main()
{
    Fruit apple = new() { Name = "Яблоко", Quantity = 5 };
}
```

Затем преобразуем объект в строку **JSON** и выведем результат в консоль:

```
// Сериализация в JSON
string json = JsonConvert.SerializeObject(apple);
Console.WriteLine("В JSON: " + json);
```

После преобразует **JSON**-строку обратно в объект типа **Fruit** и выведем в консоль:

```
// Десериализация обратно
var deserialized = JsonConvert.DeserializeObject<Fruit>(json);
Console.WriteLine($"Объект: {deserialized?.Name} – {deserialized?.Quantity} шт.");
```

Теперь разберём ещё один пример. Вам нужно самостоятельно подключать сторонний **NuGet**-пакет (**HtmlAgilityPack**) и использовать его для анализа содержимого веб-страницы.

В начале программы подключаем библиотеку **HtmlAgilityPack** для парсинга HTML в .NET:

```
using HtmlAgilityPack;
```

Затем меняем метод **Main** на асинхронный:

```
static async Task Main()
```

- async позволяет использовать await для асинхронных операций (например, HTTP-запросов).
- Task — возвращаемый тип для асинхронных методов.

Теперь мы хотим получить URL от пользователя и сохранить его в переменную:

```
Console.Write("Введите URL сайта: ");  
string? url = Console.ReadLine();
```

Сделаем проверку ввода, что наша строка не null, не пустая и не состоит из пробелов:

```
if (!string.IsNullOrEmpty(url))  
{  
    ...  
}
```

Создадим внутри блок try/catch для обработки ошибок (проверка на неправильный URL, отсутствие интернета, отсутствие тега <title>):

```
if (!string.IsNullOrEmpty(url))  
{  
    try  
    {  
        ...  
    }  
    catch (Exception ex)  
    {  
        Console.WriteLine("Ошибка: " + ex.Message);  
    }  
}  
else  
{  
    Console.WriteLine("URL не может быть пустым.");  
}
```

Делаем HTTP-запрос через HttpClient:

```
try  
{  
    HttpClient client = new HttpClient();  
    string html = await client.GetStringAsync(url);  
}
```

- HttpClient — класс для отправки HTTP-запросов.
- GetStringAsync(url) — асинхронно скачивает HTML-код страницы.
- await — приостанавливает выполнение, пока запрос не завершится.

Осуществляем парсинг HTML с помощью HtmlAgilityPack:

```
HtmlDocument doc = new HtmlDocument();
doc.LoadHtml(html);
```

- HtmlDocument — класс для работы с HTML.
- LoadHtml(html) — загружает HTML-код в объект для парсинга.

Далее ищем заголовок страницы:

```
var titleNode = doc.DocumentNode.SelectSingleNode("//title");
```

- SelectSingleNode("//title") — ищет первый тег <title> в HTML.
 - // — означает поиск по всему документу (XPath-синтаксис).

Выводим результат:

```
if (titleNode != null)
    Console.WriteLine("Заголовок страницы: " + titleNode.InnerText);
else
    Console.WriteLine("Заголовок не найден.");
```

- InnerText — возвращает текст внутри тега (без HTML-разметки).

Шаг 3. Null и ссылочные типы

В C# значение null представляет **отсутствие данных** или **пустую ссылку**. Оно используется по умолчанию для всех ссылочных типов (например, string, class). Однако начиная с C# 8.0 была введена система **Nullable Reference Types**, которая делает работу с **null** более безопасной.

Раньше любая переменная ссылочного типа могла быть null без предупреждений:

```
string name = null; // Опасность: возможен NullReferenceException!
Console.WriteLine(name.Length); // Ошибка в runtime!
```

Проблема: Компилятор не предупреждал о потенциальных NullReferenceException.

Теперь ссылочные типы по умолчанию **не допускают null**, а для разрешения нужно явно указать ?:

```
string nonNullableName = "Ария"; // Не может быть null
string? nullableName = null; // Может быть null

Console.WriteLine(nullableName?.Length); // Безопасный вызов через ?.
```


Что изменилось:

- Компилятор выдаёт предупреждения, если null присваивается non-nullable типу.
- Явное указание string? делает код понятнее.

Рассмотрим примеры использования null.

1. Методы, возвращающие null

```
// Возвращает string?, так как строка может отсутствовать
1 reference
string? GetUserName(int id)
{
    return id == 1 ? "Alice" : null;
}

var user = GetUserName(2);
Console.WriteLine(user?.ToUpper()); // Проверка на null через ?.
```

2. Парсинг ввода

```
string? input = Console.ReadLine(); // Возвращает string?
if (input != null)
    Console.WriteLine($"Вы ввели: {input}");
else
    Console.WriteLine("Вы ничего не ввели!");
```

3. Работа с коллекциями

```
List<string?> names = new() { "Anna", null, "Bob" };

foreach (var name in names)
{
    Console.WriteLine(name?.Length ?? 0); // Если null, выведет 0
}
```

Операторы для работы с null

1. Оператор ?. (null-условный)

```
string? text = null;
Console.WriteLine(text?.Length);
```

Не выбросит исключение, выведет null

2. Оператор ?? (null-объединения)

```
string? name = null;  
string result = name ?? "Default";
```

Если `name == null`, вернёт "Default"

3. Оператор ! (null-forgiving)

```
string? name = null;  
Console.WriteLine(name!.Length); // Отключаем предупреждение компилятора
```

Зачем нужно `null`?

1. Обозначение отсутствия значения:

- Например, если пользователь не ввёл данные (`Console.ReadLine()`).
- Если элемент не найден в базе данных.

2. Оптимизация памяти:

- `null` указывает, что объект не создан, экономя ресурсы.

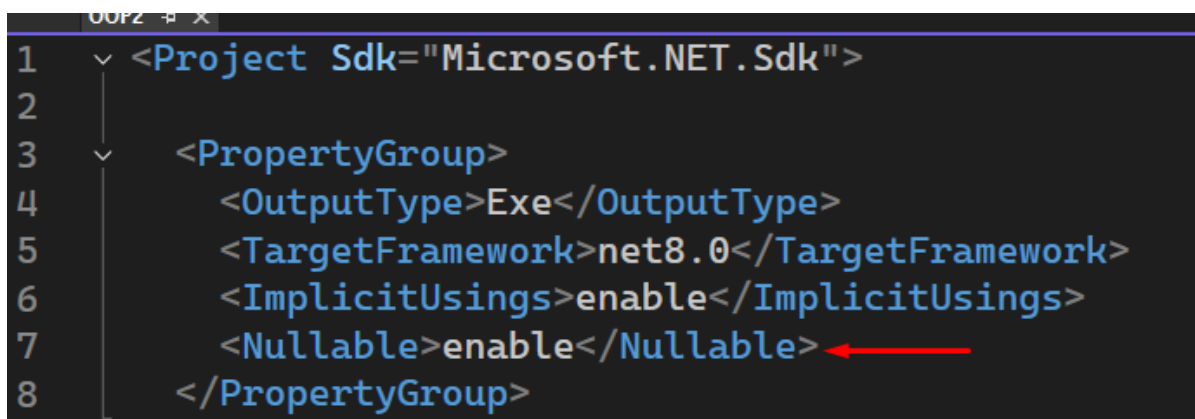
3. Работа с внешними API:

- Многие REST-сервисы возвращают `null` для отсутствующих полей.

4. Явное указание на "пустоту":

- Чёткое разделение между "значение есть" и "значения нет".

Nullable-контекст - это опция, которой мы можем управлять. Так, откроем файл проекта. Данный элемент `<Nullable>enable</Nullable>` со значением `enable` указывает, что этот nullable-контекст будет распространяться на весь проект:



```
1 <Project Sdk="Microsoft.NET.Sdk">  
2  
3 <PropertyGroup>  
4 <OutputType>Exe</OutputType>  
5 <TargetFramework>net8.0</TargetFramework>  
6 <ImplicitUsings>enable</ImplicitUsings>  
7 <Nullable>enable</Nullable>  
8 </PropertyGroup>
```

При желании мы можем отключить nullable-контекст в файле конфигурации проекта изменив значение опции `Nullable` на **"disable"**.

С помощью специальных директив, мы можем включить nullable-контекст на уровне отдельных участков кода **#nullable enable**, исключить какой-то определенный кусок кода **#nullable disable**:

```
#nullable disable
string name = null;
Console.WriteLine(name!.Length);
#nullable enable
string hero = null;
```

Шаг 4. Null и значимые типы

В отличие от ссылочных типов переменным/параметрам значимых типов нельзя напрямую присвоить значение **null**. Тем не менее нередко бывает удобно, чтобы переменная/параметр значимого типа могли принимать значение **null**. Например, получаем числовое значение из базы данных, которое в базе данных может отсутствовать. То есть, если значение в базе данных есть - получим число, если нет - то **null**.

Чтобы присвоения переменной или параметру значимого типа значения **null**, эти переменная/параметр значимого типа должны представлять тип **nullable**. Для этого после названия типа указывается знак вопроса ?

```
int? val = null;
Console.WriteLine(val);
```

Здесь переменная **val** представляет не просто тип **int**, а тип **int?** - тип, переменные/параметры которого могут принимать как значения типа **int**, так и значение **null**. В данном случае мы передаем ей значение **null**. Но также можно передать и значение типа **int**:

```
int? val = null;
IsNull(val); // null
val = 22;
IsNull(val); // 22
2 references
void IsNull(int? obj)
{
    if (obj == null) Console.WriteLine("null");
    else Console.WriteLine(obj);
}
```

Однако если переменная/параметр представляет значимый не nullable-тип, то присвоить им значение **null** не получится:

```
int val = null;
```

Стоит отметить, что фактически запись `?` для значимых типов является упрощенной формой использования структуры `System.Nullable<T>`. Параметр `T` в угловых скобках представляет универсальный параметр, вместо которого в программе подставляется конкретный тип данных. Следующие виды определения переменных будут эквивалентны:

```
int? number1 = 5;
Nullable<int> number2 = 5;
```

Свойства `Value` и `HasValue`. Метод `GetValueOrDefault`

Структура `Nullable<T>` имеет два свойства:

- **Value** - значение объекта
- **HasValue**: возвращает `true`, если объект хранит некоторое значение, и `false`, если объект равен `null`.

Мы можем использовать эти свойства для проверки наличия и получения значения:

```
PrintNullable(5); // 5
PrintNullable(null); // параметр равен null
2 references
void PrintNullable(int? number)
{
    if (number.HasValue)
    {
        Console.WriteLine(number.Value);
        // аналогично
        Console.WriteLine(number);
    }
    else
    {
        Console.WriteLine("параметр равен null");
    }
}
```

Однако если мы попробуем получить через свойство `Value` значение переменной, которая равна `null`, то мы столкнемся с ошибкой:

```
int? number = null;
Console.WriteLine(number.Value); // ! Ошибка
Console.WriteLine(number); // Ошибки нет – просто ничего не выведет
```

Также структура `Nullable<T>` имеет метод `GetValueOrDefault()`. Он возвращает значение переменной/параметра, если они не равны `null`. Если они равны `null`, то возвращается значение по умолчанию. Значение по умолчанию можно передать в метод.

Если в метод не передается данных, то возвращается значение по умолчанию для данного типа данных (например, для числовых данных это число 0).

```
int? number = null; // если значения нет, метод возвращает
    значение по умолчанию
Console.WriteLine(number.GetValueOrDefault()); // 0 - значение
    по умолчанию для числовых типов
Console.WriteLine(number.GetValueOrDefault(10)); // 10

number = 15; // если значение задано, оно возвращается методом
Console.WriteLine(number.GetValueOrDefault()); // 15
Console.WriteLine(number.GetValueOrDefault(10)); // 15
```

Шаг 5. Делегаты (Delegates)

Делегаты и события в C# — это механизмы для создания гибкого и связанного кода, особенно полезные в играх (например, когда игрок берет предмет, и нужно уведомить другие системы: интерфейс, достижения, звук).

Делегат — это **"указатель на функцию"**. Он позволяет передавать методы как аргументы или хранить их в переменных.

Зачем?

- Когда нужно вызвать разные методы в одном месте (например, обработка клика кнопки).
- Для создания кастомных событий.

В главном классе над методом Main объявите делегат:

```
internal class Program
{
    public delegate void MessageHandler(string text);

    0 references
    static void Main(string[] args)
    {
    }
}
```

Это — **тип делегата**. Он определяет сигнатуру методов, которые могут быть ему присвоены. В данном случае: любой метод, который принимает один string и ничего не

возвращает (void), может быть обработан этим делегатом. Более простыми словами делегат — как переменная, в которую мы можем "положить" метод и потом вызвать его.

Затем создадим метод, который подходит под наш делегат:

```
static void Main(string[] args)
{
    // Метод, соответствующий делегату
    0 references
    static void ShowMessage(string message)
    {
        Console.WriteLine($"Сообщение: {message}");
    }
}
```

Затем в методе Main создадим переменную **handler** типа **MessageHandler** и присваиваем ей метод **ShowMessage**:

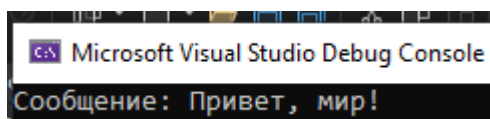
```
0 references
static void Main(string[] args)
{
    // Создаем экземпляр делегата и привязываем метод
    MessageHandler handler = ShowMessage;
}
```

Важно: мы не вызываем метод (ShowMessage()), а просто передаём ссылку на него.

И теперь вызовем делегат:

```
static void Main(string[] args)
{
    MessageHandler handler = ShowMessage;
    // Вызываем делегат (выполнится ShowMessage)
    handler("Привет, мир!");
}
```

Запустите программу и увидите на консоли вывод сообщения:



Рассмотрим ещё один пример. Удалите предыдущее создание и вызов делегата. Мы создаём тип делегата **NumberHandler**, который представляет метод с одним параметром типа `int` и возвращает `void`:

```
namespace OOP
{
    public delegate void NumberHandler(int number);
    internal class Program
    {
        static void Main(string[] args)
        {
        }
    }
}
```

Далее создаём два метода. Метод **Double** - это метод, который выводит удвоенное значение числа. Он подходит по сигнатуре к нашему делегату и метод **Square** - этот метод выводит квадрат числа и тоже подходит под сигнатуру `NumberHandler`:

```
internal class Program
{
    static void Double(int num) => Console.WriteLine($"Удвоено: {num * 2}");
    static void Square(int num) => Console.WriteLine($"Квадрат: {num * num}");
    static void Main(string[] args)
    {
    }
}
```

Далее создаём делегат в методе **Main** и связываем его с методом **Double**:

```
static void Main(string[] args)
{
    NumberHandler handler = Double; // создаём делегат
}
```

Важно: handler теперь “ссылается” на метод `Double`, но пока вызывает только его.

Далее добавляем метод в делегат:

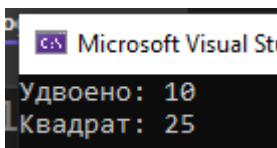
```
static void Main(string[] args)
{
    NumberHandler handler = Double; // создаём делегат
    handler += Square; // Добавляем второй метод
}
```

Мы добавляем ещё один метод к делегату. Теперь handler будет вызывать *оба метода* — сначала Double, затем Square. Такой список методов называется **списком вызовов** (*invocation list*).

И затем вызываем делегат:

```
static void Main(string[] args)
{
    NumberHandler handler = Double; // создаём делегат
    handler += Square; // Добавляем второй метод
    handler(5); // Вызов обоих методов
}
```

При запуске программы, при вызове handler(5), он последовательно выполняет оба метода:



Шаг 6. События (Events)

Событие — это "настройка" над делегатом, которая добавляет безопасность (защита от случайного удаления подписчиков).

Зачем?

- Уведомлять другие части программы о действиях (например, "игрок умер").
- Снижать связанность кода (объекты не знают друг о друге, только о событии).

Добавьте в свой проект новый класс **Player** (Add – Class). Он будет представлять игрока со здоровьем и событием смерти.

1. Объявляем событие:

```
internal class Player
{
    public event Action OnDeath;
```

- **event** — ключевое слово для создания события.
- **Action** — это встроенный делегат, который не принимает параметров и не возвращает значение (**void**). (Можно было бы использовать и кастомный делегат, но **Action** проще.)
- **OnDeath** — название события (по соглашению начинается с **On**).

Смысл: Когда игрок умирает, он "вызывает" это событие, и все, кто на него подписан, получают уведомление.

2. Добавляем поле здоровья

```
internal class Player
{
    public event Action OnDeath;
    private int health = 100;
}
```

- **private** — здоровье скрыто от внешнего кода (инкапсуляция).
- Изначально у игрока 100 HP.

3. Пишем метод смерти

```
private void Die()
{
    Console.WriteLine("Игрок погиб!");
    OnDeath?.Invoke();
}
```

- **private** — метод нельзя вызвать снаружи класса.
- **OnDeath?.Invoke()** — безопасный вызов события:
 - **?.** — проверяет, есть ли подписчики (если нет, ошибки не будет).
 - **Invoke()** — запускает все подписанные методы.

4. Пишем метод для получения урона

```
public void TakeDamage(int damage)
{
    health -= damage;
    if (health <= 0)
    {
        Die();
    }
}
```

- **public** — метод можно вызывать извне (например, из кода врага).
- Если здоровье падает до 0 или ниже, вызывается метод **Die()**.

Итоговый код:

```

1 namespace OOP;
2 internal class Player
3 {
4     public event Action OnDeath;
5     private int health = 100;
6
7     public void TakeDamage(int damage)
8     {
9         health -= damage;
10        if (health <= 0)
11        {
12            Die();
13        }
14    }
15
16    private void Die()
17    {
18        Console.WriteLine("Игрок погиб!");
19        OnDeath?.Invoke();
20    }
21 }

```

Теперь переходим в класс **Program**. Создадим событие и подпишемся на него.

5. Создадим метод **ShowGameOver()**. Его задача срабатывать при смерти игрока:

```

1 namespace OOP;
2 internal class Program
3 {
4     static void Main(string[] args)
5     {
6
7     }
8     static void ShowGameOver() => Console.WriteLine
9         ("GAME OVER");
10 }

```

6. Далее подпишемся на событие:

```

static void Main(string[] args)
{
    Player player = new Player();
    //Подписываемся на событие
    player.OnDeath += () => Console.WriteLine
        ("Враги празднуют победу!");
    player.OnDeath += ShowGameOver;
}
1 reference
static void ShowGameOver() => Console.WriteLine
    ("GAME OVER");

```

- += — добавляем подписчика (можно добавить сколько угодно методов).
- Лямбда () => ... — анонимный метод (выведет текст при смерти).
- ShowGameOver — второй метод.

7. Наносим игроку урон:

```

static void Main(string[] args)
{
    Player player = new Player();
    //Подписываемся на событие
    player.OnDeath += () => Console.WriteLine
        ("Враги празднуют победу!");
    player.OnDeath += ShowGameOver;
    player.TakeDamage(100); // Убиваем игрока
}

```

- После этого вызова:
 - health станет 0.
 - Вызовется Die().
 - Сработает событие OnDeath, и все подписчики получат уведомление.

Запустим программу и увидим вывод:

```

Microsoft Visual Studio Debug Console:
Игрок погиб!
Враги празднуют победу!
GAME OVER

```

Шаг 7. Мини-проект «Умный дом»

Давайте сделаем мини-проект на C# с использованием делегатов и событий — симулятор умного дома, где датчики (температуры, движения) уведомляют подписчиков о событиях.

Цель:

- Датчик температуры сообщает о перегреве.
- Датчик движения обнаруживает активность.
- Все события выводятся в консоль + имитация отправки уведомлений.

Добавьте в свой проект новый класс **SmartHome (Add – Class)**.

Шаг 1: Объявление делегатов

```
1 // Делегаты для событий
2 public delegate void TemperatureEventHandler(string message);
3 public delegate void MotionEventHandler(string message);
```

Здесь создаются два типа делегатов — они описывают сигнатуру методов для обработки событий.

- **TemperatureEventHandler** — будет использоваться, когда перегревается температура.
- **MotionEventHandler** — будет использоваться при обнаружении движения.

Они оба принимают строку (сообщение) и ничего не возвращают.

Шаг 2. Создаём класс TemperatureSensor

```
5 // Класс датчика температуры
6 public class TemperatureSensor
7 {
8 }
9 }
```

В нём объявляем событие **TemperatureEventHandler**:

```
6 public class TemperatureSensor
7 {
8     // Событие (основано на делегате)
9     public event TemperatureEventHandler OnOverheat;
```

Это **событие**. Оно основано на делегате и сообщает: “температура слишком высокая”. Событие может быть подписано другими методами.

Затем внутри класса **TemperatureSensor** создаём метод **CheckTemperature**:

```
6  public class TemperatureSensor
7  {
8      public event TemperatureEventHandler OnOverheat;
9
10     public void CheckTemperature(int currentTemp)
11     {
12         if (currentTemp > 30)
13         {
14             // Вызываем событие, если есть подписчики
15             OnOverheat?.Invoke($"!! Температура критическая: {currentTemp}°C!");
16         }
17     }
18 }
```

Метод **CheckTemperature** проверяет температуру, и если выше 30°C, вызывает событие. **?.Invoke(...)** гарантирует, что метод будет вызван только если есть подписчики (иначе произойдет **NullReferenceException**).

Шаг 3. Создаём класс **MotionSensor**

```
20 // Класс датчика движения
21 public class MotionSensor
22 {
23 }
24 }
```

В нём объявляем событие **MotionEventHandler**, которое будет вызываться при обнаружении движения:

```
21 public class MotionSensor
22 {
23     public event MotionEventHandler OnMotionDetected;
```

И создаём метод **DetectMotion**:

```
21 public class MotionSensor
22 {
23     public event MotionEventHandler OnMotionDetected;
24
25     public void DetectMotion(bool isMotion)
26     {
27         if (isMotion)
28         {
29             OnMotionDetected?.Invoke("!! Обнаружено движение в коридоре!");
30         }
31     }
32 }
```

При **true** — вызываем событие с сообщением.

Шаг 3. Создаём класс Класс Notifier

```
34 // Класс для уведомлений
35 public class Notifier
36 {
37
38 }
```

В нём создаём два метода. Метод, подходящий под `TemperatureEventHandler` — он выводит сообщение об опасной температуре. Может представлять отправку email, пуша, смс и т.д. И метод для логирования движения. Подходит под `MotionEventHandler`. Пишет сообщение с меткой времени:

```
35 public class Notifier
36 {
37     public static void SendTemperatureAlert(string message)
38     {
39         Console.WriteLine($"[Уведомление] {message}");
40         // Здесь могла бы быть отправка email/SMS...
41     }
42
43     public static void LogMotionEvent(string message)
44     {
45         Console.WriteLine($"[Лог] {message} (время: {DateTime.Now})");
46     }
47 }
```

Теперь возвращаемся в класс **Program**. В методе **Main** создадим датчики:

```
static void Main(string[] args)
{
    // Создаем датчики
    var tempSensor = new TemperatureSensor();
    var motionSensor = new MotionSensor();
}
```

Далее подписываем датчики на события:

```
// Подписываем методы на события
tempSensor.OnOverheat += Notifier.SendTemperatureAlert;
motionSensor.OnMotionDetected += Notifier.LogMotionEvent;
```

И создать симуляцию работы датчиков:

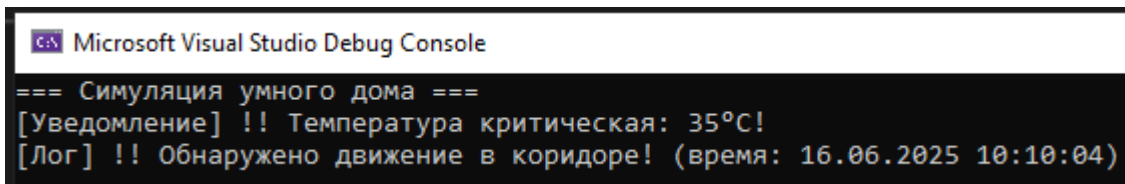
```
// Симуляция работы датчиков
Console.WriteLine("=== Симуляция умного дома ===");
tempSensor.CheckTemperature(15); // Норма
tempSensor.CheckTemperature(35); // Перегрев -> вызовет OnOverheat
motionSensor.DetectMotion(false); // Ничего
motionSensor.DetectMotion(true); // Движение -> вызовет OnMotionDetected
```

Таким образом событие — это механизм подписки/уведомления. Один вызывает, другие реагируют.

Делегат определяет тип метода, который можно подписать на событие.

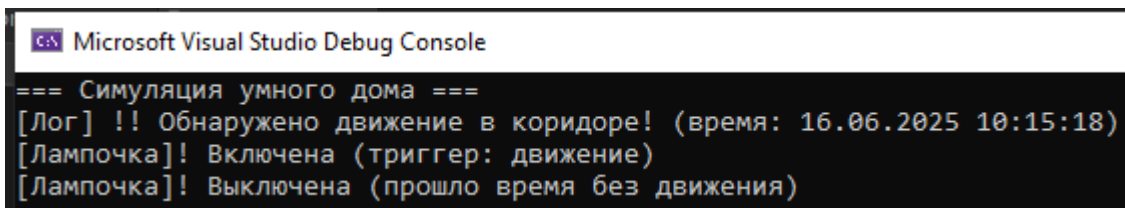
Мы можем объединить устройства и логику поведения — всё это демонстрирует принцип слабой связи (loose coupling) между компонентами.

Если мы запустим программу, то нам выведет:



```
Microsoft Visual Studio Debug Console
=== Симуляция умного дома ===
[Уведомление] !! Температура критическая: 35°C!
[Лог] !! Обнаружено движение в коридоре! (время: 16.06.2025 10:10:04)
```

Теперь **ваша задача** попробовать самостоятельно дополнить вашу программу, включив в неё лампочку, которая будет включаться при движении, и выключаться при отсутствии:



```
Microsoft Visual Studio Debug Console
=== Симуляция умного дома ===
[Лог] !! Обнаружено движение в коридоре! (время: 16.06.2025 10:15:18)
[Лампочка]! Включена (триггер: движение)
[Лампочка]! Выключена (прошло время без движения)
```

Для создания задержки в консоли, используйте класс **Thread**. Например, для задержки в 3000 миллисекунд, можно использовать следующую инструкцию:

```
Thread.Sleep(3000);
```

Ниже приводится решение задачи. Создаём в классе **SmartHome** новый класс **SmartLight**:

```

public class SmartLight
{
    private bool isOn = false;

    2 references
    public void TurnOn(string message)
    {
        if (!isOn)
        {
            isOn = true;
            Console.WriteLine("[Лампочка]! Включена (триггер: движение)");
        }
    }

    public void TurnOff()
    {
        if (isOn)
        {
            isOn = false;
            Console.WriteLine("[Лампочка]! Выключена (прошло время без движения)");
        }
    }
}

```

И вызовем в методе **Main**:

```

static void Main(string[] args)
{
    // Создаем датчики
    var tempSensor = new TemperatureSensor();
    var motionSensor = new MotionSensor();
    var smartLight = new SmartLight();

    // Подписываем методы на события
    tempSensor.OnOverheat += Notifier.SendTemperatureAlert;
    motionSensor.OnMotionDetected += Notifier.LogMotionEvent;
    motionSensor.OnMotionDetected += smartLight.TurnOn;

    // Симуляция работы датчиков
    Console.WriteLine("=== Симуляция умного дома ===");

    motionSensor.DetectMotion(true);
    smartLight.TurnOn("Обнаружение движение");
    Thread.Sleep(3000);
    smartLight.TurnOff();
}

```

- Класс SmartLight подключён к событию движения.
- Один сигнал = несколько реакций: и лог, и лампочка включается.
- Архитектура остаётся модульной: лампу можно заменить на сирену или камеру.

Самостоятельные задания

Задание 1 — Создание собственной библиотеки классов

1. Создайте новый проект типа **Class Library (.NET)** с названием MathLibrary.
2. Внутри создайте класс MathTools с методами:
 - `int Add(int a, int b)`
 - `int Multiply(int a, int b)`
3. Соберите библиотеку и подключите её к консольному приложению.
4. Используйте методы из MathTools в Main.

Задание 2 — Установка и использование NuGet-пакета

1. Создайте новое консольное приложение.
2. Установите NuGet-пакет **HtmlAgilityPack**.
3. Напишите код, который скачивает HTML-страницу (например, <https://example.com>) и выводит заголовок (`<title>`) страницы.

Подсказка: Используйте `HtmlWeb` и `HtmlDocument`.

Задание 3 — Делегаты

1. Создайте делегат `public delegate void Logger(string message);`
2. Напишите метод `LogToConsole(string message)` — просто выводит сообщение.
3. В Main, создайте переменную делегата и вызовите метод через него.

Задание 4 — Событие в классе

1. Создайте класс `Thermometer`.
2. Внутри создайте событие `TemperatureTooHigh`, которое вызывается, если температура выше 100.
3. Создайте метод `Measure(int value)`.
4. Подпишитесь на событие и выведите предупреждение в консоль, если температура превышена.

Задание 5 — Работа с null

1. Создайте метод `PrintLength(string? input)`, который:
 - Проверяет, не равна ли строка `null`.

- Если не null, выводит её длину.
- Если null, сообщает, что строка отсутствует.

Задание 6 — Мини-проект «Умный дом»

1. Создайте класс SmartLight, с полями IsOn, Brightness.
2. Реализуйте событие OnStateChanged, которое срабатывает при включении/выключении.
3. Добавьте метод Toggle() — меняет состояние лампы.
4. Подпишитесь на событие в Main, чтобы выводить сообщения: "Свет включён" / "Свет выключен".
5. Подключите библиотеку Newtonsoft.Json и сделайте сериализацию/десериализацию состояния лампы.