

Лабораторная работа №8. Продвинутое ООП: свойства и пространства имён

Цель: Углубить знания в ООП, изучить свойства (get, set), работу со структурами, а также правильно использовать пространства имён.

Задачи:

- Изучить и реализовать свойства классов.
- Разобрать отличия между структурами и классами (значимые и ссылочные типы).
- Ознакомиться с пространствами имён и их организацией в проекте.
- Использовать глобальные и встроенные пространства имён (System, global::).
- Понять порядок подключения пространств имён по умолчанию в проекте.

Шаг 1. Свойства

Кроме обычных методов в языке C# предусмотрены специальные методы доступа, которые называют свойствами. Они обеспечивают простой доступ к полям классов и структур, узнать их значение или выполнить их установку.

Стандартное описание свойства имеет следующий синтаксис:

```
[модификаторы] тип_свойства название_свойства
{
    get { действия, выполняемые при получении значения свойства }
    set { действия, выполняемые при установке значения свойства }
}
```

Вначале определения свойства могут идти различные модификаторы, в частности, модификаторы доступа. Затем указывается тип свойства, после которого идет название свойства. Полное определение свойства содержит два блока: **get** и **set**.

В блоке **get** выполняются действия по получению значения свойства. В этом блоке с помощью оператора **return** возвращаем некоторое значение.

В блоке **set** устанавливается значение свойства. В этом блоке с помощью параметра **value** мы можем получить значение, которое передано свойству.

Блоки **get** и **set** еще называются **акссесорами** или методами доступа (к значению свойства), а также **геттером** и **сеттером**.

Рассмотрим на примере, создадим пустой класс **Person**:

```

class Person
{
    private string name = "Undefined";

    0 references
    public string Name
    {
        get
        {
            return name; // возвращаем значение свойства
        }
        set
        {
            name = value; // устанавливаем новое значение свойства
        }
    }
}

```

Здесь в классе **Person** определено приватное поле **name**, которая хранит имя пользователя, и есть общедоступное свойство **Name**. Хотя они имеют практически одинаковое название за исключением регистра, но это не более чем стиль, названия у них могут быть произвольные и не обязательно должны совпадать.

Через это свойство мы можем управлять доступом к переменной **name**. В свойстве в блоке **get** возвращаем значение поля:

Теперь в методе **Main**, создадим объект:

```

0 references
static void Main(string[] args)
{
    // Создадим объект
    Person person = new Person();
}

```

В программе мы можем обращаться к этому свойству, как к обычному полю. Если мы ему присваиваем какое-нибудь значение, то срабатывает блок **set**, а передаваемое значение передается в параметр **value**:

```

// Устанавливаем свойство – срабатывает блок Set
// значение "Ivan" и есть передаваемое в свойство value
person.Name = "Ivan";

```

Если мы получаем значение свойства, то срабатывает блок **get**, который по сути возвращает значение переменной **name**:

```
// Получаем значение свойства и присваиваем его переменной – срабатывает блок Get
string personName = person.Name;
Console.WriteLine(personName); // Ivan
```

То есть по сути свойство **Name** ничего не хранит, оно выступает в роли посредника между внешним кодом и переменной **name**.

```
get { return name; }
```

А в блоке **set** устанавливаем значение переменной **name**. Параметр **value** представляет передаваемое значение, которое передается переменной **name**.

```
set { name = value; }
```

Теперь добавим в класс новое поле:

```
class Person
{
    private string name = "Undefined";
    private int age = 1;
```

И установим в нём же для него свойство:

```
0 references
public int Age
{
    set
    {
        if (value < 1 || value > 120)
            Console.WriteLine("Возраст должен быть в диапазоне от 1 до 120");
        else
            age = value;
    }
    get { return age; }
}
```

В данном случае переменная **age** хранит возраст пользователя. Напрямую мы не можем обратиться к этой переменной - только через свойство **Age**. Причем в блоке **set** мы устанавливаем значение, если оно соответствует некоторому разумному диапазону. Поэтому при передаче свойству **Age** значения, которое не входит в этот диапазон, значение переменной не будет изменяться:

```
static void Main(string[] args)
{
    Person person = new Person(); // создаём объект

    Console.WriteLine(person.Age); // 1
    person.Age = 37; // изменяем значение свойства
    Console.WriteLine(person.Age); // 37
}
```

Если же мы попробуем установить возраст не в пределах, и выведем его, то увидим, что он не изменился:

```
person.Age = -23; // пробуем передать недопустимое значение,
// возраст должен быть в диапазоне от 1 до 120
Console.WriteLine(person.Age); // 37 - возраст не изменился
```

Блоки **set** и **get** не обязательно одновременно должны присутствовать в свойстве. Если свойство определяет только блок **get**, то такое свойство доступно только для чтения - мы можем получить его значение, но не установить.

И, наоборот, если свойство имеет только блок **set**, тогда это свойство доступно только для записи - можно только установить значение, но нельзя получить. Обновим класс:

```
class Person
{
    string name = "Tom";
    int age = 1;
    // свойство только для записи
    public int Age { set { age = value; } }
    // свойство только для чтения
    public string Name { get { return name; } }

    public void Print() => Console.WriteLine($"Name: {name} Age: {age}");
}
```

Мы можем с вами получить значение свойства, но установить новое значение не сможем, получим ошибку. Также мы можем установить свойство для записи, но получить не сможем:

```
static void Main(string[] args)
{
    Person person = new Person();
    Console.WriteLine(person.Name);
    // person.Name = "Bob";    // ! Ошибка
    person.Age = 37;
    // Console.WriteLine(person.Age); // ! Ошибка
    person.Print();
}
```

Рассмотрим на двух примерах.

Мини-проект: Корзина интернет-магазина

Цель:

- Создать товары с ценой и количеством
- Геттер будет возвращать *стоимость всей позиции* (цена × количество)
- Сеттер будет **ограничивать количество**, не допуская отрицательных значений

Создайте класс **Product**. В нём создадим:

- **Публичные свойства** (Name, Price)
- **Приватное поле** (_quantity)
- **Свойство с логикой** (Quantity)
- **Вычисляемое свойство** (TotalPrice)

```
public class Product
{
    0 references
    public string Name { get; set; }
    1 reference
    public decimal Price { get; set; }
}
```

- Это **автоматические свойства** с геттером и сеттером.
- Можно читать (**product.Name**) и изменять (**product.Price = 100**) из любого места программы.

Далее объявим приватное поле:

```
private int _quantity;
```

- Поле объявлено как **private**, поэтому доступно **только внутри класса**.
- Это пример **инкапсуляции** — мы прячем внутреннюю реализацию.

Далее установим свойство **Quantity** с проверкой:

```
// Сеттер с проверкой
1 reference
public int Quantity
{
    get => _quantity;
    set
    {
        if (value < 0)
            _quantity = 0;
        else
            _quantity = value;
    }
}
```

- **Сеттер (set)** проверяет, что значение не отрицательное. Если передали -5, свойство сохранит 0.
- **Геттер (get)** просто возвращает текущее значение `_quantity`.

Создадим вычисляемое свойство **TotalPrice**:

```
// Геттер только для чтения
0 references
public decimal TotalPrice
{
    get { return Price * Quantity; }
}
```

Создадим два объекта – яблоко и банан (для него специально установим отрицательное свойство):

```
static void Main(string[] args)
{
    var apple = new Product { Name = "Яблоко", Price = 10.5m, Quantity = 5 };
    var banana = new Product { Name = "Банан", Price = 7.2m, Quantity = -2 };
}
```

Выводим в консоль наши свойства:

```
Console.WriteLine($"{apple.Name} x {apple.Quantity} = {apple.TotalPrice} Py6.");
Console.WriteLine($"{banana.Name} x {banana.Quantity} = {banana.TotalPrice} Py6.");
```

Выведите на консоль, и объясните почему у бананов стало значение равное 0.

Теперь сделаем второй мини-проект – создадим игрового персонажа.

Создаём класс **Hero**. Класс **Hero** представляет героя в игре и содержит:

- **Характеристики** (имя, здоровье, мана, уровень)
- **Методы для взаимодействия** (получение урона, лечение)
- **Логику контроля состояний** (проверка на смерть, ограничение здоровья)

Создадим публичные свойства, которые можно свободно читать и изменять вне класса:

```
public class Hero
{
    4 references
    public string Name { get; set; }
    2 references
    public int Mana { get; set; }
    2 references
    public int Level { get; set; }
}
```

Затем создадим приватные поля, где скроем прямой доступ к ним (инкапсуляция). Их значение реализуем через свойство **Health**:

```
private int _health;
private int _maxHealth;
```

Далее создадим конструктор:

```
public Hero(string name, int maxHealth)
{
    Name = name;
    _maxHealth = maxHealth;
    _health = maxHealth; // Герой начинает с полным здоровьем
    Mana = 100; // Стартовое значение маны
    Level = 1; // Начальный уровень
}
```

Далее установим свойство с валидацией:

```
public int Health
{
    get => _health;
    set
    {
        if (value < 0)
            _health = 0; // Здоровье не может быть отрицательным
        else if (value > _maxHealth)
            _health = _maxHealth; // Нельзя превысить максимум
        else
            _health = value;
    }
}
```

Теперь вычислим статус через тернарную операцию:

```
public string Status => _health > 0 ? "Жив" : "Погиб";
```

- Возвращает "Жив" или "Погиб" в зависимости от здоровья.

- Только для чтения (геттер).

Осталось создать два метода. Первый для получения урона:

```
public void TakeDamage(int damage)
{
    Health -= damage; // Используем свойство, а не поле напрямую
    Console.WriteLine($"{Name} получает {damage} урона. HP: {Health} | Статус: {Status}");
}
```

1. Уменьшает Health на damage.
2. Автоматически проверяет, не умер ли герой (через свойство Health).
3. Выводит результат в консоль.

Второй метод лечения:

```
public void Heal(int amount)
{
    Health += amount; // Нельзя превысить _maxHealth
    Console.WriteLine($"{Name} исцелен на {amount}. HP: {Health} | Статус: {Status}");
}
```

Далее в методе Main создадим объект нашего класса и установим ему здоровье.

Далее вызовем методы нанесения урона и лечения:

```
static void Main(string[] args)
{
    Hero hero = new("Ривен", 120);
    Console.WriteLine($"{hero.Name} - Уровень: {hero.Level}, Мана: {hero.Mana}, Статус: {hero.Status}");

    hero.TakeDamage(50);
    hero.Heal(30);
    hero.TakeDamage(200);
    hero.Heal(10); // не поможет - герой уже "мертв"
}
```

Шаг 2. Структуры

Наряду с классами **структуры** представляют еще один способ создания собственных типов данных в C#. Более того многие примитивные типы, например, int, double и т.д., по сути являются структурами.

Для определения структуры применяется ключевое слово **struct**:


```
struct имя_структуры
{
    // элементы структуры
}
```

Создайте новый консольный проект. В нём давайте создадим структуру:

```
struct Person
{
    ...
}
```

Начиная с версии C# 12, если структура имеет пустое определение (не содержит полей, свойств, методов), то фигурные скобки после названия типа можно не использовать:

```
struct Person;
```

Как и классы, структуры могут хранить состояние в виде полей (переменных) и определять поведение в виде методов. Например, добавим в структуру **Person** пару полей и метод:

```
struct Person
{
    public string name;
    public int age;

    0 references
    public void Print()
    {
        Console.WriteLine($"Имя: {name}  Возраст: {age}");
    }
}
```

Для использования структуры ее необходимо инициализировать. Для инициализации создания объектов структуры, как и в случае с классами, применяется вызов конструктора с оператором **new**. Даже если в коде структуры не определено ни одного конструктора, тем не менее имеет как минимум один конструктор - конструктор по умолчанию, который генерируется компилятором. Этот конструктор не принимает параметров и создает объект структуры со значениями по умолчанию.

```
new название_структуры();
```

Как и в случае с классами, для обращения к функциональности структуры - полям, методам и другим компонентам структуры применяется точечная нотация - после объекта структуры ставится точка, а затем указывается компонент структуры:

```
объект.поле_структуры  
объект.метод_структуры(параметры_метода)
```

Давайте создадим объект нашей структуры:

```
static void Main(string[] args)  
{  
    Person garen = new Person(); // вызов конструктора  
    garen.name = "Гарен"; // изменяем значение по умолчанию в поле name  
    garen.Print(); // Имя: Гарен Возраст: 0  
}
```

Если все поля структуры доступны (как в случае с полями структуры **Person**, который имеет модификатор **public**), то структуру можно инициализировать без вызова конструктора. В этом случае необходимо присвоить значения всем полям структуры перед получением значений полей и обращением к методам структуры. Например:

```
static void Main(string[] args)  
{  
    Person garen; // не вызов конструктора  
    garen.name = "Гарен";  
    garen.age = 25;  
    garen.Print();  
}
```

Начиная с версии C# 10, мы можем напрямую инициализировать поля структуры при их определении (до C# 10 это делать было нельзя):

```
struct Person  
{  
    // инициализация полей значениями по умолчанию  
    public string name = "Том";  
    public int age = 1;  
    0 references  
    public Person() { }  
    1 reference  
    public void Print() => Console.WriteLine($"Имя: {name} Возраст: {age}");  
}
```

И создадим объект:

```
static void Main(string[] args)
{
    Person timo = new Person();
    timo.Print();
}
```

Как и класс, структура может определять конструкторы. Например, добавим в структуру **Person** конструктор:

```
struct Person
{
    public string name; public int age;
    public Person(string name = "Tom", int age = 1)
    {
        this.name = name; this.age = age;
    }
    public void Print() => Console.WriteLine($"Имя: {name}  Возраст: {age}");
}
```

Создадим объекты:

```
static void Main(string[] args)
{
    Person ahri = new();
    Person brand = new("Brand");
    Person darius = new("Darius", 25);
    ahri.Print(); // !!!! Имя:  Возраст: 0
    brand.Print(); // Имя: Brand  Возраст: 1
    darius.Print(); // Имя: Darius  Возраст: 25
}
```

Начиная с версии C# 10 мы можем определить свой конструктор без параметров:

```
struct Person
{
    public string name;
    public int age;
    public Person()
    {
        name = "Undefined";
        age = 18;
    }
    public void Print() => Console.WriteLine($"Имя: {name}  Возраст: {age}");
}
```

Что использовать структуры или классы?

Структуры	Классы
<p>Легковесные объекты:</p> <p>Структуры — это значимые типы данных, которые хранятся непосредственно в стеке, а не в куче (heap). Они обычно занимают меньше памяти и имеют более простую семантику. Применяются для хранения небольших объемов данных.</p>	<p>Сложные объекты:</p> <p>Классы представляют более сложные объекты с состоянием и поведением. Они хранятся в куче и передаются по ссылке.</p>
<p>Передача по значению:</p> <p>Структуры передаются по значению (копируются), что может быть полезно в некоторых сценариях. Например, координаты точки (x, y) могут быть представлены структурой.</p>	<p>Наследование и полиморфизм:</p> <p>Классы поддерживают наследование, что позволяет создавать иерархии классов. Они также поддерживают полиморфизм, что упрощает обработку разных типов объектов через общий интерфейс.</p>
<p>Неизменяемость (Immutable):</p> <p>Структуры могут быть объявлены как неизменяемые (readonly). Это полезно для представления неизменяемых данных, таких как дата или время.</p>	<p>Мутабельность (Mutable):</p> <p>Классы могут быть изменяемыми, и вы можете изменять их состояние. Они часто используются для представления объектов с динамическим поведением.</p>

Как выбрать между структурами и классами:

Используйте структуры, если вам нужны легковесные, неизменяемые объекты или если данные маленькие и простые.

Используйте классы, если вам нужно состояние, поведение, наследование или если объекты более сложные.

Важно помнить, что правильный выбор зависит от конкретной задачи и контекста. В большинстве случаев классы являются более универсальным инструментом, но структуры могут быть полезны в определенных сценариях.

Давайте разберём один наглядный пример.

Мини-проект: Система координатных точек

Мы разрабатываем простую 2D-игру или редактор изображений, где на экране может быть **десятки тысяч точек**. Каждая точка имеет координаты и используется кратковременно — например, при расчёте траекторий, коллизий, визуализации и т.д.

Создадим структуру **Point2D**:

```
public struct Point2D
{
    // ...
}
```

Объявим публичные поля, которые будут представлять собой координаты точки:

```
public int X;
public int Y;
```

Далее создадим конструктор, для инициализации наших точек:

```
public Point2D(int x, int y)
{
    X = x;
    Y = y;
}
```

И создадим метод **DistanceTo()**, который будет вычислять расстояние между текущей точкой (**this**) и другой точкой (**other**):

```
public double DistanceTo(Point2D other)
{
    int dx = X - other.X;
    int dy = Y - other.Y;
    return Math.Sqrt(dx * dx + dy * dy);
}
```

Формула: $\sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$ (теорема Пифагора).

Затем в методе Main создадим две точки с координатами и вызовем у них метод **DistanceTo()**:

```
static void Main(string[] args)
{
    var p1 = new Point2D(10, 20);
    var p2 = new Point2D(13, 24);
    Console.WriteLine($"Расстояние: {p1.DistanceTo(p2):F2}");
}
```

Почему именно struct в этом случае?

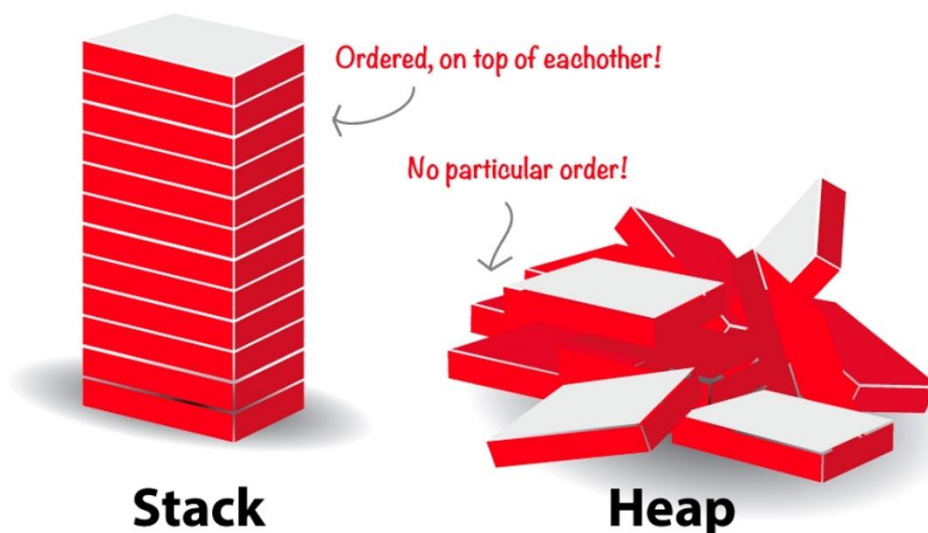
- Структуры хранятся в стеке, а не в куче — то есть работают быстрее, особенно при частом создании/удалении.
- Нет лишней нагрузки на сборщик мусора (GC).
- Идеально подходит для простых объектов с фиксированной логикой и коротким сроком жизни.
- Копируются по значению, а не по ссылке — например, передача в метод не может "изменить оригинал".

Шаг 3. Типы значений и ссылочные типы

Ранее мы рассматривали следующие элементарные типы данных: `int`, `byte`, `double`, `string`, `object` и др. Также есть сложные типы: структуры, перечисления, классы. Все эти типы данных можно разделить на типы значений, еще называемые значимыми типами, (**value types**) и ссылочные типы (**reference types**). Важно понимать между ними различия.



В .NET. память делится на два типа: **стек** и **куча (heap)**:

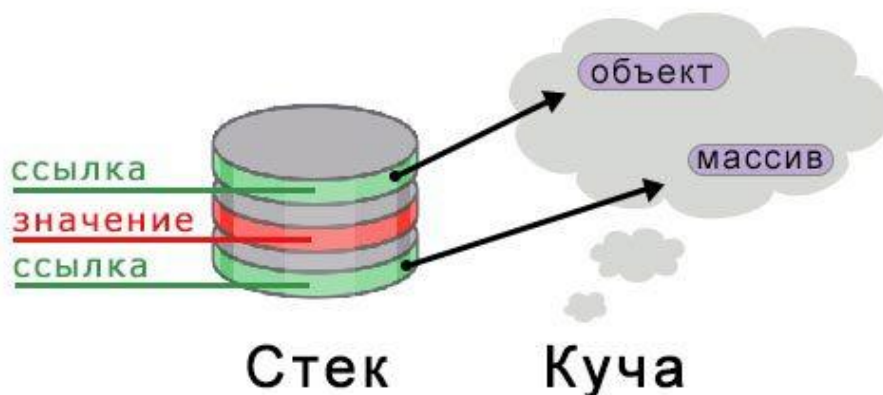


Параметры и переменные метода, которые представляют типы значений, размещают свое значение в стеке. **Стек** представляет собой структуру данных, которая растет снизу-вверх: каждый новый добавляемый элемент помещается поверх предыдущего. Время жизни переменных таких типов ограничено их контекстом. Физически стек - это некоторая область памяти в адресном пространстве.

Ссылочные типы хранятся в **куче** или **хипе**, которую можно представить, как неупорядоченный набор разнородных объектов. Физически это оставшаяся часть памяти, которая доступна процессу.

Когда программа только запускается на выполнение, в конце блока памяти, зарезервированного для стека устанавливается указатель стека. При помещении данных в стек указатель переустанавливается таким образом, что снова указывает на новое

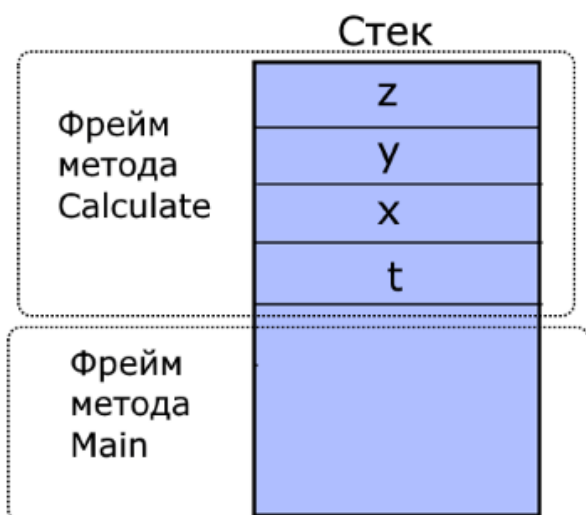
свободное место. При вызове каждого отдельного метода в стеке будет выделяться область памяти или фрейм стека, где будут храниться значения его параметров и переменных.



Рассмотрим следующий пример:

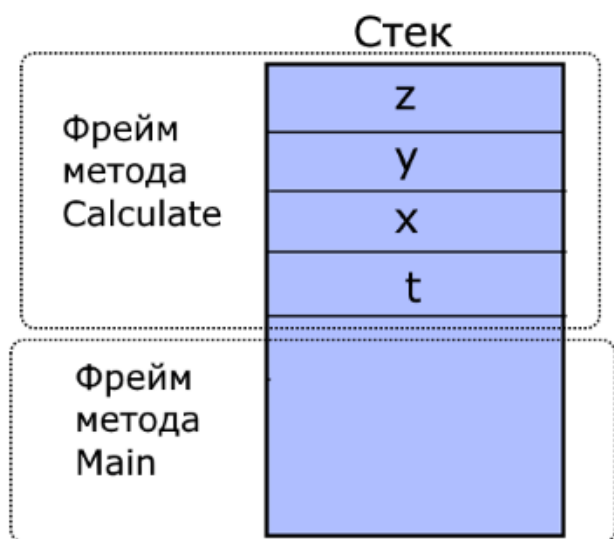
```
internal class Program
{
    0 references
    static void Main(string[] args)
    {
        Calculate(5);
    }
    1 reference
    static void Calculate(int t)
    {
        int x = 6;
        int y = 7;
        int z = y + t;
    }
}
```

При запуске такой программы в стеке будут определяться два фрейма - для метода **Main** (так как он вызывается при запуске программы) и для метода **Calculate**:



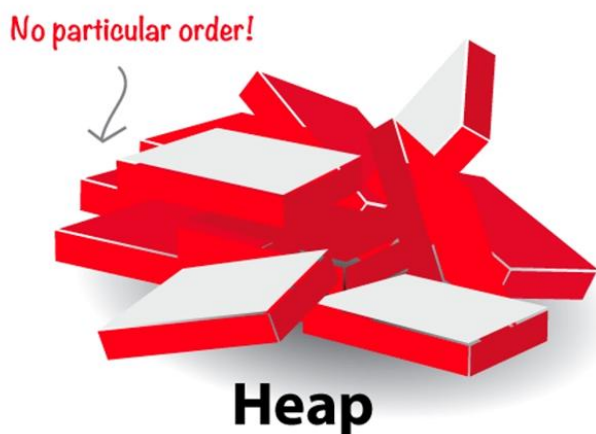
При вызове этого метода **Calculate** в его фрейм в стеке будут помещаться значения **t**, **x**, **y** и **z**. Они определяются в контексте данного метода. Когда метод отработает, область памяти, которая выделялась под стек, впоследствии может быть использована другими методами.

Причем если параметр или переменная метода представляет тип значений, то в стеке будет храниться непосредственное значение этого параметра или переменной.



Например, в данном случае переменные и параметр метода **Calculate** представляют значимый тип - тип **int**, поэтому в стеке будут храниться их числовые значения.

При создании объекта ссылочного типа в стеке помещается ссылка на адрес в **куче (хипе)**. Когда объект ссылочного типа перестает использоваться, в дело вступает автоматический сборщик мусора: он видит, что на объект в хипе нету больше ссылок, условно удаляет этот объект и очищает память - фактически помечает, что данный сегмент памяти может быть использован для хранения других данных.



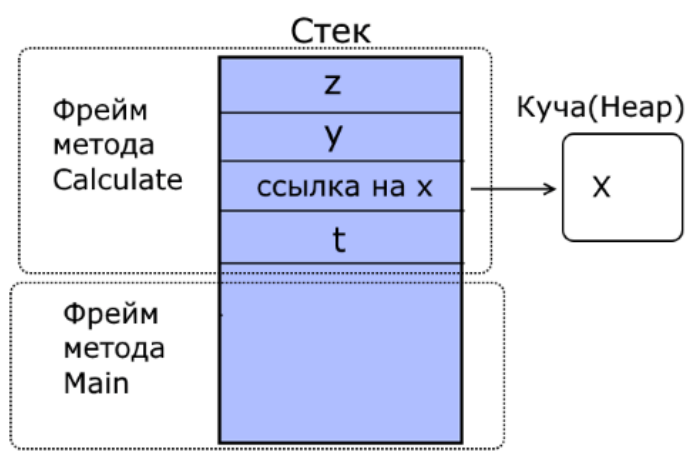
Обновим метод **Calculate()**:


```

static void Calculate(int t)
{
    object x = 6;
    int y = 7;
    int z = y + t;
}

```

Теперь значение переменной **x** будет храниться в **куче**, так как она представляет ссылочный тип **object**, а в стеке будет храниться ссылка на объект в **куче**.



Теперь рассмотрим ситуацию, когда тип значений и ссылочный тип представляют составные типы - структуру и класс:

```

struct State
{
    public int x; public int y;
}

2 references
class Country
{
    public int x; public int y;
}

```

И создадим объекты:

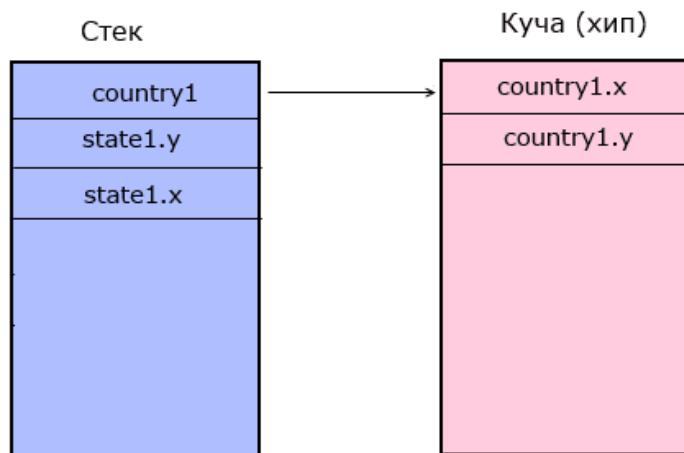
```

static void Main(string[] args)
{
    State state1 = new State();
    Country country1 = new Country();
}

```

State - структура, ее данные размещены в стеке. Country - класс, в стек помещается ссылка на адрес в хипе, а в хипе располагаются все данные объекта country1

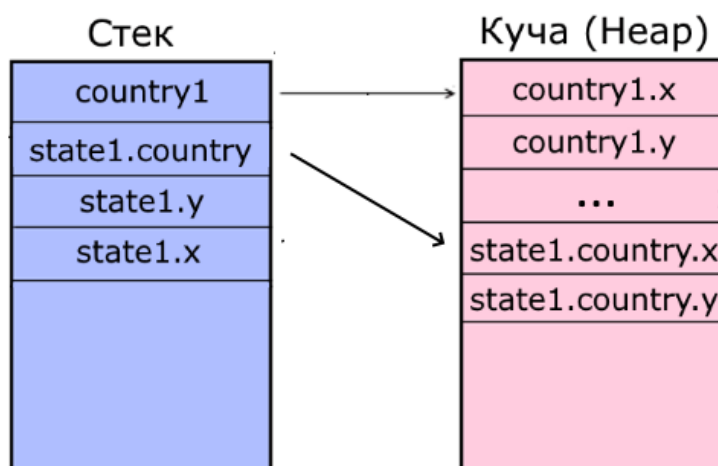
Здесь в методе **Main** в стеке выделяется память для объекта **state1**. Далее в стеке создается ссылка для объекта **country1** (**Country country1**), а с помощью вызова конструктора с ключевым словом **new** выделяется место в хипе (**new Country()**). Ссылка в стеке для объекта **country1** будет представлять адрес на место в хипе, по которому размещен данный объект. Таким образом, в стеке окажутся все поля структуры **state1** и ссылка на объект **country1** в хипе.



Но, допустим, в структуре **State** также определена переменная ссылочного типа **Country**. Где она будет хранить свое значение, если она определена в типе значений?

```
struct State
{
    public int x; public int y; public Country country;
    1 reference
    public State()
    {
        x = 0; y = 0; country = new Country();
    }
}
```

Значение переменной **state1.country** также будет храниться в куче, так как эта переменная представляет ссылочный тип:



Шаг 4. Пространства имен

Обычно определяемые классы и другие типы в .NET не существуют сами по себе, а заключаются в специальные контейнеры - пространства имен. Пространства имен позволяют организовать код программы в логические блоки, позволяют объединить и отделить от остального кода некоторую функциональность, которая связана некоторой общей идеей или которая выполняет определенную задачу.

Для определения пространства имен применяется ключевое слово **namespace**, после которого идет название пространства имен:

```
namespace имя_пространства_имен
{
    // содержимое пространства имен
}
```

Например, определим в файле **Program.cs** пространство имен, которое будет называться **Base**:

```
namespace OOP2
{
    0 references
    internal class Program
    {
        0 references
        static void Main(string[] args)
        {
        }
    }
}

namespace Base
{
    1 reference
    class Person
    {
        string name;
        0 references
        public Person(string name) => this.name = name;
        0 references
        public void Print() => Console.WriteLine($"Имя: {name}");
    }
}
```

Теперь попробуем использовать класс **Person** в методе **Main**:

```
static void Main(string[] args)
{
    Person tom = new Person("Tom");
    // Ошибка - Visual Studio не видит класс Person
    tom.Print();
}
```

Здесь на первой строке мы столкнемся с ошибкой, так как **Visual Studio** не может найти класс **Person**. Чтобы все-таки обратиться к классу **Person**, необходимо использовать полное имя этого класса с учетом пространства имен:

```
Base.Person tom = new ("Tom");
```

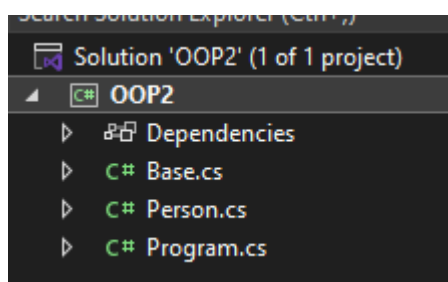
Однако полное имя класса с учетом пространства имен добавляет в код избыточность - особенно, если пространство имен содержит множество классов, которые мы хотим использовать. И чтобы не писать полное имя класса, мы можем просто подключить пространство имен с помощью директивы **using**:

```
1 using Base;
2 namespace OOP2
3 {
4     internal class Program
5     {
6         static void Main(string[] args)
7         {
8             Person tom = new ("Tom");
9         }
10    }
```

Шаг 5. Глобальные пространства имен

Если мы хотим использовать какое-нибудь пространство имен в файлах кода в проекте, то по умолчанию нам надо его подключать во все файлы, где мы планируем его использовать.

Например, пусть у нас в проекте есть три файла с кодом:



В файле **Base.cs** определяется класс **Company** в пространстве **Base**:

```
namespace Base;
class Company
{
    string title;
    public Company(string title) => this.title = title;
    public void Print() => Console.WriteLine($"Компания: {title}");
}
```

В файле **Person.cs** определен класс **Person**, который использует класс **Company**:

```
using Base;
1 reference
class Person
{
    string name;
    Company company;
    0 references
    public Person(string name, Company company)
    {
        this.name = name;
        this.company = company;
    }
    0 references
    public void Print()
    {
        Console.WriteLine($"Имя: {name}");
        company.Print();
    }
}
```

В классе **Program.cs** используются классы **Person** и **Company**:

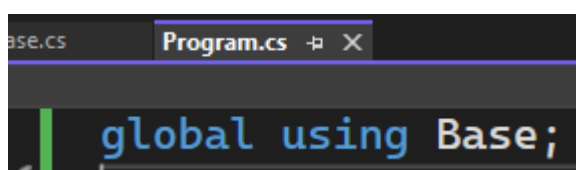
```
using Base;

Company microsoft = new("Microsoft");
Person tom = new("Tom", microsoft);
tom.Print();
```

И таких файлов, где надо подключать пространство Base (или какое-то другое) может быть множество. Это не очень удобно. И в **.NET 6** для этого предложена концепция глобальных пространств имен, который подключаются один раз, но сразу во все файлы кода в проекте. Для этого нам достаточно в одном файле прописать директиву:

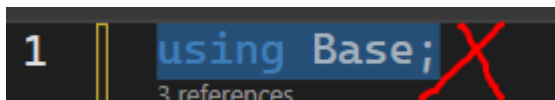
```
global using подключаемое_пространство_имен;
```

Например, изменим файл **Program.cs** следующим образом:

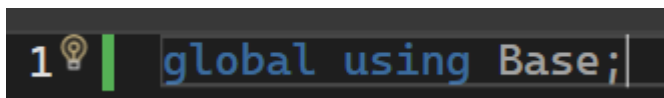
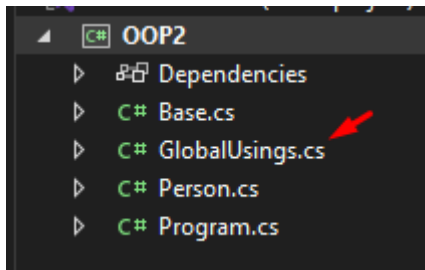


```
Program.cs
global using Base;
```

Теперь пространство **Base** подключается во все файлы кода в проекте. И из файла **Person.cs** мы можем убрать строку:



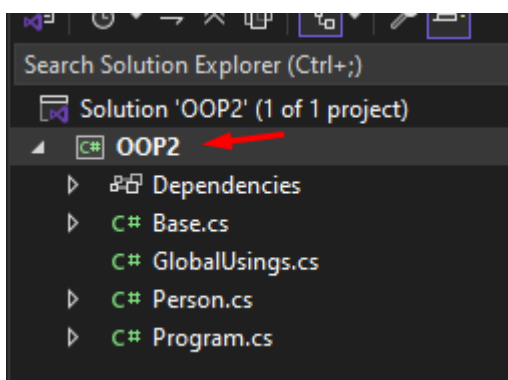
Однако вышеуказанный подход опять же может быть не очень удобным, поскольку проще определить глобальное пространство имен, которые подключаются во весь проект где-то в одном месте. И для этого в **Visual Studio 2022** мы можем добавить в проект новый файл с кодом C# и в нем определить набор подключаемых пространств имен. Например, добавим в проект файл, который назовем **GlobalUsings.cs** и в котором определим следующее содержимое:



Шаг 6. Подключение пространств имен по умолчанию

Все классы существуют в некотором пространстве имен, и чтобы эти классы использовать, необходимо подключить их пространства имен, либо использовать полное название класса с указанием его пространства имен. Однако начиная с Visual Studio 2022 и .NET 6 и C# 10 мы можем просто в файле программы написать.

Щёлкните дважды по вашему проекту:



Начиная с .NET 6 в проекте по умолчанию подключается ряд наиболее часто используемых пространств имен, поэтому нам их не надо явно подключать. Эта настройка действует на уровне всего проекта.

Элемент **<ImplicitUsings>** задает подключение некоторого набора пространств имен по умолчанию. Значение **enable** указывает, что эта настройка будет применяться.

```

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>
9
10 </Project>

```

Для отключения, нужно поменять элемент `<ImplicitUsings>` на `disable`

```

<ImplicitUsings>disable</ImplicitUsings>

```

После нам нужно будет подключить пространство имен **System**, где располагается класс **Console**, либо использовать его полное имя:

```

public void Print()
{
    System.Console.WriteLine($"Имя: {name}");
}

1  using System;

```

Самостоятельные задания

Задание 1 — Класс «Автомобиль»

Создайте класс `Car`, содержащий следующие свойства:

- `Brand` (строка)
- `Model` (строка)
- `Year` (целое число)
- `Mileage` (целое число, в километрах)

Условия:

- Свойства должны иметь модификаторы доступа с авто-свойствами (`get`, `set`).
- Добавьте метод `PrintInfo()`, который выводит всю информацию об автомобиле в консоль.
- В `Main` создайте 2 объекта `Car`, заполните их данными и вызовите `PrintInfo()`.

Задание 2 — Структура «Точка на плоскости»

Создайте структуру `Point`, описывающую координаты точки в 2D-пространстве.

Условия:

- Поля X и Y — целые числа.
- Добавьте метод DistanceTo(Point other), который рассчитывает расстояние между текущей точкой и другой.
- В Main создайте две точки и выведите расстояние между ними.

Подсказка: используйте Math.Sqrt и Math.Pow.

Задание 3 — Работа с пространствами имён

1. Внутри проекта создайте папку Models.
2. В ней создайте класс User со свойствами Name, Email, IsActive.
3. Используйте пространство имён MyApp.Models.

Условия:

- В Main создайте объект User, используя пространство имён через using.
- Выведите информацию о пользователе в консоль.

Задание 4 — Проверка типа значений

Создайте класс ValueChecker, в котором будет метод CheckType(object input), который:

- Проверяет, является ли переданное значение значимым типом или ссылочным.
- Использует is и typeof для вывода результатов.

Пример:

```
CheckType(5); // Вывод: Значимый тип (int)
CheckType("Hello"); // Вывод: Ссылочный тип (string)
```

Задание 5 — Использование глобального пространства имён

1. Создайте класс с именем System (в своём пространстве имён).
2. Попробуйте в Main вызвать System.Console.WriteLine(...) и объясните, почему возникает конфликт.
3. Используйте global::System.Console.WriteLine(...) для решения конфликта.

Задание 6 — Класс «Прямоугольник» со свойствами и проверкой

- Создайте класс Rectangle с приватными полями width и height.
- Реализуйте свойства Width и Height с валидацией: если значение меньше или равно нулю, устанавливать значение по умолчанию = 1.
- Добавьте свойство Area, которое возвращает площадь (только get).