Практическая работа № 1.  
объектно-ориентированное  
программирование

**Цель практической работы**

Закрепление теоретических знаний по разработке объектно-ориентированных программ.

**Постановка задачи**

**Общие сведение о классах и объектах**

C# является полноценным объектно-ориентированным языком. Это значит, что программу на C# можно представить в виде взаимосвязанных взаимодействующих между собой объектов.

Описанием объекта является **класс**, а объект представляет экземпляр этого класса. Можно еще провести следующую аналогию. У нас у всех есть некоторое представление о человеке, у которого есть имя, возраст, какие-то другие характеристики. То есть некоторый шаблон - этот шаблон можно назвать классом. Конкретное воплощение этого шаблона может отличаться, например, одни люди имеют одно имя, другие - другое имя. И реально существующий человек (фактически экземпляр данного класса) будет представлять объект этого класса.

В принципе ранее уже использовались классы. Например, тип **string**, который представляет строку, фактически является классом. Или, например, класс **Console**, у которого метод WriteLine() выводит на консоль некоторую информацию. Теперь же посмотрим, как мы можем определять свои собственные классы.

По сути класс представляет новый тип, который определяется пользователем. Класс определяется с помощью ключевого слова **сlass**.

После слова **class** идет имя класса и далее в фигурных скобках идет собственно содержимое класса. Например, определим в файле **Program.cs** класс Person, который будет представлять человека:

class Person

{

}

Класс может хранить некоторые данные. Для хранения данных в классе применяются **поля**. По сути **поля класса** - это переменные, определенные на уровне класса.

Кроме того, класс может определять некоторое поведение или выполняемые действия. Для определения поведения в классе применяются методы.

Итак, добавим в класс Person поля и методы:

class Person

{

public string name = "Undefined"; // имя

public int age; // возраст

public void Print()

{

Console.WriteLine($"Имя: {name} Возраст: {age}");

}

}

В данном случае в классе Person определено поле name, которое хранит имя, и поле age, которое хранит возраст человека. В отличие от переменных, определенных в методах, поля класса могут иметь модификаторы, которые указываются перед полем. Так, в данном случае, чтобы все поля были доступны вне класса Person поля определены с модификатором **public**.

При определении полей мы можем присвоить им некоторые значения, как в примере выше в случае переменной name. Если поля класса не инициализированы, то они получают значения по умолчанию. Для переменных числовых типов это число 0.

Также в классе Person определен метод Print(). Методы класса имеют доступ к его поля, и в данном случае обращаемся к полям класса name и age для вывода их значения на консоль. И чтобы этот метод был виден вне класса, он также определен с модификатором **public**.

**Библиотеки классов в Visual Studio**

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.

Рассмотрим последовательность шагов для создания библиотеки классов (рисунки 1–3).

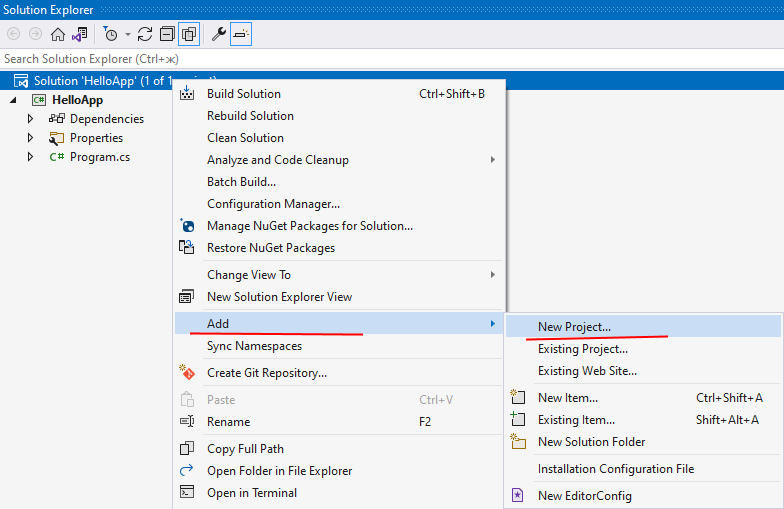


Рисунок 1 – Создание библиотеки классов, шаг 1

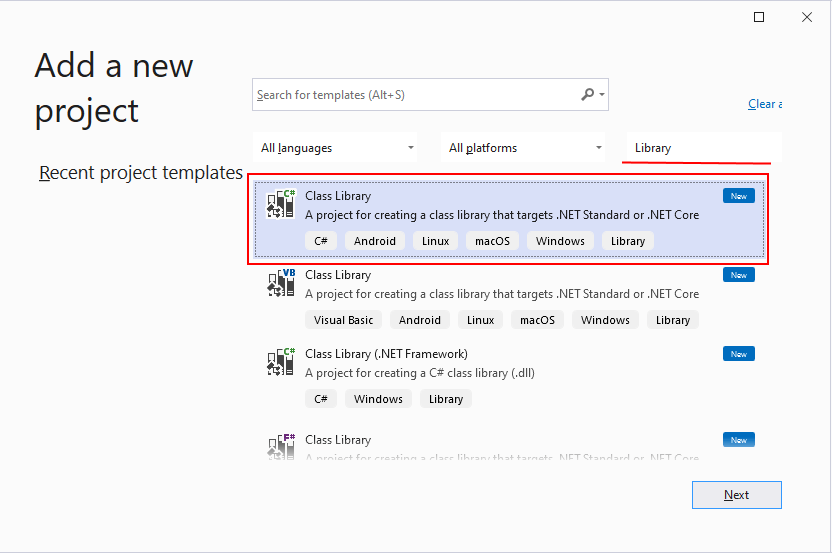


Рисунок 2 – Создание библиотеки классов, шаг 2

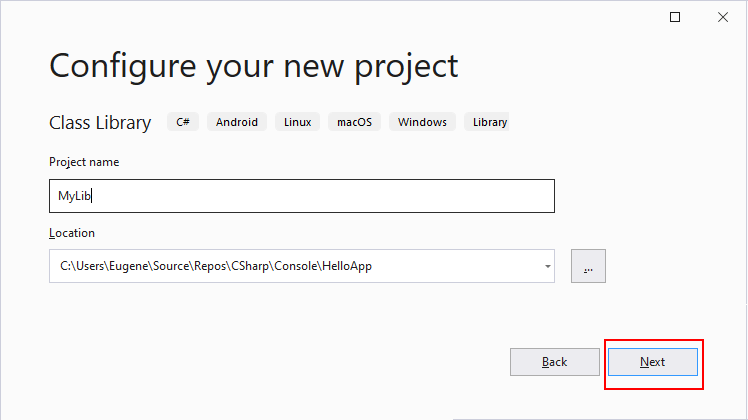


Рисунок 3 – Создание библиотеки классов, шаг 3

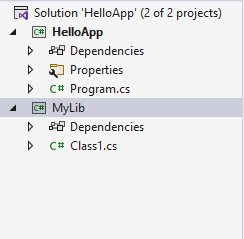


Рисунок 4 – Созданная библиотека классов

По умолчанию новый проект имеет один пустой класс Class1 в файле Class1.cs (рисунок 4). Мы можем этот файл удалить или переименовать, как нам больше нравится.

Например, переименуем файл Class1.cs в Person.cs, а класс Class1 в Person (рисунок 5). Определим в классе Person простейший код:

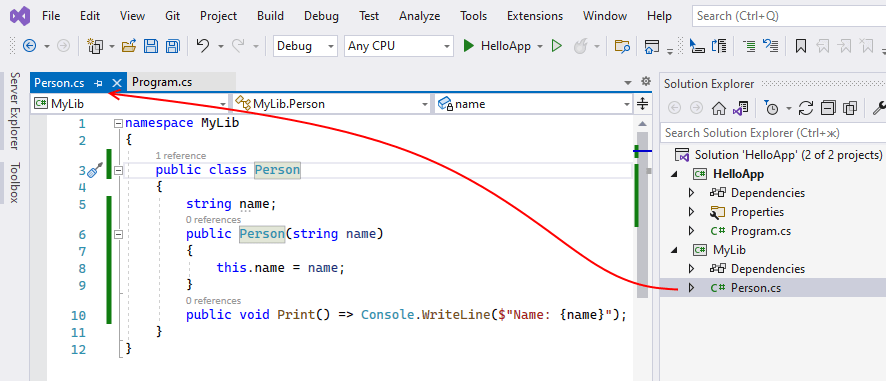


Рисунок 5 – Редактирование класса в библиотеке

namespace MyLib

{

public class Person

{

string name;

public Person(string name)

{

this.name = name;

}

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

}

}

Подключим его в основной проект (рисунки 6, 7). Для этого в основном проекте нажмем правой кнопкой на узел Dependencies и в контекстном меню выберем пункт Add Project Reference...:

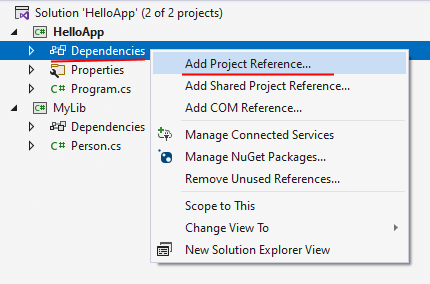


Рисунок 6 – Подключение библиотеки к проекту, шаг 1

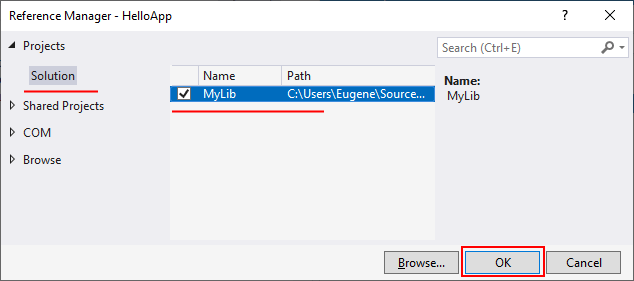


Рисунок 7 – Подключение библиотеки к проекту, шаг 2

Если наша библиотека вдруг представляет файл dll, который не связан ни с каким проектом в нашем решении, то с помощью кнопки **Browse** мы можем найти местоположение файла dll и также его подключить.

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

using MyLib; // подключение пространства имен из библиотеки классов

Person tom = new("Tom");

tom.Print(); // Name: Tom

**Преобразование типов**

**Восходящие преобразования. Upcasting**

Объекты производного типа (который находится внизу иерархии) в то же время представляют и базовый тип. Например, объект Employee в то же время является и объектом класса Person. Что в принципе естественно, так как каждый сотрудник (Employee) является человеком (Person). И мы можем написать, например, следующим образом:

Employee employee = new Employee("Tom", "Microsoft");

Person person = employee; // преобразование от Employee к Person

Console.WriteLine(person.Name);

class Person

{

public string Name { get; set; }

public Person(string name)

{

Name = name;

}

public void Print()

{

Console.WriteLine($"Person {Name}");

}

}

class Employee : Person

{

public string Company { get; set; }

public Employee(string name, string company) : base(name)

{

Company = company;

}

}

class Client : Person

{

public string Bank { get; set; }

public Client(string name, string bank) : base(name)

{

Bank = bank;

}

}

В данном случае переменной person, которая представляет тип Person, присваивается ссылка на объект Employee. Но чтобы сохранить ссылку на объект одного класса в переменную другого класса, необходимо выполнить преобразование типов - в данном случае от типа Employee к типу Person. И так как Employee наследуется от класса Person, то автоматически выполняется неявное восходящее преобразование - преобразование к типу, которые находятся вверху иерархии классов, то есть к базовому классу.

В итоге переменные employee и person будут указывать на один и тот же объект в памяти, но переменной person будет доступна только та часть, которая представляет функционал типа Person.

**Нисходящие преобразования. Downcasting**

Но кроме восходящих преобразований от производного к базовому типу есть нисходящие преобразования или downcasting - от базового типа к производному.

И может возникнуть вопрос, можно ли обратиться к функционалу типа Employee через переменную типа Person. Но автоматически такие преобразования не проходят, ведь не каждый человек (объект Person) является сотрудником предприятия (объектом Employee). И для нисходящего преобразования необходимо применить явное преобразование, указав в скобках тип, к которому нужно выполнить преобразование:

Employee employee1 = new Employee("Tom", "Microsoft");

Person person = employee1; // преобразование от Employee к Person

//Employee employee2 = person; // так нельзя, нужно явное преобразование

Employee employee2 = (Employee)person; // преобразование от Person к Employee

**Способы преобразований**

Во-первых, можно использовать ключевое слово **as**. С помощью него программа пытается преобразовать выражение к определенному типу, при этом не выбрасывает исключение. В случае неудачного преобразования выражение будет содержать значение null:

Person person = new Person("Tom");

Employee? employee = person as Employee;

if (employee == null)

{

Console.WriteLine("Преобразование прошло неудачно");

}

else

{

Console.WriteLine(employee.Company);

}

Стоит отметить, что переменная employee здесь определяется не просто как переменная Employee, а именно Employee? - после названия типа ставится вопросительный знак. Что указывает, что переменная может хранить как значение null, так и значение Employee.

Второй способ заключается в проверке допустимости преобразования с помощью ключевого слова is:

значение is тип

Если значение слева от оператора представляет тип, указаный справа от оператора, то оператор is возвращает true, иначе возвращается false.

Причем оператор **is** позволяет автоматически преобразовать значение к типу, если это значение представляет данный тип. Например:

Person person = new Person("Tom");

if (person is Employee employee)

{

Console.WriteLine(employee.Company);

}

else

{

Console.WriteLine("Преобразование не допустимо");

}

Выражение if (person is Employee employee) проверяет, является ли переменная person объектом типа Employee. И если person является объектом Employee, то автоматически преобразует значение переменной person в тип Employee и преобразованное значение сохраняет в переменную employee. Далее в блоке if мы можем использовать объект employee как значение типа Employee.

Однако, если person не является объектом Employee, как в данном случае, то такая проверка вернет значение false, и преобразование не сработает.

**Пример** кода для выполнения практики

namespace Practice2Example

{

/\* Задание:

\* Файл Input.txt содержит 2 числа A и B, необходимо посчитать НОД и записать результат в файл Output.txt

\*/

/// <summary>

/// Поиск наибольшего общего делителя

/// </summary>

public class GreatestCommonDivisorCalculator

{

private static void readInput(string path, out int a, out int b)

{

var input = File.ReadAllText(path);

var arr = input.Split(' ');

a = int.Parse(arr[0]);

b = int.Parse(arr[1]);

}

public static int CalculateGCD(int a, int b)

{

while (a != 0 && b != 0)

{

if (a > b)

a %= b;

else

b %= a;

}

return a + b;

}

private static void writeOutput(int gcd, string path)

{

File.WriteAllText(path, $"{gcd}");

}

public static void Process(string input, string output)

{

readInput(input, out var a, out var b);

var gcd = CalculateGCD(a, b);

writeOutput(gcd, output);

}

}

}

**Задание на практическую работу**

Решить практические задания с платформы **ulearn.me,** тема «Основы ООП»:

<https://ulearn.me/course/basicprogramming/Klassy_9eb68528-17cb-4d7f-99f3-40210a8f6cef>

**Задачи:**

1. **«Вектор»**

Создайте новый проект в Visual Studio. Выберите в качестве типа проекта Class Library.

В этом проекте создайте два класса, Vector и Geometry, в пространстве имен Geometry.

В классе Vector должно быть два публичных поля, X и Y, типа double.

В классе Geometry должно быть два статических метода: GetLength, который возвращает длину переданного вектора, и Add, который возвращает сумму двух переданных векторов.

1. **«Отрезок»**

Продолжаем разработку геометрической библиотеки.

Создайте класс Segment, представляющий отрезок прямой. Концы его отрезков должны задаваться двумя публичными полями: Begin и End типа Vector.

Добавьте метод Geometry.GetLength, вычисляющий длину сегмента, и метод Geometry.IsVectorInSegment(Vector, Segment), проверяющий, что задаваемая вектором точка лежит в отрезке.

Сохраните функциональность предыдущего этапа.

1. **«Нестатические методы»**

Вы вдруг поняли, что не очень-то удобно писать имя класса Geometry при выполнении любой операции с векторами и сегментами. Однако, отказаться от этого класса вы не можете, потому что за те несколько минут, пока вы сдавали предыдущую задачу, вашу библиотеку скачали и начали использовать в своих проектах тысячи человек.

Поэтому вы решили сохранить этот класс, но добавить методы Vector.GetLength(), Segment.GetLength(), Vector.Add(Vector), Vector.Belongs(Segment) и Segment.Contains(Vector) не вместо, а вместе с соответствующими методами класса Geometry.

Сделайте это! Каждый из этих методов должен вызывать уже существующий метод класса Geometry, чтобы не дублировать код.

Вся функциональность предыдущего этапа должна остаться!

**Пример разработки приложения**

**Практика «Вектор»**

Создайте новый проект в Visual Studio. Выберите в качестве типа проекта Class Library.

В этом проекте создайте два класса, Vector и Geometry, в пространстве имен Geometry.

В классе Vector должно быть два публичных поля, X и Y, типа double.

В классе Geometry должно быть два статических метода: GetLength, который возвращает длину переданного вектора, и Add, который возвращает сумму двух переданных векторов.

Оба класса разместите в одном файле. Вообще-то так обычно делать не стоит, но так удобнее для нашей автоматической проверки выполнения задания.

**Шаг 1. Создание проекта**

1. Откройте **Visual Studio**.
2. Нажмите **"Создать новый проект"**.
3. Выберите **"Библиотека классов"**.
4. Назовите проект (например, GeometryLibrary) и нажмите **"Создать"**.

**Шаг 2. Создание структуры классов**

1. В **обозревателе решений** найдите файл Class1.cs (он создаётся автоматически) и **переименуйте** его в Geometry.cs.
2. Удалите всё содержимое файла и замените его на следующий код:

namespace Geometry

{

    public class Vector

    {

        public double X;  // Координата X вектора

        public double Y;  // Координата Y вектора

    }

    public class Geometry

    {

        // Метод вычисления длины вектора

        public static double GetLength(Vector vector)

        {

           // Ваш код здесь

        }

        // Метод сложения двух векторов

        public static Vector Add(Vector v1, Vector v2)

        {

            // Ваш код здесь

        }

    }

}

**Частые ошибки и решения**

| **Проблема** | **Решение** |
| --- | --- |
| Ошибка: "Имя 'Vector' не существует в текущем контексте" | Убедитесь, что подключено пространство имён using Geometry; |
| Ошибка: "Math.Sqrt не найден" | Добавьте using System; в начало файла |
| Неверные расчёты длины вектора | Проверьте, что используется  X \* X + Y \* Y, а не X + Y |
| Не компилируется метод Add | Убедитесь, что возвращается new Vector, а не null |

# Содержание пояснительной записки

1. Постановка задачи. Приводится теоретический материал, использованный при написании приложения.

2. Формулировка задания и вариант. Приводится задание на лабораторную работу и вариант этого задания.

3. Описание выполняемых действий. Необходимо привести описание последовательности разработки программы, реализации используемых методов, алгоритмов, блок-схем.

4. Анализ результатов. Привести анализ входных и выходных данных. Показать результаты выполнения программного кода. Предоставить скриншоты обработки тестовых примеров. Сделать выводы.

5. Листинг программы. Привести листинг разработанного программного кода, содержание файлов входных и выходных данных.

# Используемое программное обеспечение

1. Среда программирования MS Visual Studio Community 2022 (Свободно распространяемое программное обеспечение (в учебных целях));
2. Microsoft Office Standard 2007 (Open License: 42267924);
3. Open Office (Свободно распространяемое программное обеспечение).
4. Браузер (Свободно распространяемое программное обеспечение).

# Список литературы

* + - 1. Мейер Б. Объектно-ориентированное программирование и программная инженерия [Электронный ресурс]/ Мейер Б. – Электрон. текстовые данные. – М.: Интернет-Университет Информационных Технологий (ИНТУИТ), 2016. – 285 c.
      2. Биллиг, В. A. Основы объектного программирования на С# (C# 3.0, Visual Studio 2008) [Электронный ресурс] : учебное пособие / В. A. Биллиг. — Электрон. текстовые данные. — Москва, Саратов : Интернет-Университет Информационных Технологий (ИНТУИТ), Вузовское образование, 2017. — 583 c. — 978-5-4487-0145-0. — Режим доступа: http://www.iprbookshop.ru/72339.html
      3. Павловская, Т. А. Программирование на языке высокого уровня C# [Электронный ресурс] / Т. А. Павловская. — 2-е изд. — Электрон. текстовые данные. — М. : Интернет-Университет Информационных Технологий (ИНТУИТ), 2016. — 245 c. — 2227-8397. — Режим доступа: http://www.iprbookshop.ru/73713.html
      4. Агапов, В. П. Основы программирования на языке С# [Электронный ресурс] : учебное пособие / В. П. Агапов. — Электрон. текстовые данные. — М. : Московский государственный строительный университет, ЭБС АСВ, 2012. — 128 c. — 978-5-7264-0576-6. — Режим доступа: http://www.iprbookshop.ru/16366.html
      5. Медведев, М. А. Программирование на СИ# [Электронный ресурс] : учебное пособие / М. А. Медведев, А. Н. Медведев ; под ред. А. В. Присяжный. — Электрон. текстовые данные. — Екатеринбург : Уральский федеральный университет, ЭБС АСВ, 2015. — 64 c. — 978-5-7996-1561-1. — Режим доступа: http://www.iprbookshop.ru/69667.html
      6. Казанский А.А. Объектно-ориентированное программирование на языке Microsoft Visual С# в среде разработки Microsoft Visual Studio 2008 и .NET Framework. 4.3 [Электронный ресурс]: учебное пособие и практикум/ Казанский А.А.— Электрон. текстовые данные.— М.: Московский государственный строительный университет, ЭБС АСВ, 2011.— 180 c
      7. Уйманова Н.А. Основы объектно-ориентированного программирования [Электронный ресурс]: практикум/ Уйманова Н.А., Таспаева М.Г.— Электрон. текстовые данные.— Оренбург: Оренбургский государственный университет, ЭБС АСВ, 2017.— 156 c.— Режим доступа: http://www.iprbookshop.ru/78808.html.— ЭБС «IPRbooks»
      8. Новиков П.В. Объектно-ориентированное программирование [Электронный ресурс]: учебно-методическое пособие к лабораторным работам/ Новиков П.В.— Электрон. текстовые данные.— Саратов: Вузовское образование, 2017.— 124 c.— Режим доступа: http://www.iprbookshop.ru/64650.html.— ЭБС «IPRbooks»