***Класс*** представляет собой шаблон, описывающий некое множество объектов. В нем указываются характеристики описывающие данное множество объектов, а также процедуры и функции для работы с объектами класса. Прототипы, строящиеся на основе класса называются ***экземплярами*** класса.

Класс создается с помощью ключевого слова class. Ниже приведена общая форма определения простого класса, содержащая только переменные экземпляра и методы:

class имя\_класса

{

// Объявление переменных класса

доступ тип переменная1;

доступ тип переменная2;

// Объявление методов класса

доступ тип метод1 (параметры)

{

// тело метода

}

доступ тип метод2 (параметры)

{

// тело метода

}

}

ДОСТУП

В C# применяются следующие модификаторы доступа:

* public: публичный, общедоступный класс или член класса. Такой член класса доступен из любого места в коде, а также из других программ и сборок.
* private: закрытый класс или член класса. Представляет полную противоположность модификатору public. Такой закрытый класс или член класса доступен только из кода в том же классе или контексте.
* protected: такой член класса доступен из любого места в текущем классе или в производных классах. При этом производные классы могут располагаться в других сборках.
* internal: класс и члены класса с подобным модификатором доступны из любого места кода в той же сборке, однако он недоступен для других программ и сборок (как в случае с модификатором public).
* protected internal: совмещает функционал двух модификаторов. Классы и члены класса с таким модификатором доступны из текущей сборки и из производных классов.
* private protected: такой член класса доступен из любого места в текущем классе или в производных классах, которые определены в той же сборке.

Пример класса

class student // объявление класса

{

// поля класса

public string FIO;

public string group;

public int Nzach;

// метод для вывода информации о студенте

public void print()

{

Console.WriteLine("студент '{0}' (группы {1}) с номером зачетки {2}", FIO, group, Nzach);

}

}

Инициализация класса (запуск..)

имя\_класса имя\_переменной = new имя\_класса();

Пример

student x = new student();

Использование

class Program

{

static void Main()

{

// создание объекта типа студент

student x = new student();

// инициализация полей объекта Ь1

x.FIO= "Иванов Иван Иванович";x.group = "103";

b1.Nzach = 1911221;

// вывод информации о студенте на экран

x.print();

}

}

 В вышеописанном примере переменные (поля) инициализировались Явно… (с использованием точки) мало того если бы использовался другой модификатор доступа, то явный доступ к переменным был бы недоступен. Для определения переменных класса (публичных и закрытых) используется специальная операция с именем «конструктор»

доступ имя\_класса([список\_параметров])

{

// тело конструктора

}

В программе может быть несколько конструкторов в зависимости от количества и типа передаваемых параметров.

class Person

{

 public string name;

 public int age;

public Person() { name = "Неизвестно"; age = 18; }      // 1 конструктор

public Person(string n) { name = n; age = 18; }         // 2 конструктор

public Person(**string n, int a) { name = n; age = a; }   // 3 конструктор**

public void GetInfo()

{

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

}

}

static void Main(string[] args)

{

Person tom = new Person(); // вызов 1-ого конструктора без параметров

Person bob = new Person("Bob"); //вызов 2-ого конструктора с одним параметром

Person sam = new Person("Sam", 25); // вызов 3-его конструктора с двумя параметрами

bob.GetInfo(); // Имя: Bob  Возраст: 18

tom.GetInfo(); // Имя: Неизвестно  Возраст: 18

sam.GetInfo(); // Имя: Sam  Возраст: 25

}

**Ключевое слово this**

Ключевое слово this представляет ссылку на текущий экземпляр класса. В каких ситуациях оно нам может пригодиться? В примере выше определены три конструктора. Все три конструктора выполняют однотипные действия - устанавливают значения полей name и age. Но этих повторяющихся действий могло быть больше. И мы можем не дублировать функциональность конструкторов, а просто обращаться из одного конструктора к другому через ключевое слово this, передавая нужные значения для параметров:

class Person

{

public string name;

public int age;

public Person() : this("Неизвестно")

{

}

public Person(string name) : this(name, 18)

{

}

public Person(string name, int age)

{

  this.name = name;

  this.age = age;

}

public void GetInfo()

{

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

}

}

В данном случае первый конструктор вызывает второй, а второй конструктор вызывает третий. По количеству и типу параметров компилятор узнает, какой именно конструктор вызывается. Например, во втором конструкторе:

|  |  |
| --- | --- |
|  | public Person(string name) : this(name, 18){} |

идет обращение к третьему конструктору, которому передаются два значения. Причем в начале будет выполняться именно третий конструктор, и только потом код второго конструктора.

Также стоит отметить, что в третьем конструкторе параметры называются также, как и поля класса.

|  |  |
| --- | --- |
|  | public Person(string name, int age){    this.name = name;    this.age = age;} |

И чтобы разграничить параметры и поля класса, к полям класса обращение идет через ключевое слово this. Так, в выражении this.name = name; первая часть this.name означает, что name - это поле текущего класса, а не название параметра name. Если бы у нас параметры и поля назывались по-разному, то использовать слово this было бы необязательно. Также через ключевое слово this можно обращаться к любому полю или методу.

Массивы объектов

Так как массив в C# - это коллекция переменных одинакового типа, обращение к которым происходит с использованием общего для всех имени, то можно организовать работу с массивами объектов.

Для работы с массивами объектов сначала объявляется класс, например student, после чего объявляется ссылочная переменная на массив, а затем для него выделяется память, и переменной массива присваивается ссылка на эту область памяти. Например, с помощью оператора

student[] students = new student [15];

создается массив объектов студенты (состоящий из 15 объектов типа student).

Доступ к отдельному элементу массива осуществляется посредством индекса, к элементу объекта - посредством имени члена класса. Например, оператор

students [3].Fio = “Иванов Иван Иванович”;

записывает год издания книги в 4-й объект.

**Свойства**

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

Стандартное описание свойства имеет следующий синтаксис:

[модификатор\_доступа] возвращаемый\_тип произвольное\_название

{

    // код свойства

}

Например:

class person

{

    private string FIO;

    public string fio

    {

        get

        {

            return FIO;

        }

        set

        {

            name = value;

        }

    }

}

Здесь у нас есть закрытое поле FIO и есть общедоступное свойство fio. 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 FIO. Стандартное определение свойства содержит блоки **get** и **set**. В блоке get мы возвращаем значение поля, а в блоке set устанавливаем. Параметр value представляет передаваемое значение.

Мы можем использовать данное свойство следующим образом:

person p = new person();

// Устанавливаем свойство - срабатывает блок Set

// значение "Tom" и есть передаваемое в свойство value

p.fio = "Иванов И.И.";

// Получаем значение свойства и присваиваем его переменной - срабатывает блок Get

string stud = p.fio;

Возможно, может возникнуть вопрос, зачем нужны свойства, если мы можем в данной ситуации обходиться обычными полями класса? Но свойства позволяют вложить дополнительную логику, которая может быть необходима, например, при присвоении переменной класса какого-либо значения. Например, нам надо установить проверку по коректности ввода возраста:

class Person

{

    private int age;

    public int Age

    {

        set

        {

            if (value < 18)

            {

                Console.WriteLine("Возраст должен быть больше 17");

            }

            else

            {

                age = value;

            }

        }

        get { return age; }

    }

}

Если бы переменная age была бы публичной, то мы могли бы передать ей извне любое значение, в том числе отрицательное. Свойство же позволяет скрыть данные объекты и опосредовать к ним доступ.

Блоки set и get не обязательно одновременно должны присутствовать в свойстве. Если свойство определяют 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:

class Person

{

    private string name;

    // свойство только для чтения

    public string Name

    {

        get

        {

            return name;

        }

    }

    private int age;

    // свойство только для записи

    public int Age

    {

        set

        {

            age = value;

        }

Свойства управляют доступом к полям класса. Однако что, если у нас с десяток и более полей, то определять каждое поле и писать для него однотипное свойство было бы утомительно. Поэтому с версии .NET 4.0 в фреймворк были добавлены автоматические свойства. Они имеют сокращенное объявление:

class Person

{

public string Name { get; set; }

public int Age { get; set; }

public Person(string name, int age)

{

Name = name;

Age = age;

}

}

На самом деле тут также создаются поля для свойств, только их создает не программист в коде, а компилятор автоматически генерирует при компиляции.

С одной стороны, автоматические свойства довольно удобны. С другой стороны, стандартные свойства имеют ряд преимуществ: например, они могут инкапсулировать дополнительную логику проверки значения.

У вас может возникнуть вопрос, а в чем тогда разница между простыми открытыми полями и автоматическими свойствами. У таких свойств остается возможность делать их только на чтение или только на запись. Для этого уже используется модификатор доступа private перед именем аксессора:

public string Name { get; private set; } // свойство только на чтение public string Name { private get; set; } // свойство только на запись

### Сокращенная запись свойств

Как и методы, мы можем сокращать свойства. Например:

class Person

{

    private string name;

    // эквивалентно public string Name { get { return name; } }

    public string Name => name;

}

Индексаторы

В C# наряду с обычными свойствами существуют свойства-массивы. Роль свойств- массивов выполняют индексаторы.

Индексаторы позволяют индексировать объекты и обращаться к данным по индексу. Фактически с помощью индексаторов мы можем работать с объектами как с массивами. По форме они напоминают свойства со стандартными блоками get и set, которые возвращают и присваивают значение.

Формальное определение индексатора:

возвращаемый\_тип this [тип параметр1, ...]

|  |  |  |  |
| --- | --- | --- | --- |
| { | get | { .. | • • } |
| } | set | { .. | • • } |
|  | В | отличие от свойств индексатор не имеет названия. Вместо его указывается ключевое |

слово this, после которого в квадратных скобках идут параметры. Индексатор должен иметь как минимум один параметр.

**Пример**. Допустим, у нас есть класс Person, который представляет человека, и класс People, который представляет группу людей. Используем индексаторы для определения класса People:

class Person

{

public string Name { get; set; }

}

class People

{

Person[] data;

public People()

{

data = new Person[5];

}

// индексатор

public Person this[int index]

{

get

{

return data[index];

}

set

{

data[index] = value;

}

}

}

Конструкция public Person this[int index] и представляет индексатор. Здесь определяем, во-первых, тип возвращаемого или присваиваемого объекта, то есть тип Person. Во-вторых, определяем через параметр int index способ доступа к элементам.

По сути, все объекты Person хранятся в классе в массиве data. Для получения их по индексу в индексаторе определен блок get:

get

{

return data[index];

}

Поскольку индексатор имеет тип Person, то в блоке get нам надо возвратить объект этого типа с помощью оператора return. Здесь мы можем определить разнообразную логику. В данном случае просто возвращаем объект из массива data.

В блоке set получаем через параметр value переданный объект Person и сохраняем его в массив по индексу.

После этого мы можем работать с объектом People как с набором объектов Person:

class Program

{

static void Main(string[] args)

{

People people = new People();

people[0] = new Person { Name = "Tom" };

people[1] = new Person { Name = "Bob" };

Person tom = people[0];

Console.WriteLine(tom.Name);

Console.ReadLine();

}

}

Давайте создадим простейший класс **Room**, в котором будут находиться данные о длине, ширине и высоте комнаты, а также будет содержаться информация о том, есть ли в ней окна.Реализуем метод, который возвращает площадь комнаты, а также реализуем метод, который возвращает объём комнаты (так сказать рефакторинг кода):

class Room {

    public double length; //длина комнаты

    public double height; //высота комнаты

    public double width; //ширина комнаты

    public bool windows; //есть ли окна?

public double SRoom(double x, double y) {

    return x \* y;

}

public double VRoom(double x, double y, double z) {

    return x \* y \* z;

}

}

class Program

    {

        static void Main(string[] args)

        {

            Room myRoom = new Room();

            myRoom.length = 7;

            myRoom.height = 2.5;

            myRoom.width = 4;

            myRoom.windows = true;

            Console.WriteLine("Объем комнаты равен = " + myRoom.VRoom(myRoom.length,myRoom.height,myRoom.width));

            Console.WriteLine("Площадь комнаты равна = " + myRoom.SRoom(myRoom.length, myRoom.width));

            if (myRoom.windows) Console.WriteLine("В комнате имеются окна."); else Console.WriteLine("В комнате нет окон.");

            Console.ReadLine();

        }

    }

Добавим поля

class Room {

     public double length{ get;

set{

            if (value < 0)

            {

                Console.WriteLine("длина комнаты должна быть больше нуля");

            }

            else

            {

                age = value;

            }

        }; }; //длина комнаты

Ну и т.д.

Создадим класс дом

class house

{

Room[] flat;

public house()

{

flat = new room[5];

}

// индексатор

public room this[int index]

{

get

{

return flat[index];

}

set

{

flat[index] = value;

}

}

}

class Program

{

static void Main(string[] args)

{

house dom = new house();

dom[0] = new room { length = 10, height =5, width =3, windows =true};

dom[1] = new Person { length = -5, height =7, width =3, windows =true };

room room1 = dom[0];

Console.WriteLine(room1.length);

Console.ReadLine();

}

}

Задания

Описать класс **Triangle** *(Треугольник)* с полями *a, b, c* - длины 3-х сторон. Определить метод, выводящий тип треугольника по сторонам (разносторонний, равнобедренный, равносторонний).

Описать класс **Parallelogram** (*Параллелограмм*) с полями *a*, *b* - стороны параллелограмма и а - угол между сторонами. Определить метод, выводящий вид параллелограмма (квадрат, прямоугольник, ромб).

Для *N* заданных треугольников определить *k1* - количество разносторонних треугольников, *k2* - количество равнобедренных треугольников, *k3 -* количество равносторонних треугольников.

Для *N* заданных фигур определить *k1* - количество квадратов, *k2* - количество прямоугольников, *k3* - количество ромбов.

Ввести информацию по N школьникам (Ф.И.О, пол, год рождения). Определить количество мальчиков и девочек. Вывести список каждых.

Ввести информацию по N перевозкам (№ рейса, пункт назначения, вес). Вывести сведения о перевозке с минимальным весом. Найти суммарный объем всех перевозок.