Министерство образования, науки и молодежной политики

Нижегородской области

Государственное бюджетное образовательное учреждение

высшего образования

«Нижегородский государственный инженерно-экономический университет»

(ГБОУ ВО НГИЭУ)

Институт информационных технологий и систем связи

Кафедра: Информационные системы и технологии

**Конспекты**

по дисциплине «Основы программирования на C#»

Выполнил:

студент группы 19ИЗ,

заочной формы обучения

института ИТиСС

Володин Евгений Викторович

Проверил:

старший преподаватель

Маслов Никита Сергеевич

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

(подпись)

г. Княгинино

2021

**Содержание**

1. Конспект сведений о работе с файлами (класс StreamWriter, класс StreamReader)……………………………………………………………..3
2. Конспект сведений об операторах проверки и операторе безусловного перехода goto……………………………………………………………..7
3. Конспект сведений о циклах. Вложенные циклы……………………..12
4. Конспект сведений об инициализации массивов……………………...14
5. Конспект сведений о классе Array……………………………………...15
6. Конспект сведений о спецификаторах классов и методах……………17
7. Список используемых источников……………………………………..22

**Работа с файлами в Си-шарп. Классы StreamReader и StreamWriter**

**Файл** – это набор данных, который хранится на внешнем запоминающем устройстве (например на жестком диске). Файл имеет имя и расширение. Расширение позволяет идентифицировать, какие данные и в каком формате хранятся в файле.  
  
Под работой с файлами подразумевается:  
  
- cоздание файлов;  
- удаление файлов;  
- чтение данных;  
- запись данных;  
  
- изменение параметров файла (имя, расширение…);  
- другое.  
  
В Си-шарп есть пространство имен **System.IO**, в котором реализованы все необходимые нам классы для работы с файлами. Чтобы подключить это пространство имен, необходимо в самом начале программы добавить строку using System.IO. Для использования кодировок еще добавим пространство using System.Text;

using System;  
using System.Collections.Generic;  
using System.Linq;  
using System.Text;  
using System.IO;

**Как создать файл?**  
  
Для создания пустого файла, в классе **File** есть метод **Create()**. Он принимает один аргумент – путь. Ниже приведен пример создания пустого текстового файла new\_file.txt на диске D:

static void Main(string[] args)  
{  
   File.Create("D:\\new\_file.txt");  
}

Если файл с таким именем уже существует, он будет переписан на новый пустой файл.  
  
Метод **WriteAllText()** создает новый файл (если такого нет), либо открывает существующий и записывает текст, заменяя всё, что было в файле:

static void Main(string[] args)  
{  
   File. WriteAllText("D:\\new\_file.txt", "текст");  
}

Метод **AppendAllText()** работает, как и метод WriteAllText() за исключением того, что новый текст дописывается в конец файла, а не переписывает всё что было в файле:

static void Main(string[] args)  
{  
   File.AppendAllText("D:\\new\_file.txt", "текст метода AppendAllText ()"); //допишет текст в конец файла  
}

**Как удалить файл?**  
  
Метод **Delete()** удаляет файл по указаному пути:

static void Main(string[] args)  
{  
   File.Delete("d:\\test.txt"); //удаление файла  
}

Кроме того, чтобы читать/записывать данные в файл с Си-шарп можно использовать потоки.  
  
**Поток** – это абстрактное представление данных (в байтах), которое облегчает работу с ними. В качестве источника данных может быть файл, устройство ввода-вывода, принтер.  
  
Класс **Stream** является абстрактным базовым классом для всех потоковых классов в Си-шарп. Для работы с файлами нам понадобится класс **FileStream** (файловый поток).  
  
**FileStream** - представляет поток, который позволяет выполнять операции чтения/записи в файл.

static void Main(string[] args)  
{  
   FileStream file = new FileStream("d:\\test.txt", FileMode.Open  
, FileAccess.Read); //открывает файл только на чтение  
}

Режимы открытия **FileMode**:  
  
- *Append* – открывает файл (если существует) и переводит указатель в конец файла (данные будут дописываться в конец), или создает новый файл. Данный режим возможен только при режиме доступа FileAccess.Write.  
- *Create* - создает новый файл(если существует – заменяет)  
- *CreateNew* – создает новый файл (если существует – генерируется исключение)  
- *Open* - открывает файл (если не существует – генерируется исключение)  
- *OpenOrCreate* – открывает файл, либо создает новый, если его не существует  
- *Truncate* – открывает файл, но все данные внутри файла затирает (если файла не существует – генерируется исключение)

static void Main(string[] args)  
{  
   FileStream file1 = new FileStream("d:\\file1.txt", FileMode.CreateNew); //создание нового файла  
   FileStream file2 = new FileStream("d:\\file2.txt", FileMode.Open); //открытие существующего файла  
   FileStream file3 = new FileStream("d:\\file3.txt", FileMode.Append); //открытие файла на дозапись в конец файла  
}

Режим доступа **FileAccess**:  
  
- *Read* – открытие файла только на чтение. При попытке записи генерируется исключение  
- *Write* - открытие файла только на запись. При попытке чтения генерируется исключение  
- *ReadWrite* - открытие файла на чтение и запись.  
  
**Чтение из файла**  
  
Для чтения данных из потока нам понадобится класс **StreamReader**. В нем реализовано множество методов для удобного считывания данных. Ниже приведена программа, которая выводит содержимое файла на экран:

static void Main(string[] args)  
{  
   FileStream file1 = new FileStream("d:\\test.txt", FileMode.Open); //создаем файловый поток  
   StreamReader reader = new StreamReader(file1); // создаем «потоковый читатель» и связываем его с файловым потоком  
   Console.WriteLine(reader.ReadToEnd()); //считываем все данные с потока и выводим на экран  
   reader.Close(); //закрываем поток  
   Console.ReadLine();  
}

Метод **ReadToEnd()** считывает все данные из файла. **ReadLine()** – считывает одну строку (указатель потока при этом переходит на новую строку, и при следующем вызове метода будет считана следующая строка).  
  
Свойство **EndOfStream** указывает, находится ли текущая позиция в потоке в конце потока (достигнут ли конец файла). Возвращает *true* или *false*.  
  
**Запись в файл**  
  
Для записи данных в поток используется класс **StreamWriter**. Пример записи в файл:

static void Main(string[] args)  
{  
   FileStream file1 = new FileStream("d:\\test.txt", FileMode.Create); //создаем файловый поток  
  StreamWriter writer = new StreamWriter(file1); //создаем «потоковый писатель» и связываем его с файловым потоком  
   writer.Write("текст"); //записываем в файл  
   writer.Close(); //закрываем поток. Не закрыв поток, в файл ничего не запишется  
}

Метод **WriteLine()** записывает в файл построчно (то же самое, что и простая запись с помощью Write(), только в конце добавляется новая строка).  
  
Нужно всегда помнить, что после работы с потоком, его нужно закрыть (освободить ресурсы), использовав метод **Close()**.  
  
**Кодировка**, в которой будут считываться/записываться данные указывается при создании StreamReader/StreamWriter:

static void Main(string[] args)  
{  
   FileStream file1 = new FileStream("d:\\test.txt", FileMode.Open);  
   StreamReader reader = new StreamReader(file1, Encoding.Unicode);  
   StreamWriter writer = new StreamWriter(file1, Encoding.UTF8);  
}

Кроме того, при использовании StreamReader и StreamWriter можно не создавать отдельно файловый поток FileStream, а сделать это сразу при создании StreamReader/StreamWriter:

static void Main(string[] args)  
{  
   StreamWriter writer = new StreamWriter("d:\\test.txt"); //указываем путь к файлу, а не поток  
   writer.WriteLine("текст");  
   writer.Close();  
}

**Как создать папку?**  
  
С помощью статического метода **CreateDirectory()** класса **Directory**:

static void Main(string[] args)  
{  
  Directory.CreateDirectory("d:\\new\_folder");  
}

**Как удалить папку?**  
  
Для удаления папок используется метод **Delete()**:

static void Main(string[] args)  
{  
   Directory.Delete("d:\\new\_folder"); //удаление пустой папки  
}

Если папка не пустая, необходимо указать параметр рекурсивного удаления - true:

static void Main(string[] args)  
{  
   Directory.Delete("d:\\new\_folder", true); //удаление папки, и всего, что внутри  
}

**Условные операторы в Си-шарп. Тернарный оператор, оператор безусловного перехода Goto**

Условные операторы служат для ветвления программы. В зависимости от некоторого условия выполняется тот или другой набор команд.  
  
В Си-шарп есть три условных оператора: «if-else», «switch» и «?:» - тернарный оператор.  
  
**Оператор «if-else»**  
  
Данный оператор имеет следующую структуру:

if ([условное выражение])  
{  
   Блок кода, который нужно выполнить при удовлетворении условия,   [условное выражение] = true (истина)  
}  
else  
{  
  Блок кода, который нужно выполнить при неудовлетворении условия,   [условное выражение] = false (ложь)  
}

Часть else не является обязательной и может отсутствовать.  
  
Пример использования оператора «if-else» в программе, которая проверяет вводимое число на чётность:

static void Main(string[] args)  
{  
   int a;  
   Console.WriteLine("Введите число:");  
   a = Convert.ToInt32(Console.ReadLine()); // вводим данные с клавиатуры\*  
   if (a % 2 == 0) //проверяем число на чётность путем нахождения остатка от деления числа на 2  
   {  
     Console.WriteLine("Число " + a + " - чётное");  
   }  
   else  
   {  
     Console.WriteLine("Число " + a + " - нечётное");  
   }  
   Console.ReadKey();  
}

\* Функция Console.ReadLine() позволяет ввести данные с клавиатуры. Данные вводятся как строка, а так как нужно число, мы преобразовываем ее в числовой тип. Для преобразования мы используем функцию Convert.ToInt32().  
  
Если после if или else необходимо выполнить лишь одну команду, фигурные скобки можно опускать:  
  
if ([условное выражение])  
[команда1] // команда1 выполнится лишь если условное выражение истинно  
[команда2]// команда2 выполнится в любом случае  
  
Оператор if может иметь несколько условий:  
  
if ([логическое выражение1])  
{блок1}  
else if ([логическое выражение2])  
{блок2}  
else  
{блок3}  
  
Пример программы, которая определяет, какое из двух введенных чисел больше:

static void Main(string[] args)  
{  
   int a, b;  
   Console.WriteLine("Введите первое число:");  
   a = Convert.ToInt32(Console.ReadLine());  
   Console.WriteLine("Введите второе число:");  
   b = Convert.ToInt32(Console.ReadLine());  
   if (a > b)  
     Console.WriteLine("Первое число больше второго");  
   else if (a < b)  
     Console.WriteLine("Второе число больше первого");  
   else  
     Console.WriteLine("Числа равны");  
  
   Console.ReadKey();  
}

Логическое выражение может быть сложнее. Здесь и используются логические операторы «!», «||» и «&&».  
  
Пример программы которая дает совет, что делать, в зависимости от температуры на дворе:

static void Main(string[] args)  
{  
   int t;  
   Console.WriteLine("Введите температуру во дворе");  
   t = Convert.ToInt32(Console.ReadLine());  
   if (t < -20 || t > 40) //если температура меньше -20 или больше 40  
     Console.WriteLine("Вам лучше посидеть дома!");  
   else  
     Console.WriteLine("Можете идти гулять");  
   Console.ReadKey();  
}

**Оператор switch**  
  
В некоторых случаях удобно использовать условный оператор «switch» вместо «if-else». Он имеет следующую структуру:  
  
switch (выражение)  
{  
   case значение1:  
     блок1;  
     break;  
   case значение2:  
     блок2;  
     break;  
     ...  
   case значениеN:  
     блокN;  
     break;  
   default:  
     блокN+1;  
     break;  
}  
  
Выражение сравнивается последовательно со значениями. Если выражение равно значению – выполняется соответственный блок кода и при достижении ключевого слова break оператор switch заканчивает работу. Если выражение не будет соответствовать ни одному значению, тогда выполнится блок после default.  
  
Пример программы с использованием switch, которая выводит на экран название дня недели соответственно вводимому порядковому номер дня:

static void Main(string[] args)  
{  
   int a;  
   Console.WriteLine("Введите порядковый номер дня недели:");  
   a = Convert.ToInt32(Console.ReadLine());  
   switch (a)  
   {  
     case 1:  
       Console.WriteLine("Понедельник");  
       break;  
     case 2:  
       Console.WriteLine("Вторник");  
       break;  
     case 3:  
       Console.WriteLine("Среда");  
       break;  
     case 4:  
       Console.WriteLine("Четверг");  
       break;  
     case 5:  
       Console.WriteLine("Пятница");  
       break;  
     case 6:  
       Console.WriteLine("Суббота");  
       break;  
     case 7:  
       Console.WriteLine("Воскресенье");  
       break;  
     default :  
       Console.WriteLine("Ошибка");  
       break;  
     }  
     Console.ReadKey();  
}

**Тернарный оператор «?:»**  
  
Этот оператор используется для сокращения объема кода. Им можно заменять простые по сложности операторы if-else. Тернарный оператор имеет такую структуру:  
  
логическое выражение ? выражение1 : выражение2  
  
Сначала вычисляется логическое выражение. Если оно истинно, то вычисляется выражение1, в противном случае - вычисляется выражение2.  
  
Пример использования тернарного оператора «?:» в той же программе для проверки числа на чётность:

static void Main(string[] args)  
{  
   int a;  
   Console.WriteLine("Введите число:");  
   a = Convert.ToInt32(Console.ReadLine());  
   Console.WriteLine( a % 2 == 0 ? "Число чётное" : "Число нечётное" );  
   Console.ReadKey();  
}

«?:» также можно использовать для присваивания значений. Пример программы, которая находит большее число из двух вводимых:

static void Main(string[] args)  
{  
   int a, b, max;  
   Console.WriteLine("Введите первое число:");  
   a = Convert.ToInt32(Console.ReadLine());  
   Console.WriteLine("Введите второе число:");  
   b = Convert.ToInt32(Console.ReadLine());  
   max = a > b ? a : b;  
}

**Оператор безусловного перехода goto**

Оператор безусловного перехода goto имеет формат:

**goto <метка>;**

В теле той же функции должна присутствовать ровно одна конструкция вида:

**<метка>: <оператор>;**

Оператор *goto* передает управление на помеченный меткой оператор. Рассмотрим пример использования оператора *goto*:

static void Main()

{

float x;

metka: Console.WriteLine("x="); //оператор, отмеченный меткой

x = float.Parse(Console.ReadLine());

if (x!=0) Console.WriteLine("y({0})={1}", x, 1 / x );

else

{

Console.WriteLine("функция не определена");

goto metka; // передача управление метке

}

}

Следует учитывать, что использование оператора *goto* затрудняет чтение больших по объему программ, поэтому использовать метки нужно только в крайних случаях, например, в операторе *switch*.

**Циклы в Си-шарп.**

**Цикл while**  
  
Слово while переводится, как «пока», что хорошо его характеризует. Он продолжает выполнятся до тех пор, пока «истинно» некоторое условие. Он имеет такую структуру:

while (условие продолжения)  
{  
   //блок кода, который будет повторяться  
}

Сначала проверяется условие, а дальше выполняется блок кода.  
  
Пример той же программы, которая выводит на экран числа 0, 1, 2, 3, 4:

int i = 0;  
while (i < 5)  
{  
   Console.WriteLine(i);  
   i++;  
}

Цикл может выполнятся «вечно», если задать всегда истинное условие:

while (true)  
{  
   Console.WriteLine("Вечный цикл");  
}

**Цикл do-while**  
  
Этот тот же цикл while, только здесь сначала выполняется блок кода, а уже потом идет проверка условия. Это гарантирует хотя бы один проход цикла.

do  
{  
//блок кода, который будет повторяться  
}  
while (условие продолжения);

Пример программы, которая не завершит работу, пока с клавиатуры не введут число 5:

static void Main(string[] args)  
{  
   int number;  
   do  
   {  
      Console.WriteLine("Введите число 5");  
      number = Convert.ToInt32(Console.ReadLine());  
   }  
   while (number != 5);  
}

Как и условные конструкции, циклы также могут быть вложены друг в друга.

Вложенные циклы – это циклы, организованные в теле другого цикла. Вложенный цикл в тело другого цикла, называется внутренним циклом. Цикл, в теле которого существует вложенный цикл, называется внешним.

Полное число исполнений внутреннего цикла, всегда равно произведению числа итераций внутреннего цикла на произведение чисел итераций всех внешних циклов, например, если внешний цикл имеет 5 итераций, а внутренний 10, то общее число итераций внутреннего цикла будет 5 \* 10 = 50 итераций.

Пример с вложением одного цикла for в другой цикл for:

for (int i = 0; i < 10; i++)

{

// Выводим одну строку из 10 звездочек.

for (int j = 0; j < 10; j++)

{

Console.Write("\*");

}

// Переход на новую строку.

Console.WriteLine();

}

**Примечание**: для переменных счётчиков также действуют правила областей видимости, поэтому во вложенном цикле в качестве переменной счётчика нельзя снова создать переменную i – при попытке создания будет ошибка!

**Массивы в Си-шарп. Инициализация.**

**Массив** – это набор однотипных данных, которые располагаются в памяти последовательно друг за другом. Доступ к элементам массива осуществляется по индексу (номеру) элемента. Массив может содержать элементы любого типа данных, можно даже создавать массив массивов (ступенчатый массив). Количество элементов в массиве называется размером массива. Массивы относятся к ссылочным [типам данных](http://mycsharp.ru/post/7/2013_04_15_peremennye_tipy_dannyx_konstanty_v_si-sharp.html).  
  
Массивы в Си-шарп могут быть одномерными и многомерными.  
  
**Одномерные массивы**  
  
Одномерный массив по-другому еще называется вектором, и для доступа к его элементам используется только один индекс. Выглядит вектор примерно так:  
  
одномерный массив в Си-шарп  
  
В Си-шарп объявление массива имеет такую структуру:  
  
тип[] имя\_массива = new тип[размер массива];  
  
Пример:

int[] array = new int[5]; // создаем массив целых чисел  
string[] seasons = new string[4] {"зима","весна","лето","осень"}; //объявление массива строк и его инициализация значениями

Если происходит инициализация, оператор new можно упускать:

string[] seasons = {"зима","весна","лето","осень"}; //корректно

Доступ к элементам осуществляется по индексу. Следует помнить, что индексация начинается с нуля – первый элемент массива имеет индекс 0, а последний n-1, где n – размер массива.

static void Main(string[] args)  
{  
   int[] numbers = new int[5];  
   numbers[0] = 5;  
   numbers[1] = 2;  
   numbers[4] = 3;  
   numbers[5] = 2; // ошибка, индекс вне рамок массива  
}

В качестве массива можно представить, например, список студентов в группе (имена), показатели температуры воздуха за последние несколько дней и так далее.

## **Класс Array**

Все массивы в C# построены на основе класса **Array** из пространства имен System. Этот класс определяет ряд свойств и методов, которые мы можем использовать при работе с массивами. Основные свойства и методы:

* Свойство **Length** возвращает длину массива
* Свойство **Rank** возвращает размерность массива
* Статический метод **BinarySearch()** выполняет бинарный поиск в отсортированном массиве
* Статический метод **Clear()** очищает массив, устанавливая для всех его элементов значение по умолчанию
* Статический метод **Copy()** копирует часть одного массива в другой массив
* Статический метод **Exists()** проверяет, содержит ли массив определенный элемент
* Статический метод **Find()** находит элемент, который удовлеворяет определенному условию
* Статический метод **FindAll()** находит все элементы, которые удовлеворяет определенному условию
* Статический метод **IndexOf()** возвращает индекс элемента
* Статический метод **Resize()** изменяет размер одномерного массива
* Статический метод **Reverse()** располагает элементы массива в обратном порядке
* Статический метод **Sort()** сортирует элементы одномерного массива

Разберем самые используемые методы. Например, изменим порядок элементов и размер массива:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | int[] numbers = { -4, -3, -2, -1,0, 1, 2, 3, 4 };    // расположим в обратном порядке  Array.Reverse(numbers);    // уменьшим массив до 4 элементов  Array.Resize(ref numbers, 4);    foreach(int number in numbers)  {      Console.Write($"{number} \t");  } |

Результат программы:

4 3 2 1

В функции Resize первым параметром является изменяемый массив, а второй - количество элементов, которые должны быть в массиве. Если второй параметр меньше длины массива, то массив усекается. Если значение параметра, наоборот, больше, то массив дополняется дополнительными элементами, которые имеют значение по умолчанию.

Метод **Copy** копирует часть одного массива в другой:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | int[] numbers = { -4, -3, -2, -1,0, 1, 2, 3, 4 };  int[] numbers2 = new int[5];    // копируем из numbers с 2-го индекса 5 элементов  // и поместим их в массив numbers2, начиная с 0-го индекса  Array.Copy(numbers, 2, numbers2, 0, 5);    foreach(int number in numbers2)  {      Console.Write($"{number} \t");  } |

Результат программы:

-2 -1 0 1 2

Отсортируем массив с помощью метода **Sort()**:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | int[] numbers = { -3, 10, 0, -5, 12, 1, 22, 3};    Array.Sort(numbers);    foreach(int number in numbers)  {      Console.Write($"{number} \t");  } |

Результат работы программы:

-5 -3 0 1 3 10 12 22

**Спецификаторы классов и методов.**

Все члены класса - поля, методы, свойства - все они имеют **модификаторы доступа**. Модификаторы доступа позволяют задать допустимую область видимости для членов класса. То есть модификаторы доступа определяют контекст, в котором можно употреблять данную переменную или метод. В предыдущих темах мы уже с ним сталкивались, когда объявляли поля класса публичными (то есть с модификатором public).

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.

Мы можем явно задать модификатор доступа, например:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | private protected class State  {      internal int a;      protected void Print()      {          Console.WriteLine($"a = {a}");      }  } |

Либо можем не указывать:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | class State  {      int a;      void Print()      {          Console.WriteLine($"a = {a}");      }  } |

Если для полей и методов не определен модификатор доступа, то по умолчанию для них применяется модификатор **private**.

Классы и структуры, объявленные без модификатора, по умолчанию имеют доступ **internal**.

Все классы и структуры, определенные напрямую в пространствах имен и не являющиеся вложенными в другие классы, могут иметь только модификаторы public или internal.

Посмотрим на примере и создадим следующий класс State:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38 | public class State  {      // все равно, что private int defaultVar;      int defaultVar;      // поле доступно только из текущего класса      private int privateVar;      // доступно из текущего класса и производных классов, которые определены в этом же проекте      protected private int protectedPrivateVar;      // доступно из текущего класса и производных классов      protected int protectedVar;      // доступно в любом месте текущего проекта      internal int internalVar;      // доступно в любом месте текущего проекта и из классов-наследников в других проектах      protected internal int protectedInternalVar;      // доступно в любом месте программы, а также для других программ и сборок      public int publicVar;        // по умолчанию имеет модификатор private      void defaultMethod() => Console.WriteLine($"defaultVar = {defaultVar}");        // метод доступен только из текущего класса      private void privateMethod() => Console.WriteLine($"privateVar = {privateVar}");        // доступен из текущего класса и производных классов, которые определены в этом же проекте      protected private void protectedPrivateMethod() => Console.WriteLine($"protectedPrivateVar = {protectedPrivateVar}");        // доступен из текущего класса и производных классов      protected void protectedMethod()=> Console.WriteLine($"protectedVar = {protectedVar}");        // доступен в любом месте текущего проекта      internal void internalMethod() => Console.WriteLine($"internalVar = {internalVar}");        // доступен в любом месте текущего проекта и из классов-наследников в других проектах      protected internal void protectedInternalMethod() => Console.WriteLine($"protectedInternalVar = {protectedInternalVar}");        // доступен в любом месте программы, а также для других программ и сборок      public void publicMethod() => Console.WriteLine($"publicVar = {publicVar}");      } |

Так как класс State объявлен с модификатором public, он будет доступен из любого места программы, а также из других программ и сборок. Класс State имеет пять полей для каждого уровня доступа. Плюс одна переменная без модификатора, которая является закрытой (private) по умолчанию.

Также имеются шесть методов, которые будут выводить значения полей класса на экран. Обратите внимание, что так как все модификаторы позволяют использовать члены класса внутри данного класса, то и все переменные класса, в том числе закрытые, у нас доступны всем его методам, так как все находятся в контексте класса State.

Теперь посмотрим, как мы сможем использовать переменные нашего класса в программе (то есть в методе Main класса Program), если классы State и Program находятся **в одном проекте**:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33 | class Program  {      static void Main(string[] args)      {          State state1 = new State();            // присвоить значение переменной defaultVar у нас не получится,          // так как она имеет модификатор private и класс Program ее не видит          // И данную строку среда подчеркнет как неправильную          state1.defaultVar = 5; //Ошибка, получить доступ нельзя            // то же самое относится и к переменной privateVar          state1.privateVar = 5; // Ошибка, получить доступ нельзя            // присвоить значение переменной protectedPrivateVar не получится,          // так как класс Program не является классом-наследником класса State          state1.protectedPrivateVar =5; // Ошибка, получить доступ нельзя            // присвоить значение переменной protectedVar тоже не получится,          // так как класс Program не является классом-наследником класса State          state1.protectedVar = 5; // Ошибка, получить доступ нельзя            // переменная internalVar с модификатором internal доступна из любого места текущего проекта          // поэтому спокойно присваиваем ей значение          state1.internalVar = 5;            // переменная protectedInternalVar так же доступна из любого места текущего проекта          state1.protectedInternalVar = 5;            // переменная publicVar общедоступна          state1.publicVar = 5;      }  } |

Таким образом, мы смогли установить только переменные internalVar, protectedInternalVar и publicVar, так как их модификаторы позволяют использовать в данном контексте.

Аналогично дело обстоит и с методами:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21 | class Program  {      static void Main(string[] args)      {          State state1 = new State();            state1.defaultMethod(); //Ошибка, получить доступ нельзя            state1.privateMethod(); // Ошибка, получить доступ нельзя            state1.protectedPrivateMethod(); // Ошибка, получить доступ нельзя            state1.protectedMethod(); // Ошибка, получить доступ нельзя            state1.internalMethod();    // норм            state1.protectedInternalMethod();  // норм            state1.publicMethod();      // норм      }  } |

Здесь нам оказались доступны только три метода: internalMethod, protectedInternalMethod, publicMethod, которые имееют соответственно модификаторы internal, protected internal, public.

Благодаря такой системе модификаторов доступа можно скрывать некоторые моменты реализации класса от других частей программы.

Несмотря на то, что модификаторы public и internal похожи по своему действию, но они имеют большое отличие. Классы и члены класса с модификатором public также будут доступны и другим программам, если данных класс поместить в динамическую библиотеку dll и потом ее использовать в этих программах.

**Список используемых источников**

1. Данные интернет – ресурса metanit.com/sharp
2. C# 7.0. Справочник. Полное описание языка. Авторы: Джозеф Албахари и Бен Албахари
3. Язык программирования C#. Классика Computers Science. 4-е издание. Авторы: А. Хейлсберг, М. Торгерсен, С. Вилтамут, П. Голд