Тема: «Использование алгоритмов и программ с методами»

Цель: ознакомление со структурой программ с применением пользовательских методов, методов массивов и строк с использованием языка программирования C#

Время выполнения: 4 часа.

ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ

Методы

Если переменные хранят некоторые значения, то методы содержат собой набор инструкций, которые выполняют определенные действия. По сути ***метод*** - это именованный блок кода, который выполняет некоторые действия.

Общее определение методов выглядит следующим образом:

```
[модификаторы] тип_возвращаемого_значения название_метода ([параметры])
{
    // тело метода
}
```

Модификаторы и параметры необязательны.

Ранее мы уже использовали как минимум один метод - Console.WriteLine(), который выводит информацию на консоль. Теперь рассмотрим, как мы можем создавать свои методы.

Определение метода

Определим один метод:

```
void SayHello()
{
    Console.WriteLine("Hello");
}
```

Здесь определен метод SayHello, который выводит некоторое сообщение. К названиям методов предъявляются в принципе те же требования, что и к названиям переменных. Однако, как правило, названия методов начинаются с большой буквы.

Перед названием метода идет возвращаемый тип данных. Здесь это тип void, который указывает, что фактически ничего не возвращает, он просто производит некоторые действия.

После названия метода в скобках идет перечисление параметров. Но в данном случае скобки пустые, что означает, что метод не принимает никаких параметров.

После списка параметров в круглых скобках идет блок кода, который представляет набор выполняемых методом инструкций. В данном случае блок метода SayHello содержит только одну инструкцию, которая выводит строку на консоль:

```
Console.WriteLine("Hello");
```

Но если мы запустим данный проект, то мы не увидим никакой строки, которую должен выводить метод SayHello. Потому что после определения метод еще надо вызвать, чтобы он выполнил свою работу.

Вызов методов

Чтобы использовать метод SayHello, нам надо его вызвать. Для вызова метода указывается его имя, после которого в скобках идут значения для его параметров (если метод принимает параметры).

название_метода (значения_для_параметров_метода);

Например, вызов метода SayHello будет выглядеть следующим образом:

```
SayHello();
```

Поскольку метод не принимает никаких параметров, то после названия метода идут пустые скобки.

Объединим определение и вызов метода:

```
void SayHello()
{
    Console.WriteLine("Hello");
}

SayHello(); // Hello
SayHello(); // Hello
```

Преимуществом методов является то, что их можно повторно и многократно вызывать в различных частях программы. Например, в примере выше два раза вызывается метод SayHello.

При этом в данном случае нет разницы, сначала определяется метод, а потом вызывается или наоборот. Например, мы могли бы написать и так:

```
SayHello(); // Hello
SayHello(); // Hello

void SayHello()
{
    Console.WriteLine("Hello");
}
```

Определим и вызовем еще несколько методов:

```
void SayHelloRu()
{
    Console.WriteLine("Привет");
}
void SayHelloEn()
{
    Console.WriteLine("Hello");
}
void SayHelloFr()
{
    Console.WriteLine("Salut");
}


string language = "en";

switch (language)
{
    case "en":
        SayHelloEn();
        break;
    case "ru":
        SayHelloRu();
        break;
    case "fr":
        SayHelloFr();
        break;
}
```

Здесь определены три метода SayHelloRu(), SayHelloEn() и SayHelloFr(), которые также имеют тип void, не принимают никаких параметров и также выводит некоторую строку на консоль. Условно говоря, они выводят приветствие на определенном языке.

В конструкции switch проверяется значение переменной language, которая условно хранит код языка, и в зависимости от ее значения вызывается определенный метод. Так, в данном случае на консоль будет выведено

Hello

Сокращенная запись методов

Если метод в качестве тела определяет только одну инструкцию, то мы можем сократить определение метода. Например, допустим у нас есть метод:

```
void SayHello()
{
    Console.WriteLine("Hello");
}
```

Мы можем его сократить следующим образом:

```
1
void SayHello() => Console.WriteLine("Hello");
```

То есть после списка параметров ставится оператор =>, после которого идет выполняемая инструкция.

Параметры методов

В прошлой теме был определен метод SayHello, который выводит на консоль некоторое сообщение:

```
void SayHello()
{
    Console.WriteLine("Hello");
}

SayHello(); // Hello
```

Но минусом подобного метода является то, что он выводит одно и то же сообщение. И было бы неплохо, если бы мы могли бы динамически определять, какое сообщение будет выводить метод на экран, то есть передать из вне в метод это сообщение. Для этого в языке C# мы можем использовать параметры.

Параметры позволяют передать в метод некоторые входные данные. Параметры определяются через заятую в скобках после названия метода в виде:

```
тип_метода имя_метода (тип_параметра1 параметр1, тип_параметра2 параметр2, ...)
{
    // действия метода
}
```

Определение параметра состоит из двух частей: сначала идет тип параметра и затем его имя.

Например, определим метод PrintMessage, который получает извне выводимое сообщение:

```
void PrintMessage(string message)
{
    Console.WriteLine(message);
}

PrintMessage("Hello work");           // Hello work
PrintMessage("Hello METANIT.COM");    // Hello METANIT.COM
PrintMessage("Hello C#");             // Hello C#
```

Здесь метод PrintMessage() принимает один параметр, который называется message и имеет тип string.

Чтобы выполнить метод, который имеет параметры, при вызове после имени метода в скобках ему передаются значения для его параметров, например:

```
PrintMessage("Hello work");
```

Здесь параметру message передается строка "Hello work". Значения, которые передаются параметрам, еще называются аргументами. То есть передаваемая строка "Hello work" в данном случае является аргументом.

Иногда можно встретить такие определения как формальные параметры и фактические параметры. Формальные параметры - это собственно параметры метода (в данном случае message), а фактические параметры - значения, которые передаются формальным параметрам. То есть фактические параметры - это и есть аргументы метода.

Определим еще один метод, который складывает два числа:

```
void Sum(int x, int y)
{
    int result = x + y;
    Console.WriteLine($"{x} + {y} = {result}");
```

```
}

Sum(10, 15);    // 10 + 15 = 25
```

Метод Sum имеет два параметра: x и y. Оба параметра представляют тип int. Поэтому при вызове данного метода нам обязательно надо передать на место этих параметров два числа. Внутри метода вычисляется сумма переданных чисел и выводится на консоль.

При вызове метода Sum значения передаются параметрам по позиции. Например, в вызове Sum(10, 15) число 10 передается параметру x, а число 15 - параметру y.

Также параметры могут использоваться в сокращеной версии метода:

```
void Sum(int x, int y) => Console.WriteLine($"{x} + {y} = { x + y }");

Sum(10, 15);    // 10 + 15 = 25
```

Передаваемые параметру значения могут представлять значения переменных или результат работы сложных выражений, которые возвращают некоторое значение:

```
void Sum(int x, int y) => Console.WriteLine($"{x} + {y} = { x + y }");

int a = 10, b = 15, c = 6;
Sum(a, b);                  // 10 + 15 = 25
Sum(3, c);                  // 3 + 6 = 9
Sum(14, 4 + c);             // 14 + 10 = 24
```

Если параметрами метода передаются значения переменных, то таким переменным должно быть присвоено значение. Например, следующая программа не скомпилируется:

```
void Sum(int x, int y)
{
    Console.WriteLine($"{x} + {y} = { x + y }");
}

int a;
int b = 15;
Sum(a, b);  // ! Ошибка
```

Соответствие параметов и аргументов по типу данных

При передаче значений параметрам важно учитывать тип параметров: между аргументами и параметрами должно быть соответствие по типу. Например:

```
void PrintPerson(string name, int age)
{
    Console.WriteLine($"Name: {name}  Age: {age}");
}

PrintPerson("Tom", 24); // Name: Tom  Age: 24
```

В данном случае первый параметр метода PrintPerson() представляет тип string, поэтому при вызове метода мы должны передать этому параметру значение типа string, то есть строку. Второй параметр представляет тип int, поэтому должны передать ему целое число, которое соответствует типу int.

```
PrintPerson("Tom", 24);
```

Также мы можем передать параметрам значения тех типов, которые автоматически могут быть преобразованы в тип параметров. Например:

```
void PrintPerson(string name, int age)
{
    Console.WriteLine($"Name: {name}  Age: {age}");
}

byte b = 37;
PrintPerson("Tom", b); // Name: Tom  Age: 37
```

Здесь параметру типа int передается значение типа byte, но комилятор может автоматически преобразовать значение типа byte к тиу int. Поэтому здесь ошибки не возникнет.

Данные других типов мы передать параметров не можем. Например, следующий вызов метода PrintPerson будет ошибочным:

```
1
PrintPerson(45, "Bob"); // Ошибка! несоответствие значений типам параметров
```

Необязательные параметры

По умолчанию при вызове метода необходимо предоставить значения для всех его параметров. Но C# также позволяет использовать необязательные параметры. Для таких параметров нам необходимо объявить

значение по умолчанию. Также следует учитывать, что после необязательных параметров все последующие параметры также должны быть необязательными:

```
void PrintPerson(string name, int age = 1, string company = "Undefined")
{
    Console.WriteLine($"Name: {name}  Age: {age}  Company: {company}");
}
```

Здесь параметры age и company являются необязательными, так как им присвоены значения. Поэтому при вызове метода мы можем не передавать для них данные:

```
void PrintPerson(string name, int age = 1, string company = "Undefined")
{
    Console.WriteLine($"Name: {name}  Age: {age}  Company: {company}");
}

PrintPerson("Tom", 37, "Microsoft");  // Name: Tom  Age: 37 Company: Microsoft
PrintPerson("Tom", 37);               // Name: Tom  Age: 37 Company: Undefined
PrintPerson("Tom");                   // Name: Tom  Age: 1 Company: Undefined
```

Консольный вывод программы:

```
Name: Tom  Age: 37  Company: Microsoft
Name: Tom  Age: 37  Company: Undefined
Name: Tom  Age: 1   Company: Undefined
```

Именованные параметры

В предыдущих примерах при вызове методов значения для параметров передавались в порядке объявления этих параметров в методе. То есть аргументы передавались параметрам по позиции. Но мы можем нарушить подобный порядок, используя именованные параметры:

```
void PrintPerson(string name, int age = 1, string company = "Undefined")
{
    Console.WriteLine($"Name: {name}  Age: {age}  Company: {company}");
}
```

```
    PrintPerson("Tom", company:"Microsoft", age: 37);  // Name: Tom  Age: 37  Company: Microsoft
    PrintPerson(age:41, name: "Bob");              // Name: Bob  Age: 41  Company: Undefined
    PrintPerson(company:"Google", name:"Sam");  // Name: Sam  Age: 1   Company: Google
```

Для передачи значений параметрам о имени при вызове метода указывается имя параметра и через двоеточие его значение: name:"Tom"

Консольный вывод программы:

```
Name: Tom  Age: 37  Company: Microsoft
Name: Bob  Age: 41  Company: Undefined
Name: Sam  Age: 1   Company: Google
```

Возвращение значения и оператор return

Метод может возвращать значение, какой-либо результат. В примере выше были определены два метода, которые имели тип void. Методы с таким типом не возвращают никакого значения. Они просто выполняют некоторые действия.

Но методы также могут возвращать некоторое значение. Для этого применяется оператор return, после которого идет возвращаемое значение:

return возвращаемое значение;

Например, определим метод, который возвращает значение типа string:

```
string GetMessage()
{
    return "Hello";
}
```

Метод GetMessage имеет тип string, следовательно, он должен возвратить строку. Поэтому в теле метода используется оператор return, после которого указана возвращаемая строка.

При этом методы, которые в качестве возвращаемого типа имеют любой тип, кроме void, обязательно должны использовать оператор return для возвращения значения. Например, следующее определение метода некорректно:

```
string GetMessage()
{
    Console.WriteLine("Hello");
}
```

Также между возвращаемым типом метода и возвращаемым значением после оператора return должно быть соответствие. Например, в следующем

случае возвращаемый тип - string, но метод возвращает число (тип int), поэтому такое определение метода некорректно:

```
string GetMessage()
{
     return 3;    // Ошибка! Метод должен возвращать строку, а не число
}
```

Результат методов, который возвращают значение, мы можем присвоить переменным или использовать иным образом в программе:

```
string GetMessage()
{
    return "Hello";
}
```

string message = GetMessage(); // получаем результат метода в переменную message

```
Console.WriteLine(message);      // Hello
```

Метод GetMessage() возвращает значение типа string. Поэтому мы можем присвоить это значение какой-нибудь переменной типа string: string message = GetMessage();

Либо даже передать в качестве значения параметру другого метода:

```
string GetMessage()
{
    return "Hello";
}
void PrintMessage(string message)
{
    Console.WriteLine(message);
}
PrintMessage(GetMessage());
```

В вызове PrintMessage(GetMessage()) сначада вызывается метод GetMessage() и его результат передается параметру message метода PrintMessage.

После оператора return также можно указывать сложные выражения или вызовы других методов, которые возвращают определенный результат. Например, определим метод, который возвращает сумму чисел:

```
int Sum(int x, int y)
{
    return x + y;
}
```

```
int result = Sum(10, 15);   // 25
Console.WriteLine(result);   // 25

Console.WriteLine(Sum(5, 6));   // 11
```

Метод Sum() имеет тип int, следовательно, он должен возвратить значение типа int - целое число. Поэтому в теле метода используется оператор return, после которого указано возвращаемое число (в данном случае результат суммы переменных x и y).

Сокращенная версия методов с результатом

Также мы можем сокращать методы, которые возвращают значение:

```
string GetMessage()
{
    return "hello";
}
```

аналогичен следующему методу:

```
string GetMessage() => "hello";
```

А метод

```
int Sum(int x, int y)
{
    return x + y;
}
```

аналогичен следующему методу:

```
1
int Sum(int x, int y) => x + y;
```

Выход из метода

Оператор return не только возвращает значение, но и производит выход из метода. Поэтому он должен определяться после остальных инструкций. Например:

```
string GetHello()
{
    return "Hello";
    Console.WriteLine("After return");
}
```

С точки зрения синтаксиса данный метод корректен, однако его инструкция Console.WriteLine("After return") не имеет смысла - она никогда

не выполнится, так как до ее выполнения оператор return возвратит значение и произведет выход из метода.

Однако мы можем использовать оператор return и в методах с типом void. В этом случае после оператора return не ставится никакого возвращаемого значения (ведь метод ничего не возвращает). Типичная ситуация - в зависимости от опеределенных условий произвести выход из метода:

```
void PrintPerson(string name, int age)
{
    if(age > 120 || age < 1)
    {
        Console.WriteLine("Недопустимый возраст");
        return;
    }
    Console.WriteLine($"Имя: {name}  Возраст: {age}");
}

PrintPerson("Tom", 37);         // Имя: Tom  Возраст: 37
PrintPerson("Dunkan", 1234);    // Недопустимый возраст
```

Здесь метод PrintPerson() в качестве параметров принимает имя и возраст пользователя. Однако в методе вначале мы проверяем, соответствует ли возраст некоторому диапазону (меньше 120 и больше 0). Если возраст находится вне этого диапазона, то выводим сообщение о недопустимом возрасте и с помощью оператора return выходим из метода. После этого метод заканчивает свою работу.

Однако если возраст корректен, то выводим информацию о пользователе на консоль.

Передача параметров по ссылке и значению. Выходные параметры

Существует два способа передачи параметров в метод в языке C#: по значению и по ссылке.

Передача параметров по значению

Наиболее простой способ передачи параметров представляет передача по значению, по сути это обычный способ передачи параметров:

```
void Increment(int n)
{
    n++;
    Console.WriteLine($"Число в методе Increment: {n}");
}

int number = 5;
```

```
    Console.WriteLine($"Число до метода Increment: {number}");
    Increment(number);
    Console.WriteLine($"Число после метода Increment: {number}");
```

Консольный вывод:

```
Число до метода Increment: 5
Число в методе Increment: 6
Число после метода Increment: 5
```

При передаче аргументов параметрам по значению параметр метода получает не саму переменную, а ее копию и далее работает с этой копией независимо от самой переменной.

Так, выше при вызове метод Increment получает копию переменной number и увеличивает значение этой копии. Поэтому в самом методе Increment мы видим, что значение параметра n увеличилось на 1, но после выполнения метода переменная number имеет прежнее значение - 5. То есть изменяется копия, а сама переменная не изменяется.

Передача параметров по ссылке и модификатор ref

При передаче параметров по ссылке перед параметрами используется модификатор ref:

```
void Increment(ref int n)
{
    n++;
    Console.WriteLine($"Число в методе Increment: {n}");
}

int number = 5;
Console.WriteLine($"Число до метода Increment: {number}");
Increment(ref number);
Console.WriteLine($"Число после метода Increment: {number}");
```

Консольный вывод:

```
Число до метода Increment: 5
Число в методе Increment: 6
Число после метода Increment: 6
```

При передаче значений параметрам по ссылке метод получает адрес переменной в памяти. И, таким образом, если в методе изменяется значение параметра, передаваемого по ссылке, то также изменяется и значение переменной, которая передается на его место..

Так, в метод Increment передается ссылка на саму переменную number в памяти. И если значение параметра n в Increment изменяется, то это приводит и к изменению переменной number, так как и параметр n и переменная number указывают на один и тот же адрес в памяти.

Обратите внимание, что модификатор ref указывается как перед параметром при объявлении метода, так и при вызове метода перед аргументом, который передается параметру.

Выходные параметры. Модификатор out

Выше мы использовали входные параметры. Но параметры могут быть также выходными. Чтобы сделать параметр выходным, перед ним ставится модификатор out:

```
void Sum(int x, int y, out int result)
{
    result = x + y;
}
```

Здесь результат возвращается не через оператор return, а через выходной параметр result. Использование в программе:

```
void Sum(int x, int y, out int result)
{
    result = x + y;
}

int number;

Sum(10, 15, out number);

Console.WriteLine(number);   // 25
```

Причем, как и в случае с ref ключевое слово out используется как при определении метода, так и при его вызове.

Также обратите внимание, что методы, использующие такие параметры, обязательно должны присваивать им определенное значение. То есть следующий код будет недопустим, так как в нем для out-параметра не указано никакого значения:

```
void Sum(int x, int y, out int result)
{
    Console.WriteLine(x + y);
}
```

Прелесть использования подобных параметров состоит в том, что по сути мы можем вернуть из метода не одно значение, а несколько. Например:

```
    void GetRectangleData(int width, int height, out int rectArea, out int rectPerimetr)
    {
            rectArea = width * height;              // площадь прямоугольника - произведение ширины на высоту
            rectPerimetr = (width + height) * 2; // периметр прямоугольника - сумма длин всех сторон
    }

    int area;
    int perimetr;

    GetRectangleData(10, 20, out area, out perimetr);

    Console.WriteLine($"Площадь прямоугольника: {area}"); // 200
    Console.WriteLine($"Периметр прямоугольника: {perimetr}"); // 60
```

Здесь у нас есть метод GetRectangleData, который получает ширину и высоту прямоугольника (параметры width и height). А два выходных параметра мы используем для подсчета площади и периметра прямоугольника.

При этом можно определять переменные, которые передаются out-параметрам в непосредственно при вызове метода. То есть мы можем сократить предыдущий пример следующим образом:

```
    void GetRectangleData(int width, int height, out int rectArea, out int rectPerimetr)
    {
        rectArea = width * height;
        rectPerimetr = (width + height) * 2;
    }

    GetRectangleData(10, 20, out int area, out int perimetr);

    Console.WriteLine($"Площадь прямоугольника: {area}"); // 200
    Console.WriteLine($"Периметр прямоугольника: {perimetr}"); // 60
```

При этом, если нам неизвестен тип значений, которые будут присвоены параметрам, то мы можем для их определения использовать оператор var:

```
    GetRectangleData(10, 20, out var area, out var perimetr);
```

```
    Console.WriteLine($"Площадь прямоугольника: {area}");
// 200
    Console.WriteLine($"Периметр прямоугольника: {perimetr}");
// 60
```

Входные параметры. Модификатор in

Кроме выходных параметров с модификатором out метод может использовать входные параметры с модификатором in. Модификатор in указывает, что данный параметр будет передаваться в метод по ссылке, однако внутри метода его значение параметра нельзя будет изменить. Например, возьмем следующий метод:

```
    void GetRectangleData(in int width, in int height, out int
rectArea, out int rectPerimetr)
    {
        //width = 25; // нельзя изменить, так как width –
входной параметр
        rectArea = width * height;
        rectPerimetr = (width + height) * 2;
    }

    int w = 10;
    int h = 20;
    GetRectangleData(w, h, out var area, out var perimetr);

    Console.WriteLine($"Площадь прямоугольника: {area}");
// 200
    Console.WriteLine($"Периметр прямоугольника: {perimetr}");
// 60
```

В данном случае через входные параметры width и height в метод передаются значения, но в самом методе мы не можем изменить значения этих параметров, так как они определены с модификатором in.

Передача по ссылке в некоторых случаях может увеличить производительность, а использование оператора in гарантирует, что значения переменных, которые передаются параметрам, нельзя будет изменить в этом методе.

Массив параметров и ключевое слово params

Во всех предыдущих примерах мы использовали постоянное число параметров. Но, используя ключевое слово params, мы можем передавать неопределенное количество параметров:

```
    void Sum(params int[]  numbers)
```

```
{
    int result = 0;
    foreach (var n in numbers)
    {
        result += n;
    }
    Console.WriteLine(result);
}

int[] nums = { 1, 2, 3, 4, 5};
Sum(nums);
Sum(1, 2, 3, 4);
Sum(1, 2, 3);
Sum();
```

Сам параметр с ключевым словом params при определении метода должен представлять одномерный массив того типа, данные которого мы собираемся использовать. При вызове метода на место параметра с модификатором params мы можем передать как отдельные значения, так и массив значений, либо вообще не передавать параметры. Количество передаваемых значений в метод неопределено, однако все эти значения должны соответствовать типу параметра с params.

Если же нам надо передать какие- то другие параметры, то они должны указываться до параметра с ключевм словом params:

```
void Sum(int initialValue, params int[]  numbers)
{
    int result = initialValue;
    foreach (var n in numbers)
    {
        result += n;
    }
    Console.WriteLine(result);
}

int[] nums = { 1, 2, 3, 4, 5};
Sum(10,  nums);    // число 10 - передается параметру initialValue
Sum(1, 2, 3, 4);
Sum(1, 2, 3);
Sum(20);
```

Здесь метод Sum имеет обязательный параметр initialValue, поэтому при вызове метода для него нужно обязательно передать значение. Поэтому первое значение при вызове метода будет передаваться этому параметру.

Однако после параметра с модификатором params мы НЕ можем указывать другие параметры. То есть следующее определение метода недопустимо:

```
//Так НЕ работает
void Sum(params int[] numbers, int initialValue)
{}
```

Массив в качестве параметра

Также этот способ передачи параметров надо отличать от передачи массива в качестве параметра:

```
void Sum(int[] numbers, int initialValue)
{
    int result = initialValue;
    foreach (var n in numbers)
    {
        result += n;
    }
    Console.WriteLine(result);
}

int[] nums = { 1, 2, 3, 4, 5};
Sum(nums, 10);
```

```
// Sum(1, 2, 3, 4);     // так нельзя - нам надо передать массив
```

Так как метод Sum принимает в качестве параметра массив без ключевого слова params, то при его вызове нам обязательно надо передать в качестве первого параметра массив. Кроме того, в отличие от метода с параметром params после параметра-массива могут располагаться другие параметры.

Методы строк

Конкатенация строк или объединение может производиться как с помощью операции +, так и с помощью метода **Concat**:

```
string s1 = "hello";
string s2 = "world";
string s3 = s1 + " " + s2; // результат: строка "hello world"
string s4 = string.Concat(s3, "!!!"); // результат: строка "hello world!!!"
```

Для сравнения строк применяется статический метод **Compare**:

Данный метод принимает две строки и возвращает число. Если первая строка по алфавиту стоит выше второй, то возвращается число меньше нуля. В противном случае возвращается число больше нуля. И третий случай - если строки равны, то возвращается число 0. Например:

```
string s1 = "hello";
string s2 = "world";

int result = string.Compare(s1, s2);
if (result<0)
{
    Console.WriteLine("Строка s1 перед строкой s2");
}
else if (result > 0)
{
    Console.WriteLine("Строка s1 стоит после строки s2");
}
else
{
    Console.WriteLine("Строки s1 и s2 идентичны");
}
// результатом будет "Строка s1 перед строкой s2"
```

С помощью метода **IndexOf** мы можем определить индекс первого вхождения отдельного символа или подстроки в строке:

```
string s1 = "hello world";
char ch = 'o';
int indexOfChar = s1.IndexOf(ch); // равно 4
Console.WriteLine(indexOfChar);

string substring = "wor";
int indexOfSubstring = s1.IndexOf(substring); // равно 6
```

Подобным образом действует метод **LastIndexOf**, только находит индекс последнего вхождения символа или подстроки в строку.

Еще одна группа методов позволяет узнать начинается или заканчивается ли строка на определенную подстроку. Для этого предназначены методы StartsWith и EndsWith. Например, в массиве строк хранится список файлов, и нам надо вывести все файлы с расширением exe:

```
var files = new string[]
{
    "myapp.exe",
    "forest.jpg",
    "main.exe",
    "book.pdf",
    "river.png"
};

for (int i = 0; i < files.Length; i++)
{
    if (files[i].EndsWith(".exe"))
        Console.WriteLine(files[i]);
}
```

С помощью метода **Split** мы можем разделить строку на массив подстрок. В качестве параметра метод **Split** принимает массив символов или строк, которые и будут служить разделителями. Например, подсчитаем количество слов в сроке, разделив ее по пробельным символам:

```
string text = "И поэтому все так произошло";

string[] words = text.Split(new char[] { ' ' });

foreach (string s in words)
{
    Console.WriteLine(s);
}
```

Для обрезки начальных или концевых символов используется метод **Trim**:

```
string text = " hello world ";

text = text.Trim(); // результат "hello world"
text = text.Trim(new char[] { 'd', 'h' }); // результат "ello worl"
```

Обрезать определенную часть строки позволяет метод **Substring**:

```
string text = "Хороший день";
// обрезаем начиная с третьего символа
text = text.Substring(2);
// результат "роший день"
Console.WriteLine(text);
```

```
// обрезаем сначала до последних двух символов
text = text.Substring(0, text.Length - 2);
// результат "роший де"
 Console.WriteLine(text);
```

Для вставки одной строки в другую применяется метод **Insert:**

```
string text = "Хороший день";
string substring = "замечательный ";

text = text.Insert(8, substring);
Console.WriteLine(text);    // Хороший замечательный день
```

Удалить часть строки помогает метод **Remove:**

```
string text = "Хороший день";
// индекс последнего символа
int ind = text.Length - 1;
// вырезаем последний символ
text = text.Remove(ind);
Console.WriteLine(text);    // Хороший ден

// вырезаем первые два символа
text = text.Remove(0, 2);
Console.WriteLine(text);    // роший ден
```

Чтобы заменить один символ или подстроку на другую, применяется метод **Replace:**

```
string text = "хороший день";

text = text.Replace("хороший", "плохой");
Console.WriteLine(text);    // плохой день

text = text.Replace("о", "");
Console.WriteLine(text);    // плхй день
```

Для приведения строки к верхнему и нижнему регистру используются соответственно методы **ToUpper()** и **ToLower():**

```
string hello = "Hello world!";

Console.WriteLine(hello.ToLower()); // hello world!
Console.WriteLine(hello.ToUpper()); // HELLO WORLD!
```

Методы массивов

Поиск индекса элемента может осуществляться при помощи различных методов, например:

```
    string[] people = { "Tom", "Sam", "Bob", "Kate", "Tom", "Alice" };

    // находим индекс элемента "Bob"
    int bobIndex = Array.BinarySearch(people, "Bob");
    // находим индекс первого элемента "Tom"
    int tomFirstIndex = Array.IndexOf(people, "Tom");
    // находим индекс последнего элемента "Tom"
    int tomLastIndex = Array.LastIndexOf(people, "Tom");
    // находим индекс первого элемента, у которого длина строки больше 3
    int lengthFirstIndex = Array.FindIndex(people, person => person.Length > 3);
    // находим индекс последнего элемента, у которого длина строки больше 3
    int lengthLastIndex = Array.FindLastIndex(people, person => person.Length > 3);

    Console.WriteLine($"bobIndex:                {bobIndex}");  // 2
    Console.WriteLine($"tomFirstIndex:       {tomFirstIndex}");  // 0
    Console.WriteLine($"tomLastIndex:         {tomLastIndex}");  // 4
    Console.WriteLine($"lengthFirstIndex:  {lengthFirstIndex}");  // 3
    Console.WriteLine($"lengthLastIndex:    {lengthLastIndex}");  // 5
```

Если элемент не найден в массиве, то методы возвращают -1.

Поиск элемента по условию:

```
    string[] people =  { "Tom", "Sam", "Bob", "Kate", "Tom", "Alice" };

    // находим первый и последний элементы
    // где длина строки больше 3 символов
```

```
    string? first = Array.Find(people, person => person.Length
> 3);
    Console.WriteLine(first); // Kate
    string?   last   =   Array.FindLast(people,   person   =>
person.Length > 3);
    Console.WriteLine(last); // Alice

    // находим элементы, у которых длина строки равна 3
    string[]   group   =   Array.FindAll(people,   person   =>
person.Length == 3);
    foreach (var person in group) Console.WriteLine(person);
    // Tom Sam Bob Tom

    Изменение порядка элементов массива:

    string[] people =  { "Tom", "Sam", "Bob", "Kate", "Tom",
"Alice" };

    Array.Reverse(people);

    foreach (var person in people)
        Console.Write($"{person} ");
    // "Alice", "Tom", "Kate", "Bob", "Sam", "Tom"

    Также можно изменить порядок только части элементов:

    string[] people =  { "Tom", "Sam", "Bob", "Kate", "Tom",
"Alice" };

    // изменяем порядок 3 элементов начиная с индекса 1
    Array.Reverse(people, 1, 3);

    foreach (var person in people)
        Console.Write($"{person} ");
    // "Tom", "Kate", "Bob", "Sam", "Tom", "Alice"
```

Для изменения размера массива применяется метод **Resize**. Его первый параметр - изменяемый массив, а второй параметр - 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. Причем первый параметр передается по ссылке:

```
    string[] people =  { "Tom", "Sam", "Bob", "Kate", "Tom",
"Alice" };
```

```
// уменьшим массив до 4 элементов
Array.Resize(ref people, 4);

foreach (var person in people)
    Console.Write($"{person} ");
// "Tom", "Sam", "Bob", "Kate"
```

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

```
string[] people =  { "Tom", "Sam", "Bob", "Kate", "Tom", "Alice" };

var employees = new string[3];

// копируем 3 элемента из массива people с индекса 1
// и вставляем их в массив employees начиная с индекса 0
Array.Copy(people,1, employees,0, 3);

foreach (var person in employees)
    Console.Write($"{person} ");
// Sam Bob Kate
```

Для сортировки массива используется метод **Sort()**

Индивидуальные задания для лабораторной работы

1.Написать метод, который принимает строку. В качестве возвращаемого значения должна быть строка, из которой удалены все гласные.

2. Написать метод, который принимает строку любой длины и содержания. Данная строка это пинкод. Метод возвращает true если пинкод валидный и false иначе. Пинкод может содержать только 4 или 6 цифр.

3. Написать метод, который принимает строку. В качестве возвращаемого значения должна быть строка, из которой удалены все согласные.

3. Написать метод, который принимает любое количество целочисленных аргументов. В случае если количество четных аргументов больше возвращает их сумму, если количество нечетных аргументов больше возвращает их произведение, если количество четных и нечетных аргументов одинаково возвращает 0.

4. Есть купюры номиналом 1р., 5р., 10р., 20р., и 50р. Написать метод, который принимает число и выводит минимальное количество купюр, которые необходимы для составления данного числа.

5. Написать метод, который принимает в качестве аргумента строку вида «фамилия и.о.». Любая буква в данной строке может быть как заглавной, так и строчной, ваша задача вернуть строку, приведенную к виду «Фамилия И.О.»

6. Написать метод, который принимает в качестве аргумента массив, состоящий из массивов разной длины. Метод должен вернуть индекс самого длинного подмассива.

7. Написать метод, который принимает в качестве аргумента массив, состоящий из массивов разной длины. Метод должен вернуть индекс подмассива сумма элементов которого максимальна.

8. Написать метод, который принимает массив строк и букву. Метод должен вернуть новый массив, который состоит из строк из первого массива, в которых встречается данная буква.

9. Написать метод, который принимает строку и число n. Возвращать метод должен новую строку, полученную путем сдвига каждой буквы исходной строки на n. (шифр цезаря).

10. В небольшом городе численность населения на начало года равна 1000 человек. Население регулярно увеличивается на 2 процента в год, более того, в город приезжает 50 новых жителей в год. Написать метод, который посчитает сколько лет понадобится городу, чтобы его население превысило или стало равным 2000 жителей.

11. Написать метод, который на вход принимает массив из нулей и единиц, которые представляют собой число в двоичной системе исчисления. Вернуть метод должен число в десятичной системе.

12. Написать метод, который на вход принимает любое неотрицательное целое число. Вернуть метод должен максимально возможное число, составленное из цифр принятого числа.

13. Написать метод, который на вход принимает строку, состоящую из букв, цифр, пробелов. Вернуть метод должен отсортированную строку (сначала цифры, потом буквы), из которой были удалены все пробелы.

14. Написать метод, который на вход принимает два целых числа (числа могут быть как положительными, так и отрицательными). Вернуть метод должен сумму всех целых чисел, стоящих между ними (с учетом знаков, стоящих при числе).

15. Написать метод, который на вход принимает строку, состоящую из букв. Вернуть данный метод должен строку, в которой каждая буква заменена ее позицией в алфавите.

ПОРЯДОК ВЫПОЛНЕНИЯ РАБОТЫ

1. Изучить теоретическую часть лабораторной работы.

2. Реализовать индивидуальное задание согласно варианту, сделать скриншоты работающих программ. Написать комментарии.

3. Написать отчет, содержащий:

1. Титульный лист, на котором указывается:

а) полное наименование министерства образование и название учебного заведения;

б) название дисциплины;

в) номер практического занятия;

г) фамилия преподавателя, ведущего занятие;

д) фамилия, имя и номер группы студента;

е) год выполнения лабораторной работы.

2. Индивидуальное задание (листинг кода программы, скриншоты консоли с результатами работы).

3. Вывод о проделанной работе.