**Лекция 3:**

# **Методы: основные понятия**

### *Методы: основные понятия. Перегрузка методов.*

#### Методы: основные понятия

Метод – это функциональный элемент класса, который реализует вычисления или другие действия, выполняемые классом или его экземпляром (объектом). Метод представляет собой законченный фрагмент кода, к которому можно обратиться по имени. Он описывается один раз, а вызываться может многократно. Совокупность методов класса определяет, что конкретно может делать класс. Например, стандартный класс Math содержит методы, которые позволяют вычислять значения математических функций.

Синтаксис метода:

[атрибуты] [спецификаторы] тип\_возвращаемого\_результата имя\_метода ([список\_параметров])

{

тело\_метода;

return значение; // если тип возвращаемого значения не void

}

где:

1. Атрибуты и спецификаторы являются необязательными элементами синтаксиса описания метода. На данном этапе атрибуты нами использоваться не будут, а из всех спецификаторов мы в обязательном порядке будем использовать спецификатор static, который позволит обращаться к методу класса без создания его экземпляра.

**Замечание**. Остальные спецификаторы мы рассмотрим в разделе "классы".

1. *Тип\_возвращаемого\_результата* определяет тип значения, возвращаемого методом. Это может быть любой тип, включая типы классов, создаваемые программистом. Если метод не возвращает никакого значения, необходимо указать тип void (в этом случае в теле метода отсутсвует оператор return ).
2. *Имя\_метода* – идентификатор, заданный программистом с учетом требований, накладываемыми на идентификаторы в С#, отличный от тех, которые уже использованы для других элементов программы в пределах текущей области видимости.
3. *Список\_параметров* представляет собой последовательность пар, состоящих из типа данных и идентификатора, разделенных запятыми. Параметры (формальные переменные) — это переменные или константы, которые получают значения, передаваемые методу при вызове. Если метод не имеет параметров, то *список\_параметров* остается пустым, но скобки () указываются обязательно.
4. *Значение* определяет значение, возвращаемое методом. Тип значения должен соответствовать *типу\_возвращаемого\_результата* или приводится к нему.

Рассмотрим простейший пример метода:

class Program

{

static void Func() //дополнительный метод

{

Console.Write("x= ");

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

double y = 1 / x;

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

}

static void Main() //точка входа в программу

{

Func(); //первый вызов метода Func

Func(); //второй вызов метода Func

}

}

Рассмотрим другой пример:

class Program

{

static int Func( int x, int y)

{

return (x>y)? x:y;

}

static void Main()

{

Console.Write("a=");

int a = int.Parse(Console.ReadLine());

Console.Write("b=");

int b = int.Parse(Console.ReadLine());

Console.Write("c=");

int c = int.Parse(Console.ReadLine());

int max = Func(Func(a, b), c);

Console.WriteLine("max({0}, {1}, {2})={3}", a, b, c, max);

}

}

В данном примере метод Func имеет два целочисленных параметра – x, y, а в качестве результата метод возвращает наибольшее из них. На этапе описания метода указываются *формальные* параметры, на этапе вызова в метод передаются *фактические* параметры, которые по количеству и по типу совпадают с формальными параметрами. Если количество фактических и формальных параметров будет различным, то компилятор выдаст соответствующее сообщение об ошибке. Если параметры будут отличаться типами, то компилятор попытается выполнить *неявное преобразование типов*. Если *неявное преобразование* невозможно, то также будет сгенерирована ошибка.

Обратите внимание на то, что при вызове метода Func использовалось вложение одного вызова в другой.

В общем случае параметры используются для обмена информацией между вызывающим и вызываемым методами. В С# для обмена предусмотрено четыре типа параметров: параметры-значения, параметры-ссылки, выходные параметры, параметры-массивы.

При передаче параметра *по значению* метод получает копии параметров, и операторы метода работают с этими копиями. Доступа к исходным значениям параметров у метода нет, а, следовательно, нет и возможности их изменить.

**Замечание**. Все примеры, рассмотренные ранее, использовали передачу данных по значению.

При передаче параметров *по ссылке* метод получает копии адресов параметров, что позволяет осуществлять доступ к ячейкам памяти по этим адресам и изменять исходные значения параметров. Для того чтобы параметр передавался по ссылке, необходимо при описании метода перед формальным параметром и при вызове метода перед соответствующим фактическим параметром поставить служебное слово ref.

class Program

{

static void Func(int x, ref int y)

{

x += 10; y += 10; //изменение параметров

Console.WriteLine("In Func: {0}, {1}", x, y);

}

static void Main()

{

int a=10, b=10; // строка 1

Console.WriteLine("In Main: {0}, {1}", a, b);

Func(a, ref b);

Console.WriteLine("In Main: {0}, {1}", a, b);

}

}

*Результат работы программы*:

In Main: 10 10

In Func: 20 20

In Main: 10 20

В данном примере в методе Func были изменены значения формальных параметров х и y. Эти изменения не отразились на фактическом параметре а, т.к. он передавался по значению, но значение b было изменено, т.к. он передавался по ссылке.

Передача параметра по ссылке требует, чтобы аргумент был инициализирован до вызова метода (см. строку 1). Если в этой строке не проводить инициализацию переменных, то компилятор выдаст сообщение об ошибке.

Однако в некоторых случаях бывает невозможно инициализировать параметр до вызова метода. Тогда параметр следует передавать как выходной, используя спецификатор out.

class Program

{

static void Func(int x, out int y)

{

x += 10; y = 10; // определение значения выходного параметра y

Console.WriteLine("In Func: {0}, {1}", x, y);

}

static void Main()

{

int a=10, b;

Console.WriteLine("In Main: {0}", a);

Func(a, out b);

Console.WriteLine("In Main: {0}, {1}", a, b);

}

}

*Результат работы программы*:

In Main: 10

In Func: 20 10

In Main: 10 10

В данном примере в методе Func формальный параметр y и соответствующий ему фактический параметр b метода Main были помечены спецификатором out. Поэтому значение b до вызова метода Func можно было не определять, но изменение параметра y отразились на изменении значения параметра b.

**Замечание**. Параметры-массивы будут рассмотрены позже.

#### Перегрузка методов

Иногда бывает удобно, чтобы методы, реализующие один и тот же алгоритм для различных типов данных, имели одно и то же имя. Использование нескольких методов с одним и тем же именем, но различными типами и количеством параметров называется *перегрузкой методов*. Компилятор определяет, какой именно метод требуется вызвать, по типу и количеству фактических параметров.

Рассмотрим следующий пример:

class Program

{

static int max(int a) //первая версия метода max

{

int b = 0;

while (a > 0)

{

if (a % 10 > b) b = a % 10;

a /= 10;

}

return b;

}

static int max(int a, int b) //вторая версия метода max

{

if (a > b) return a;

else return b;

}

static int max(int a, int b, int c) //третья версия метода max

{

if (a > b && a > c) return a;

else if (b > c) return b;

else return c;

}

static void Main()

{

int a = 1283, b = 45, c = 35740;

Console.WriteLine(max(a));

Console.WriteLine(max(a, b));

Console.WriteLine(max(a, b, c));

}

}

При вызове метода max компилятор выбирает вариант, соответствующий типу и количеству передаваемых в метод аргументов. Если точного соответствия не найдено, выполняются *неявные преобразования типов* в соответствии с общими правилами. Если преобразование невозможно, выдается сообщение об ошибке. Если выбор *перегруженного метода* возможен более чем одним способом, то выбирается "лучший" из вариантов (вариант, содержащий меньшие количество и длину преобразований в соответствии с правилами преобразования типов). Если существует несколько вариантов, из которых невозможно выбрать лучший, выдается сообщение об ошибке.

*Перегрузка методов* является проявлением *полиморфизма*, одного из основных свойств ООП. Программисту гораздо удобнее помнить одно имя метода и использовать его для работы с различными типами данных, а решение о том, какой вариант метода вызвать, возложить на компилятор. Этот принцип широко используется в классах библиотеки .NET. Например, в стандартном классе Console метод WriteLine перегружен 19 раз для вывода величин разных типов.

**Примеры:**

* 1. Разработать метод min(a,b) для нахождения минимального из двух чисел. Вычислить с помощью него значение выражения z=min(3x,2y)+min(x-y,x+y).

using System;

namespace Hello

{

class Program

{

static double min(double a, double b)

{

return (a < b) ? a : b;

}

static void Main(string[] args)

{

Console.Write("x=");

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

Console.Write("y=");

double y = double.Parse(Console.ReadLine());

double z = min(3 \* x, 2 \* y) + min(x - y, x + y);

Console.WriteLine("z=" + z);

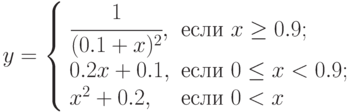
}

}

}

1. Постройте таблицу значений функции y=f(x)  для х \in [a, b] с шагом h.

*Замечание*. Для решения задачи использовать вспомогательный метод.

* 1. 

**Пример**:

class Program

{

static double f (double x)

{

double y;

if (x >= 0.9) y = 1 / Math.Pow(1 + x, 2);

else if (x >= 0) y = 0.2 \* x + 0.1;

else y = x \* x + 0.2;

return y;

}

static void Main(string[] args)

{

Console.Write("a=");

double a = double.Parse(Console.ReadLine());

Console.Write("b=");

double b = double.Parse(Console.ReadLine());

Console.Write("h=");

double h = double.Parse(Console.ReadLine());

for (double i = a; i <= b; i += h)

Console.WriteLine("f({0:f2})={1:f4}", i, f(i));

}

}

**Рекурсивные методы**

Рекурсивным называют метод, если он вызывает сам себя в качестве вспомогательного. Классическим примером *рекурсивного метода* является метод, вычисляющий *факториал*.

Из курса математики известно, что 0!=1!=1, n!=1\*2\*3…\*n. С другой стороны n!=(n-1)!\*n. Таким образом, известны два частных случая параметра n, а именно n= 0 и n=1, при которых мы без каких-либо дополнительных вычислений можем определить *значение* факториала. Во всех остальных случаях, то есть для n>1, *значение* факториала может быть вычислено через *значение* факториала для параметра n-1. Таким образом, *рекурсивный метод* будет иметь вид:

{

static long F(int n) //рекурсивный метод

{

if (n==0 || n==1)

return 1; //нерекурсивная ветвь

else return n\*F(n-1); //шаг рекурсии - повторный вызов метода с другим параметром

}

static void Main()

{

Console.Write("n=");

int n =int.Parse( Console.ReadLine());

long f=F(n); //нерекурсивный вызов метода F

Console.WriteLine("{0}!={1}",n, f);

}

}

Рассмотрим работу описанного выше *рекурсивного метода* для n=3.

Изображение выглядит как текст

Автоматически созданное описание

Первый *вызов метода* осуществляется из метода Main, в нашем случае командой f=F(3). Этап вхождения в рекурсию обозначим жирными стрелками. Он продолжается до тех пор, пока *значение* переменной n не становится равной 1. После этого начинается *выход* из рекурсии (тонкие стрелки). В результате вычислений получается, что F(3)=3\*2\*1.

Рассмотренный вид рекурсии называют *прямой*. Метод с *прямой* рекурсией обычно содержит следующую структуру:

if (<условие>)

<оператор>;

else <вызов данного метода с другими параметрами>;

В качестве <условия> обычно записываются некоторые граничные случаи параметров, передаваемых рекурсивному методу, при которых результат его работы заранее известен, поэтому далее следует простой оператор или блок, а в ветви else происходит *рекурсивный вызов* данного метода с другими параметрами.

Что необходимо знать для реализации рекурсивного процесса? Со входом в рекурсию осуществляется *вызов метода*, а для выхода необходимо помнить точку возврата, т.е. то *место* программы откуда мы пришли и куда нам нужно будет возвратиться после завершения метода. *Место* хранения точек возврата называется стеком вызовов и для него выделяется определенная область оперативной памяти. В этом стеке запоминаются не только адреса точек возврата, но и копии значений всех параметров. *По* этим копиям восстанавливается при возврате вызывающий метод. При развертывании рекурсии за счет создания копий параметров возможно *переполнение* стека. Это является основным недостатком *рекурсивного метода*. С другой стороны, *рекурсивные методы* позволяют перейти к более компактной записи алгоритма.

Следует понимать, что любой *рекурсивный метод* можно преобразовать в обычный метод. И практически любой итерационный метод можно преобразовать в рекурсивный, если выявить *рекуррентное соотношение* между вычисляемыми в методе значениями.

Далее для сравнения приведены решения с использованием обычного и рекурсивного методов:

**Пример 1**: Найти сумму цифр числа А.

Известно, что любое *натуральное число* A=an an-1... a1 a0, где an an-1... a1 a0 - цифры числа, можно представить следующим образом:

A=an an-1... a1 a0 = A=an\*10n + an-1\*10n-1 + ... a1\*101 + a0\*100 = ((...((an\*10 + an-1)\*10+ an-2)\*10...)\*10 + a1)\*10 + a0

Например, число 1234 можно представить как:

1234 = 1\*103+ 2\*102+ 3\*101+ 4\*100= ((1\*10 + 2)\*10 + 3)\*10 + 4

Из данного представления видно, что получить последнюю цифру можно, если найти *остаток от деления* числа на 10. В связи с этим для разложения числа на составляющие его цифры можно использовать следующий *алгоритм*:

1. Находим остаток при делении числа А на 10, т.е. получаем крайнюю правую цифру числа.
2. Находим целую часть числа при делении A на 10, т.е. отбрасываем от числа A крайнюю правую цифру.
3. Если преобразованное A > 0, то переходим на пункт 1. Иначе число равно нулю и отделять от него больше нечего.

Данный *алгоритм* будет использоваться при разработке нерекурсивного метода.

С другой стороны, сумму цифр числа 1234 можно представить следующим образом sum(1234)=sum(123)+4=(sum(12)+3)+4=(((sum(1)+2)+3)+4)=(((sum(0)+1)+2)+3)+4. Таким образом, если А=0, то сумма цифр числа также равна нулю, т.е. sum=0. В противном случае сумму цифр числа A можно представить *рекуррентным соотношением* sum(A)=sum(A/10)+A%10. Полученное *рекуррентное соотношение* будем использовать при разработке *рекурсивного метода*.

class Program

{

static long Sum(long a) //нерекурсивный метод

{

long sum=0;

while (a>0) //пока a больше нуля

{

sum+=a%10; //добавляем к сумме последнюю цифру числа а

a/=10; //отбрасываем от числа а последнюю цифру

}

return sum; //возвращаем в качестве результата сумму цифр числа a

}

static long SumR(long a) //рекурсивный метод

{

if (a==0) //если a =0, то

return 0; // возвращаем 0

else return SumR(a/10)+ a%10; //иначе обращаемся к рекуррентному соотношению

}

static void Main()

{

Console.Write("n=");

long n=long.Parse(Console.ReadLine());

Console.WriteLine("Нерекурсивный метод: "+Sum(n));

Console.WriteLine("Рекурсивный метод: "+SumR(n));

}

}

}

**Пример 2**: вычислить n-ный член последовательности Фиббоначи.

Первые два члена последовательности Фиббоначи равны 1, остальные получаются *по* рекуррентной формуле an=an-1+an-2.

class Program

{

static int Fb(int n) //нерекурсивный алгоритм

{

int a, a1=1, a2=1;

if (n==1||n==2) return 1;

else

{

for (int i=2; i<=n; ++i)

{

a=a1+a2;

a1=a2;

a2=a;

}

return a1;

}

}

static int FbR(int n) //рекурсивный алгоритм

{

if (n==1 || n==2 )return 1;

else return FbR(n-1)+FbR(n-2);

}

static void Main()

{

Console.Write("n=");

int n=int.Parse(Console.ReadLine());

Console.WriteLine("Нерекурсивный метод: "+Fb(n));

Console.WriteLine("Рекурсивный метод: "+FbR(n));

}

}

# Массивы

*Массив* - набор элементов одного и того же типа, объединенных общим именем. Массивы в С# можно использовать по аналогии с тем, как они используются в других языках программирования. Однако С#-массивы имеют существенные отличия: они относятся к *ссылочным типам данных*, более того - реализованы как объекты. Фактически имя массива является ссылкой на область кучи (динамической памяти), в которой последовательно размещается набор элементов определенного типа. Выделение памяти под элементы происходит на этапе инициализации массива. А за освобождением памяти следит система сборки мусора - неиспользуемые массивы автоматически утилизируются данной системой.

Рассмотрим различные типы массивов.

#### Одномерные массивы

Одномерный массив — это фиксированное количество элементов одного и того же типа, объединенных общим именем, где каждый элемент имеет свой номер (индекс). Нумерация элементов массива в С# начинается с нуля, то есть, если массив состоит из 10 элементов, то его элементы будут иметь следующие номера: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9.

Одномерный массив в С# реализуется как объект, поэтому его создание представляет собой двухступенчатый процесс. Сначала объявляется ссылочная переменная на массив, затем выделяется память под требуемое количество элементов базового типа, и ссылочной переменной присваивается адрес нулевого элемента в массиве. Базовый тип определяет тип данных каждого элемента массива. Количество элементов, которые будут храниться в массиве, определяется размер массива.

В общем случае процесс объявления переменной типа массив, и выделение необходимого объема памяти может быть разделено. Кроме того, на этапе объявления массива можно произвести его инициализацию. Поэтому для объявления одномерного массива может использоваться одна из следующих форм записи:

|  |  |
| --- | --- |
| **Форма записи** | **Пояснения** |
| базовый\_тип [] имя\_\_массива;  *Например*:  int [] a; | Описана ссылка на одномерный массив, которая в дальнейшем может быть использована:   1. для адресации на уже существующий массив; 2. передачи массива в метод в качестве параметра 3. отсроченного выделения памяти под элементы массива. |
| базовый\_тип [] имя\_\_массива = new базовый\_тип [размер];  *Например*:  int []a=new int [10]; | Объявлен одномерный массив заданного типа и выделена память под одномерный массив указанной размерности. Адрес данной области памяти записан в ссылочную переменную. Элементы массива равны нулю.  **Замечание**. Надо отметить, что в C# элементам массива присваиваются начальные значения по умолчанию в зависимости от базового типа. Для арифметических типов - нули, для ссылочных типов - null, для символов - пробел. |
| базовый\_тип [] имя\_\_массива={список инициализации};  *Например*:  int []a={0, 1, 2, 3}; | Выделена память под одномерный массив, размерность которого соответствует количеству элементов в списке инициализации. Адрес этой области памяти записан в ссылочную переменную. Значение элементов массива соответствует списку инициализации. |

Обращения к элементам массива происходит с помощью индекса, для этого нужно указать имя массива и в квадратных скобках его номер. Например, a[0], b[10], c[i].

Так как массив представляет собой набор элементов, объединенных общим именем, то обработка массива обычно производится в цикле. Рассмотрим несколько простых примеров работы с одномерными массивами.

**Пример 1**.

static void Main()

{

int[] myArray = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };

int i;

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

Console.WriteLine(myArray[i]);

}

**Пример 2**.

static void Main()

{

int[] myArray = new int[10];

int i;

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

myArray[i] = i \* i;

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

Console.WriteLine(myArray[i]);

}

##### Массив как параметр

Так как имя массива фактически является ссылкой, то он передается в метод по ссылке и, следовательно, все изменения элементов массива, являющегося формальным параметром, отразятся на элементах соответствующего массива, являющимся фактическим параметром.

Рассмотрим пример передачи массива как параметра:

class Program

{

static void Print(int n, int[] a) //n - размерность массива, а - ссылка на массив

{

for (int i = 0; i < n; i++) Console.Write("{0} ", a[i]);

Console.WriteLine();

}

static void Change(int n, int[] a)

{

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

if (a[i] > 0) a[i] = 0; // изменяются элементы массива

}

static void Main()

{

int[] myArray = { 0, -1, -2, 3, 4, 5, -6, -7, 8, -9 };

Print(10, myArray);

Change(10, myArray);

Print(10, myArray);

}

}

##### Массив как объект

Мы уже говорили о том, что массивы в С# реализованы как объекты. Если говорить более точно, то они реализованы на основе базового класса Array, определенного в пространстве имен System. Данный класс содержит различные свойства и методы. Например, свойство Length позволяет определять количество элементов в массиве. Преобразуем предыдущий пример:

class Program

{

static void Print(int[] a) // передаем только ссылку на массив

{

for (int i = 0; i < a.Length; i++) Console.Write("{0} ", a[i]);

Console.WriteLine();

}

static void Change(int[] a)

{

for (int i = 0; i < a.Length; i++)

if (a[i] > 0) a[i] = 0;

}

static void Main()

{

int[] myArray = { 0, -1, -2, 3, 4, 5, -6, -7, 8, -9 };

Print(myArray);

Change(myArray);

Print(myArray);

}

}

Другие свойства и методы класса Array приведены в следующей таблице:

|  |  |  |
| --- | --- | --- |
| **Элемент** | **Вид** | **Описание** |
| Length | свойство | Количество элементов массива (по всем размерностям) |
| *BinarySearch* | статический метод | Двоичный поиск в отсортированном массиве |
| Clear | статический метод | Присваивание элементам массива значений по умолчанию |
| Copy | статический метод | Копирование заданного диапазона элементов одного массива в другой |
| CopyTo | *экземплярный метод* | Копирование всех элементов текущего одномерного массива в другой массив |
| GetValue | *экземплярный метод* | Получение значения элемента массива |
| IndexOf | статический метод | Поиск первого вхождения элемента в одномерный массив |
| *LastIndexOf* | статический метод | Поиск последнего вхождения элемента в одномерный массив |
| Reverse | статический метод | Изменение порядка следования элементов на обратный |
| SetValue | *экземплярный метод* | Установка значения элемента массива |
| Sort | статический метод | Упорядочивание элементов одномерного массива |

Вызов статических методов происходит через обращение к имени класса, например, Array.Sort(myArray). В данном случае мы обращаемся к статическому методу Sort класса Array и передаем данному методу в качестве параметра объект myArray - экземпляр класса Array.

Обращение к свойству или вызов *экземплярного метода* производится через обращение к экземпляру класса, например, myArray.Length или myArray.GetValue(i).

#### **Многомерные массивы**

Многомерные массивы имеют более одного измерения (индекса). Чаще всего используются двумерные массивы, которые представляют собой таблицы. Каждый элемент массива имеет два индекса, первый определяет номер строки, второй - номер столбца, на пересечении которых находится элемент. Нумерация строк и столбцов начинается с нуля.

Объявить двумерный массив можно одним из предложенных способов:

тип [,] имя\_\_массива;

тип [,] имя\_\_массива = new тип [размер1, размер2];

тип [,] имя\_\_массива={{элементы 1-ой строки}, … , {элементы n-ой строки}};

тип [,] имя\_\_массива= new тип [,]{{элементы 1-ой строки}, … ,{элементы n-ой строки}};

строки}};

Например:

int [,] a;

int [,] a= new int [3, 4];

int [,] a={{0, 1, 2}, {3, 4, 5}};

int [,] a= new int [,]{{0, 1, 2}, {3, 4, 5}};

**Замечания**.

1. Как и в случае с одномерными массивами, последние два описания являются избыточными.
2. При работе с многомерными массивами можно использовать приемы, которые мы рассмотрели для одномерных массивов.
3. При обращении к свойству Length для двумерного массива мы получим общее количество элементов в массиве. Чтобы получить количество строк нужно обратиться к методу GetLength с параметром 0. Чтобы получить количество столбцов - к методу GetLength с параметром 1.

**Пример**:

class Program

{

static void PrintArray(string a, int[,] mas)

{

Console.WriteLine(a);

for (int i = 0; i < mas.GetLength(0); i++)

{

for (int j = 0; j < mas.GetLength(1); j++)

Console.Write("{0} ", mas[i, j]);

Console.WriteLine();

}

}

static void Change(int[,] mas)

{

for (int i = 0; i < mas.GetLength(0); i++)

for (int j = 0; j < mas.GetLength(1); j++)

if (mas[i, j] % 2 == 0) mas[i, j] = 0;

}

static void Main()

{

try

{

int[,] MyArray ={ { 1, 2, 3 }, { 4, 5, 6 }, { 7, 8, 9 } };

PrintArray("исходный массив:", MyArray);

Change(MyArray);

PrintArray("итоговый массив", MyArray);

}

catch (FormatException)

{

Console.WriteLine("неверный формат ввода данных");

}

catch (OverflowException)

{

Console.WriteLine("переполнение");

}

catch (OutOfMemoryException)

{

Console.WriteLine("недостаточно памяти для создания нового объекта");

}

}

}

Объявление ступенчатого массива:

тип [][] имя\_массива;

Например:

int [][]a;

Фактически мы объявили одномерный массив ссылок на целочисленные одномерные массивы. При таком описании потребуется не только выделять память под одномерный массив ссылок, но и под каждый из целочисленных одномерных массивов. Такое распределение памяти позволяет определять произвольную длину каждой строки массива (отсюда и произошло название массива - ступенчатый). Например:

int [][] a= new int [3][]; // Создаем три строки

a[0]=new int [2]; // 0-ая строка ссылается на 2-х элементый одномерный массив

a[1]=new int [3]; // 1-ая строка ссылается на 3-х элементый одномерный массив

a[2]=new int [10]; // 2-ая строка ссылается на 10-ти элементый одномерный массив

Другой способ выделения памяти:

int [][] a= {new int [2], new int [3], new int [10]};

Так как каждая строка ступенчатого массива фактически является одномерным массивом, то с каждой строкой можно работать как с экземпляром класса Array. Это является преимуществом ступенчатых массивов перед двумерными массивами.

Пример:

class Program

{

static void Main()

{

try

{

int[][] MyArray;

Console.Write("Ввведите количество строк: ");

int n = int.Parse(Console.ReadLine());

MyArray = new int[n][];

for (int i = 0; i < MyArray.Length; i++)

{

Console.Write("введите количество элементов в {0} строке: ", i);

int j = int.Parse(Console.ReadLine());

MyArray[i] = new int[j];

for (j = 0; j < MyArray[i].Length; j++)

{

Console.Write("a[{0}][{1}]= ", i, j);

MyArray[i][j] = int.Parse(Console.ReadLine());

}

}

PrintArray("исходный массив:", MyArray);

for (int i = 0; i < MyArray.Length; i++) Array.Sort(MyArray[i]);

PrintArray("измененный массив", MyArray);

}

catch (FormatException)

{

Console.WriteLine("неверный формат ввода данных");

}

catch (OverflowException)

{

Console.WriteLine("переполнение");

}

catch (OutOfMemoryException)

{

Console.WriteLine("недостаточно памяти для создания нового объекта");

}

}

static void PrintArray(string a, int[][] mas)

{

Console.WriteLine(a);

for (int i = 0; i < mas.Length; i++)

{

for (int j = 0; j < mas[i].Length; j++) Console.Write("{0} ", mas[i][j]);

Console.WriteLine();

}

}

}

### Оператор foreach и его использование при работе с массивами

Оператор foreach применяется для перебора элементов в специальным образом организованной группе данных, в том числе и в массиве. Удобство этого вида цикла заключается в том, что нам не требуется определять количество элементов в группе и выполнять перебор по индексу - мы просто указываем на необходимость перебрать все элементы группы. Синтаксис оператора:

foreach (<тип> <имя> in <группа>) <тело цикла>

где *имя* определяет локальную по отношению к циклу переменную, которая будет по очереди принимать все значения из указанной *группы*, а *тип* соответствует базовому типу элементов *группы*.

Ограничением оператора foreach является то, что с его помощью можно только просматривать значения элементов в группе данных, но нельзя их изменять.

Рассмотрим несколько примеров использования оператора foreach:

1. для работы с одномерными массивами:

static void PrintArray(string a, int [] mas)

{

Console.WriteLine(a);

foreach (int x in mas)Console.Write("{0} ", x);

Console.WriteLine();

}

1. для работы с двумерными массивами:

static int Sum (int [,] mas)

{

int s=0;

foreach (int x in mas) s += x;

return s;

}

1. для работы со ступенчатыми массивами:

static void PrintArray3(string a, int[][] mas)

{

Console.WriteLine(a);

for (int i = 0; i < mas.Length; i++)

{

foreach (int x in mas[i]) Console.Write("{0} ", x);

Console.WriteLine();

}

}