Государственный университет – Высшая школа экономики

Факультет Бизнес-информатики
Отделение Программной Инженерии

Конспект лекций по дисциплине "Информатика и программирование"

Москва – 2007

Аннотация

Данные материалы предназначены для обеспечения лекционных занятий по дисциплине "Информатика и программирование" со студентами первого курса по направлениям "бизнес информатика" и "программная инженерия".

Материалы охватывают все темы дисциплины "Информатика и программирование" и могут использоваться в качестве конспекта лекций. Темы по разработке приложений с развитым графическим интерфейсом и использованию визуальных средств программирования изложены в виде опорного конспекта и сопровождаются графическим иллюстративным материалом.

Оглавление

АЛГОРИТМ И ЕГО СВОЙСТВА

1 Определение алгоритма

Информатика – это научная и прикладная область знаний, изучающая информационные процессы. *Информационным процессом* называются процесс получения, хранения, обработки и передачи информации с помощью компьютерных и других технических средств.

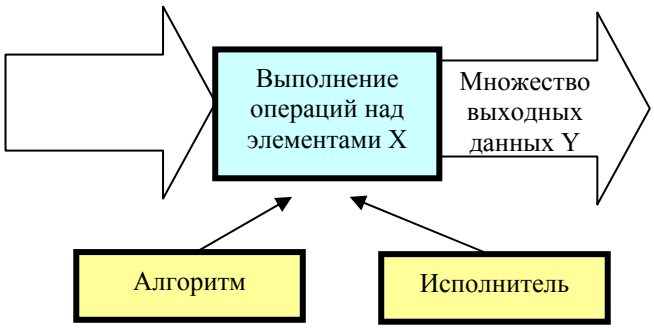


Рис. 1

Предметом изучения данного курса являются процессы обработки информации с помощью средств вычислительной техники с целью получения информации нового качества. Информацию любой природы, зафиксированную тем или иным способом, принято называть данными.

Таким образом, задача обработки информации будет рассматриваться как задача получения выходных данных путем обработки входных данных. Решение этой задачи в самом общем виде приведено на Рис.1. Для решения задачи необходимо иметь:

1. Точное описание действий (операций), выполняемы над данными, и последовательность выполнения этих действий.
2. Исполнителя, способного выполнить требуемые действия.

На основании указанных требований можно дать следующее определение алгоритма: *алгоритм – это точное описание* ***последовательности действий*** *над* ***входными данными,*** *выполнение которых* ***исполнителем*** *приводит к получению выходных* ***данных****.*

Пример. Входные данные представляет собой коэффициенты квадратного уравнения $AX^2 + BX + C = 0$. Вычислить корни уравнения X_1, X_2. Вычисления повторять до ввода нулевого значения коэффициента A.Решение задачи возлагается на исполнителя, который умеет выполнять операции с вещественными числами и вычислять типовые математические функции.

Алгоритм решения задачи представим в виде перечня действий. Каждый пункт этого перечня будем называть шагом:

1. Ввести значения коэффициентов A,B,C
2. Если A равно 0, то перейти к шагу 9
3. Вычислить $D = B^2 - 4AC$.
4. Вычислить $X_1 = \dfrac{-B + \sqrt{D}}{2A}$
5. Вычислить $X_2 = \dfrac{-B - \sqrt{D}}{2A}$
6. Вывести значение X_1 и значение X_2
7. Ввести значения коэффициентов A,B,C
8. Перейти к шагу 2.
9. СТОП – прекратить выполнения алгоритма.

Действия выполняются последовательно в порядке перечисления шагов. Такой порядок выполнения шагов называется естественным поряд-

ком. Для изменения естественного порядка в алгоритм включены специальные действия, выполняемые на шаге 2 и шаге 8.

2. Свойства алгоритма

Схематично основные свойства алгоритма приведены на Рис.2. Рассмотрим эти свойства подробнее.

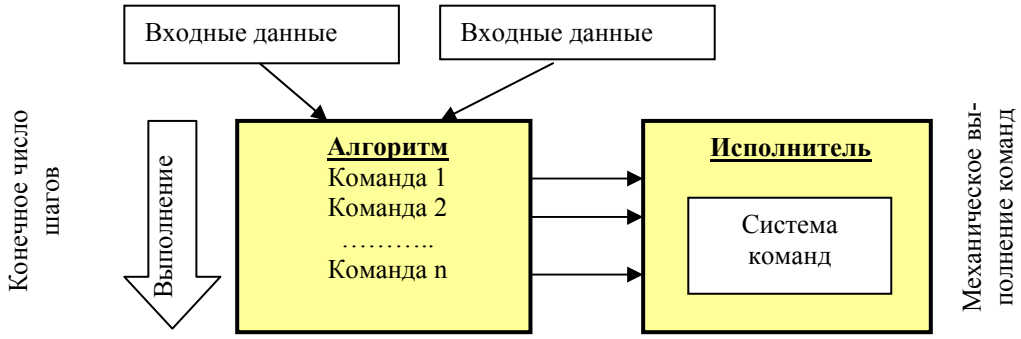


Рис. 2

Дискретность - алгоритм должен быть записан в виде конечного числа шагов. Выполнение каждого шага начинается после завершения выполнения предыдущего.

Конечность - выполнение алгоритма завершается после выполнения конечного числа шагов. При выполнении алгоритма некоторые шаги могут выполняться многократно. В рассмотренном примере шаги с 2-го по 8-ой выполняются до ввода нулевого значения коэффициента A.

Понятность - исполнитель должен знать, что ему делать. *Отдельные указания исполнителю, содержащиеся в каждом шаге, называются командами.* Каждый исполнитель характеризуется набором команд, которые он понимает и может выполнить. Совокупность таких команд называется *системой команд* исполнителя. Другими словами, понятность означает, что алгоритм содержит только те команды, которые входят в систему команд исполнителя.

Определенность – алгоритм не должен допускать произвольной трактовки шагов со стороны исполнителя. Исполнитель должен действовать в строгом соответствии с командами, которые указаны в каждом шаге. У исполнителя не должно возникать необходимости предпринимать действия, не предусмотренные алгоритмом. Другими словами, алгоритм рассчитан на механическое выполнение устройством, не обладающим "здравым" смыслом.

Приведенный алгоритм содержит элемент неопределенности. Отрицательное значение D приводит к невозможности вычисления $\sqrt{D}$, а действия исполнителя в этой ситуации не определены. *В алгоритме рекомендуется указывать команды, которые выполняют контроль входных данных и определяют действия исполнителя при получении некорректных данных*.

Массовость означает, что один и тот же алгоритм можно использовать для решения многих однотипных задач, отличающихся количеством и/или значениями входных данных.

3. Способы записи алгоритма

Исполнителем алгоритма может быть человек или техническое устройство. Форма записи алгоритма должна учитывать особенности исполнителя. Алгоритм, предназначенный для исполнения человеком, может записываться на естественном языке или в виде графических схем. Для документирования алгоритма применяются графические схемы.

Пример записи алгоритма на естественном языке был рассмотрен выше.

Для записи алгоритма в виде графических схем используется несколько нотаций. Наиболее распространенными из них являются:

1. Схема алгоритма. Правила выполнения схемы алгоритма регламентируются стандартом ГОСТ 19.002-80 и ГОСТ 19.003-80 (в измененной редакции объединены в ГОСТ 19.701-90).
2. Структурограмма.

Алгоритм, предназначенный для исполнения техническим устройством, должен быть записан на формализованном языке, "понятном" для этого устройства. В этом случае алгоритм представляет собой последовательность команд исполнителя.

Для составления алгоритма необходимо знать систему команд исполнителя. Это создает большие проблемы при замене исполнителя. Для решения проблемы используется запись алгоритма на языке, не зависящем от особенностей исполнителя. Такой язык не воспринимается техническим устройством и требует предварительного перевода на язык команд исполнителя. *Запись обрабатываемых данных и алгоритма на языке, доступном для восприятия исполнителем непосредственно или через систему перевода, называется программой.*

3. Типовые управляющие конструкции и их условные графические обозначения

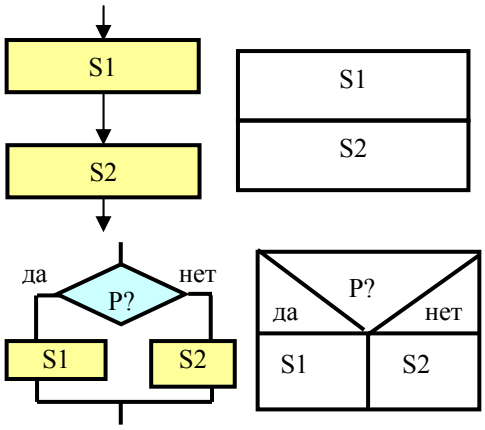
Алгоритм любой степени сложности может быть составлен путем комбинации трех основных вариантов управления действиями исполнителя: *следование, ветвление и цикл.* Эти варианты управления принято называть типовыми управляющими конструкциями.

Кроме того, используются дополнительные управляющие конструкции, представляющие собой модификацию основных конструкций. К ним относят *сокращенное ветвление и множественное ветвление (переключатель).*

Управляющие конструкции определяют последовательность выполнения фрагментов алгоритма. В дальнейшем фрагменты алгоритма будем называть блоками. Все современные языки программирования имеют средства для записи типовых управляющих конструкций и их модификаций.

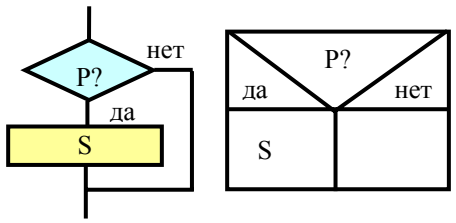
Рассмотрим условные графические обозначения управляющих конструкций применительно к схеме алгоритма и структурограмме. При изображении управляющих конструкций будем использовать следующие обозначения: **S** – ,блок, **P** – проверяемое условие. Результат проверки условия определяет следующий шаг алгоритма.

Схема алгоритма Структурограмма

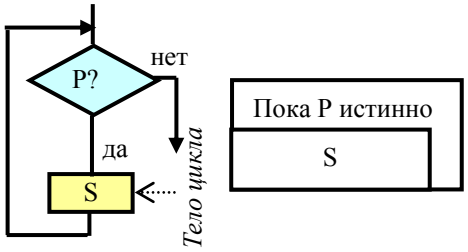


Следование предписывает последовательное выполнение блоков. Первым выполняется блок **S1**, а после его завершения – блок **S2**.

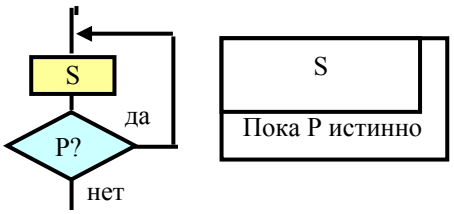
Ветвление предписывает выбор одного из двух блоков в зависимости от того, выполняется условие **P** или нет. В приведенной конструкции при положительном результате проверки условия выполняется блок **S1**, а при отрицательном – блок **S2**.



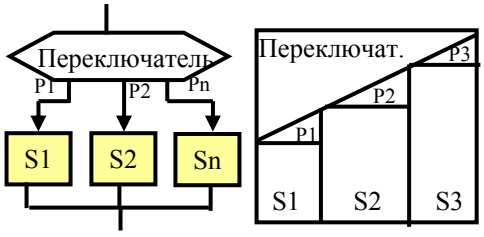
Сокращенное ветвление предписывает выполнить блок **S** при положительном результате проверки условия **P**.



Цикл с предусловием предписывает повторять выполнение блока **S**. Выполнению предшествует проверка условия повтора **P.** Блок, выполняемый в цикле, называется телом цикла, а однократное выполнение тела цикла – шагом цикла или итерацией. Условие повтора может быть сформулировано так, что ни одного шага не будет выполнено.



Цикл с постусловием предписывает повторять выполнение блока **S**. Условие повтора **P** проверяется после выполнения блока**.** Гарантируется выполнение тела цикла хотя бы один раз.



Множественное ветвление (переключатель) определяет выбор одного из **n** блоков (n>2). Для выполнения выбирается тот блок, условие которого совпадает со значением переключателя. На рисунке приведен вариант множественного ветвления для выбора одного из трех блоков.

С точки зрения последовательности выполнения блоков все рассмотренные конструкции имеют один вход и один выход. *Это означает, что любой блок может быть представлен его более детальным описанием в виде одной из типовых управляющих конструкций.*

Некоторые блоки могут быть оформлены в виде отдельных процедур. Процедура представляет собой самостоятельную часть алгоритма и предназначена для решения частного процесса обработки. Процедуру можно вызвать из любого места алгоритма и передать ей данные. После выполнения процедуры автоматически происходит возврат в то место алгоритма, откуда процедура была вызвана, при этом передаются данные, полученные при выполнении процедуры.

Процедура, как правило, используются для оформления типовых процессов обработки, которые отличаются только значениями обрабатываемых данных. При записи алгоритма на языке команд исполнителя вместо термина "процедура" используется термин "подпрограмма". Типичной является ситуация, когда исполнитель предоставляет разработчику алгоритма помимо команд заранее разработанный набор (библиотеку) стандартных подпрограмм.

Начало или окончание | Ввод или вывод данных | Вызов подпрограммы

При составлении схем алгоритмов применяются специальные обозначения для начала и окончания выполнения алгоритма, получения входных и выдачи выходных данных и вызова подпрограмм.

В учебных целях допускается использовать комбинированные обозначения: схем алгоритмов и структурограмм.

4. Структурный подход к разработке алгоритма

Структурный подход предполагает пошаговую детализацию алгоритма по принципу "сверху вниз". Рассмотрим основные правила разработки алгоритма при этом подходе:

1. Определите входные и выходные данные.
2. Представьте процесс обработки входных данных в виде одного блока.
3. Выполните шаг детализации. Для этого разбейте блок на более простые блоки по схеме одной из управляющих конструкций. Определите промежуточные данные, необходимость в которых возникла на данном шаге детализации.
4. Для блоков, которые могут быть выражены на доступном для исполнителя языке, прекратите детализацию. Остальные блоки детализируйте дальше и к каждому из них примените правило 3.

Рассмотрим структурный подход к разработке алгоритма на примере вычисления корней квадратного уравнения.

Процесс детализации исходной задачи показан в виде структурограмм, приведенных на Рис.3. Структурограммы следует рассматривать слева направо и сверху вниз.

Жирной линией показаны блоки, которые необходимо детализировать на следующих шагах, поскольку выполняемое в них действие не может быть выражено в виде команды исполнителя или вызова стандартной подпрограммы.

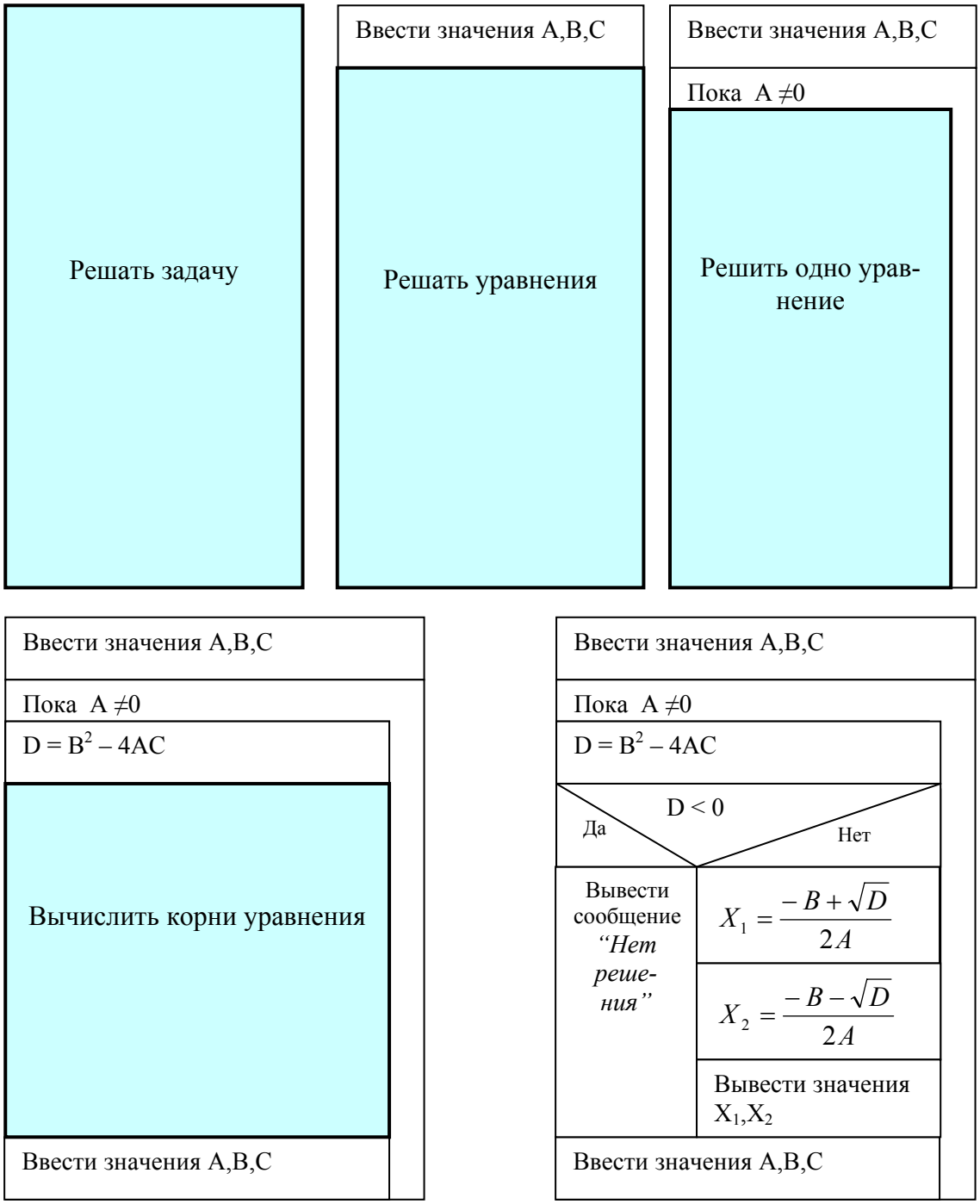


Рис.3

В окончательном виде структурограмма отражает все уровни детализации и содержит информацию о структурной организации программы. Это позволяет формализовать процесс преобразования алгоритма в программу на языке, в котором реализованы типовые управляющие конструкции. Динамика развития процесса обработки данных в структурограмме выражена в неявном виде и в этом плане структурограмма менее наглядна, чем схема алгоритма.

Структурный подход можно применять и для разработки схемы алгоритма. Пример схемы алгоритма для задачи вычисления корней квадратного уравнения приведен на Рис.4.

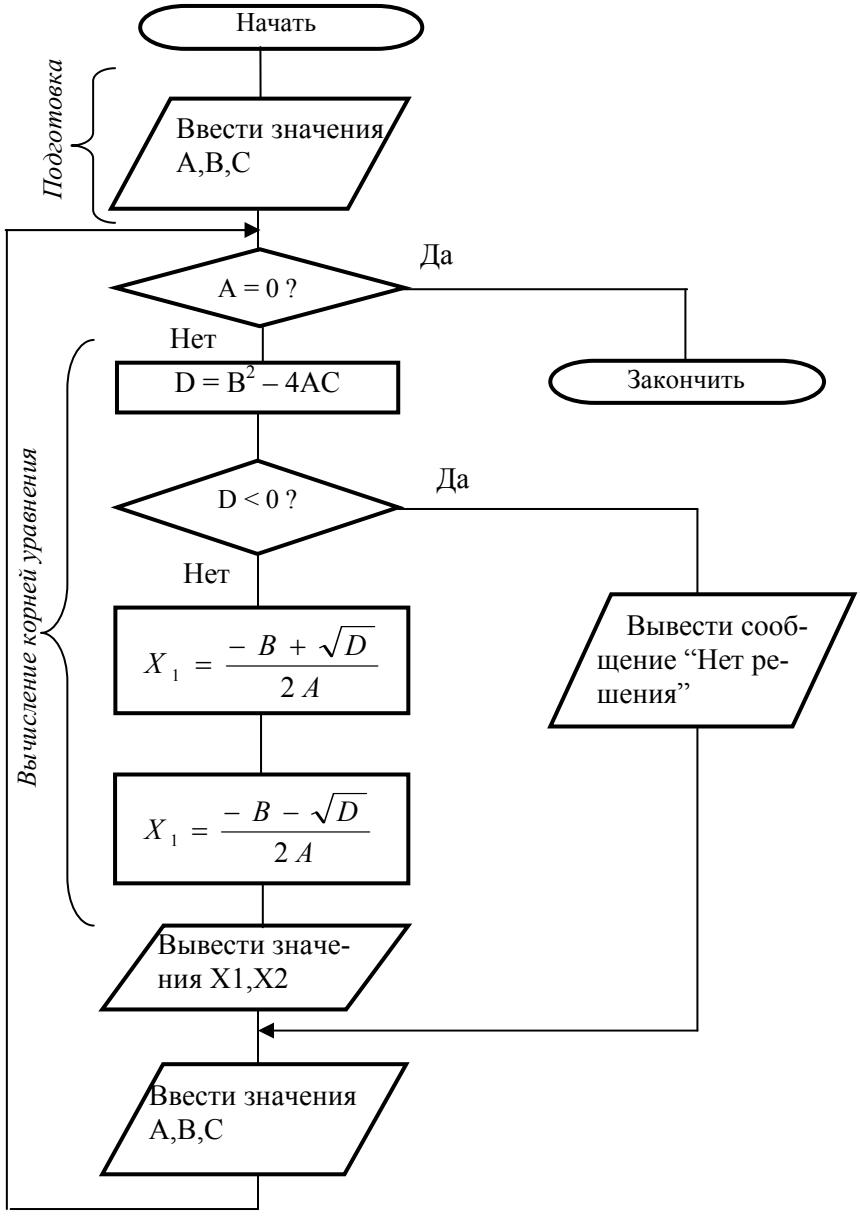


Рис.4

В окончательном виде схема алгоритма отражает только последний уровень детализации. Схема алгоритма более наглядно отражает динамику развития процесса обработки данных, но не дает представления о структурной организации программы.

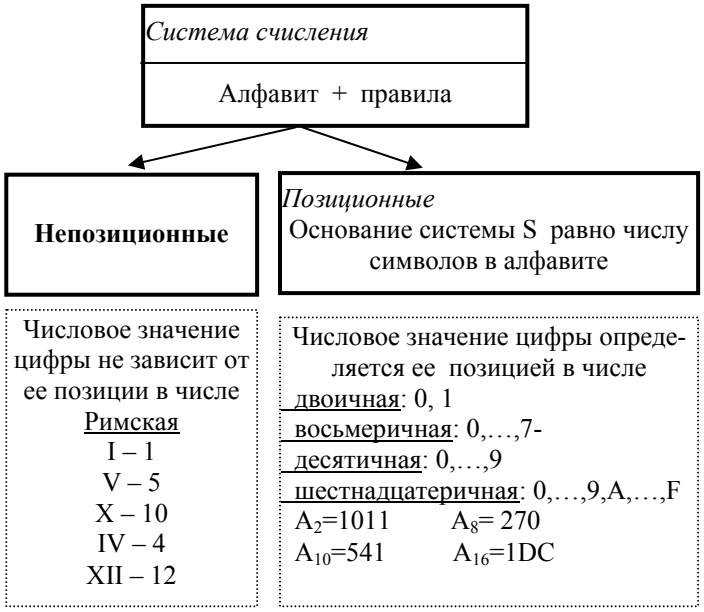
Развитие процесса указывается на схеме алгоритма в виде линий со стрелками. Стрелка указывает направление развития процесса обработки данных. Направление сверху вниз и направление слева направо приняты по умолчанию. Стрелки для этих направлений в схеме алгоритма могут не указываться.

МАТЕМАТИЧЕСКИЕ ОСНОВЫ ПОСТРОЕНИЯ ЭВМ

Основным средством реализации информационных процессов являются электронно-вычислительные машины (ЭВМ). ЭВМ представляет собой техническое устройство, предназначенное для автоматического выполнения алгоритма. Современные ЭВМ способны обрабатывать информацию различных видов: числовую, текстовую, звуковую, изображения и т.д. Информация любого вида в конечном итоге преобразуется к числовой форме. Это преобразование называют оцифровкой данных. Представление чисел и их обработка в ЭВМ составляет основу информационных процессов.

1. Системы счисления

Система счисления – совокупность символов и правил для наименования и обозначения чисел. Символы называются цифрами. Классификация систем счисления в зависимости от способа определения числового эквивалента цифр приведена на рисунке.



Позиционные системы счисления имеют более простые правила выполнения арифметических операций, что обусловило их повсеместное использование. При обработке данных в ЭВМ в основном используются двоичная, восьмеричная, шестнадцатеричная и десятичная позиционные системы счисления.

Аппаратные средства ЭВМ рассчитаны на представление и обработку двоичных данных. Объясняется это простотой реализации электронного устройства, имеющего два устойчивых состояния. Одно из состояний принимается за **0**, другое за **1**. За простоту реализации приходится платить большим количеством цифр в записи числа. Двоичное число в среднем в 3,3 раза длиннее десятичного.

Восьмеричная и шестнадцатеричная системы широко используется для записи двоичных чисел в компактной форме. Такая форма применяется в технической документации для ЭВМ и в программах на машинно-ориентированных языках.

Для записи данных в программах на языках высокого уровня используется, в основном, привычная для человека десятичная система счисления. Преобразование к двоичному виду выполняется при переводе программы на машинный язык.

В общем виде число в позиционной системе счисления записывается в виде полинома:

$$A=\underbrace{a_{n-1}S^{n-1}+\ldots+a_1S^1+a_0S^0}_{\text{Целая часть}}+\underbrace{a_{-1}S^{-1}+\ldots+a_{-m}S^{-m}}_{\text{Дробная часть}} \quad (1)$$

где S – основание системы, a_i – одна из цифр алфавита этой системы счисления.

При сокращенной записи числа основание системы счисления подразумевается, а цифры целой и дробной части разделяются точкой:

$$A=\underbrace{a_{n-1}\,a_{n-2}\ldots a_1a_0}_{\text{Целая часть}}.\ \underbrace{a_{-1}\,a_{-2}\ldots a_{-m}}_{\text{Дробная часть}}$$

Каждая цифра в записи числа называется разрядом. Для двоичных чисел вместо термина разряд часто используют термин бит. Каждый левый разряд имеет вес в S раз больший, чем предыдущий. Крайний правый разряд имеет наименьший вес и называется младшим разрядом. Крайний левый разряд имеет наибольший вес и называется старшим разрядом.

Пусть на запись числа отведено n+m разрядов, из них n на целую часть и m на дробную часть. Число рассматривается как число без знака. Определим максимальное и минимальное число, которое может быть записано в этой разрядной сетке:

Максимальное число $\mathbf{Amax = (S^n - 1) + (1 - S^{-m})}$

максимальная целая часть

максимальная дробная часть

n=3				m=2	
9	**9**	**9**	**.**	**9**	**9**

Например, для десятичной системы при n=3 и m=2 максимальное число

$Amax = (10^3-1) + (1-10^{-2}) = (1000-1) + (1-0.01) = 999 + 0.99 = 999.99$

Для получения записи максимального числа необходимо занести во все разряды наибольшую цифру системы счисления.

Минимальное нуля число Amin= **0 + S^{-m}**

минимальная целая часть

минимальная дробная часть

отличное от

n=3				m=2	
0	**0**	**0**	**.**	**0**	**1**

Например, для десятичной системы при n=3 и m=2 минимальное число

$$Amin = 10^{-2} = 0.01$$

Для получения записи минимального числа необходимо занести в младший разряд единицу, а в остальные разряды нули.

Количество различных комбинаций цифр Nк определяет количество различных по значению чисел, которые могут быть записаны в заданных разрядах:

$$Nк = S^{m+n}$$

Для рассмотренного примера количество десятичных чисел $Nк=10^5$

При работе с технической документацией и анализе машинных программ приходится выполнять преобразования чисел из одной системы счисления в другую. Общие правила преобразования будут рассмотрены ниже.

В таблице приведена запись первых 20-ти чисел в разных системах счисления:

A_{10}	A_2	A_8	A_{16}
0	0	0	0
1	1	1	1
2	10	2	2
3	11	3	3
4	100	4	4
5	101	5	5
6	110	6	6
7	111	7	7
8	1000	10	8
9	1001	11	9

A_{10}	A_2	A_8	A_{16}
10	1010	12	A
11	1011	13	B
12	1100	14	C
13	1101	15	D
14	1110	16	E
15	1111	17	F
16	10000	20	10
17	10001	21	11
18	10010	22	12
19	10011	23	13

2. Преобразования из любой системы счисления в десятичную систему

Выполняется по полиномиальной записи числа (1). Основание исходной системы и значения цифр рассматриваются как десятичные числа. Все операции выполняются по правилам десятичной арифметики. Рассмотрим примеры преобразования чисел в десятичную систему.

$A_{[2]} = 101.1 = 1*2^2 + 0*2^1 + 1*2^0 + 1*2^{-1} = 4 + 0 + 1 + 0.5 = 5.5_{[10]}$

$A_{[16]} = AB.8 = 10*16^1 + 11*16^0 + 8*16^{-1} = 160 + 11 + 0.5 = 171.5_{[10]}$

$A_{[8]} = 173.4 = 1*8^2 + 7*8^1 + 3*8^0 + 4*8^{-1} = 64 + 56 + 3 + 0.5 = 123.5_{[10]}$

Удобно сразу же над цифрами исходного числа проставлять вес разрядов

$$A_{[2]} = \overset{8}{1}\ \overset{4}{0}\ \overset{2}{0}\ \overset{1}{1}\ .\ \overset{0.5}{1}\ \overset{0.25}{1} = 8 + 1 + 0.5 + 0.25 = 9.75_{[10]}$$

3. Преобразования из десятичной системы счисления в любую другую систему

Преобразование выполняется отдельно для целой и дробной части числа. Все операции выполняются по правилам десятичной арифметики. Основание новой системы рассматривается как десятичное число.

Преобразование целой части выполняется путем последовательного деления целой части на основание новой системы. На каждом шаге вычисляется целая часть и остаток от деления. Остатки и есть значение цифр числа в новой системе счисления. Полученные остатки записываются в новой системе счисления. На первом шаге будет получена младшая цифра. Процесс продолжается, пока очередная целая часть не окажется меньше основания новой системы счисления. Эта целая часть есть старшая цифра числа.

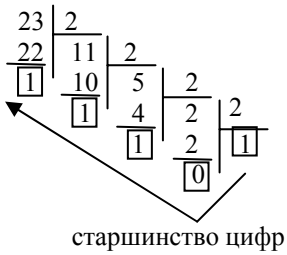
Правила перевода следуют из формы записи числа в позиционной системе счисления:

Шаг 1 $A_{10} = a_{n-1}S^{n-1} + \ldots + a_1S^1 + a_0 \mid :S \Rightarrow \underbrace{a_{n-1}S^{n-2} + \ldots + a_1S^0}_{\text{Целая часть}} + \underbrace{\frac{a_0}{S}}_{\text{Остаток}} \rightarrow \boldsymbol{a_0}$

Шаг 2 $A_{10} = a_{n-1}S^{n-2} + \ldots + a_2S^1 + a_1 \mid :S \Rightarrow \underbrace{a_{n-1}S^{n-3} + \ldots + a_2S^0}_{\text{Целая часть}} + \underbrace{\frac{a_1}{S}}_{\text{Остаток}} \rightarrow \boldsymbol{a_1}$

Шаг n-1 $A_{10} = a_{n-1}S^1 + a_{n-2} \mid : S \Rightarrow \underbrace{\boldsymbol{a_{n-1}}}_{\text{Целая часть}} + \underbrace{\frac{a_{n-2}}{S}}_{\text{Остаток}} \rightarrow \boldsymbol{a_{n-2}}$

Пример: $\mathbf{23_{[10]} \rightarrow ?_{[2]}}$



$\mathbf{10111_{[2]}}$

Пример: $\mathbf{23_{[10]} \rightarrow ?_{[8]}}$

$\mathbf{27_{[8]}}$

Пример: $\mathbf{23_{[10]} \rightarrow ?_{[16]}}$

$\mathbf{17_{[16]}}$

Преобразование дробной части выполняется путем последовательного умножения дробной части на основание новой системы. На каждом шаге выделяется новая целая часть и новая дробная часть. Последовательность целых частей дает значение цифр числа в новой системе счисления. На первом шаге будет получена старшая цифра дробной части в новой системе. Процесс продолжается до получения заданного количества значащих цифр или нулевого значения дробной части.

Правила перевода следуют из формы записи числа в позиционной системе счисления:

Шаг 1 $A_{10} = a_{-1}S^{-1} + a_{-2}S^{-2} + \ldots + a_{-m}S^{-m} \mid * S \Rightarrow \underbrace{\boldsymbol{a_{-1}}}_{\text{Целая часть}} + \underbrace{a_{-2}S^{-1} + \ldots + a_{-m}S^{-m+1}}_{\text{Дробная часть}}$

Шаг 2 $A_{10} = a_{-2}S^{-1} + \ldots + a_{-m}S^{-m+1} \mid * S \Rightarrow \underbrace{\boldsymbol{a_{-2}}}_{\text{Целая часть}} + \underbrace{a_{-3}S^{-1} + \ldots + a_{-m}S^{-m+2}}_{\text{Дробная часть}}$

Пример: $\mathbf{0.125_{[10]} \rightarrow ?_{[2]}}$

В данном примере преобразование завершено при получении нулевой дробной части.

$\mathbf{0.125_{[10]} \rightarrow 0.001_{[2]}}$

Целая часть	Дробная часть
0	125 * 2
0	250 * 2
0	500 * 2
1	000

Пример: $\mathbf{0.55_{[10]} \rightarrow ?_{[16]}}$

В данном примере преобразование проводится до получения трех значащих цифр дробной части.

$\mathbf{0.55_{[10]} \rightarrow 0.8CC_{[16]}}$

Целая часть	Дробная часть
0	55 * 16
8	800 *16
C	800 *16
C	800

4. Преобразования из любой системы счисления в любую другую систему счисления

Необходимо преобразовать число из системы счисления S1 в систему счисления S2. Это наиболее общий вариант преобразования чисел из одной системы счисления в другую. Преобразование выполняется по тем же правилам, что и из десятичной системы в любую другую. Операции выполняются по правилам арифметики в системе счисления S1, что соз-

дает определенные неудобства. На практике такие преобразования выполняются через промежуточное преобразование в десятичную систему:

Число в системе S1 → Число в десятичной системе → Число в системе S2

Такая схема позволяет выполнять все операции в привычной для человека десятичной системе.

Важным для практики случаем является преобразование из двоичной системы в восьмеричную или шестнадцатеричную систему и обратное преобразование. Такие преобразования можно выполнить без промежуточного преобразования в десятичную систему.

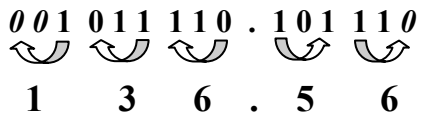
5. Двоично-восьмеричные и двоично-шестнадцатеричные преобразования

Эти преобразования наиболее просты, так как восьмеричные и шестнадцатеричные числа представляют собой не что иное, как компактную форму записи двоичных чисел. Преобразование базируются на том, что основание одной системы является степью двойки основания другой системы:

$2^3 = 8$, поэтому восьмеричную цифру можно представить группой из трех двоичных цифр. Группа из трех двоичных цифр называется триадой;

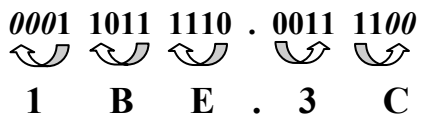
$2^4 = 16$, поэтому шестнадцатеричную цифру можно представить группой из четырех двоичных цифр. Группа из четырех двоичных цифр называется тетрадой.

Преобразование «2 → 8». Двигаясь от точки влево и вправо разбить цифры исходного двоичного числа на триады. При необходимости дополнить число слева и справа не значащими нулями. Каждую триаду заменить восьмеричной цифрой.



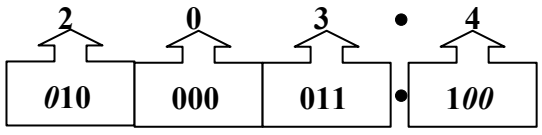
На рисунке приведен пример преобразования двоичного числа 1011110.10111 в восьмеричное число 136.56. Для образования триад слева добавлены два нуля, а справа один.

Преобразование «2 → 16». Правила аналогичны преобразованию «2 → 8», но исходное двоичное число разбивается на тетрады.



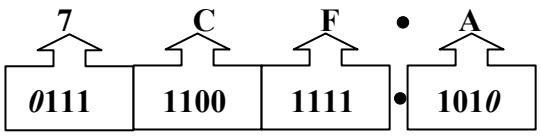
На рисунке приведен пример преобразования двоичного числа 110111110.001111 в шестнадцатеричное число 1BE.3C. Для образования тетрад слева добавлены три нуля, а справа два.

Преобразование «8 → 2». Это преобразование противоположно преобразованию «2 → 8». Каждая цифра исходного восьмеричного числа заменяется триадой, содержащей двоичный эквивалент восьмеричной цифры. Незначащие левые и правые нули можно отбросить.



На рисунке приведен пример преобразования восьмеричного числа 203.4 в двоичное число 10000011.1. Слева отброшен один незначащий ноль, а справа - два незначащих нуля.

<u>Преобразование «16 → 2»</u>. Это преобразование противоположно «2 → 16». Каждая цифра исходного шестнадцатеричного числа заменяется тетрадой, содержащей двоичный эквивалент шестнадцатеричной цифры. Незначащие левые и правые нули можно отбросить.



На рисунке приведен пример преобразования шестнадцатеричного числа 7CF.A в двоичное число 11111001111.101. Слева и справа отброшено по одному незначащему нулю.

Для получения двоичной тетрады, эквивалентной восьмеричной цифре, можно использовать правило **"421"**, в основе которого лежит представление числа в виде суммы степеней двойки. В триаде необходимо записать единицы на местах цифр, сумма которых дает значение восьмеричной цифры. На местах остальных цифр записать ноль.

Примеры: $7_{[8]} = 4 + 2 + 1 \Rightarrow 111_{[2]}$
$3_{[8]} = 2 + 1 \Rightarrow 011_{[2]}$
$6_{[8]} = 4 + 2 \Rightarrow 110_{[2]}$
$5_{[8]} = 4 + 1 \Rightarrow 101_{[2]}$

Для получения двоичной тетрады, эквивалентной шестнадцатеричной цифре, можно использовать правило **"8421".** В тетраде необходимо записать единицы на местах цифр, сумма которых дает значение шестнадцатеричной цифры. На местах остальных цифр записать ноль

Примеры**:** $F_{[16]} = 15_{[10]} = 8 + 4 + 2 + 1 \Rightarrow 1111_{[2]}$
$C_{[16]} = 12_{[10]} = 8 + 4 \Rightarrow 1100_{[2]}$
$D_{[16]} = 13_{[10]} = 8 + 4 + 1 \Rightarrow 1101_{[2]}$
$9_{[16]} = 9_{[10]} = 8 + 1 \Rightarrow 1001_{[2]}$

6. Двоичная арифметика

Правила арифметических операций во всех позиционных системах одинаковы и основываются на таблицах сложения, вычитания и умножения одноразрядных чисел. В силу наибольшей распространенности двоичной системы более детально остановимся на ней.

Таблица. сложения	Таблица. вычитания	Таблица. Умножения
	(заем)	
0 + 0 = 0	10 – 1 = 1	0 * 0 = 0
0 + 1 = 1	1 – 1 = 0	0 * 1 = 0
1 + 0 = 1	0 – 0 = 0	1 * 0 = 0
1 + 1 = 10 (перенос)	1 – 0 = 1	1 * 1 = 1

<u>Сложение двоичных чисел.</u> Выполняется поразрядно, начиная с младшего разряда. В сложении участвуют одноименные разряды слагаемых и перенос в текущий разряд. Результатом является сумма в текущем разряде и перенос в следующий разряд. В зависимости от значения слагаемых перенос будет равен 0 (нет переноса) или 1 (есть перенос). ЭВМ имеет специальное устройство, предназначенное для сложения двоичных чисел. Это устройство называется сумматором. Сумматор реализует сложе-

```
  1 1 1 1 0 1 . 0 1   ⇒ 29.25
+   1 0 1 1 . 1 0     ⇒ 11.5
-------------------
1 0 1 0 0 0 . 1 1     ⇒ 40.75
```

ние через комбинацию логических операций. Логические операции будут рассмотрены позже.

Вычитание двоичных чисел. Выполняется поразрядно, начиная с младшего. При необходимости выполняется заем двойки из соседнего старшего разряда. ЭВМ обычно не имеет специального вычитающего устройства. За счет применения специальных кодов операция вычитания заменяется на операцию сложения.

```
  1 1 0 1 1 . 1 0  ⇒ 27.5
- 1 1 0 1 . 0 1    ⇒ 13.25
  1 1 1 0 . 0 1    ⇒ 14.25
```

Умножение двоичных чисел. Выполняется путем последовательного умножения цифр множителя на множимое и сложение частичных сумм. Количество цифр в дробной части результата равно суммарному количеству цифр в дробных частях сомножителей. Пример умножения приведен на Рис.1. В ЭВМ умножение реализуется путем сдвига множимого и прибавление его к сумме, если цифра множителя равна единице.

```
    1 0 1 . 1 0     ⇒ 5.5
*     1 0 . 0 1     ⇒ 2.25
  + 1 0 1 1 0
  + 0 0 0 0 0
+ 0 0 0 0 0
1 0 1 1 0
1 1 0 0. 0 1 1 0    ⇒ 12.375
         2+2 = 4 цифры
```

Рис. 1

$$\underbrace{1100.011}_{12.375} : \underbrace{101.1}_{5.5} = \underbrace{11000.11}_{123.75} : \underbrace{1011}_{55} = \underbrace{10.01}_{2.25}$$

```
 11000.11 | 1011
-1011     | 10.01
  -1011
   1011
      0
```

Рис. 2

Деление двоичных чисел. Реализуется путем последовательных умножений цифр результата на делитель и вычитания из делимого. Пример деления приведен на Рис.2.

При выполнении операций сложения и вычитания вручную удобнее пользоваться восьмеричной или шестнадцатеричной системой. Рассмотрим особенности выполнения операций в этих системах счисления.

Если **при сложении** одноименных разрядов и переноса получено число N<S, то оно берется в качестве суммы разряда, а перенос в следующий разряд равен 0. Если получено число N≥S, то в качестве суммы берется разность N-S и формируется единица переноса в следующий разряд. *При сложении нескольких чисел сумма одноименных разрядов может превысить основание системы счисления в несколько раз. В этом*

Пример: $267_{[8]}+136_{[8]}=425_{[8]}$

```
  2 6 7   ⇒ 183[10]
+ 1 3 6   ⇒  94[10]
  4 2 5   ⇒ 277[10]
```

13-8=5

10-8=2

Пример: $2FA_{[16]}+3B2_{[16]}=6AC_{[16]}$

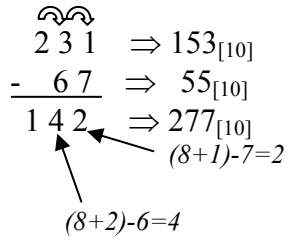
```
  2 F A   ⇒  762[10]
+ 3 B 2   ⇒  946[10]
  6 A C   ⇒ 1708[10]
```

26-16=10

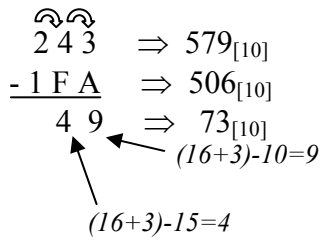
случае необходимо пользоваться общим правилом: в качестве суммы берется остаток от целочисленного деления N/S, а значение переноса в следующий разряд есть целая часть от деления N/S.

При вычитании разрядов при необходимости берется заем из старшего разряда. Вычитание производится из суммы $S+A_i$, где A_i – значение i-го разряда вычитаемого. Значение разряда, из которого производился заем, уменьшается на 1.

Пример: $231_{[8]}-67_{[8]}=142_{[8]}$



Пример: $243_{[16]}-1FA_{[16]}=49_{[16]}$



Операции умножения и деления в этих системах менее удобны для ручного выполнения, так как требуют знания таблиц умножения для восьмеричных или шестнадцатеричных чисел.

ЛОГИЧЕСКИЕ ОСНОВЫ ПОСТРОЕНИЯ ЭВМ

1. Логические переменные, функции и операции

ЭВМ преобразует информацию, представленную в виде двоичных кодов. Каждый разряд кода принимает значение 0 или 1. Именно поэтому для описания функционирования устройств ЭВМ широко используется аппарат алгебры логики.

Алгебра логики изучает логические функции. Функция $Y(x_0,x_1,\ldots,x_n)$ является логической, если ее аргументы и она сама принимают только одно из двух значений: "истина" или "ложь". Значение **"истина"** принято кодировать как **1**, а значение **"ложь"** - как **0**.

Аргументы логической функции называют логическими переменными. В алгебре логики логические переменные принято обозначать строчными латинскими буквами, а функции – прописными латинскими буквами.

Часто логические переменные и функции называют булевыми от имени английского математика Джорджа Буля, занимавшегося разработкой основ алгебры логики.

Логические функции строятся на основе элементарных логических операций. Так как значение логических переменных и функций ограничено значениями 0 и 1, определение смысла логических операций удобно производить табличным способом. В таблице приводятся все комбинации значений переменных и результат операции. Таблицу такого рода в алгебре логики называют таблицей истинности. Приведем таблицы истинности для элементарных логических операций.

1. Логическое отрицание (НЕ, инверсия: $\bar{x}$

x	$\bar{x}$
0	**1**
1	**0**

2. Логическое умножение (И, конъюнкция): $x_1 \wedge x_2$.

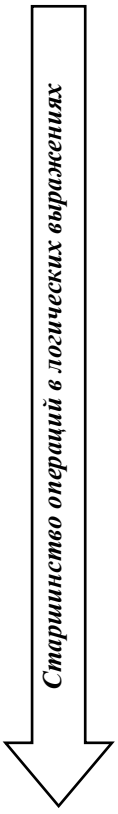
Иногда эта операция обозначается через знак умножения или его пропуск: $x_1 \bullet x_2$ или $x_1 x_2$.

x_1	x_2	$x_1 \wedge x_2$
0	0	**0**
0	1	**0**
1	0	**0**
1	1	**1**

3. Логическое сложение (ИЛИ, дизъюнкция): $x_1 \vee x_2$.

Иногда эта операция обозначается через знак сложения: x_1+x_2.

x_1	x_2	$x_1 \vee x_2$
0	0	**0**
0	1	**1**
1	0	**1**
1	1	**1**



Выражение, содержащее логические переменные и логические операции, называется логическим выражением. Порядок выполнения операций определяется их старшинством (приоритетом). Приоритет логических операций приведен на рисунке. Порядок выполнения операций

можно регулировать скобками. Операции в скобках выполняются в первую очередь.

Подобно алгебраическим операциям, для логических операций существуют законы их выполнения, свойства и следствия из них.

Основные законы логических операций:

Переместительный закон: $a \vee b = b \vee a$; $a \bullet b = b \bullet a$

Сочетательный закон: $a \vee (b \vee c) = (a \vee b) \vee c = b \vee (a \vee c)$

$a \bullet (b \bullet c) = (a \bullet b) \bullet c = b \bullet (a \bullet c)$.

Распределительные законы: $a \bullet (b \vee c) = a \bullet b \vee a \bullet c$ (первый)

$a \vee (b \bullet c) = (a \vee b) \bullet (a \vee c)$ (второй)

Закон двойственности (закон де Моргана):

$$\overline{a \bullet b} = \bar{a} \vee \bar{b}$$

$$\overline{a \vee b} = \bar{a} \bullet \bar{b}$$

Закон двойственного отрицания: $\bar{\bar{a}} = a$

Закон исключения: $a \vee \bar{a} = 1$

Закон противоречия: $a \bullet \bar{a} = 0$

Основные свойства логических операций:

Свойства логического умножения: $0 \bullet a = 0$, $1 \bullet a = a$

Свойства логического сложения: $0 \vee a = a$, $1 \vee a = 1$

Свойство повторения: $a \bullet a \bullet a \bullet \ldots \bullet a \bullet a = a$; $a \vee a \vee a \vee \ldots \vee a \vee a = a$.

Из перечисленных законов и свойств вытекает широко использующихся следствия:

Склеивание $a \bullet x \vee a \bullet \bar{x} = a \bullet (x \vee \bar{x}) = a \bullet 1 = a$

$(a \vee x) \bullet (a \vee \bar{x}) = a \vee (x \bullet \bar{x}) = a \bullet 0 = a$

Неполное склеивание:

$a \bullet x \vee a \bullet \bar{x} \vee a = a \bullet (x \vee \bar{x} \vee 1) = a \bullet 1 = a$

Поглощение: $a \vee a \bullet x = a \bullet (1 \vee x) = a \bullet 1 = a$

$a \vee a \bullet \bar{x} = a \bullet (1 \vee \bar{x}) = a \bullet 1 = a$

$a \bullet (a \vee x) = a \bullet a \vee a \bullet x = a \vee a \bullet x = a \bullet (1 \vee x) = a \bullet 1 = a$

$a \bullet (a \vee \bar{x}) = a \bullet a \vee a \bullet \bar{x} = a \vee a \bullet \bar{x} = a \bullet (1 \vee \bar{x}) = a \bullet 1 = a$

Устройства ЭВМ реализуют логические функции. Для сокращения элементов в устройстве логическую функцию максимально упрощают. Упрощение логической функции преследует цель сократить количество переменных и операций до минимума. Этот процесс называют минимизацией. Минимизация проводится на основе законов, свойств и следствий алгебры логики. Рассмотрим пример минимизации:

$$Y = \overline{x_2}\,\overline{x_1} x_0 + \overline{x_2} x_1 x_0 + x_2 \overline{x_1} x_0 + x_2 x_1 x_0 =$$

$$(\overline{x_2}\,\overline{x_1} x_0 + x_2 \overline{x_1} x_0) + (\overline{x_2} x_1 x_0 + x_2 x_1 x_0) =$$

$$\overline{x_1} x_0 (\overline{x_2} + x_2) + x_1 x_0 (\overline{x_2} + x_2) = \overline{x_1} x_0 1 + x_1 x_0 1 =$$

$$\overline{x_1} x_0 + x_1 x_0 = x_0 (\overline{x_1} + x_1) = x_0 1 = x_0$$

2. Способы описания логических функций

Применяются три основных способа описания логических функций: аналитический, табличный и графический.

При аналитическом способе функция записываются в виде логического выражения.

При табличном способе функция записывается в виде таблицы истинности. Число строк в таблице истинности равно числу комбинаций переменных, т.е 2^n, где n - число логических переменных. Таблицу истинности можно составить по аналитической записи путем подстановки всех комбинаций значений переменных.

Преобразование таблицы истинности в аналитическую запись выполняется по методу совершенной дизъюнктивной нормальной формы (СДНФ) или методу совершенной конъюнктивной нормальной формы (СКНФ).

СДНФ –функция, образованная сложением всех конституант единицы. Конституанта единицы составляется для тех комбинаций переменных, на которых функция равна единице. Конституанта единицы – это логическое умножение всех переменных, причем переменные, имеющие значение 0, берутся с инверсией. Составим аналитическую запись логической функции двух переменных по заданной таблице истинности.

Номер комбинации	x_1	x_2	Y
1	0	0	0
2	0	1	**1**
3	1	0	**1**
4	1	1	0

В рассматриваемом примере логических переменных две, т.о. количество комбинаций $2^2=4$. Комбинациям в строчках 2 и 3 соответствует единичное значение функции. Именно для них составляются конституанты 1.В результате сложения конституант 1 получим аналитическую запись функции по методу СДНФ:

$$Y = \bar{x}_1 \bullet x_2 + x_1 \bullet \bar{x}_2$$

Метод СДНФ целесообразно применять, когда в таблице истинности преобладают нулевые значения функции.

СКНФ *–функция, образованная умножением всех конституант нуля. Конституанта нуля составляется для тех комбинаций переменных, на которых значение логической функции равно нулю. Конституанта нуля – это логическое сложение всех переменных, причем переменные, имеющие значение 1, берутся с инверсией. Составим аналитическую запись логической функции по таблице истинности, рассмотренной в предыдущем примере. Комбинациям в строчках 1 и 4 соответствует нулевое значение функции. Именно для них составляются конституанты 0. Путем умножения конституант 0 получим СКНФ:*

$$Y = (x_1 + x_2) \bullet (\bar{x}_1 + \bar{x}_2)$$

Метод СКНФ целесообразно применять, когда в таблице истинности преобладают единичные значения функции.

Логические функции считаются тождественными, если их значения совпадают при всех комбинациях аргументов. Т.о. тождественность логических функций может быть проверена путем сопоставления их таблиц истинности.

Альтернативным путем проверки логических функций на тождественность является преобразование логических функций до совпадения их аналитической записи. Проверим тождественность полученных выше логических функций аналитическим путем.

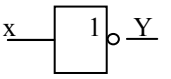
$$Y=(x_1+x_2)(\bar{x}_1+\bar{x}_2)=x_1\bar{x}_1+x_2\bar{x}_1+x_1\bar{x}_2+x_2\bar{x}_2=x_2\bar{x}_1+x_1\bar{x}_2$$

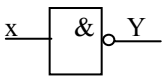
Имеем совпадение аналитической записи обеих логических функций, следовательно, они тождественны.

3. Графическое описание логической функции

Графическое представление чаще всего применяется для описания схемы устройства, которое реализует заданную функцию. Логические функции в этом случае называют переключательными функциями. Рассмотрим основные переключательные функции и их условные графические обозначения.

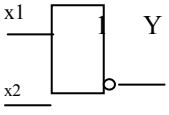
$Y=\bar{x}$ **(НЕ)**





Функция инверсии.

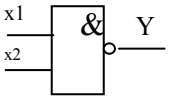
$Y=\overline{x_1+x_2}$ **(ИЛИ-НЕ)**



Функцию ИЛИ-НЕ иногда называют стрелкой Пирса.

Функция определена для любого числа аргументов.

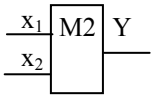
$Y=\overline{x_1 \bullet x_2}$ **(И-НЕ)**



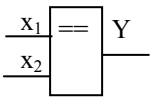
Функцию И-НЕ иногда называют стрелкой Пирса.
Функция определена для любого числа аргументов.

Набор функций, из которого методом суперпозиции можно получить любую переключательную функцию, называется логически (функционально) полным. Функционально полный набор образуют логическое сложение, умножение и инверсия. Причем сложение по закону де Моргана можно заменить умножением с инверсией аргументов. Аналогично умножение можно заменить сложением с инверсией аргументов. Таким образом, каждая из функций ИЛИ-НЕ и И-НЕ является функционально полной.

В ЭВМ широко используются функции двух переменных для проверки их равнозначности или неравнозначности.Формально они не относятся к основным, так как их можно реализовать на базе рассмотренных выше функций. Но ввиду распространенности на практике эти функции часто относят к основным функциям. Рассмотрим условные обозначения и логику этих функций.



$Y=x_1 \oplus x_2=x_2\bar{x}_1+x_1\bar{x}_2$ **(М2)**. Функция неравнозначности. Используется для арифметического сложения аргументов по модулю 2 и проверки переменных на несовпадение их значений.



$Y=\overline{x_1 \oplus x_2}$ **(= =)** . Функция равнозначности. Используется для проверки переменных на совпадения их значений. Эта функция представляет собой инверсию функции неравнозначности.

Таблица истинности функции неравнозначности (М2)

Таблица истинности функции равнозначности $\left(\overline{M2}\right)$

x_1	x_2	Y
0	0	**0**
0	1	**1**
1	0	**1**
1	1	**0**

x_1	x_2	Y
0	**0**	**1**
0	**1**	**0**
1	**0**	**0**
1	**1**	**1**

4. Примеры построения и преобразования переключательных функций

Пример 1. Дана таблица истинности, определяющая функции одноразрядного сумматора. В суммировании участвуют одноразрядные двоичные числа **a** и **b** и входной перенос **p**. Результатом работы сумматора является арифметическая сумма **s** и перенос в следующий разряд. $\mathbf{p_s}$. Получить аналитическую запись функций сумматора и его схему.

a	b	p		s	p_s
0	0	0		**0**	**0**
0	0	1		**1**	**0**
0	1	0		**1**	**0**
0	1	1		**0**	**1**
1	0	0		**1**	**0**
1	0	1		**0**	**1**
1	1	0		**0**	**1**
1	1	1		**1**	**1**

Построим СДНФ для функции получения суммы и минимизируем ее.

$$S = \bar{a}\bar{b}p + \bar{a}b\bar{p} + a\bar{b}\bar{p} + abp = (\bar{a}b\bar{p} + a\bar{b}\bar{p}) + (\bar{a}\bar{b}p + abp) = (\bar{a}b + a\bar{b})\bar{p} + (\bar{a}\bar{b} + ab)p =$$

$$= (a \oplus b)\bar{p} + \overline{\overline{(\bar{a}\bar{b} + ab)}}p = (a \oplus b)\bar{p} + \overline{\overline{(\bar{a}\bar{b}} \bullet \overline{ab)}}p = (a \oplus b)\bar{p} + \overline{(a + b)(\bar{a} + \bar{b})}p =$$

$$= (a \oplus b)\bar{p} + \overline{(a\bar{a} + b\bar{a} + a\bar{b} + b\bar{b})}p = (a \oplus b)\bar{p} + \overline{(0 + b\bar{a} + a\bar{b} + 0)}p = (a \oplus b)\bar{p} + \overline{(a \oplus b)}p = a \oplus b \oplus p$$

Построим СДНФ для функции получения переноса и минимизируем ее.

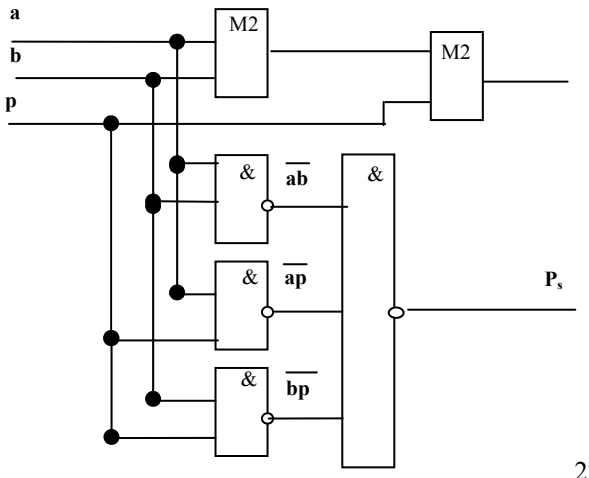
$$p_s = \bar{a}bp + a\bar{b}p + ab\bar{p} + abp$$

Допишем в функцию два раза (**abp)**, в силу свойства повторения это не повлияет на значение функции получения переноса.

$$p_s = (\bar{a}bp + abp) + (a\bar{b}p + abp) + (ab\bar{p} + abp) = bp(\bar{a} + a) + ap(\bar{b} + b) + ab(\bar{p} + p) =$$

$$= bp \bullet 1 + ap \bullet 1 + ab \bullet 1 = bp + ap + ab = \overline{\overline{bp + ap + ab}} = \overline{\overline{bp} \bullet \overline{ap} \bullet \overline{ab}}$$

Используя условные графические обозначения для основных переключательных функций, изобразим схему одноразрядного сумматора.



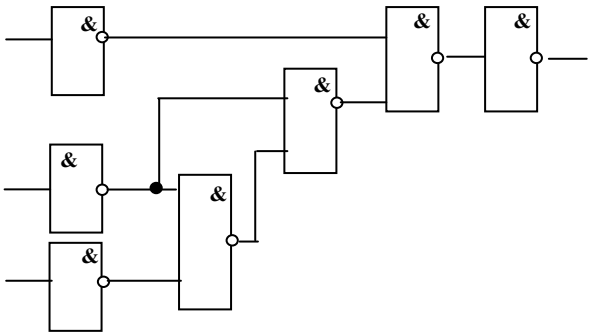
Соединив последовательно по линии распространения переноса N таких схем, можно получить сумматор для арифметического сложения N-разрядных двоичных чисел.

Пример 2. Минимизировать переключательную функцию и получить схему ее реализации на основе функции И-НЕ.

$$Y = \overline{x_1}\,\overline{x_2}\,\overline{x_3} + \overline{x_1}x_2x_3 + \overline{x_1}\,\overline{x_2}x_3 + \overline{x_1}x_2(x_3 + \overline{x_1}x_2x_3) = \overline{x_1}\,\overline{x_2}\,\overline{x_3} + \overline{x_1}x_3(x_2 + \overline{x_2}) + \overline{x_1}x_2x_3(1 + \overline{x_1}x_2) =$$

$$= \overline{x_1}\,\overline{x_2}\,\overline{x_3} + \overline{x_1}x_3 \bullet 1 + \overline{x_1}x_2x_3 \bullet 1 = \overline{x_1}\,\overline{x_2}\,\overline{x_3} + \overline{x_1}x_3 + \overline{x_1}x_2x_3 = \overline{x_1}\,\overline{x_2}\,\overline{x_3} + \overline{x_1}x_3(1 + x_2) =$$

$$= \overline{x_1}\,\overline{x_2}\,\overline{x_3} + \overline{x_1}x_3 \bullet 1 = \overline{x_1}\,\overline{x_2}\,\overline{x_3} + \overline{x_1}x_3 = \overline{x_1}(x_3 + \overline{x_2}x_3) = \overline{x_1}\,\overline{(x_3 + \overline{x_2}x_3)} = \overline{x_1} \bullet (\overline{x_3} \bullet \overline{x_2} \bullet \overline{x_3})$$



ЭВМ КАК СРЕДСТВО ОБРАБОТКИ ИНФОРМАЦИИ

1. Понятие архитектуры ЭВМ

ЭВМ – многофункциональное электронное устройство, предназначенное для накопления, обработки и передачи информации. Применительно к обработке информации ЭВМ рассматривается как исполнитель алгоритма. Для автоматического выполнения алгоритма исполнитель должен:

- Помнить алгоритм и обрабатываемые данные;
- Уметь выполнять команды, заданные в алгоритме.

Электронные устройства, обладающие этими качествами, имеются в любой ЭВМ. Это основная память и процессор, образующие центральную часть ЭВМ.

Каждая ЭВМ имеет определенную архитектуру. Под архитектурой ЭВМ понимается ее структура, логическая организация и ресурсы, т.е. те средства, которые могут быть выделены процессу обработки данных на определенный интервал времени.

Под структурой ЭВМ понимается состав основных устройств ЭВМ, их взаимное соединение и информационные связи между устройствами.

Логическая организация ЭВМ определяет организацию основной памяти, систему команд, принцип организации процесса обработки данных.

Принципиальные отличия архитектуре проявляются в логической организации. Центральная часть большинства современных ЭВМ имеет логическую организацию ОКОД (одиночный поток команд – одиночный поток данных). При этой организации процесс обработки данных рассматривается как процесс выполнения процессором потока команд над потоком данных.

Принципы такой логической организации сформулированы Джоном фон Нейманом и могут быть сведены к следующим положениям:

- Принцип программного управления: программа состоит из набора команд, которые выполняются процессором друг за другом в последовательности, заданной алгоритмом.
- Принцип однородности основной памяти: программы и данные хранятся в одной и той же памяти. Ни команды, ни данные не имеют признаков, по которым их можно отличить друг от друга в месте хранения. Над командами можно выполнять те же действия, что и над данными.
- Принцип адресности: основная память состоит из дискретных элементов – пронумерованных ячеек памяти.

ЭВМ, построенные на этих принципах, имеют так называемую классическую архитектуру. Для ЭВМ классической архитектуры структура и логическая организация рассматриваются более подробно в следующих пунктах.

2.Типовая структура ЭВМ

Состав устройств ЭВМ во многом определяется рассмотренными принципами. Отличия в структуре проявляются в основном в соединении устройств и информационных связях между ними. Типовая структура ЭВМ приведена на Рис.1.

Основная память предназначена для хранения программы, входных, выходных и промежуточных данных. Данные и команды хранятся в виде двоичных кодов. Основная память энергозависима, после выключения

ЭВМ информация в основной памяти теряется. При включении ЭВМ основная память, как правило, не обнуляется. Поэтому в ячейках памяти, не задействованных под размещение программы и определенных в программе данных, будут содержаться случайные двоичные коды.

Процессор предназначен для автоматического выполнения команд в порядке, заданном алгоритмом. Процессору доступны для обработки только те данные, которые хранятся в основной памяти.

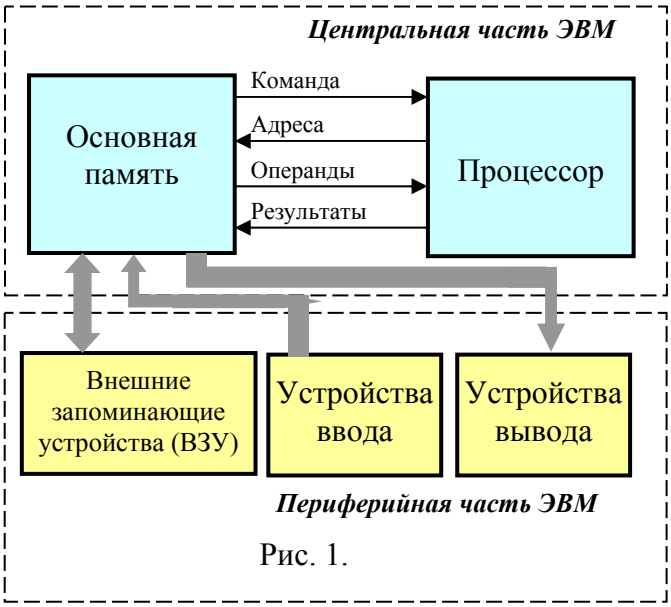


Рис. 1.

Для обеспечения связи центральной части с источниками и потребителями данных ЭВМ имеет *внешние устройства*, образующих периферийную часть. Все внешние устройства делятся на внешние запоминающие устройства (ВЗУ) и устройства ввода-вывода (УВВ).

Передача данных из основной памяти во внешние устройства называется выводом данных, а передача данных из внешних устройств в основную память – вводом данных.

Управление обменом данными между основной памятью и внешними устройствами в простейшем случае возлагается на процессор. В ЭВМ с более развитой архитектурой управление обменом возлагается на дополнительный специализированный процессор. В этом случае процессор называют центральным процессором, а специализированный процессор – процессором ввода-вывода.

Внешние запоминающие устройства обеспечивают длительное хранение программ и данных. Наиболее широко распространены ВЗУ на основе магнитных и оптических носителей информации: магнитные диски, ленты, барабаны, компакт-диски

Устройства ввода-вывода предназначены для обмена данными между центральной частью и пользователями. Пользователем может быть человек или техническое устройство.

Обмен данными с техническими устройствами, подключаемыми к ЭВМ, производится, как правило, в том виде, в каком они хранятся в основной памяти.

При взаимодействии с человеком данные преобразуются к форме, привычной для человека. Устройства ввода принимают от человека информацию в алфавитно-цифровом или графическом виде и обеспечивают их преобразование и передачу в центральную часть. В качестве устройств ввода широко применяются клавиатура, знакокоординатные устройства, устройства тактильного ввода, электронные перья и т.д.

Устройства вывода обеспечивают преобразование данных к алфавитно-цифровому или графическому виду и выдачу их человеку. Наиболее используемыми устройствами вывода являются дисплеи на базе электронно-лучевых трубок и жидкокристаллические дисплеи. Кроме того, широко используются принтеры, графопостроители, графические планшеты, синтезаторы звука и т.д.

3. Логическая организация основной памяти

Каждая ячейка имеет свой номер, называемый *исполнительным адресом (АИ)* или *абсолютным адресом (АА)*. Для записи в память или считывания из памяти требуется указать исполнительный адрес ячейки памяти. Время записи и считывания для всех ячеек одинаково и не зависит от их исполнительного адреса.

Ячейка имеет фиксированный размер. Стандартным размером ячейки является *байт – 8 бит*. Следовательно, байт – наименьший объем информации, который может быть адресован в основной памяти. Во многих ЭВМ процессор способен одновременно получать из основной памяти, обрабатывать и записывать несколько байт. Поле из нескольких байт, одновременно обрабатываемых процессором, называют *машинным словом.* Адресом слова является адрес младшего байта.

Для указания объема памяти используют более крупные единицы информации:

- 1 Кб (килобайт) – 1024 байта;
- 1 Мб (мегабайт) – 1024 Кб;
- 1 Гб (гигабайт) – 1024 Мб.

4. Программная модель процессора

Выполнение программы сводится к выполнению машинных команд в заданной последовательности. Каждая команда *прямо или косвенно* должна однозначно определять следующую информацию::

- Операцию, которую процессор должен выполнить над данными. Данные, участвующие в операции и результат операции принято называть *операндами*. Процессор может выполнять строго определенный набор операций. Каждой операции присвоен свой уникальный номер, называемый *кодом операции (КОП).*
- Исполнительные адреса операндов.
- Адрес следующей команды.

Для указания этой информации каждой команде при разработке процессора назначается свой формат. Формат команды определяет ее разбиение на отдельные смысловые поля, длину этих полей и способ кодирования информации в каждом из полей.

В предельном случае команда должна содержать пять полей, показанных на Рис.2. Поле КОП определяет код операции, а остальные четыре поля – адреса операндов и следующей команды

Процессор содержит два основных устройства, показанных на Рис.3, на которые возлагается выполнение команд в заданной последовательности.

Устройство управления (УУ) предназначено для декодирования команды, которое заключается в извлечении информации из полей команды в соответствии с форматом, определения адресов операндов и адреса следующей команды.

Операционное устройство (ОУ) предназначено для выполнения над операндами операции, предписанной полем КОП.

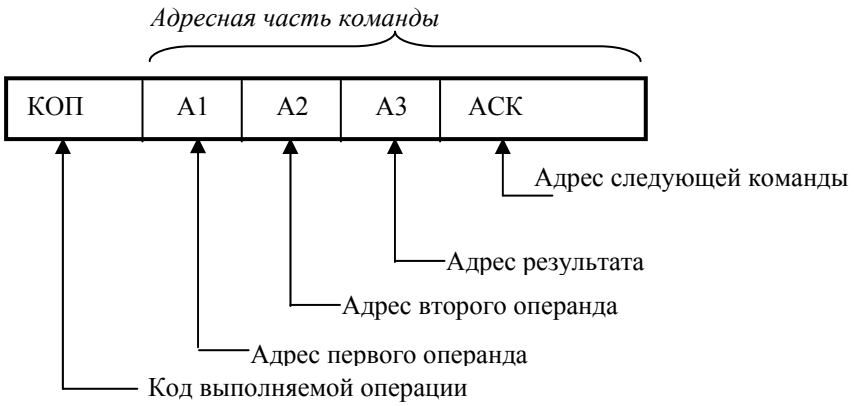


Рис. 2.

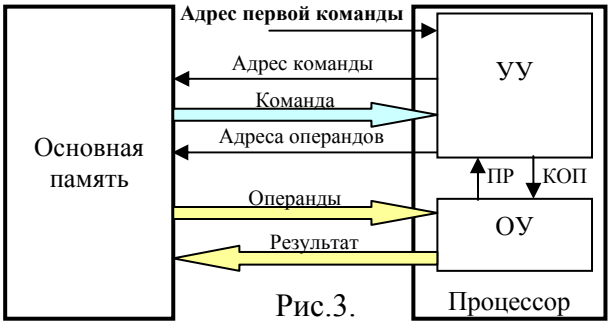


Рис.3.

Рассмотрим основные фазы выполнения машинной программы.

1. Для запуска программы в устройство управления заносится адрес первой команды.
2. По адресу команды из памяти считывается содержимое ячейки памяти и передается в устройство управления.
3. Устройство управления выделяет поля команды.
4. На основании адресной части определяются адреса операндов.
5. По адресам операндов считывается содержимое указанных ячеек памяти и направляется в операционное устройство.
6. Операционное устройство выполняет заданную операцию.
7. Результат операции сохраняется в ячейке памяти по указанному адресу.
8. Устройство управления определяет адрес следующей команды.

Рассмотренные действия повторяются с пункта 2 до выполнения специальной команды СТОП, предписывающей завершение программы.

Помимо результата многие команды формируют признак результата ПР. Типичными признаками результата являются:

- Результат равен нулю
- Результат меньше нуля
- Результат больше нуля

В систему команд могут включаться специальные команды сравнения, основным назначением которых является формирование признака результата. Типичными признаками для команд сравнения являются признаки:

- Операнды равны по значению
- Значение первого операнда меньше значения второго операнда
- Значение первого операнда больше значения второго операнда.

Сформированный признак результата может анализироваться специальными командами управления для принятия решения в процессе вычисления адреса следующей команды в ветвящихся алгоритмах.

Процессор может иметь свою собственную память, используемую для кратковременного хранения информации. Она состоит из специальных электронных узлов – *регистров*. Разрядность регистров обычно совпадает с разрядностью машинного слова. Регистры доступны программе и могут использоваться ею. для хранения промежуточных результатов. Такого рода регистры называются регистрами общего назначения (РОН). Каждый регистр имеет номер, который может указываться в качестве адреса в команде. Адреса регистров не совпадают с адресами основной памяти.

По сравнению с основной памятью, объем которой может достигать нескольких Гб, объем регистровой памяти невелик, порядка десятков РОН.. Время же считывания и записи информации для РОН на несколько порядков меньше, чем для ячейки основной памяти.

Основная память может быть распределена для одновременного размещения в ней нескольких программ. Регистры же выделяются в монопольное использование выполняющейся в данный момент программы

Команды с четырьмя адресными полями наиболее удобны с точки зрения программиста, но реализовать команды такого формата в аппаратуре процессора при ограниченной длине машинного слова практически невозможно.

Рассмотрим пример. Пусть ЭВМ имеет основную память объемом 1Мб, а процессор способен выполнять до двухсот различных операций, Длина каждого адресного поля составит 20 бит (2^{20} = 1Мб), а поля КОП – 8бит. Длина команды 8+4*20 =88 бит. С другой стороны, машинное слово современных процессоров, как правило, не превышает 32 бита. Современные ЭВМ имеют значительные объемы памяти, измеряемые сотнями мегабайт и гигабайтами, что еще более обостряет проблему.

Сокращение длины команды достигается двумя способами:

- Сокращение количества адресных полей.
- Сокращение длины адресных полей.

При первом способе некоторые адреса определяются по умолчанию (косвенно) и в команде не указываются. Сокращение количества адресных полей оборачивается увеличением количества команд в программе.

При втором способе в команде указывается не исполнительный адрес, а так называемый *логический адрес (АЛ)*. Логический адрес содержит информацию, на основе которой можно вычислить исполнительный адрес. Сокращение длины адресных полей в общем случае приводит к увеличению времени выполнения программы, т.к. необходимо дополнительные действия по вычислению адресов.

4. Особенности выполнения команд с сокращенным количеством адресных полей

Рассмотрим особенности выполнения команд с сокращенным количеством адресных полей на примере двухадресных команд, достаточно широко применяемых в современных ЭВМ. Состав полей двухадресной команды приведен на Рис.4.

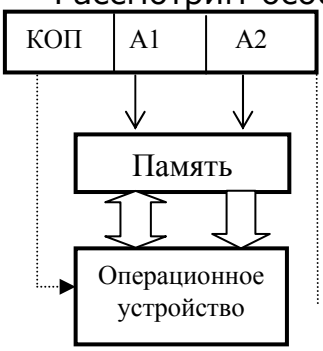


Рис. 4.

Исключение поля АСК компенсируется включением в состав устройства управления специализированного регистра – счетчика команд (СК).

При запуске программы в счетчик заносится адрес первой команды программы. После считывания команды из памяти значение счетчика увеличивается на ее длину. Таким образом, в счетчике окажется адрес следующей по порядку команды.

Такой прием позволяет выполнять команды в порядке их записи в программе. Такой порядок выполнения команд называют *естественным порядком*.

Для изменения естественного порядка команд в программу включаются специальные команды управления, которые принудительно изменяют текущее состояние счетчика путем занесения в него адреса перехода, т.е. адреса команды, которая должна быть выполнена следующей. После выполнения перехода естественный порядок выполнения программы возобновляется.

Исключение поля А3 компенсируется за счет записи результата по адресу одного из операндов, чаще всего первого. Формализованной записью схемы выполнения двухадресной команды будут выражения:

1. А1 $\Leftarrow$ (А1) $\otimes$ (А2)
2. СК $\Leftarrow$ (СК) + Lк,

где Lк – длина выполняемой команды;

знак $\Leftarrow$ обозначает запись по указанному слева от знака адресу данных, указанных справа от знака;

знак $\otimes$ обозначает операцию, которую необходимо выполнить.

Адрес, заключенный в круглые скобки, означает чтение данных, другими словами – содержимое ячейки памяти или регистра.

Если первый операнд нужен для дальнейших вычислений, придется выполнить предварительное копирование первого операнда в рабочую ячейку памяти или регистр общего назначения. Обозначим адрес этой ячейки как A_S. Т.о. придется выполнить следующие действия:

1. A_S $\Leftarrow$ (А1)
2. А1 $\Leftarrow$ (А1) $\otimes$ (А2).
3. СК $\Leftarrow$ (СК) + Lк

5. Понятие способа адресации

Способ адресации – это правила записи в команде логического адреса и правила вычисления на его основе исполнительного адреса. Применение того или иного способа адресации во многом зависит от требований к размещению программы в основной памяти.

Специализированные ЭВМ имеют фиксированный набор программ. Размещение программ в основной памяти можно спланировать заранее. Основной целью кодирования адресов является сокращение длины команды.

Набор программ для ЭВМ общего назначения заранее не известен. Предварительно спланировать размещение программ в памяти невозможно. Кодирование адресов должно обеспечить размещение программы в любом месте памяти без внесения изменений в программу. Это свойство называют *перемещаемостью* программы. Другими словами, изменение результата вычисления АИ=F(АЛ) должно достигаться без изменения логического адреса.

Дадим краткое описание наиболее используемых способов адресации.

Прямая адресация**.** В команде указывается исполнительный адрес. Правило декодирования: АИ = АЛ. Время вычисления исполнительного адреса минимально. Перемещаемость программ не поддерживается. Длина логического адреса максимальна. Применяется для адресации данных в регистрах общего назначения

Косвенная адресация. В команде указывается адрес, по которому хранится исполнительный адрес. Правило декодирования: АИ = (АЛ). Поддерживается перемещаемость программ. При хранении исполнительного адреса в основной памяти длина логического адреса максимальна. Для получения исполнительного адреса необходимо дополнительное обращение в память. При хранении исполнительного адреса в регистре общего назначения сокращается длина логического адреса и время получения исполнительного адреса.

Относительная адресация

Относительная адресация в различных вариантах реализации является основной в ЭВМ общего значения и изначально ориентирована на поддержку перемещаемости программ. Суть относительной адресации показана на рисунке 5.

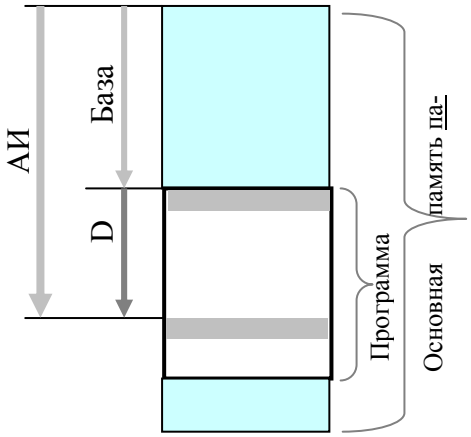


Рис.5.

При составлении каждой программы отсчет логических адресов производится с нуля. При размещении программы в выделенной ей памяти для программы назначается база. База представляет собой исполнительный адрес начала выделенной области памяти.

При вычислении исполнительного адреса логический адрес рассматривается как смещение D от базы АИ = База + D.

Изменение базы при неизменных логических адресах в командах позволяет "привязать" программу к любому месту основной памяти.

Обычно значение базы устанавливается при загрузке программы в основную память и хранится либо в специальном программно недоступном регистре или одном из регистров общего назначения.

Разновидностью относительной адресации является индексная адресация:

АИ = База + Индекс + D.

Значение индекса хранится в специальном регистре индекса или одном из регистров общего назначения. Индекс позволяет обрабатывать данные в последовательно расположенных ячейках памяти без изменения команды. Для изменения исполнительного адреса достаточно изменить значение индекса.

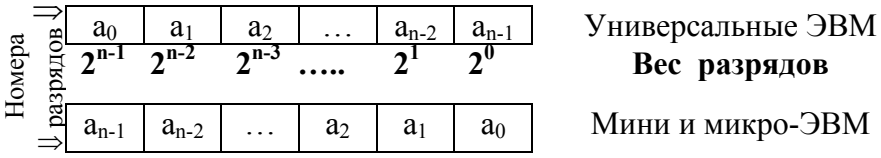
Непосредственная адресация

Операнд записывается непосредственно в команде на месте логического адреса. Декодирование логического адреса не требуется, так как Операнд=АЛ. Непосредственная адресация применяется для записи констант.

КОДЫ ДВОИЧНЫХ ЧИСЕЛ

1. Разрядная сетка

Все данные в центральной части ЭВМ хранятся и обрабатываются в виде двоичных кодов. Для размещения кода отводится строго фиксированное число двоичных разрядов, образующих разрядную сетку. Без привязки к разрядной сетке понятие кода не определено. Каждый разряд кода нумеруется, начиная с нуля. В разных ЭВМ принята разная система нумерации. В универсальных ЭВМ разряды нумеруются слева направо, а в мини и микро-ЭВМ – справа налево. Пример нумерации для n двоичных разрядов приведен ниже.



При любой системе нумерации левый разряд имеет наибольший вес, а правый – наименьший. Положение точки, отделяющей целую и дробную часть, подразумевается. В большинстве ЭВМ считается, что точка находится после младшего разряда. В этом случае в разрядной сетке размещаются коды целых чисел.

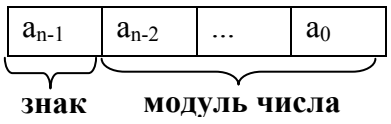
Наиболее просто кодируются целые числа без знака. В этом случае в разрядную сетку записывается двоичное представление числа. Каждая цифра числа занимает разряд в соответствии со своим весом.

Для кодирования целых чисел со знаком применяют:

- Прямой код.
- Обратный код.
- Дополнительный код.

2. Прямой код

Прямой n-разрядный двоичный код состоит из знакового разряда и n-1 разрядов для записи модуля числа. В качестве знакового разряда используется старший разряд.



Знак "плюс" кодируется как **0,** а знак "минус" – как **1**.

Определим диапазон чисел, представимых в прямом n-разрядном двоичном коде. Введем следующие обозначения: A^+max – максимальное положительное число, A^-min - минимальное отрицательное число:

$$\mathbf{A^+max = 2^{n-1} - 1; \quad A^-min = -(2^{n-1}-1)}$$

Пример: Получить прямой 5-ти разрядный код числа $A_{[10]} = -6$. Определить диапазон чисел для этой разрядной сетки.

1.$|A_{[10]}| = 6 \Rightarrow |A_{[2]}| = 0110$ (на запись модуля выделено 4 разряда)
2.Знаковый разряд = **1**
3.$(A_{[2]})_{пр}$ = | **1** | 0 | 1 | 1 | 0 |

Определим диапазон чисел:

$A^+max = 2^4 - 1 = 15 \quad A^-max = -(2^4 - 1) = -15$

Пример: Получить прямой 8-ми разрядный код числа $A_{[16]}$ = 3C. Определить диапазон чисел для этой разрядной сетки.

1.$| A_{[16]} | = 3C \Rightarrow | A_{[2]} | = 0111100$ (на запись модуля выделено 7 разрядов)

2.Знаковый разряд = **0**

3.$(A_{[2]})_{пр}$ = | **0** | 0 | 1 | 1 | 1 | 1 | 0 | 0 |

Определим диапазон чисел:

$A^+max = 2^7 - 1 = 127 \qquad A^-max = -(2^7 - 1) = -127$

Прямой код для кодирования целых чисел со знаком в ЭВМ практически не применяется и используется как составная часть более сложных кодов.

Недостатки:

1. Сложение и вычитание чисел в прямом коде выполняется по правилам алгебры:

   $(C)_{пр} = (A)_{пр} + (B)_{пр}$

   $(C)_{пр} = (A)_{пр} - (B)_{пр}$,

   поэтому процессор должен иметь два разных устройства: сумматор и вычитатель.
2. Знаковый разряд обрабатывается отдельно от остальных по особым правилам.
3. Неоднозначность кодировки нуля:

   +0 | 0 | 0 | 0 | 0 | 0 | при n=5

   -0 | 1 | 0 | 0 | 0 | 0 | при n=5

3. Обратный код

Обратный код предполагает, что при кодировании числа и выполнении операций с кодами используются одинаковые правила обработки всех разрядов, включая знаковый. Обратный n-разрядный двоичный код – это код образованный по правилу:

$$(A_2)_{обр} = \begin{cases} A_2 & \text{при } A_2 > 0 \\ (2^n - 1) - |A_2| & \text{при } A_2 < 0 \end{cases}$$

Операция вида $(2^n - 1) - |A_2|$ дает дополнение до наибольшего числа без знака, представимого в в n-разрядной сетке. Другими словами, кодирование числа можно выполнить следующим образом:

- Для положительного числа в n-разрядную сетку записывается его двоичное представление.
- Для отрицательного числа в n-разрядную сетку записывается разность между наибольшим числом без знака и модулем кодируемого числа.

В обоих случаях значение знакового разряда будет получено автоматически.

Пример: Получить обратные коды чисел $A_{[10]} = 4$, $B_{[10]} = -6$ для n = 5.

Кодируем отрицательное число

1. $|B_{[10]}| = 6$
2. $|B_{[2]}| = 00110$
3. $(B_{[2]})обр$= 1 1 1 1 1 — Это инверсия модуля

   − 0 0 1 1 0

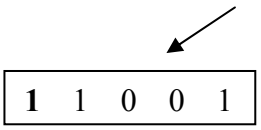
   1 1 0 0 1

   $(B_{[2]})обр$= *Знак* $\overset{32}{\rightarrow}$ ***1***

Кодируем положительное число

1. $A_{[10]} = 4$
2. $A_{[2]} = 00100$
3. $(A_{[2]})обр=$ | **0** 0 1 0 0 |

Знак → ***0***



Замечания:

- Операция дополнения для двоичного обратного кода эквивалентна инверсии всех разрядов. Инверсия удобна для реализации в операционном устройстве.
- Если длина разрядной сетки кратна 4, дополнение удобно вычислять в шестнадцатеричной системе счисления. Такая разрядная сетка характерна для большинства ЭВМ.
- Если длина разрядной сетки кратна 3, дополнение удобно вычислять в восьмеричной системе счисления. Такая разрядная сетка не характерна для ЭВМ. Однако для мини-ЭВМ с 16-ти разрядной сеткой при документировании программ часто пользуются восьмеричной записью кодов чисел, при этом старший разряд рассматривается как двоичная цифра.

Пример: Определить обратный код числа $A_{[10]} = -21$ при n=16.

Получим обратный код во всех трех указанных формах. Так как кодируется отрицательное число, необходимо вычислить дополнение и использовать его в качестве обратного кода.

- Двоичное кодирование.

1. $|A_{[10]}| = 21$
2. $|A_{[2]}| = 10101$
3. $|A_{[2]}| = 0000\ 0000\ 0001\ 0101$ (Это модуль кодируемого числа в разрядной сетке)
4. $(A_{[2]})обр=$ $\underbrace{\mathbf{1}111}_{\mathbf{F}}\ \underbrace{1111}_{\mathbf{F}}\ \underbrace{1110}_{\mathbf{E}}\ \underbrace{1010}_{\mathbf{A_{[16]}}}$

- Шестнадцатеричное кодирование

1. $|A_{[10]}| = 21$
   $|A_{[16]}| = 15$
2. $(A_{[16]})обр =$ F F F F (Это максимальное число без знака)
   - 0 0 1 5 (Это модуль кодируемого числа в разрядной сетке)
   F F E A

- Восьмеричное кодирование

1. $|A_{[10]}| = 21$
   $|A_{[8]}| = 25$
2. $(A_{[8]})обр =$ 1 7 7 7 7 7 (Это максимальное число без знака)
   - 0 0 0 0 2 5 (Это модуль кодируемого числа в разрядной сетке)
   1 7 7 7 5 2

Диапазон чисел в обратном коде составляет: $\mathbf{A^+max = 2^{n-1} - 1;\quad A^-min = -(2^{n-1}-1)}$

Достоинства:

1 Сложение и вычитание можно выполнять на одном устройстве: сумматоре, поскольку вычитание может быть заменено сложением с дополнением вычитаемого:

(C)обр = (A)обр + (B)обр

(C)обр = (A)обр - (B)обр = (A)обр + ((B)обр)обр

2 Знаковый разряд обрабатывается по общим правилам. Его значение при выполнении операции устанавливается автоматически.

Недостатки:

Неоднозначность кодировки нуля:

+0 | 0 | 0 | 0 | 0 | 0 | при n=5

-0 | 1 | 1 | 1 | 1 | 1 | при n=5

Чаще всего в ЭВМ используется модификация обратного кода – дополнительный код.

4. Дополнительный код

Дополнительный код обладает всеми достоинствами обратного и устраняет его главный недостаток – неоднозначность кодирования нуля. Дополнительный n-разрядный двоичный код представляет собой код, образованный по правилу:

$$(A_2)_{доп} = \begin{cases} A_2 & \text{при } A_2 > 0 \\ 2^n - |A_2| & \text{при } A_2 < 0 \end{cases}$$

Операция $2^n - |A_2|$ называется операцией дополнения до следующего за большим числом без знака для данной разрядной сетки. Другими словами, кодирование числа можно выполнить следующим образом:

- Для положительного числа в n-разрядную сетку занести его двоичное представление.
- Для отрицательного числа в n-разрядную сетку занести разность между следующим

  за большим числом без знака и модулем кодируемого числа.

Дополнение при использовании дополнительного кода больше по значению на 1 от дополнения при использовании обратного кода:

$$\underbrace{(2^n - 1) - |A_2|}_{\text{Дополнение при обратном коде}} + 1 = \underbrace{2^n - |A_2|}_{\text{Дополнение при дополнительном коде}}$$

Следствие: дополнительный код отрицательного числа может быть получен из его обратного кода путем прибавления единицы в младший разряд.

Пример: Определить дополнительный код числа $A_{[16]} = -1D8$ при n=16.

Кодируется отрицательное число, следовательно, необходимо вычислить дополнение и использовать его в качестве кода.

❖ Двоичное кодирование.

1. $|A_{[16]}| = 1D8$
2. $|A_{[2]}| = 111011000$
3. $|A_{[2]}| = 0000\ 0001\ 1101\ 1000$ (Это модуль кодируемого числа в разрядной сетке)
4. $(A_{[2]})обр = \mathbf{1}111\ 1110\ 0010\ 0111$ (Это обратный код)

$$+ \qquad\qquad 1$$

$(A_{[2]})доп = \underbrace{\mathbf{1}111}_{\mathbf{F}}\ \underbrace{1110}_{\mathbf{E}}\ \underbrace{0010}_{\mathbf{2}}\ \underbrace{1000}_{\mathbf{8}_{[16]}}$ (Это дополнительный код)

❖ Шестнадцатеричное кодирование

1. $|A_{[16]}| = 1D8$
2. $(A_{[16]})обр$ F F F F (Это максимальное число без знака)

   \- 0 1 D 8 (Это модуль кодируемого числа в разрядной сетке)

   F E 2 7 (Это обратный код)

$+\quad 1$

$(A_{[16]})$доп= F E 2 8 (Это дополнительный код)

❖ Восьмеричное кодирование

1. $|A_{[8]}| = 730$
2. $(A_{[8]})$обр = 1 7 7 7 7 7 (Это максимальное число без знака)
   $-\underline{0\ 0\ 0\ 7\ 3\ 0}$ (Это модуль кодируемого числа в разрядной сетке)
   1 7 7 0 4 7 (Это обратный код)
   $+\quad 1$
   $(A_{[8]})$доп= 1 7 7 0 5 0 (Это дополнительный код)

Дополнительный код обеспечивает однозначное представление нуля:

+0 | 0 | 0 | 0 | 0 | 0 | при n=5

-0 | 1 | 1 | 1 | 1 | 1 | при n=5

\+ 1

| 0 | 0 | 0 | 0 | 0 |

Диапазон представимых чисел в дополнительном коде: $\mathbf{A^+max = 2^{n-1} - 1;\ A^-min = -2^{n-1}}$

5. Операции сложения и вычитания в дополнительном коде.

Операции сложения и вычитания производятся путем арифметического сложения кодов чисел. При вычитании перед выполнением операции для вычитаемого берется дополнение. Результат операции представлен в дополнительном коде.

(C)доп = (A)доп + (B)доп

(C)доп = (A)доп - (B)доп = (A)доп + $((B)$доп$)$доп

Пример: вычислить C=A+B и D=A-B. Исходные числа $A_{[10]}=3$, $B_{[10]}= -4$. Сетка n=5.

Определим дополнительный код для A и B.

1. $A_{[10]} = 3$ 1. $|B_{[10]}| = 4$,
   $A_{[2]} = 11$ $|B_{[2]}| = 100$
2. $(A_{[2]})$доп = 00011 2. $(B_{[2]})$доп = 00100
   11011
   $+\quad 1$
   11100

Выполним сложение (A) + (B):
00011
+11100

Наличие единицы в знаковом разряде результата свидетельствует о том, что результат представляет собой код отрицательного числа. Определим десятичный эквивалент результата. Для этого возьмем дополнение для результата.

11111
(Инверсия) $\rightarrow$00000
$+\quad 1$
00001 $\rightarrow 1_{[10]}$ Результат равен **–1**.

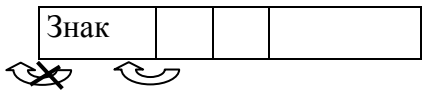
Выполним вычитание (A) - (B):
Берем дополнение от $(B_{[2]})$доп
$((B_{[2]})$доп$)$доп = 11100
(Инверсия) $\rightarrow$ 00011
+ 1
00100
Сложим
$(A_{[2]})$доп+$(B_{[2]})$доп$)$доп
00011
+00100

Наличие нуля в знаковом разряде результата свидетельствует о том, что результат представляет собой код положительного числа. Определим десятичный эквивалент результата. Для этого достаточно преоб-

разовать двоичный результат в десятичное число. Результат равен +7.

Т.к. разрядная сетка ограничена, то при сложении кодов одинакового знака значение результата может выйти за границы диапазона чисел в данной разрядной сетке. Такая ситуация называется переполнением и обнаруживается по одному из признаков:

1. Знаковые разряды слагаемых одинаковы, а знак результата противоположный.
2. Наличие переноса в знаковый разряд и отсутствие переноса из знакового разряда.



Этот признак легко реализуется аппаратным способом в сумматоре. Обычно при возникновении переполнении формируется специальный сигнал. Однако во многих ЭВМ этот сигнал игнорируется. В этом случае контроль диапазонов данных и результата возлагается на программиста.

Пример. Даны числа $A_{[10]}$=342 и $B_{[10]}$= -430. Длина разрядной сетки 16. Представить числа в дополнительном коде и выполнить операции сложения A+B и вычитания A-B. Получить десятичные эквиваленты результатов операций.

$A_{[10]}$= 342, следовательно $A_{[16]}$= 156 и $A_{[8]}$= 526

$B_{[10]}$= -430, следовательно $B_{[16]}$= -1AE и $B_{[8]}$= -656

❖ Шестнадцатеричное кодирование

Определим дополнительные коды чисел и заранее подготовим дополнение для кода вычитаемого:

$(A_{[16]})$доп=0156

$(B_{[16]})$доп=FFFF
-01AE
FE51
+ 1
FE52

$((B_{[16]})$доп$)$доп=FFFF
- FE52
01AD
+ 1
01AE

Сложение (А)доп+(В)доп

0156
+FE52
FFA8

Старший двоичный разряд результата равен единице. Результат соответствует отрицательному числу. Берем дополнение для него:

FFFF
- FFA8
0057
+ 1
0058

Результат= (80+8)= 88

Вычитание, т.е. (А)доп+((В)доп)доп

015 6
+01AE
03 04

Старший двоичный разряд результата равен нулю. Результат соответствует положительному числу.

Результат= (3*256+4)=772$_{[10]}$

❖ Восьмеричное кодирование

$(A_{[8]})$доп=000526 $(B_{[8]})$доп=177777 $((B_{[8]})$доп)доп= 177777

-000656 - 177122

177121 000655

\+ 1 + 1

177122 000656

Сложение (А)доп+(В)доп

000526
+177122
177650

Старший двоичный разряд результата равен единице. Результат соответствует отрицательному числу. Берем дополнение для него:

177777
- 177650
000127
+ 1
000130

Результат= (64+24)= 88

Вычитание, т.е. (А)доп+((В)доп)доп

000526
+000656
001404

Старший двоичный разряд результата равен нулю. Результат соответствует положительному числу.

Результат= (1*512+4*64+4)=772$_{[10]}$

Пример. В ЭВМ выполнена операция сложения A+B=R. Числа представлены в дополнительном коде. Длина разрядной сетки 16. Код первого слагаемого $(A_{[16]})$доп=006D, а код результата $(R_{[16]})$доп=005E. Определить код второго слагаемого и десятичные эквиваленты слагаемых и результата.

(R)доп=(A)доп+(B)доп, отсюда (B)доп=(R)доп-(A)доп или

(B)доп=(R)доп+((A)доп))доп.

$((A_{[16]})$доп)доп=FFFF

-006D

FF92

\+ 1

FF93

Коды первого слагаемого и результата соответствуют положительным числам.

$A_{[10]} = 6 \cdot 16^1 + 13 \cdot 16^0 = 109$

$R_{[10]} = 5 \cdot 16^1 + 14 \cdot 16^0 = 94$

Сложим . (R)доп+((A)доп)доп
005E
+FF93
FFF1
Старший двоичный разряд кода B равен единице. Слагаемое B соответствует отрицательному числу. Берем дополнение для него:
FFFF
- FFF1
000E
+ 1
000F
Слагаемое B = -15

6. Двоично-десятичный код

В некоторых универсальных ЭВМ для кодирования целых чисел применяется смешанный двоично-десятичный код. Своим появлением двоично-десятичный код обязан стремлению производить запись и обработку целых чисел в привычной для человека десятичной системе счисления. Десятичные цифры в этом коде представлены четырехразрядными двоичными числами.

Для кодировки знака используется дополнительная группа бит, расположенная за младшей десятичной цифрой. Знак положительных чисел кодируется как "**1100**" (**C**), а знак отрицательных чисел как "**1101**" (**D**). Суммарная длина кода должна быть кратна байту. Если длина полученного кода не кратна байту, то перед старшей десятичной цифрой числа добавляется незначащий десятичный ноль.

Пример: Записать число $A_{10} = 951$ в двоично-десятичном коде. Код представить в двоичной и шестнадцатеричной форме.

9	**5**	**1**	**C**
цифра	цифра	цифра	знак
1001	0101	0001	1100

Пример: Записать число $A_{10} = -87$ в двоично-десятичном коде. Код представить в двоичной и шестнадцатеричной форме.

0	**8**	**7**	**D**
цифра	цифра	цифра	Знак
0000	1000	0111	1101

ПРЕДСТАВЛЕНИЕ ДАННЫХ В ЦЕНТРАЛЬНОЙ ЧАСТИ ЭВМ

1.Понятие формата данных

ЭВМ оперирует данными, представленными в виде двоичных кодов. Каждому типу данных соответствует свой формат представления (в дальнейшем просто формат). Базовыми типами данных являются числовые данные. Формат однозначно определяет количество разрядов в коде числа и правила кодирования данных. Числовые типы данных делятся на две группы: целые числа и вещественные числа.

2. Представление целых чисел

Для представления целых чисел используется формат с фиксированной точкой и двоично-десятичный формат.

Формат с фиксированной точкой. Разрядная сетка имеет строго фиксированное число разрядов. Типичные разрядные сетки: 1байт, 2 байта, 4байта. Положение точки подразумевается справа от младшего разряда. Формат с фиксированной точкой используется для кодирования целых чисел без знака и целых чисел со знаком.

Целые числа без знака. В разрядную сетку записывается двоичное представление числа. Диапазон целых чисел без знака для типичных разрядных сеток показан в таблице 1.

Таблица 1

Разрядность, (байт)	Диапазон значений	
	Минимальное значение	Максимальное значение
1	0	255
2	0	65 535
4	0	4 294 967 295

Целые числа со знаком. В разрядную сетку записывается дополнительный код числа. Диапазон целых чисел со знаком для типичных разрядных сеток показан в таблице 2.

Таблица 2

Разрядность, (байт)	Диапазон значений	
	Минимальное значение	Максимальное значение
1	-128	127
2	-32 768	32 767
4	-2 147 483 648	2 147 483 647

Формат с фиксированной точкой является основным форматом представления целых чисел в ЭВМ.

Двоично-десятичный формат. Количество разрядов заранее не фиксируется и зависит от количества десятичных цифр в кодируемом числе. Данный формат соответствует двоично-десятичному коду числа. В силу сложности и трудоемкости выполнения арифметических операций двоично-десятичный формат используется редко и в дальнейшем не рассматривается.

3. Представление вещественных чисел

Для представления вещественных чисел используется формат с плавающей точкой. Вещественные числа всегда рассматриваются как числа со знаком. Разрядная сетка имеет строго фиксированное число разрядов.

В основе формата лежит представление чисел в экспоненциальной (научной) форме. Число в этой форме записывается в виде мантиссы и порядка:

$$\mathbf{A = \pm M \bullet S^{\pm P}},$$

где **M** – мантисса,

S – основание системы счисления,

p – порядок.

Пример. $A_{[10]} = 0.0225 \bullet 10^3 = \underline{0.225 \bullet 10^2} = 2.25 \bullet 10^1 = 22.5 \bullet 10^0 = 225 \bullet 10^{-1} = 22.5$

В формате с плавающей точкой кодируется: мантисса, порядок, знак числа и знак порядка. На каждую из кодируемых компонент отводится строго фиксированное число разрядов. Основание системы счисления заранее оговаривается и не кодируется.

Фиксированная по длине мантисса позволяет закодировать ограниченное количество значащих цифр. Поэтому вещественные числа в общем случае хранятся и обрабатываются с определенной степенью точности. Точность определяется способом кодировки мантиссы, а диапазон чисел – способом кодировки порядка.

Способ кодировки мантиссы и порядка в различных ЭВМ может быть разным. Отметим общие подходы при кодировании вещественных чисел.

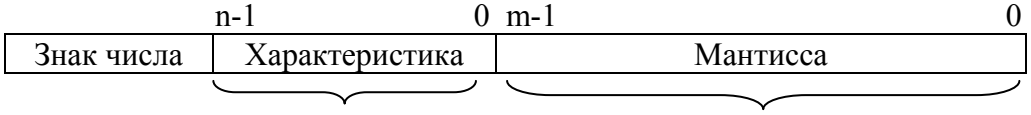
Кодируемое число предварительно нормализуется. За счет сдвига мантиссы и изменения порядка добиваются выполнения условия

$$S^{-1} \le |M| < 1. \qquad (1)$$

В рассмотренном примере этому условию соответствует запись числа в виде $A_{[10]} = 0.225 \bullet 10^2$.

В нормализованном числе целая часть мантиссы будет нулем, а старшая цифра дробной части будет отличной от нуля. Этот прием позволяет отказаться от хранения целой части и закодировать в мантиссе максимально возможное количество значащих цифр, то есть добиться максимальной точности. Мантисса записывается в прямом коде, причем знак мантиссы (он же знак числа) заносится в старший разряд. Знак положительного числа кодируется как **0**, а знак отрицательного – как **1**.

Порядок кодируется в виде так называемой характеристики. Характеристика Z представляет собой смещенный порядок: Z = P + D. Величина смещения D подбирается так, чтобы характеристика для минимального порядка была равна нулю. Этот прием позволяет использовать в качестве характеристики число без знака и не кодировать специально знак порядка. Обобщенный вид формата с плавающей точкой показан на рисунке:



Типичные для персональных ЭВМ варианты формата с плавающей точкой приведены в таблице 3.

Таблица 3

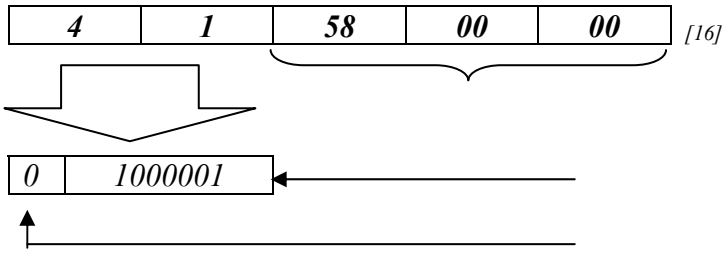
Длина формата (бит)	Длина Мантиссы, (бит)	Длина характеристики, (бит)	Диапазон чисел по модулю в десятичном эквиваленте	Точность представления, (десятичные цифры)
32	23	8	$\approx \lvert 10^{-38}\rvert \div \lvert 10^{38}\rvert$	7
64	52	11	$\approx \lvert 10^{-308}\rvert \div \lvert 10^{308}\rvert$	15

Отличия в кодировании вещественных чисел для разных ЭВМ носят количественный и качественный характер. Отличия количественного характера проявляются в различной длине мантиссы и порядка, а качественные – в системе счисления, в которой выполняется нормализация и определение порядка. Чаще всего применяется шестнадцатеричная или двоичная нормализация.

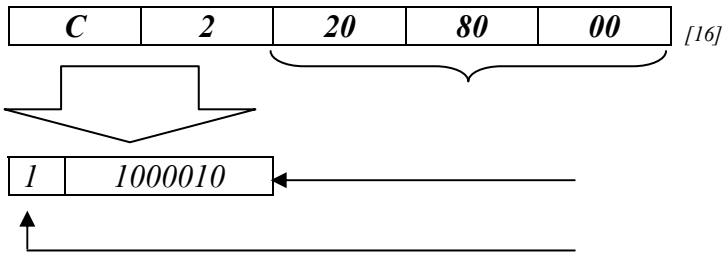
Шестнадцатеричная нормализация. Число нормализуется по рассмотренному выше общему правилу (1), чтобы $16^{-1} \le |M| < 1$. В этом случае старшая шестнадцатеричная цифра мантиссы отлична от нуля. Шестнадцатеричная нормализация обычно используется в универсальных ЭВМ.

Рассмотрим типичный вариант формата.

Длина формата – 32 разряда.
Длина мантиссы **m**=24.
Длина характеристики **n**=7.
Знак – 1 разряд.
Правила формирования характеристики: $Z_{[16]} = 40_{[16]} + P_{[16]}$.
Пример: $A_{[10]} = 5.5 \Rightarrow A_{[16]} = 5.8 \bullet 16^0$
Нормализуем $A_{[16]} = 0.58 \bullet 16^1$.
Получаем $\mathbf{Z}_{[16]} = 40+1 = 41$
$\mathbf{M}_{[16]} = 0.58$
Знак = 0 (+)



Пример: *$A_{[10]} = -32.5 \Rightarrow A_{[16]} = -20.8 \bullet 16^0$*
Нормализуем $A_{[16]} = -0.208 \bullet 16^2$
Получаем $\mathbf{Z}_{[16]} = 40 + 2 = 42$
$\mathbf{M}_{[16]} = 0.208$
***Знак** = 1 (-)*



Двоичная нормализация. Нормализация выполняется по правилу, несколько отличающемуся от общего: за счет сдвига мантиссы и изменения

порядка двоичного числа добиваются выполнения условия $1 \le |M| < 2$. В этом случае целая часть мантиссы будет содержать одну двоичную цифру, причем она будет равна единице. Обязательную единицу можно отбросить и в коде не хранить. Это позволяет использовать дополнительный разряд для хранения дробной части и тем самым увеличить точность представления числа. Двоичная нормализация обычно используется в мини и микро- ЭВМ. Рассмотрим типичный вариант формата.

Длина формата – 32 разряда.

Число разрядов мантиссы **m**=23.

Число разрядов характеристики **n**=8.

Знак – 1 разряд.

Правила формирования характеристики: $Z_{[2]} = 2^7 - 1 + P_{[2]}$ или $Z_{[10]} = 127_{[10]} + P_{[10]}$.

Пример: $A_{[10]} = 0.75 \Rightarrow A_{[2]} = 0.11 \bullet 2^0$

Нормализуем $A_{[2]} = 1.1 \bullet 2^{-1}$ и отбрасываем старшую единицу.

Получаем $\mathbf{Z}_{[10]} = 127\text{-}1 = 126$

$\mathbf{M}_{[2]}$ = .10000000000000000000000

Знак = 0 (+)

0	011 1111 0	100 0000 0000 0000 0000 0000
	3 F	**4 0 0 0 0 0**$_{[16]}$

При одинаковых количественных характеристиках (длине мантиссы и длине характеристики) шестнадцатеричная нормализация позволяет получить больший диапазон чисел, но имеет меньшую точность.

4. Представление символьных (текстовых) данных

Символьные данные не имеют специально разработанных для них форматов. При обработке текстов ЭВМ рассматривает символ как целое число без знака, представленное в формате с фиксированной точкой. Значение этого числа рассматривается как порядковый номер символа в таблице кодировки. В настоящее время используются таблицы кодировки, в которых на кодирование символа отведено 8 бит или 16 бит. Наибольшее распространение получили системы кодировки:

Кодировка EBCDIC (Expanded Binary Coded Decimal Interchange Code) обычно используется на универсальных ЭВМ. В России используется во многом совпадающая с ней кодировка ДКОИ (двоичный код обмена информацией). Для кодировки символа используется 8 бит. Количество кодируемых символов 2^8 = 256 символов.

Кодировка ASCII (American Standard Coding for Information Intorhange) обычно используется на мини и микро-ЭВМ. Для кодировки символа используется 8 бит. Количество кодируемых символов 2^8 = 256 символов.

В силу широкой распространенности кодировки ASCII рассмотрим ее подробнее.

Таблица кодировки ASCII состоит из двух частей.

Первая часть таблицы содержит коды от $0_{[10]}$ до $127_{[10]}$. Соглашения о кодируемых в этой половине символах являются общепринятыми. Первая часть таблицы содержит коды управляющих символов, латинских букв, десятичных цифр и знаков препинания.

Вторая часть таблицы содержит коды от $128_{[10]}$ до $255_{[10]}$. Эти коды используются для кодировки символов национальных алфавитов и других символов по правилам, принятым в данной стране.

Кодировка UNICODE использует для кодировки одного символа 16 бит, что позволяет закодировать 65535 символов. Кодировка символов с кодами 0 – 127 совпадает с кодировкой ASCII.

Таблицы кодировки ASCII и UNICODE можно найти в справочной литературе. Остановимся только на некоторых закономерностях кодировки, которые используются программистами при обработке текстовых данных. Все рассматриваемые коды приводятся в десятичной системе счисления.

Коды десятичных цифр образуют непрерывную возрастающую последовательность. Наименьший код имеет цифра ‘0’ – 48, а наибольший код - цифра ‘9’ – 57.

Это свойство часто используется для получения числового эквивалента цифры. Для этого из кода цифры необходимо вычесть код цифры ‘0’, например:

Числовой эквивалент цифры ‘4’ = ‘4’ – ‘0’ = 52 – 48 = 4.

Коды латинских прописных (больших) букв образуют непрерывную возрастающую последовательность. Наименьший код у буквы ‘A’ – 65, а наибольший - у буквы ‘Z’ – 90.

Коды латинских строчных (маленьких) букв образуют непрерывную возрастающую последовательность. Наименьший код у буквы ‘a’ – 97, а наибольший - у буквы ‘z’ – 122.

Свойство упорядоченности часто используется для получения порядкового номера буквы в латинском алфавите. Рассмотрим пример (отсчет номеров букв начинается с нуля):

Порядковый номер буквы ‘D’ = ‘D’ – ‘A’ = 68 – 65 = 3

Порядковый номер буквы ‘d’ = ‘d’ – ‘a’ = 100 – 97 = 3

Разность кодов одноименных строчных и прописных букв постоянна и эта разность составляет 32, что соответствует коду пробела. Это свойство используется для преобразования строчных букв в прописные и наоборот, например:

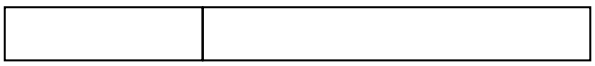
Преобразуем ‘D’ $\Rightarrow$ ‘d’: ‘D’ + (‘a’ – ‘A’) = ‘d’

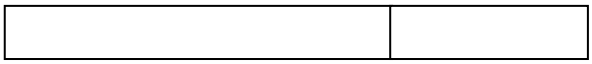
68 + (97 – 65) = 100

Преобразуем ‘d’ $\Rightarrow$ ‘D’: ‘d’ - (‘a’ – ‘A’) = ‘D’

100 - (97 – 65) = 68

Особый вид текстовых данных представляют символьные строки. Символьная строка рассматривается как последовательность символов. Каждому символу соответствует код символа. Строки имеют разную длину.





Длина строки может указываться явно или косвенно.

Явное указание. Перед символами строки располагается поле фиксированного размера, в котором указана длина строки.
Косвенное указание. За значащими символами строки размещается дополнительный символ с заранее оговоренным кодом. Этот символ играет роль признака конца строки. Код этого символа не должен совпадать ни с одним из символов строки. В качестве такового обычно используют символ с кодом 0 и называют его ноль-символом.

СИСТЕМА КОМАНД

1. Типовые операции

В силу принципа активности команд содержимое ячеек памяти трактуется по месту их обработки в процессоре. Поэтому команда помимо адресов операндов должна содержать следующую информацию:

- Вид обработки операнда (собственно операция).
- Тип операнда, т.е. длину ячейки памяти и способ трактовки ее содержимого – целое число, вещественное число и т.д.
- Способ адресации операнда.

Обычно эта информация кодируется в поле КОП. Каждому значению КОП соответствует своя команда процессора (машинная команда). Совокупность команд, которые может выполнять процессор, называют системой команд.

Разнообразие типов операндов и способов адресации приводит к тому, что процессор имеет свыше сотни команд при небольшом количестве видов обработки. Вид обработки отражает потребности алгоритма в преобразовании данных. Именно вид обработки будем понимать под термином "операция" в рамках этого занятия.

В зависимости от вида обработки выделяют следующие основные группы операций:

- арифметические операции;
- логические операции;
- сдвиговые операции;
- операции пересылки;
- операции управления;
- операции ввода-вывода.
-

2. Арифметические операции

К основным арифметическим операциям относятся операции с целыми числами: сложение, вычитание, инкремент (+1), декремент (-1), изменение знака числа, сравнение чисел. Операционное устройство выполняет арифметические преобразования данных в формате машинного слова или более коротких форматах.

Операции с длинными целыми числами, операции умножения и деления, а также операции с вещественными числами выполняются специализированным математическим сопроцессором или программным путем.

Выполнение арифметических операций на уровне машинных команд сопровождается контролем диапазона значений. При программировании на языках высокого уровня переполнение при операциях с целыми числами игнорируется, поэтому отслеживать диапазон значений должен сам программист. При операциях с вещественными числами выход за диапазон значений фиксируется как переполнение порядка или потеря значимости.

3. Логические операции

Логические операции предназначены для формирования признаков, используемых при управлении ходом выполнения программы. Эти операции рассматривают операнд с точки зрения его соответствия одному из двух значений: "Истина" или "Ложь". Типовые логические операции:

НЕ, И, ИЛИ. Результат определяется таблицей истинности и представляет собой код, соответствующий "Истина" или "Ложь".

Особое место занимают **битовые операции.** Иногда их рассматривают как разновидность логических операций. В других случаях их выделяют в отдельную группу. С логическими операциями их сближает то, что битовые операции рассматривают данные как логические значения. Но, в отличие от логических операций, в качестве логического значения рассматривается каждый бит обрабатываемого кода.

Типовые битовые операции:

- инверсия битов (битовое НЕ),
- битовое умножение (битовое И),
- битовая неэквивалентность (битовое М2),
- битовое сложение (битовое ИЛИ).

Схемы выполнения битовых операций приведены на Рис.1.

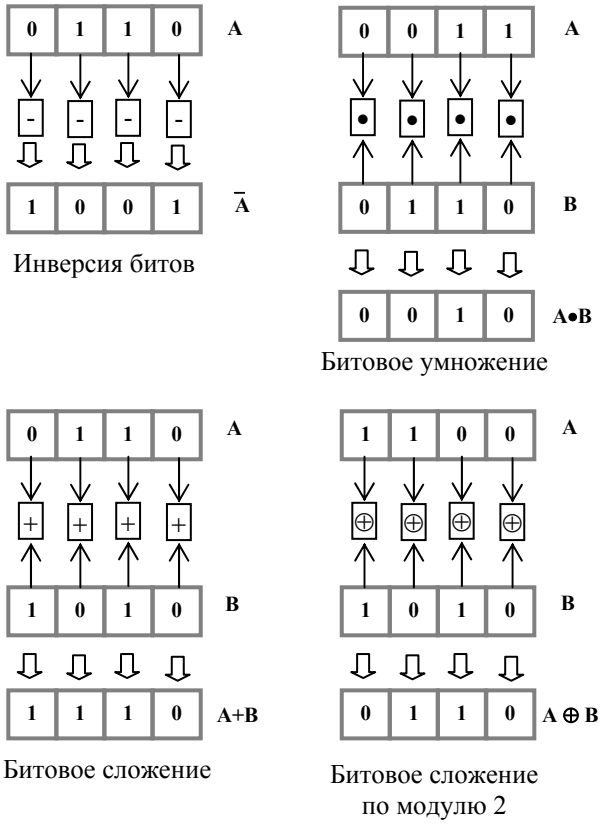


Рис. 1.

Битовые операции используются для обработки отдельных бит кода. Такая необходимость часто возникает из-за того, что поля битов в составе единого кода имеют разное смысловое значение. Такой прием обычно используется в кодах, описывающих состояние аппаратных средств ЭВМ. Например, байт атрибутов, который определяет режим отображения символа на экране, имеет три разных по смыслу поля:

Биты с 0 по 3 – цвет символа

Биты с 4 по 6 – цвет фона

Бит 7 – признак мерцания символа

Обработка бит сводится к четырем основным задачам:

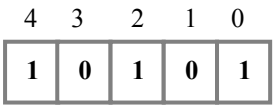
1.Проверка состояния заданного бита

2.Установка заданных бит в нулевое состояние

3.Установка заданных бит в единичное состояние

4.Инверсия заданных бит

В основе решения лежит выполнение битовой операции над исходным кодом и специально подготовленным кодом, который называется маской. Решение задач будем рассматривать на примере исходного пятиразрядного кода, приведенного на рисунке.



Проверка состояния заданного бита. Маска должна содержать единицу в проверяемом бите. Остальные биты маски нули. Выполняется битовая операция И и

результат рассматривается как целое число. Если оно отлично от нуля, то бит находится в единичном состоянии, а если равно нулю – то в нулевом.

Пример. Проверить состояние бита с номером 1 и бита с номером 4.

1 0 1 0 1	(Исходный код)	1 0 1 0 1
• 0 0 0 1 0	(Маска)	• 1 0 0 0 0
0 0 0 0 0	(Результат)	1 0 0 0 0

Вывод: бит с номером 1 находится в нулевом состоянии, а бит с номером 4 в единичном.

Установка заданных бит в нулевое состояние. Маска должна содержать нули в интересующих битах. Остальные биты маски единичные. Выполняется битовая операция И и ее результат записывается на место исходного кода. Единичные биты маски гарантируют сохранение состояния всех бит, кроме интересующих бит.

Пример. Установить биты с номерами 1 и 2 в нулевое состояние.

1 0 1 0 1	(Исходный код)
• 1 1 0 0 1	(Маска)
1 0 **0 0** 1	(Результат)

Установка заданных бит в единичное состояние. Маска должна содержать единицы в интересующих битах. Остальные биты маски нулевые. Выполняется битовая операция ИЛИ. Результат записывается на место исходного кода. Нулевые биты маски гарантируют сохранение состояния всех бит, кроме интересующих бит.

Пример. Установить биты с номерами 1 и 2 в единичное состояние.

1 0 1 0 1	(Исходный код)
∨ 0 0 1 1 0	(Маска)
1 0 **1 1** 1	(Результат)

Инверсия заданных бит. Маска должна содержать единицы в интересующих битах. Остальные биты маски нулевые. Выполняется битовая операция М2. Результат записывается на место исходного кода. Нулевые биты маски гарантируют сохранение состояния всех бит, кроме интересующих бит.

Пример. Инвертировать биты с номерами 0,1 и 4.

1 0 1 0 1	(Исходный код)
⊕ 1 0 0 1 1	(Маска)
0 0 1 **1 0**	(Результат)

Достаточно часто возникает задача занесения в поле заданного кода. Решение этой задачи может потребовать установки битов в разное состояние. В качестве примера рассмотрим задачу занесения кода 001в биты с номерами 3,2,1.

В этом случае можно использовать один из следующих методов:

1. Последовательная установка бит сначала в одно состояние, а потом в другое.

1 0 1 0 1	(Исходный код)
∨ 0 0 0 1 0	(Маска для установки единицы)
1 0 1 **1** 1	(Единица занесена)
• 1 0 0 1 1	(Маска для установки нулей)
1 **0 0 1** 1	(Нули занесены)

2. Обнуление заданного поля и занесение в него кода через операцию битового ИЛИ.

Маска в этом случае должна содержать в интересующих битах заданный код. Остальные биты нулевые.

```
  1 0 1 0 1      (Исходный код)
• 1 0 0 0 1      (Маска для обнуления)
  1 0 0 0 1      (Нули занесены)
∨ 0 0 0 1 0      (Маска для занесения кода)
  1 0 0 1 1      (Код занесен)
```

Предпочтительным является второй метод. Он обеспечивает решение задачи при заранее неизвестном по значению коде.

4. Сдвиговые операции

Выполняют направленное перемещение битов в пределах разрядной сетки. Сдвиг в сторону старших разрядов называют сдвигом влево, а в сторону младших разрядов - сдвигом вправо. В зависимости от способа заполнения освободившихся разрядов различают три вида сдвигов:

- Арифметический;
- Логический;
- Циклический.

Арифметический сдвиг. Вытолкнутые биты теряются. При сдвиге влево освобождающиеся правые биты заполняются нулем. При сдвиге вправо освобождающиеся левые биты заполняются значением старшего (знакового) бита. Схемы арифметического сдвига на примере пятиразрядного кода показаны на Рис.2.

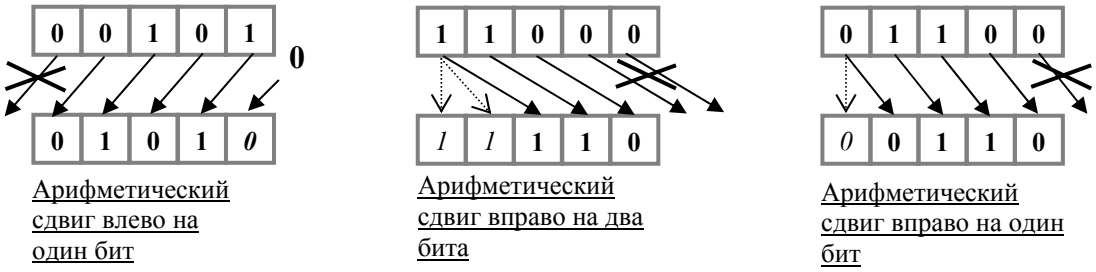


Рис. 2.

Логический сдвиг. Независимо от направления сдвига освобождающиеся биты заполняются нулями. Схемы логического сдвига на примере пятиразрядного кода показаны на Рис.3.

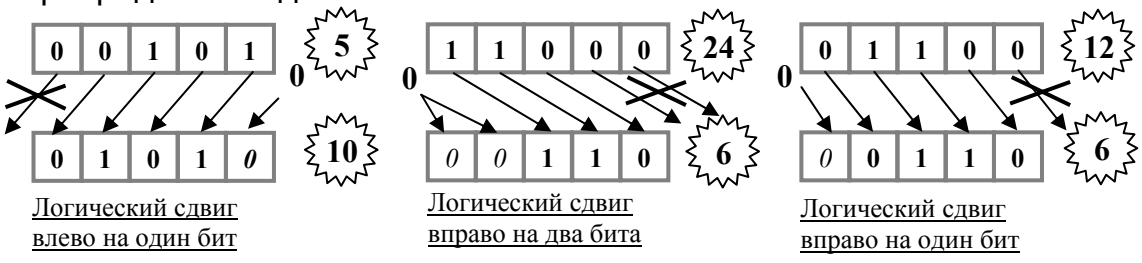


Рис. 3.

Если сдвигаемый код представляет собой код целого числа без знака, то логический сдвиг влево на n разрядов равносилен умножению числа на 2^n, а логический сдвиг вправо - делению числа на 2^n.

Циклический сдвиг. При логическом и арифметическом сдвиге вытолкнутые за разрядную сетку биты безвозвратно теряются. При циклическом сдвиге вытолкнутые биты записываются на место освободившихся битов. Схемы циклического сдвига на примере пятиразрядного кода показаны на Рис.4.

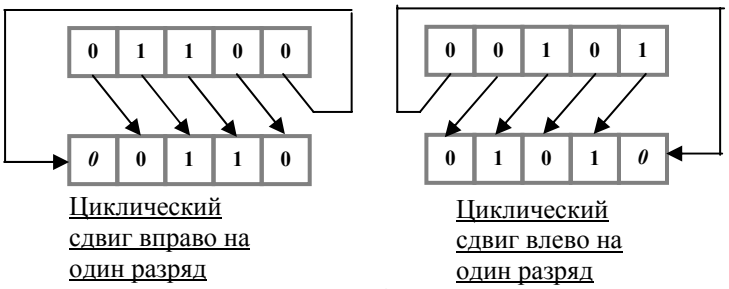


Рис. 4.

Циклический сдвиг влево или вправо на n битов, где n равно длине разрядной сетки, приведет к получению исходного двоичного кода. Для рассмотренных примеров исходный код будет получен при сдвиге на пять бит

Типичной областью применения сдвигов является программная реализация умножения и деления.

5.Операции пересылки

Применяются для перемещения данных в памяти. Чаще всего приходится пересылать данные между ячейками основной памяти и регистрами общего назначения.

6. Операции управления

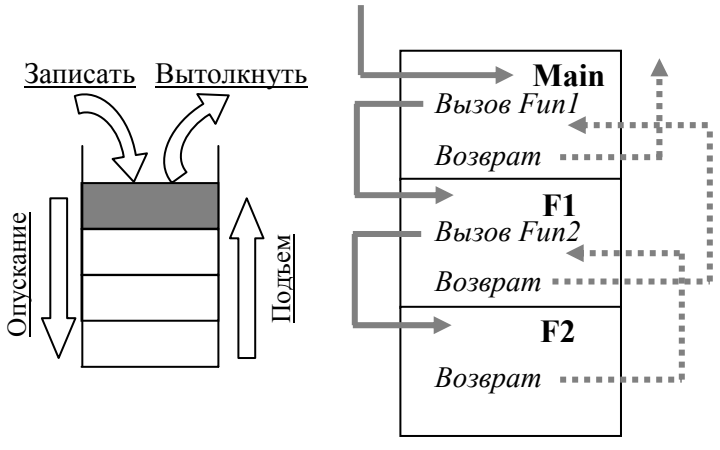
Предназначены для организации переходов при выполнении программы. В команде в явном виде указывается адрес следующей команды АСК. Для организации условных переходов кроме АСК указывается признак, по которому происходит переход. Обычно значение признака устанавливается предыдущей командой. Типичные признаки:

- Результат равен нулю;
- Результат отрицательный;
- Результат положительный;
- Переполнение.

Некоторые операции не формируют признак результата. Для выполнения условного перехода после такой операции в программу включают специальную команду, формирующую признак результата. Типичным примером такой команды является команда “Сравнить”.

Для организации вызова подпрограмм применяют операции типа "переход с возвратом". Перед выполнением перехода эта операция запоминает в памяти “точку возврата” - текущее значение счетчика команд. После завершения подпрограммы в счетчик заносится запомненное значение, то есть возобновляется выполнение программы.

Для управления вызовами подпрограмм используется специальный механизм, называемый стеком. Сущность этого механизма приведена на рисунке.



Стек представляет собой область памяти. Начало этой области памяти называется вершиной стека. В вершину стека заносятся данные выполняю-

щейся подпрограммы. Вызов подпрограммы приводит к "опусканию" стека, после чего в вершине запоминается счетчик команд выполнявшейся подпрограммы. Возврат из подпрограммы вызывает "подъем" стека. Это приводит к выталкиванию из вершины стека и занесению в счетчик команд запомненного адреса очередной команды и восстановлению в вершине стека данных предыдущей подпрограммы.

Таким образом, в вершине стека всегда находятся данные выполняемой в данный момент подпрограммы. Программа считается выполненной, если стек становится пустым. Рассмотрим этот процесс на примере выполнения приведенной на рисунке программы. На схеме отражен процесс запоминания данных выполняемой подпрограммы.

Действие: Вызов **Main**>Вызов **F1**>Вызов **F2**>Возврат из **F2**>Возврат из **F1**>Возврат из **Main**

Стек: *Пусто* **Main** **F1** **Main** **F2** **F1** **Main** **Main** *Пусто*

7. Операции ввода-вывода

Операции рассмотренных выше групп предназначены для обработки данных в центральной части ЭВМ. В силу общих принципов обработки данных в центральной части ЭВМ эти операции реализуются по схожим схемам.

Операции ввода-вывода предназначены для пересылки данных между основной памятью и внешними устройствами. Так как ввод-вывод организуется в разных ЭВМ по разному, реализация этих операций существенно зависит от типа ЭВМ.

Рассмотрим крайние ситуации.

При наличии ПВВ набор операций ввода-вывода основного процессора ограничен несколькими командами, которые управляют работой ПВВ, например, "Начать ввод-вывод", "Завершить ввод-вывод".

При централизованном управлении специальных команд ввода-вывода может не быть вообще. За каждым внешним устройством закрепляется одна или несколько ячеек памяти, через которые организуется обмен данными. В этом случае обмен может быть выполнен с помощью операций пересылки.

Все остальные варианты операций ввода-вывода занимают промежуточное положение между этими схемами.

СИСТЕМА ПРЕРЫВАНИЙ

1. Понятие прерывания

Во время выполнения программы могут возникать особые ситуации, требующие немедленной реакции ЭВМ. Источником таких ситуаций могут быть:

1. Аппаратура центральной части ЭВМ (внутренние прерывания). Например, обнаружение ошибки при передаче данных между основной памятью и процессором.

2. Аппаратура периферийной части ЭВМ (внешние прерывания). Например, нажатие клавиши на клавиатуре, нажатие кнопки мыши и т.п.

3. Программа, которую в данный момент выполняет ЭВМ (программные прерывания). Особые ситуации в этом случае могут возникать при "нештатном" режиме работы программы или же создаваться программой преднамеренно. Примером "нештатной" работы программы может быть попытка деления на ноль, извлечение квадратного корня из отрицательного числа и т.п. Преднамеренно эта ситуация создается с помощью команд вызова прерывания.

Реакция ЭВМ на особые ситуации заключается в приостановке выполнения текущей программы и переходе к выполнению специальной программы обработки особой ситуации. После завершения программы обработки ЭВМ возвращается к выполнению текущей программы. Этот процесс называется прерыванием программы, а аппаратно программные средства обработки особых ситуаций – системой прерываний.

Программа, выполнявшаяся до возникновения особой ситуации, называется прерываемой программой, а программа обработки особой ситуации – прерывающей. Процесс обработки особой ситуации приведен

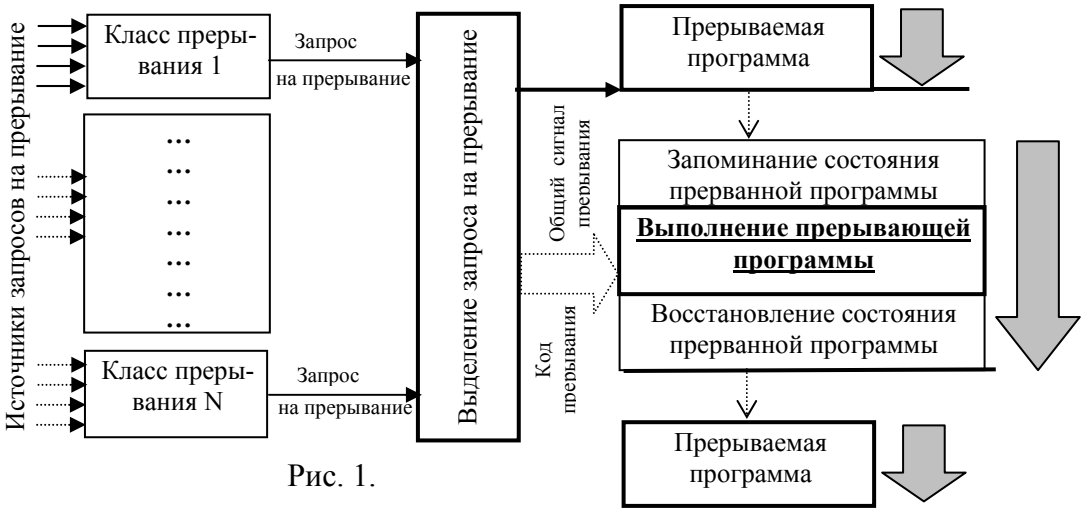


Рис. 1.

на Рис.1.

При возникновении особой ситуации формируется сигнал оповещения, называемый запросом на прерывание. Запрос для внутренних и внешних прерываний формируется аппаратными средствами, а для программных – программными средствами.

Реакция ЭВМ на особые ситуации для некоторых источников может быть одной и той же. Другими словами, этим источникам соответствует одна и та же обрабатывающая программа.

Множество особых состояний, обслуживаемых одной и той же программой, образует класс прерывания. Каждому классу прерываний назначается номер, называемый кодом прерывания. Код прерывания однозначно определяет, какая обрабатывающая программа должна обрабатывать возникшую особую ситуацию.

Количество источников может достигать сотен, а количество классов значительно меньше, как правило, не больше нескольких десятков.

Типичными классами прерываний являются:

- ❖ Прерывания от схем контроля;
- ❖ Прерывания от пульта управления ЭВМ и других внешних устройств;
- ❖ Прерывания от системы ввода-вывода;
- ❖ Прерывания при обращении программ к запрещенным для использования ресурсам;
- ❖ Программные прерывания.

Многие ЭВМ допускают прерывание прерывающей программы. Количество программ, которые могут последовательно прерывать друг друга, называется глубиной прерывания.

2. Выделение запроса на прерывание

Одновременно может возникнуть сразу несколько особых ситуаций. В этом случае одновременно будет существовать несколько запросов на прерывание. Последовательность обработки запросов зависит от важности возникшей ситуации. Для этого каждому классу прерываний назначается свой приоритет.

Существует две основные схемы выделения запроса с наибольшим приоритетом:

- ❖ Опрос источников прерывания;
- ❖ Векторное прерывание.

При первом способе запросы на прерывание у всех источников однотипны, что не позволяет сразу определить код прерывания. При появлении любого запроса формируется общий сигнал прерывания. Этот сигнал инициирует опрос источников прерываний в соответствии с их приоритетом. Только после выявления "виновника" удается сформировать код прерывания и определить программу его обработки.

Способ прост в реализации, но требует много времени на опрос. Применяется в основном в микро-ЭВМ при небольшом количестве источников прерываний.

При втором способе запрос каждого источника носит индивидуальный характер. Каждый источник формирует запроса на прерывание и устанавливает свой код прерывания. Поскольку программы обработки прерывания каждого класса хранится в заранее оговоренном месте основной памяти, то часто код прерывания представляет собой адрес программы обработки. Для реализации этого способа нужны дополнительные аппаратные средства, приведенные на Рис.2.

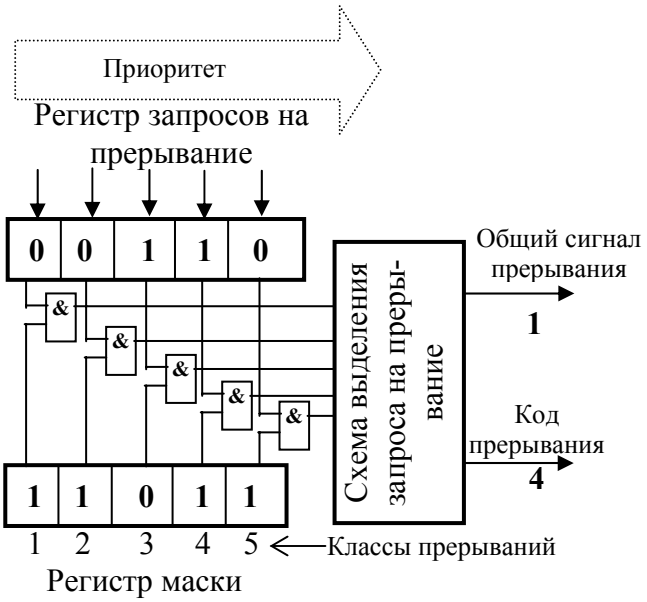


Рис. 2.

Запросы заносятся в регистр запросов на прерывание (РЗП). Число разрядов РЗП соответствует числу классов прерываний. Каждый класс имеет строго закрепленный за ним разряд РЗП. Чтобы вызвать прерывание с заданным классом требуется установить соответствую-щий разряд регистра в 1.

Некоторые запросы можно проигнорировать – замаскировать. Маска прерываний хранится в специальном регистре масок (РМ).

Одноименные разряды РЗП и РМ соединены по схеме И. Маска устанавливается специальными командами. Схема выделения запроса выявляет самую левую незамаскированную единицу в регистре запросов и формирует общий сигнал прерывания и его код. Код определяется по номеру разряда РЗП.

3. Обработка прерывания

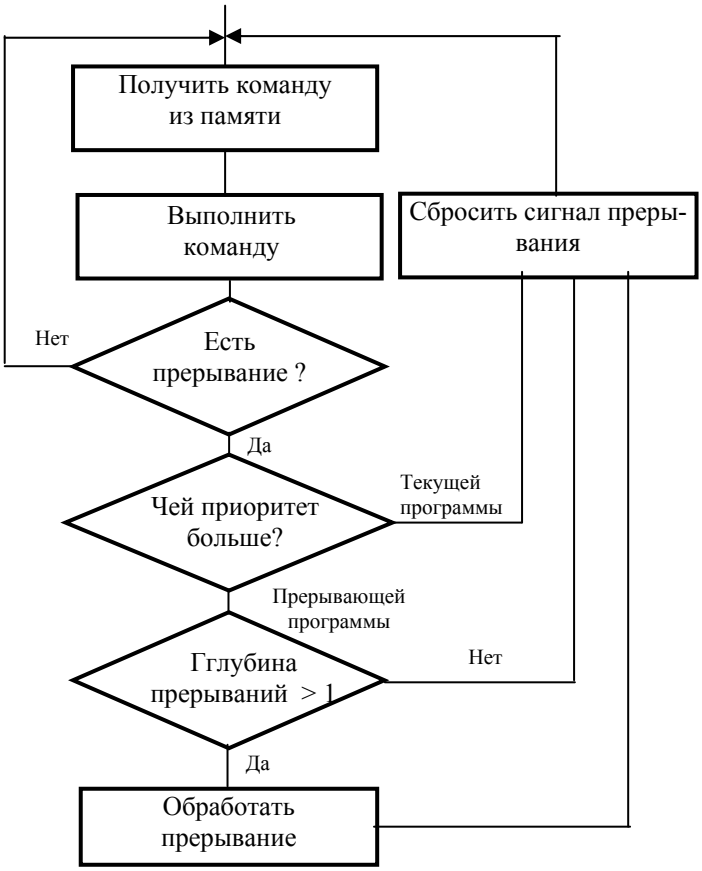


Рис. 3.

Основные этапы обработки выделенного прерывания были приведены на Рис.1. Уточним содержание некоторых из них.

Любая выполняемая программа монопольно использует регистры процессора. Содержимое регистров характеризует состояние выполняемой программы. Без сохранения состояния регистров выполняемой программы возврат к ней может оказаться невозможным.

Поэтому процесс перехода к прерывающей программе и возврат к прерванной должен состоять из следующих этапов:

- Сохранение регистров процессора в памяти (запоминание текущего состояния прерываемой программы);

- ❖ Загрузка в регистры состояния прерывающей программы и ее выполнение;
- ❖ Загрузка в регистры процессора запомненного состояния прерванной программы (восстановление состояния программы) и продолжение ее выполнения.

Обычно сохранение и восстановление регистров возлагается на прерывающую программу. Если такая возможность не предусмотрена, сохранение и восстановление регистров должна выполнять сама прерываемая программа. На Рис.3 показано взаимодействие процесса выполнения программы и процесса обработки прерывания.

ЖИЗНЕННЫЙ ЦИКЛ ПРОГРАММЫ

1. Понятие парадигмы программирования

Для решения задачи обработки данных разрабатывается и используется программа. Программа определяет перечень действий над данными, которые должен выполнить исполнитель для решения поставленной задачи. Не претендуя на полную детализацию, отметим основных участников решения задачи и их роли:

1. Заказчик – определяет требования к разрабатываемой программе. Различают функциональные и нефункциональные требования. Функциональные требования представляют собой перечень возможностей, предоставляемых программой пользователю при решении задачи (функциональность программы). К нефункциональным требованиям относится: состав входных и выходных данных, способ получения входных данных от пользователя и способ выдачи выходных данных пользователю, ограничения на время выполнения программы, диапазон значений данных, точность представления данных и т.п. Зафиксированный установленным способом перечень требований будем называть *спецификацией* программы.
2. Проектировщик – определяет структуру разрабатываемой программы, распределение функциональности между частями программы и их взаимодействие по управлению и обмену данными.
3. Разработчик – определяет способ реализации требуемой функциональности в каждой из частей программы и разрабатывает код программы на языке, доступном исполнителю.
4. Пользователь – применяет разработанную программу по назначению для получения результатов обработки данных.

При разработке спецификации выделяется часть реального мира – предметная область (Рис.1). Предметная область включает только те предметы и понятия, которые имеют отношение к решаемой задаче.

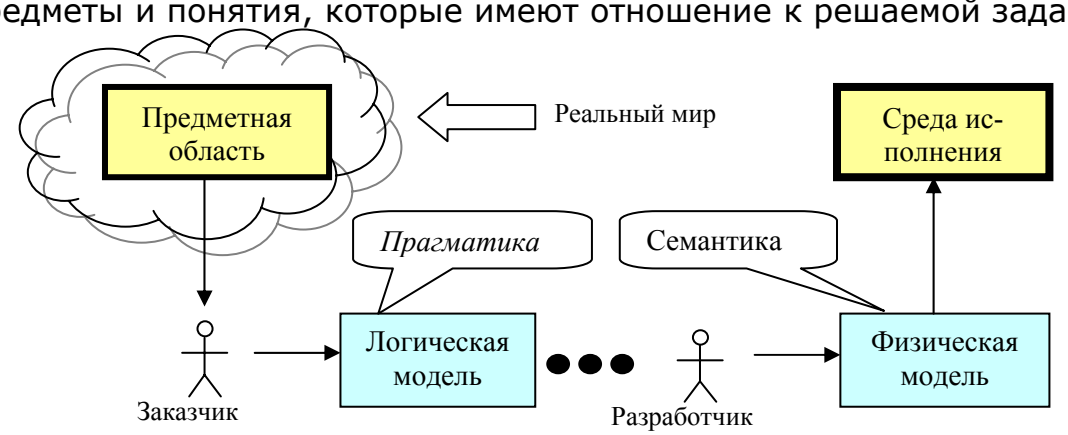


Рис. 1

Для выделенной предметной области составляется ее формализованное описание – *модель* предметной области. Модель предметной области и определение спецификации программы выполняется, как правило, системным аналитиком и согласовывается с заказчиком и проектировщиком. Модель предметной области, отражающую взгляд на нее заказчика, называют *логической моделью*, а представление о программе с точки зрения заказчика называют *прагматикой* программы.

Очевидно, что для заказчика естественным является желание использовать для построения логической модели понятийный аппарат и средства формализации, принятые в предметной области.

С другой стороны, разработчик должен определить семантику программы. *Семантика* программы - это представление о программе с точки зрения ЭВМ, которая будет ее выполнять. Другими словами, разработчик должен сформировать *физическую* модель предметной области, выраженную с использованием понятий и средств формализации, доступных для восприятия ЭВМ.

Для аппаратных средств ЭВМ непосредственно доступна программа, выраженная в терминах машинных команд. Разработчик для определения семантики программы может использовать так называемые *языки высокого уровня (ЯВУ)*. Понятийный аппарат ЯВУ может существенно отличаться от понятийного аппарата машинных команд. При использовании ЯВУ ЭВМ должна быть снабжена комплексом программных средств, которые обеспечивают выполнение на ЭВМ программ, написанных на ЯВУ. Аппаратные средства ЭВМ с программными средствами выполнения программ образуют *среду исполнения* программ.

В дальнейшем при рассмотрении семантики программы и физической модели предметной области будем исходить из того, что они определяются разработчиком на языке высокого уровня.

Понятийный аппарат, используемый для разработки моделей предметной области, называют *парадигмой программирования*.

Наиболее распространенными парадигмами программирования являются процедурно-ориентированное программирование и объектно-ориентированное программирование. Существуют и другие парадигмы программирования, в частности логическое программирование и функциональное программирование. Рассмотрение этих парадигм выходит за рамки данного курса.

Рассмотрим основные понятия, лежащие в основе процедурно-ориентированного и объектно-ориентированного программирования.

2. Процедурно-ориентированное программирование.

В основе парадигмы лежит понятийный аппарат, отражающий принципы логической организации ЭВМ классической архитектуры. Предметная область рассматривается как процесс воздействия на входные данные с целью их преобразования в выходные данные.

Для представления логической модели используются графические диаграммы, дополненные текстовым описанием.

В логической модели определяются:

- Входные данные
- Источники входных данных
- Выходные данные
- Потребители выходных данных
- Данные, подлежащие долговременному хранению (накопители данных)
- Процессы преобразования входных данных в выходные данные

Логическая модель имеет иерархическую структуру. Каждая диаграмма представляет собой детализацию процесса предыдущего уровня и, при необходимости, детализацию данных. Пример логической модели в виде диаграммы потоков данных (ДПД) для задачи вычисления итого-

вой оценки студента по дисциплине "Информатика и программирование" после выполнения экзаменационной работы приведена на Рис.2 и Рис.3.

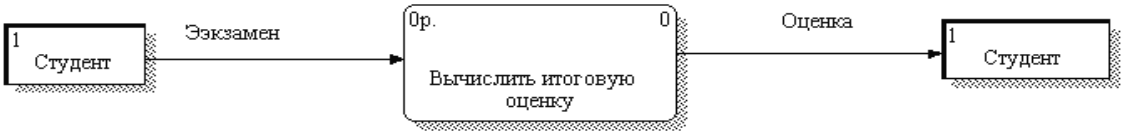


Рис.2

На Рис.2 показана диаграмма верхнего уровня, на которой определен источник входных и потребитель выходных данных (**Студент**), поток входных данных (**Экзамен**), выходной поток данных (**Оценка**) и функциональность программы в виде процесса **Вычислить итоговую оценку**

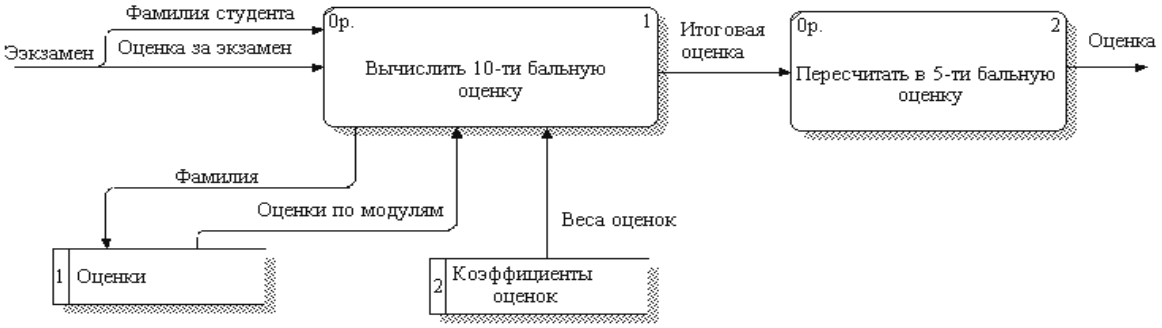


Рис.3

На Рис.3 показана диаграмма детализации, в которой процесс решения задачи разделен на два подчиненных процесса: **Вычислить 10-ти бальную оценку** и **Пересчитать в 5-ти бальную оценку**. Детализация входных данных показана в виде разветвления стрелки входного потока. Для решения задачи из накопителя данных **Оценки** считываются оценки студента по всем модулям, а из накопителя **Коэффициенты оценок** считываются веса оценок по каждому модулю.

Структурно программа представляет собой набор процедур – подпрограмм. Взаимодействие подпрограмм организовано по иерархическому принципу. Выполнение программы начинается с главной подпрограммы. Выполняющаяся подпрограмма может вызвать подчиненную подпрограмму. Выполнение подпрограммы приостанавливается до завершения вызванной подпрограммы и возврата из нее. Программа считается выполненной после завершения главной подпрограммы. На Рис. 4 приведена программа из трех подпрограмм и показана очередность их вызова в соответствии с обозначениями, принятыми на схемах алгоритмов. Главная подпрограмма обозначена именем Main.

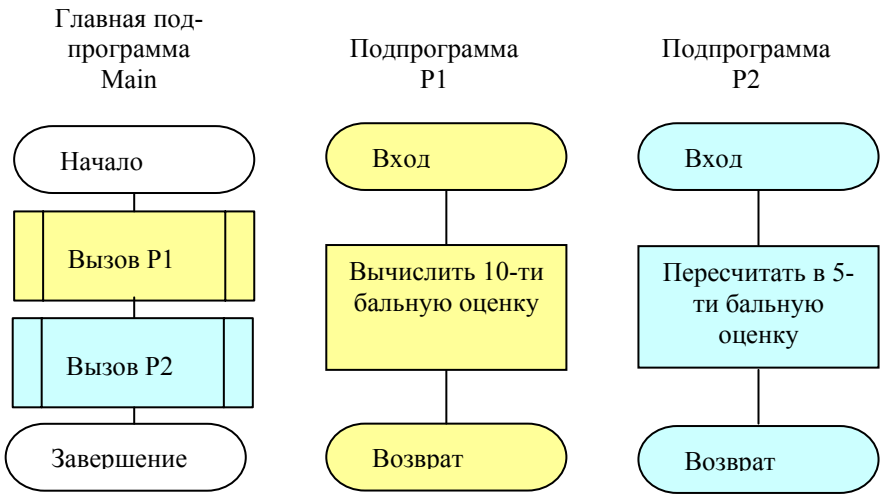


Рис. 4

Процесс выполнения программы можно представить в виде последовательности вызовов-возвратов:

- Запуск Main
- Вызов P1
- Обработка данных в P1 и возврат в Main
- Вызов P2
- Обработка данных в P2 и возврат в Main
- Завершение программы

При проектировании программы функциональность обычно распределяют таким образом, чтобы основная обработка данных выполнялась подпрограммами нижнего уровня иерархии, а на подпрограммы верхнего уровня возлагаются задачи проверки входных данных на корректность, приведение выходных данных к форме, требуемой пользователю и управление вызовами подчиненных подпрограмм.

Достоинства процедурно-ориентированного программирования:

- простая структура и предсказуемость поведения программы
- умеренные требования к ресурсам среды исполнения

Недостатки:

- Невозможность отразить структуру предметной области в виде взаимосвязанных сущностей
- Не поддерживается создание программ с распределенной обработкой
- Сложность модификации программ в случае изменения функциональности

Для разработки программ на основе процедурно-ориентированного программирования используются языки Fortran, Pascal, Basic,C.

3. Объектно-ориентированное программирование.

В основе лежит представление предметной области в виде множества объектов, взаимодействующих между собой. Под *объектом* понимается мыслимая или реальная сущность, обладающая характерным поведением и отличительными характеристиками и являющаяся важной для данной предметной области. Характеристики объекта называют *атрибутами.* Значение атрибутов в данный момент времени определяет *состояние объекта.*

Объект может иметь определенный набор действий (операций), которые можно произвести над атрибутами объекта. Другими словами, набор операций определяет возможные варианты поведения объекта.

Множество объектов, которые имеют одинаковый набор атрибутов и операций, образуют *класс объектов*. Объединение объектов в класс позволяет рассматривать задачу в общем виде.

Модели предметной области разрабатываются в виде графических диаграмм. Наиболее широко используются диаграммы, определенные в языке UML (Unified Modeling Language). В UML определены 9 типов диаграмм, с помощью которых разрабатываются модели предметной области от логической модели до физической модели.

Коротко остановимся на трех диаграммах, которые используются при разработке практически каждой программы.

Диаграмма вариантов использования. Данная диаграмма определяет функциональность программы в терминах актеров и вариантов использования. Актер является инициатором выполнения варианта использования, а вариант использования определяет функциональность.

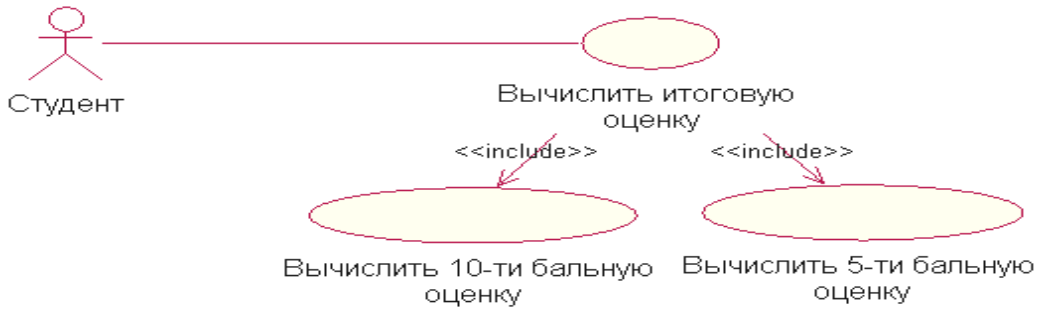


Рис.5

Пример диаграммы вариантов использования для предметной области "Экзамен по дисциплине "Информатика и программирование" приведен на Рис.5. Актером является **Студент**, варианты использования изображены в виде овалов. Основной вариант **Вычислить итоговую оценку** в обязательном порядке *включает* варианты **Вычислить 10-ти бальную оценку** и **Вычислить 5-ти бальную оценку.** Варианты использования, которые *расширяют* основной вариант и выполняются не всегда, а лишь при определенных условиях, помечаются стрелкой в обратном направлении и пометкой <<extended>>.

Диаграмма классов. Структурно программа представляет собой набор классов, находящихся между собой в определенных отношениях. Обозначение класса включает три секции, показанные на Рис.6.

Если класс не содержит атрибутов или операций – соответствующая секция остается не заполненной. Приведенная на Рис.6 диаграмма классов описывает предметную область "Экзамен по дисциплине "Информатика и программирование".

При разработке программ вместо термина атрибут используют термин поле, а вместо термина операция – термин метод.

Отношения между классами будут рассмотрены в разделе, посвященном программированию с использованием определяемых в программе классов. Пока же ограничимся рассмотрением без излишней детализации основополагающего отношения – наследования.

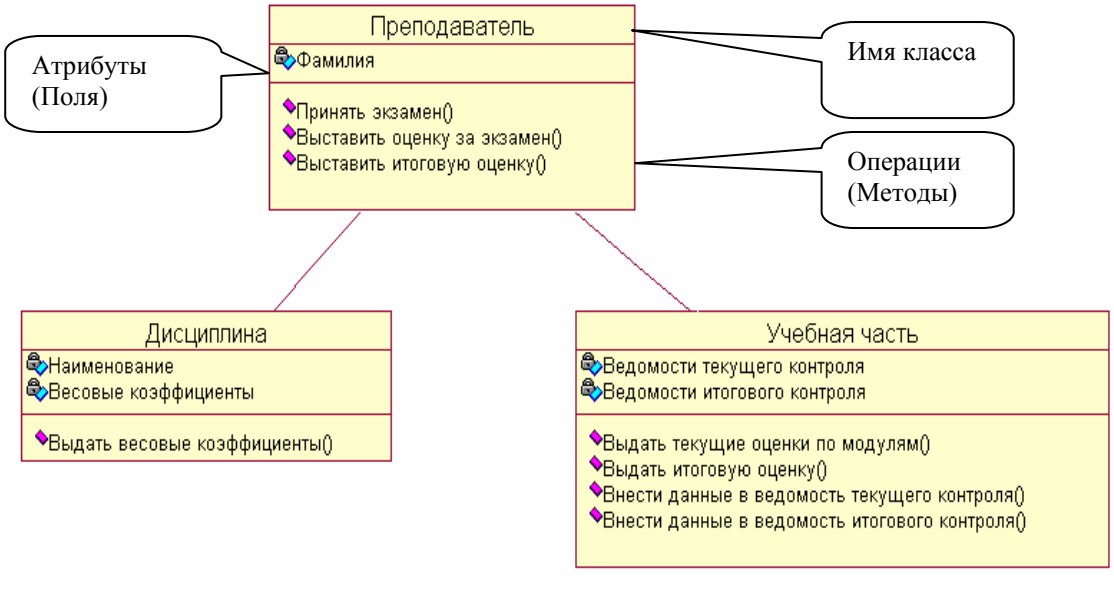


Рис.6

Каждый класс может быть использован в качестве предка для определения класса-наследника. Класс–наследник имеет поля и методы, унаследованные от предка, и может определять свои собственные поля и методы, а при необходимости переопределять унаследованные поля и методы. Наследование позволяет сократить затраты на разработку программного кода за счет использования ранее разработанного и проверенного кода.

Диаграмма кооперации. Рассмотренная диаграмма классов определяет статическое представление предметной области и разрабатывается на уровне классов. Динамические свойства предметной области проявляются в выполнении конкретными объектами операций и изменении состояния. Поэтому диаграммы для отражения динамических свойств создаются на уровне объектов.

Следует помнить, что смысл понятия класс несколько отличается при составлении логической модели и физической модели. В первом случае класс играет роль классификатора при объединении существующих объектов в группу. Во втором случае разработчика программы рассматривает класс некий шаблон, определяющий правила, по которым программа создает конкретные экземпляры класса - объекты.

Диаграмма кооперации – одна из диаграмм для отражения динамических свойств. Данная диаграмма определяет взаимодействие между объектами путем передачи сообщений между ними. В ответ на полученное сообщение объект выполняет заданную операцию и, в общем случае, изменяет свое состояние. Изменение состояния объекта может повлечь за собой передачу сообщения другому объекту. Сообщение может передаваться сразу нескольким объектам. Объект, передавший сообщение, может дожидаться ответа или же продолжать работу.

В простейшем случае сообщение представляет собой вызов из объекта одного класса метода объекта другого класса и переход в состояние ожидания ответа, как показано на Рис.7.

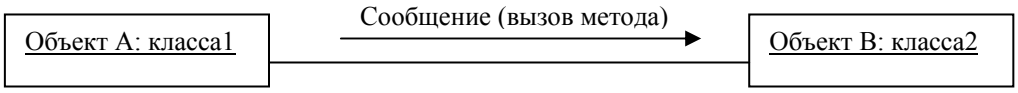


Рис.7

В этом случае поля объекта напоминают данные в процедурно-ориентированном программировании, а методы – подпрограммы. Принципиальным отличием является то, что в процедурно-ориентированном программировании данные и подпрограммы могут существовать как самостоятельные сущности независимо друг от друга, а поля и методы объединены (инкапсулированы) в единый объект и раздельно друг от друга не существуют.

Основная цель диаграммы кооперации - показать какие операции объектов используются при реализации того или иного варианта использования. Поэтому диаграмм кооперации должно быть столько же, сколько вариантов использования определено в диаграмме вариантов использования.

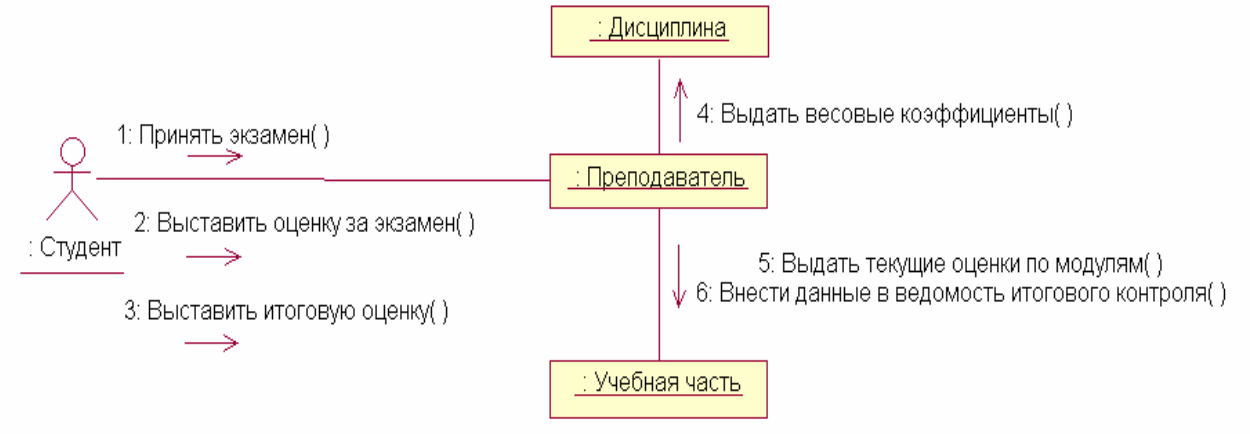


Рис.8

Пример диаграммы кооперации для варианта использования **Вычислить итоговую оценку** приведен на Рис.8. Сплошной линией показана связь между объектами. Рядом с линией связи нанесена стрелка вызова метода. Стрелка направлена на объект, метод которого вызывается, а рядом со стрелкой указан вызываемый метод. Вызовы методов пронумерованы. Нумерация вызовов соответствует последовательности их выполнения. В данной диаграмме конкретное имя объекта не представляется логически важным, поэтому оно не указано.

Достоинства объектно-ориентированного программирования:

- отражение структуры предметной области в виде взаимосвязанных сущностей
- более простая модификация программ за счет наследования полей и методов и возможности изменять код базовых классов-предков без внесения изменений в код классов-наследников.
- поддержка создания программ с распределенной обработкой

Недостатки:

- меньшая предсказуемость поведения программы
- повышенные требования к ресурсам среды исполнения

Для разработки программ на основе объектно-ориентированного программирования используются языки Java, C#.

Отметим, что существуют языки, позволяющие использовать в одной программе сочетание обеих рассмотренных парадигм программированиям. К таким языкам относятся языки C++ и Object Pascal.

5. Жизненный цикл программы

Под жизненным циклом программы понимают совокупность взаимосвязанных и следующих во времени этапов, начиная от разработки требований к программе и заканчивая полным отказом от ее использования.

Существует несколько моделей (схем) жизненного цикла, которые отличаются количеством этапов и задачами, решаемыми на каждом из них, но главное – хронологической последовательностью этапов. В этом отношении полярными являются так называемая каскадная схема жизненного цикла и итерационная схема жизненного цикла.

Каскадная схема предполагает строго последовательное выполнение типовых этапов, показанных на Рис.9.

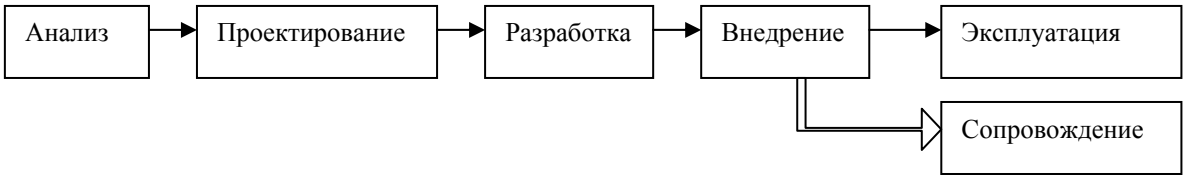


Рис.9

Исключение составляет этап сопровождения, который выполняется параллельно этапу эксплуатации. Кратко отметим основное содержание этапов.

Анализ – разработка логической модели и определение спецификации.

Проектирование – определение структуры программы и распределение функциональности по элементам этой структуры.

Разработка – определение способа реализации требуемой функциональности в каждом из элементов структуры и разработка физической модели предметной области на некотором языке программирования.

Внедрение – конфигурирование программы под конкретную среду исполнения и проведение испытаний на соответствие программы требованиям заказчика.

Эксплуатация – применение программы по назначению.

Сопровождение – модификация программы с целью исправления ошибок, выявленных при эксплуатации.

Каждый этап не начинается до завершения предыдущего и оформления документации по завершенному этапу.

Каскадная схема изначально преследует цель минимизации модификаций физической модели за счет тщательной проработки каждого этапа. Под каскадную схему ориентировано несколько методик разработки программ, в частности действующий в России стандарт ГОСТ 34.601-90.

Основным недостатком каскадной схемы является исключение заказчика из большинства этапов жизненного цикла, что чревато риском проявления ошибок, допущенных на начальных этапах, лишь при внедрении программы.

Итерационная схема предполагает несколько фаз в жизненном цикле. Каждая фаза может состоять из одной или нескольких итераций. На каждой итерации выполняются виды деятельности, направленные на уточ-

нение результатов анализа, проектирования и разработки, полученных на предыдущей итерации (Рис.10). На каждой итерации должен быть получен вариант физической модели. Результаты каждой фазы документируются.

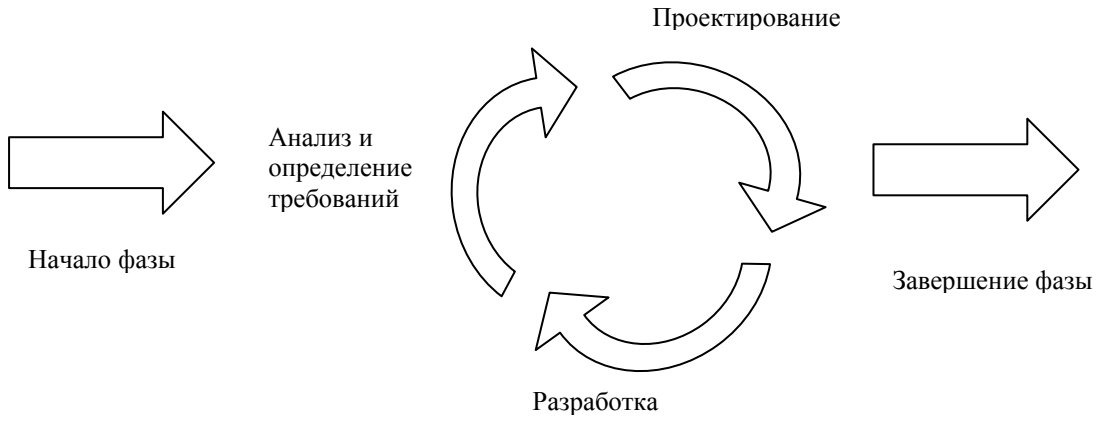


Рис. 10

Программа, полученная на каждой фазе, отличается более полной реализацией функциональности по сравнению с программой, полученной на предыдущей фазе. Итерационная схема предполагает участие заказчика на всех фазах жизненного цикла и тем самым ориентирована на снижение рисков неправильной трактовки спецификации проектировщиками и разработчиками.

Под итерационную схему ориентирована методика RUP (Rational Unified Process), при этом для документирования результатов каждой фазы используются одни и те же диаграммы языка UML, отличающиеся только степенью их детализации.

Недостатком схемы является необходимость многократной модификации физической модели.

СИСТЕМА И СРЕДА ПРОГРАММИРОВАНИЯ

1. Понятие системы и среды программирования

Под системой программирования понимают язык программирования и совокупность программных средств, поддерживающих разработку и исполнение программ, написанных на этом языке.

Для выполнения программа должна быть загружена в среду исполнения. В случае использования ЯВУ загрузке программы может предшествовать ряд преобразований, целью которых является приведение программы к виду, необходимому для загрузки в среду исполнения.

Для долговременного хранения программа на ЯВУ и программа после каждого преобразования размещается на внешнем запоминающем устройстве в виде файлов. Часть программы, которая хранится в одном файле, называется *модулем*. В простейшем случае вся программа хранится в одном файле. Имена файлов, как правило, назначает разработчик, а расширения файлов назначаются автоматически по правилам, принятым в среде исполнения.

Модуль, содержащий программу на языке высокого уровня, называется *исходным* модулем. Текст исходного модуля состоит из отдельных предложений, называемых *операторами.*

Модуль, содержащий программу в виде, готовом для загрузки в среду исполнения, называется *исполняемым* модулем.

Различают две основные схемы преобразования исходного модуля в исполняемый модуль: трансляция и интерпретация.

Схема трансляции используется для представления исполняемого модуля в виде машинных команд. Это означает, что исходный модуль должен быть предварительно переведен на язык машинных команд. Перевод выполняется специальной программой - транслятором. Схема трансляции приведена на Рис.1.

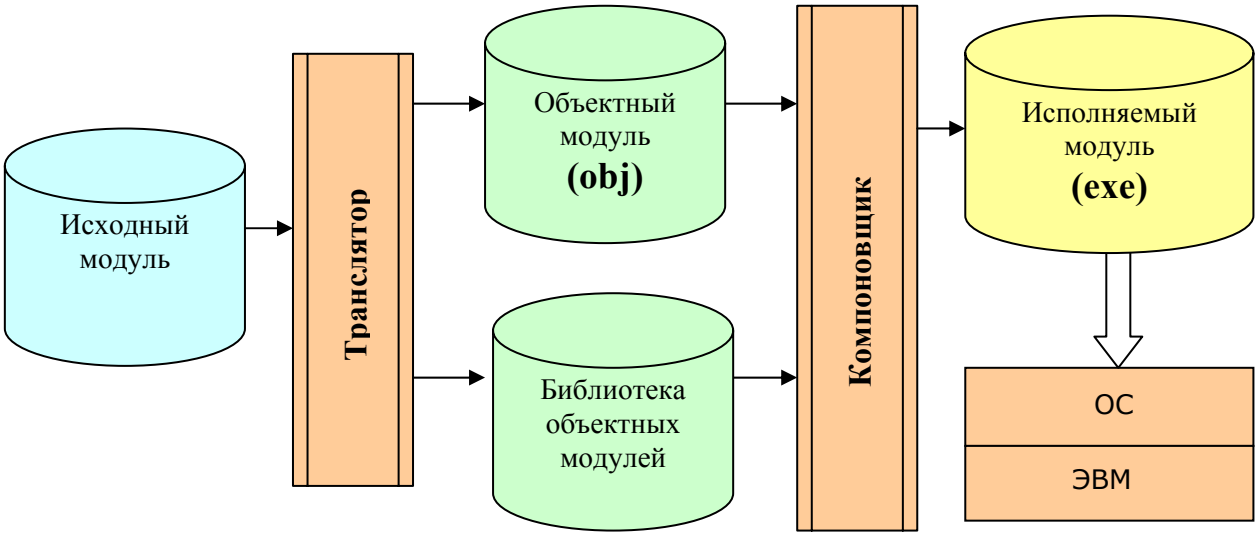


Рис. 1

При разработке программ обычно используются ранее созданные подпрограммы, которые хранятся в библиотеке стандартных подпрограмм в виде, пригодном для загрузки в среду исполнения. Подключение стандартных подпрограмм может выполняться в ходе выполнения про-

граммы (динамически компонуемые библиотеки) или предварительно до загрузки исполняемого кода в среду исполнения (статически компонуемые библиотеки). В последнем случае модуль, полученный транслятором, называют объектным модулем.

Подключение стандартных подпрограмм возлагается на специальную программу - компоновщик (редактор связей). Обозначения расширений для модулей типично для большинства систем программирования в операционной системе Windows. Транслятор и компоновщик являются составными частями системы программирования.

Отметим, что рассматривать библиотеку просто как набор подпрограмм можно только в предельно упрощенном виде. Операции по добавлению подпрограмм в библиотеку и удаления подпрограмм из библиотеки выполняются специальными программами, которые будем рассматривать как составную часть библиотеки.

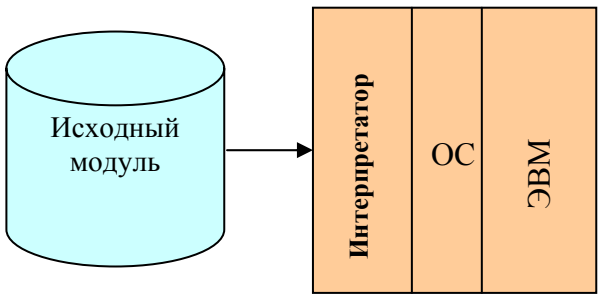


Рис. 2

Схема интерпретации используется для непосредственного распознавания и выполнения операторов исходного модуля (Рис.2). Распознавание и выполнение операторов возлагается на специальную программу – интерпретатор. Понятия исходного и исполняемого модуля в этом случае совпадают.

Таким образом, можно уточнить составные части системы программирования, как показано на Рис.3

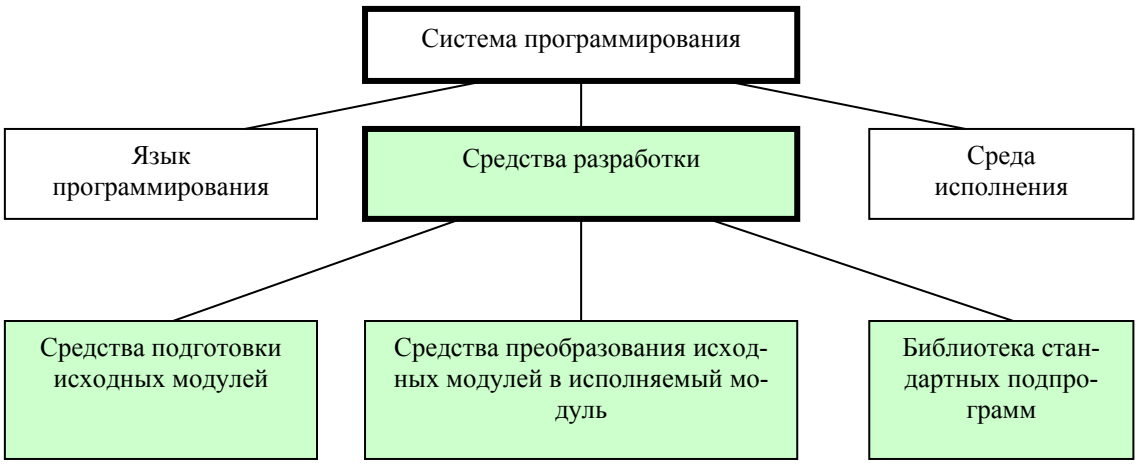


Рис. 3

Средства разработки могут использоваться автономно или объединятся в систему. В первом случае запуск каждого из средств инициируется разработчиком путем ввода команды операционной системы.

Средства разработки, объединенные в систему на основе общего интерфейса и общей базы данных, образуют *среду программирования*.

Приведенные выше составные части системы программирования необходимо рассматривать как достаточно типичные. Конкретные системы программирования могут иметь как более сложную, так и более простую структуру. Рассмотрим отличия на примере платформы Microsoft.Net (MSDN)

2. Общая характеристика платформы MSDN

Платформа MSDN предназначена для разработки и исполнения приложений различных типов:

- автономное консольное приложение с использованием текстового интерфейса пользователя;
- автономное Windows-приложение с использованием графического интерфейса пользователя;
- библиотека классов, которые предназначены для использования в других приложениях;
- Web-приложение, доступ к которому выполняется через браузер и которое по запросу формирует Web-страницу и отправляет ее клиенту по сети;
- Web-сервис – компонент, методы которого могут вызываться через Интернет.

Приложение может выполняться в режиме управляемого кода или небезопасного кода.

В первом случае исходный код должен быть переведен на специально разработанный для платформы промежуточный язык MSIL. Для исполнения кода на промежуточном языке приложения используется специальная программная компонента платформы – общеязыковая среда исполнения CLR.

Во втором случае исходный код должен быть переведен на язык машинных команд. Машинный код исполняется непосредственно под управлением операционной системы.

Основные преимущества платформы проявляются в режиме управляемого кода. Этот режим принят по умолчанию. Все сказанное в дальнейшем относится к этому режиму.

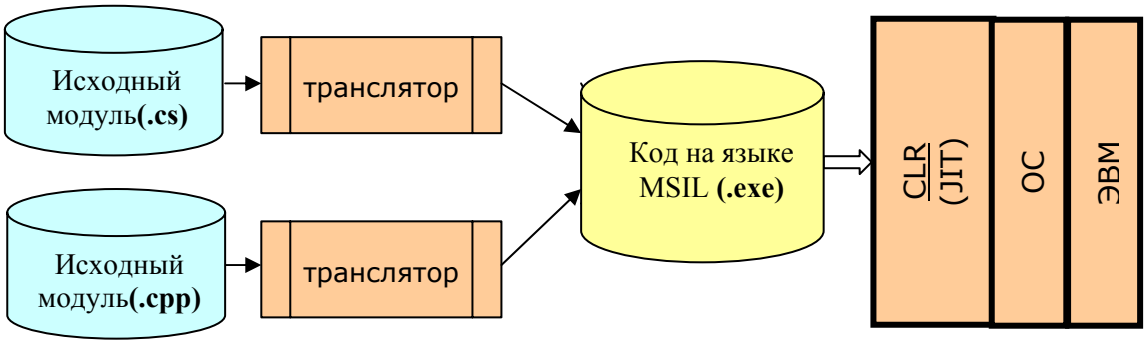


Рис. 4

Платформа поддерживает разработку приложений на нескольких языках. Для этого платформа содержит трансляторы для языков программирования C#, C++, Visual Basic, J#. Платформа открыта для включения трансляторов сторонних разработчиков с языков, удовлетворяющих требования общеязыковой спецификации типов (CTS).

Платформа обеспечивает возможность межязыкового взаимодействия, использование обширного набора готовых программных компонент, отслеживание несанкционированных действий со стороны программы по использованию основной памяти и внешних устройств.

Исходные модули транслируются на промежуточный язык MSIL, как показано на Рис.4. Код на промежуточном языке рассматривается средой исполнения CLR как исполняемый модуль. Среда исполнения представляет собой программу, надстроенную над операционной системой Windows ME,98,2000 или XP и выполняемую под ее управлением. С другой стороны, среда исполнения представляет собой функциональный аналог ЭВМ, - виртуальную машину, в которой выполняются программы на промежуточном языке.

При вызове метода среда исполнения активизирует транслятор JIT, который переводит код метода с промежуточного языка в машинный код и сохраняет его в памяти. При повторном вызове метода повторная трансляция не выполняется, используется машинный код, сохраненный в памяти. За счет такого подхода достигается эффективность, соизмеримая с эффективностью неуправляемого кода и экономия расхода основной памяти за счет перевода в машинный код не всей программы, а только тех методов, которые были реально вызваны в процессе выполнения программы.

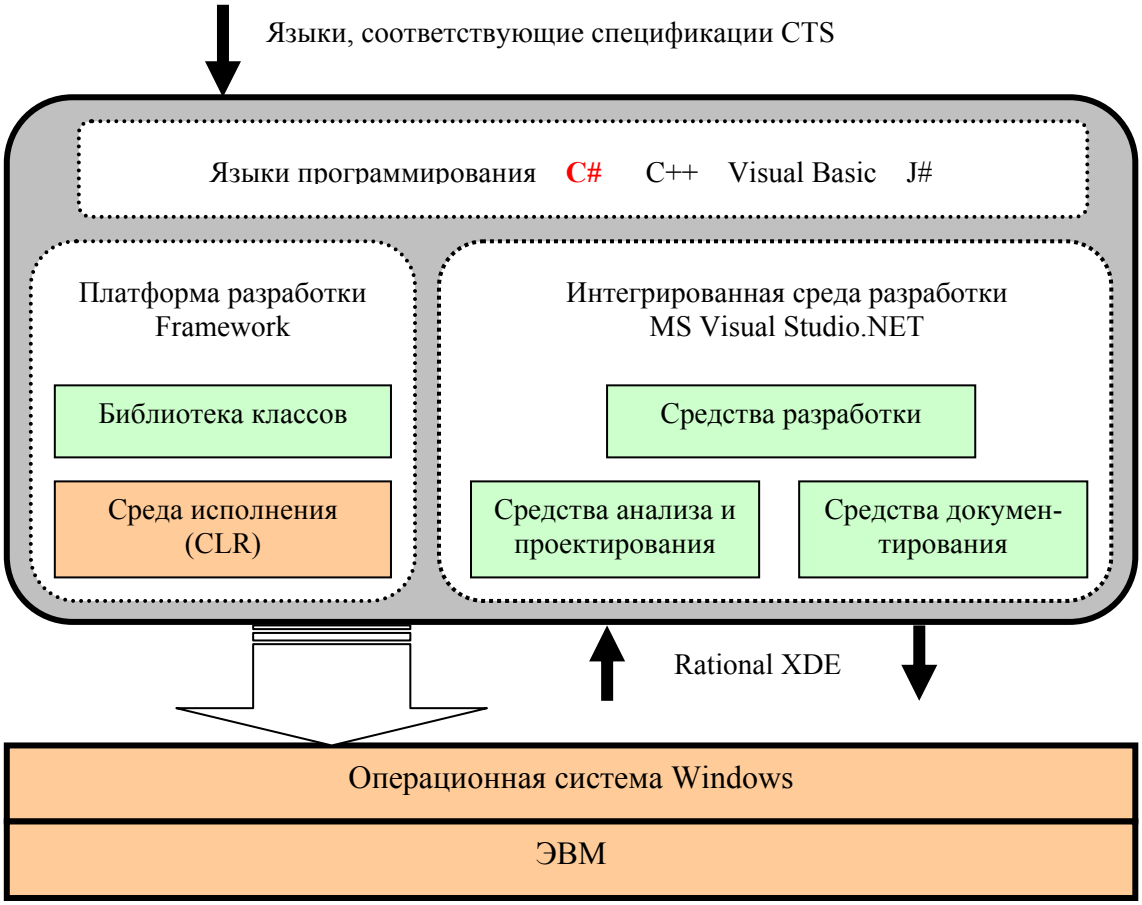


Рис. 5

Таким образом, каркас платформы образуют две компоненты, показанные на Рис.5:

- Статическая компонента – базовая библиотека классов, содержащая обширный набор готовых к использованию программных компонент на промежуточном языке. Базовая библиотека классов является общей для всех языков программирования, поддерживаемых в платформе.

- Динамическая компонента – общеязыковая среда исполнения (CLR).

Указанные компоненты являются обязательными для исполнения программ на промежуточном языке MSIL в случае использования на ЭВМ операционных систем Windows ME,98,2000 или XP. В перспективных операционных системах семейства Windows предполагается включение базовой библиотеки классов и средств исполнения в состав операционной системы.

Интегрированная среда разработки MS Visual Studio.NET представляет собой программную компоненту, поддерживающую процесс разработки программ. Возможности интегрированной среды для приложений на всех языках примерно равноценны, но в наибольшей степени возможности среды раскрываются при разработке программ на языке C#. С помощью средств MS Visual Studio.NET выполняется редактирование исходного кода, выполнение приложения в отладочном режиме, визуальное отображение логической структуры приложения, выдачf справочной информации по самой среде, платформе и языкам программирования, что является традиционным для большинства интегрированных сред. Особенностью MS Visual Studio.NET является возможность автоматической генерации исходного кода по визуальному представлению диаграммы классов в нотации UML и наоборот, построение визуального представления диаграммы классов по исходному коду программы.

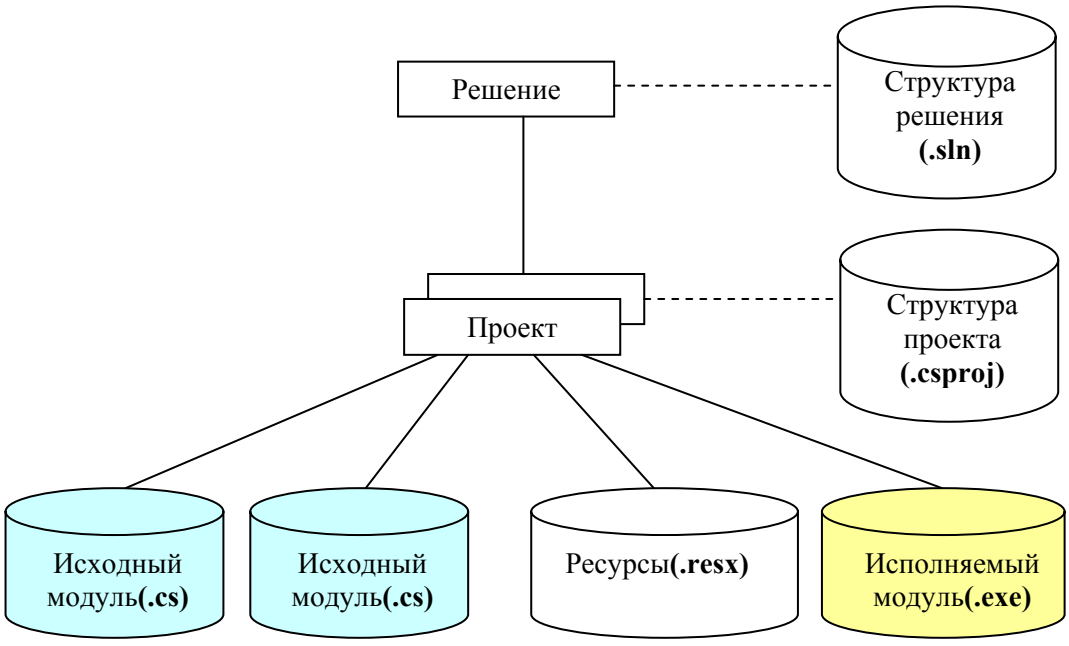


Рис. 6

В MS Visual Studio.NET может быть интегрировано специализированное инструментальное средство Rational XDE, ориентированное на решение задач анализа и проектирования программ на языке C# на основе объектно-ориентированного подхода, в результате чего возможности среды по анализу, проектированию и документированию существенно расширяются.

Приложение в процессе разработки называют проектом. Проект логически объединяет все необходимые для создания приложения файлы,

папки и прочие ресурсы. Типовая структура консольного приложения с некоторыми упрощениями приведена на Рис. 6.

Описание структуры проекта хранится в специальном файле с расширением **csproj.**

Несколько проектов логически могут быть объединены в одно решение. Структура решения хранится в файле с расширением **sln**. С помощью инспектора решения Solution Explorer, входящего в состав интегрированной среды можно просматривать логическую структуру всех проектов, включенных в решение и выполнять операции по изменению логической структуры решения и проектов.

СТАНДАРТНЫЕ ПРОСТЫЕ ТИПЫ ДАННЫХ

К *простым типам данных* относятся типы, которые не могут быть представлены в виде множества более простых элементов. В этом смысле переменную простого типа можем рассматривать как переменную, имеющую одно значение в текущий момент времени.

Все *стандартные простые данные* определены в MSDN на основе структур, т.е. относятся к типам значений. Поскольку эти типы определены в MSDN, ими могут пользоваться программы на любых языках, определенных на этой платформе. Язык C# имеет собственные синонимы этих типов.

Для обработки данных простых типов используются предопределенные *операции*, обозначаемые специальными знаками. Кроме того, можно использовать *поля и методы*, определенные в типе.

У всех простых типов есть статические поля, хранящие максимальное (MaxValue) и минимальное (MinValue) значения для данного типа данных.

Статический метод преобразования строки символов в значение заданного типа:

Тип.Parse (строка символов) => значение заданного типа

Метод объекта для преобразования значения в строку символов:

Объект.ToString() => строка символов, в которую преобразовано значение объекта.

Данные в программе могут присутствовать либо в виде переменных, либо в виде констант. Форма записи константы однозначно определяет ее значение и тип. Тип переменной должен быть определен путем явного объявления. Переменные при объявлении могут быть инициализированы любым значением, определенным на момент объявления.

1.Целочисленные типы

*Целые числа **со знаком***

Тип C#	Тип CTS	Длина, байт
sbyte	Sbyte	1
short	Int16	2
int	Int32	4
long	Int64	8

*Целые числа **без знака***

Тип C#	Тип CTS	Длина, байт
byte	Byte	1
ushort	UInt16	2
uint	UInt32	4
ulong	UInt64	8

Константы:

- Десятичные ±XXX, где X - десятичная цифра
- Шестнадцатеричные ±**0x**YYY,где Y – 16-ричная цифра

Тип константы определяется как первый подходящий, начиная с типа *int*. Возможно указание типа константы в явном виде (с помощью суффикса):

255L – имеет тип long (L-это суффикс)
255U – беззнаковая
255UL – беззнаковая длинная (ulong)

Примеры объявления переменных

int a, b=10, c=int.Parse("355"), d=b+c, e=short.MaxValue;

2. Вещественные типы

Тип C#	CTS	Длина,байт	Точность,цифр	Диапазон
float	Single	4	7	$\approx\|10^{-45}\|\div\approx\|10^{38}\|$
double	Double	8	15	$\approx\|10^{-324}\|\div\approx\|10^{308}\|$

Константы

- форма F ±XXX.YYY
- форма E ±МантиссаE ±Порядок

Примеры констант:

5.5 => 5,5 5.0 => 5,0 5. => 5,0
0.5 => 0,5 .5 => 0,5 2.5 E2 => $2,5*10^2$ => 250,0

Все константы имеют тип данных double. Тип константы может быть изменен с помощью суффикса F:
25.5 => double
25.5F => float

Примеры объявления переменных

double a, d=.5, c=double.Parse("5.5");

ОШИБКА! 5,5 - верно

3. Десятичный тип (денежный тип)

Представляет собой вещественное значение, причем способ кодирования этого значения не совпадает ни с одним способом кодирования вещественных чисел. Поэтому денежный тип несовместим ни с одним вещественным типом.

Тип C#	CTS	Длина,байт	Точность,цифр	Диапазон
decimal	Decimal	16	28	$\approx\|10^{-28}\|\div\approx\|10^{28}\|$

Константы

Целая или вещественная константа с суффиксом M: **25.5M**

4. Логический тип

Тип C#	CTS
bool	Bool

Константы

false – ложь
true - истина

Логический тип несовместим ни с каким другим типом.

5 Символьный тип

Тип C#	CTS
char	Char

На один символ отводится область памяти длиной 2 байта. В этой области содержится код символа в таблице кодировки Unicode.

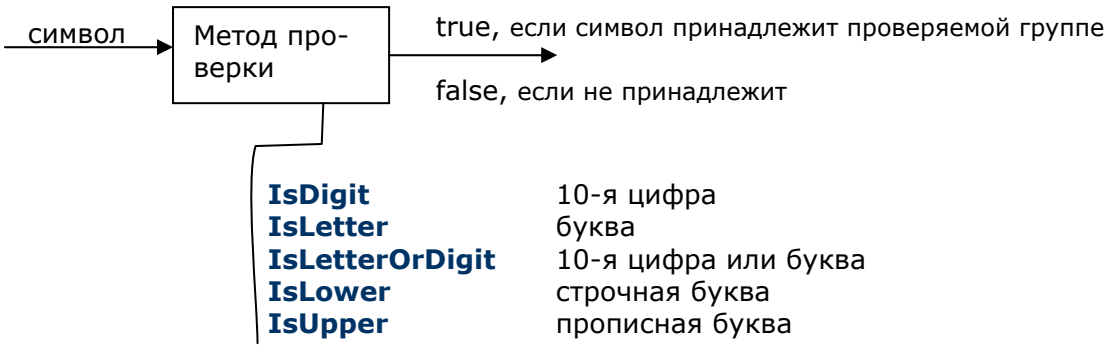
Константы записываются в виде управляющей последовательности или в виде символа, заключенного в апострофы:

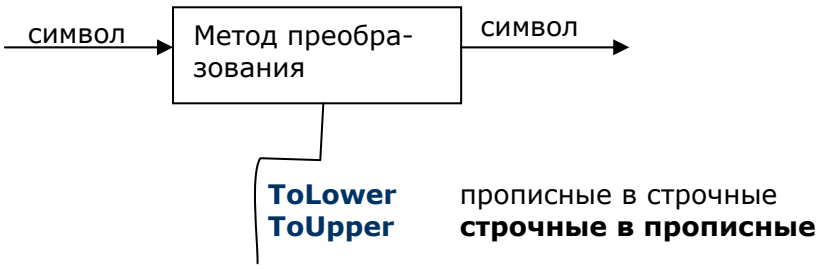
´\uXXXX´ - где XXXX – 16-й код символа по таблице кодировки

'Символ'

Пример: 'A' или ´\u0041' - это код латинской буквы A

Символьный тип имеет статические методы:





ОПЕРАЦИИ СО СТАНДАРТНЫМИ ПРОСТЫМИ ТИПАМИ ДАННЫХ

1 Выражение и оператор

Выражение – операнды, объединенные знаками операций. Любое выражение, завершенное ";" – *оператор*. Одиночный символ ";", не относящийся ни к одному оператору – пустой оператор.

Операндами могут быть:

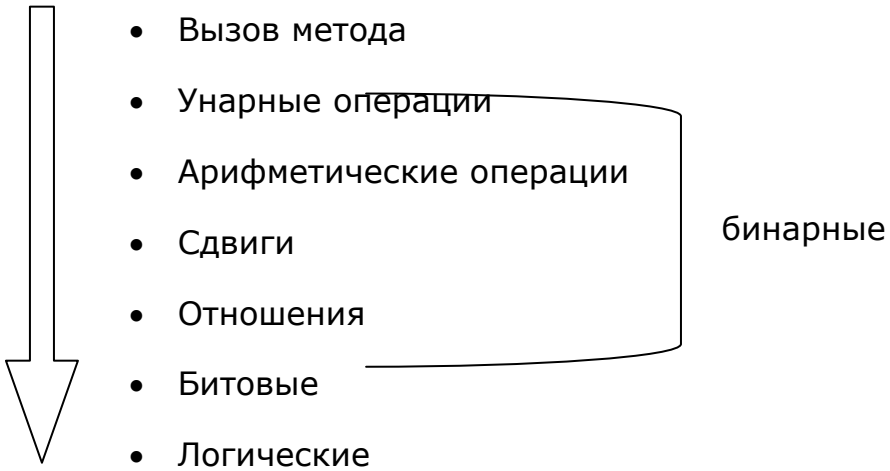
1. Вызов метода
2. Переменная
3. Константа
4. Выражение

Операции имеют *приоритет*. Приоритет предписывает очередность выполнения операций в выражении. Операции одного приоритета выполняются слева направо. Исключения будут оговорены особо. Порядок операций можно регулировать скобками "()".

В зависимости от числа операндов различают:

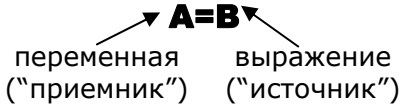
1) Унарные
2) Бинарные
3) Тернарные

Приоритеты операций по группам:



2 Операции присваивания

– Простое присваивание

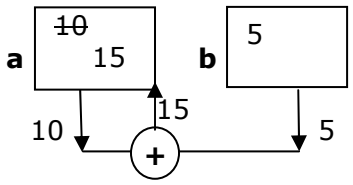


1. Вычисляется значение **B**

2. Если тип переменной **A** и тип выражения **B** не совпадают, то значение **B** преобразуется к типу **A**

3. Значение **B** заносится в область памяти, отведенную для переменной **A**

```
int a, b;

a=10;

b=5;

a=a+b;
```



Операции присваивания выполняются справа налево.

```
int a, b, c;

a=b=c=100;
```

- Составное присваивание объединяет присваивание и одну из бинарных операций

  A * = B => A = A * B, где * – бинарная операция

  ```
  int a=10;

  a+=3; // a=a+3; a=13
  ```

Преобразование типов выполняется автоматически при выполнении следующих условий:

1. Типы совместимы
2. Преобразование не приведет к потери информации

Совместимые типы

Приемник	Источник
bool	bool
char	char
decimal	decimal, char, целый тип
целый тип	целый тип, char
вещественный тип	вещественный тип, целый тип, char

Если типы совместимы, проверяется возможность потери информации при преобразовании. Потери возможны:

1. При присваивании целой переменной вещественного значения (теряется дробная часть)

2. При присваивании переменной с меньшим диапазоном значений значения переменной с большим диапазоном значений.

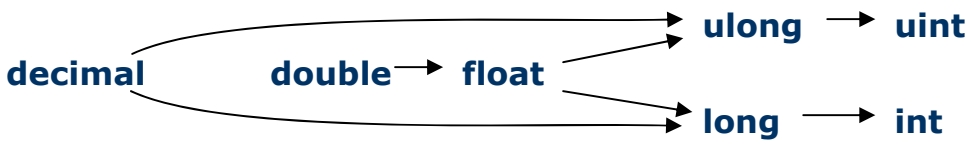
В случае возможности потери информации фиксируется ошибка.

Рассмотренная схема преобразования типов уникальна только для операции присваивания, во всех остальных операциях действует другая схема.

2. Преобразование типов операндов при выполнении операций

Перед выполнением операции все операнды целого типа со знаком, которые по длине короче, чем int преобразуются в int, а беззнаковые операнды короче uint, преобразуются в uint.

Перед выполнением операции операнд младшего типа преобразуется к операнду старшего типа. Старшим считается тип, у которого больше диапазон значений.



Больше Меньше

Исключение

Особая ситуация: операнды типов **uint** и **int**. Эти типы имеют одинаковый по величине диапазон значений, но смещенный на числовой оси. В этом случае оба операнда преобразуются к типу long**.**

Тип результата операции будет совпадает с типом операндов после преобразования.

Преобразования выполняются при выполнении операции и не влияют на хранение значений операндов в памяти.

4 Арифметические операции

Унарные

- Изменение знака
- Унарный +
- Инкремент
- Декремент

Изменение знака

```
int a=10, b=20, c, d;
c=-a; //   c=-10
d=+b; //  d=20
```

Инкремент предназначен для увеличения значения переменной на единицу, а ***декремент*** – для уменьшения значения переменной на единицу.

```
int a=10;
a++; // a=a+1; a=11
```

или

```
++a; // a=a+1; a=11
int a=10;
a--; // a=a-1; a=9
```

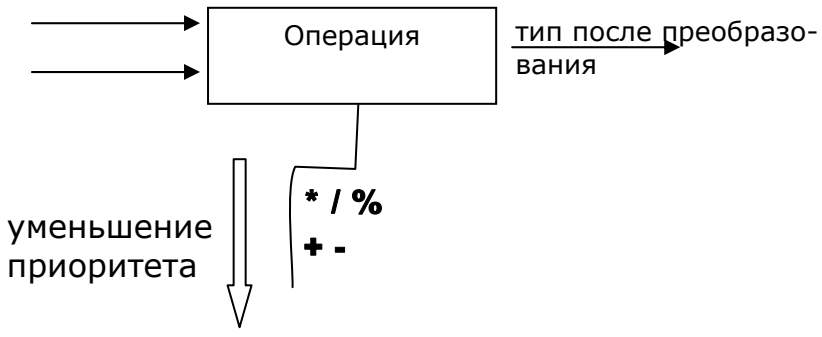
или

```
--a; // a=a-1; a=9
```

Операции инкремент и декремента могут использоваться как в префиксной форме (знак перед переменной), так и в постфиксной форме (знак после переменной). Отличия будут проявляться при использовании операции в выражении: префиксная – сначала изменяется значение переменной, потом используется новое значение; постфиксная – сначала используется старое значение переменной, потом изменяется значение переменной.

```
int a=10, b=10, x, y;
x=a++;  //.a=11 → x=10
y=++b;  // b=11 → y=11
```

Бинарные операции




```
int a, b;
double c;
a=10/4;      //a=2
c=10/4.;     //c=2.5
b=10./4;     //ошибка!!! (10./4.=2.5)
```

int ← **double** при присваивании возможна потеря информации!!!

```
int a;
a=10%3;     //a=1
a=-10%3;    //a=-1
a=10%-3;    //a=1
a=-10%-3;   //a=-1
```

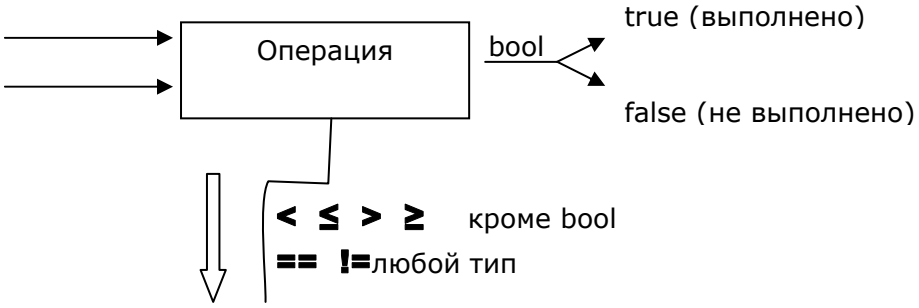
5 Операция явного преобразования типа

(Тип приемника) операнд

преобразование к типу приемника

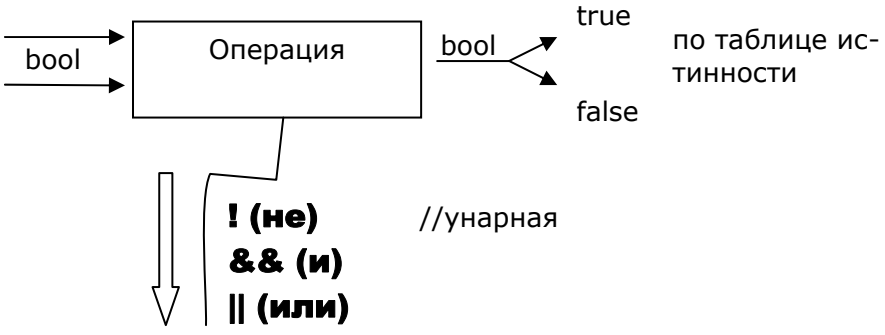
```
int a;
double b;
a=(int)3.5;              //a=3
b=(int)3.5+5.5;//b=3+5.5=8.5
```

6 Операция явного преобразования типа



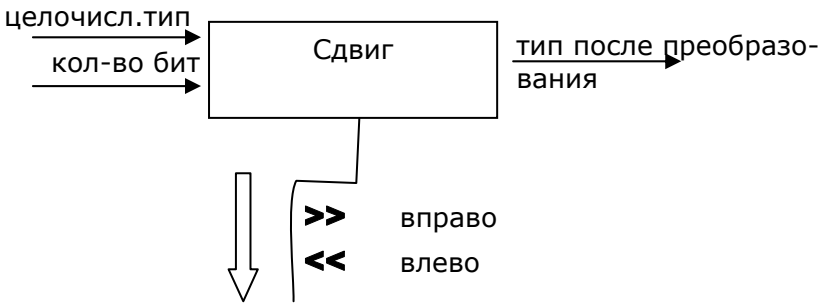

```
bool a, b, c;
a= 10>5;        // true
b= 'B'>'A';     // true
c= 65=='A';     // true
```

7. Логические операции




```
bool a;
a= !(10>5 && 3==8);        //true
```

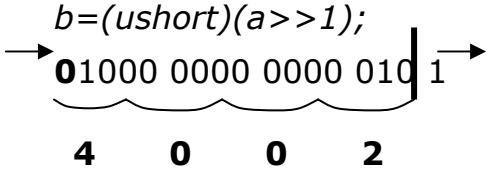
8. Операции сдвига



Для знаковых типов – арифметический сдвиг, для беззнаковых – логический.

ushort a=0x8005, b;

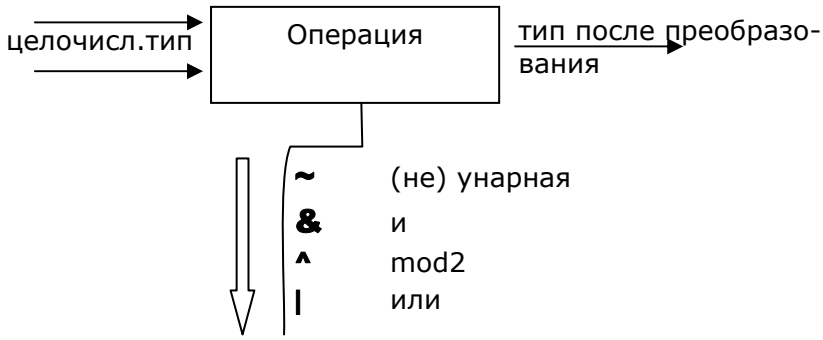
b=(ushort)(a>>1);



short a=-0x1005, b;

b=(short)(a>>1);

9. Битовые операции



byte a=0x1F, b=0x3A, d;

d=(byte)(~(a^b));

Операции выполняются поразрядно:

0001 1111

0011 1010

0010 0101 **a^b**

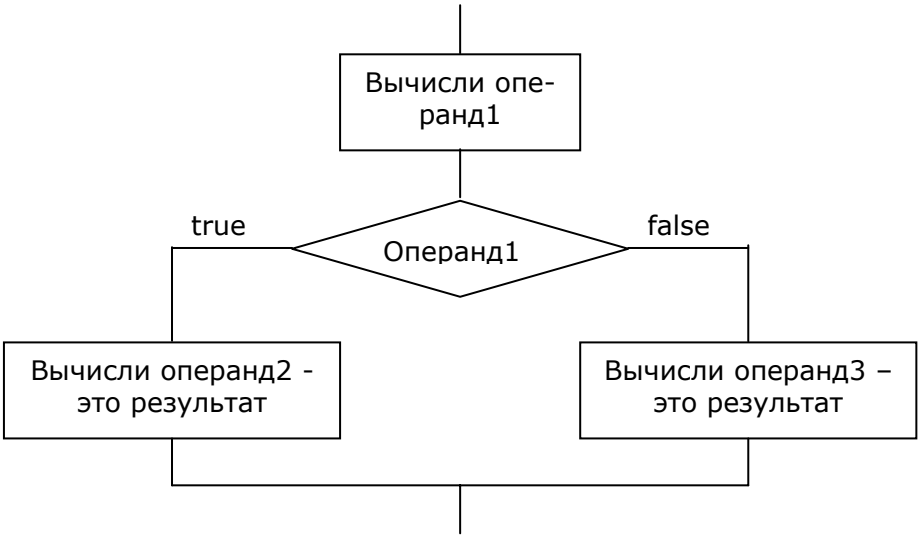
1101 1010 **~(a^b)**

10. Тернарная операция

Предназначена для реализации простейшего варианта ветвления.

Операнд1? Операнд2:Операнд3

Выполнение операции иллюстрируется приведенной ниже схемой, а применение операции для вычисления минимального из трех целых чисел – фрагментом программы.




```
int a, b, c, min;
…
a=10; b=5; c=8;
min = a;
min = b < min ?  b  :mn;
min = c < min ? c  :mn;
```

ОПЕРАТОРЫ УПРАВЛЕНИЯ

В языке C# реализован полный набор управляющих конструкций структурного программирования: следование, ветвление, цикл. Кроме того, имеются дополнительные операторы управления.

1. Следование

Операторы по умолчанию выполняются последовательно в порядке записи. Операторы, заключенные в фигурные скобки {} образуют блок операторов. Блок операторов рассматривается транслятором как один составной оператор.

{оператор … оператор} *блок операторов*

В качестве оператора могут быть:

- объявление
- выражение, заканчивающееся символом точки с запятой
- управление
- блок операторов

Поскольку блок операторов может содержать объявление переменных, то возникает проблема области видимости локальной переменной. Проблема решается следующим образом: переменная, объявленная в блоке, видна в этом блоке и во вложенных блоках, но не видна в охватывающем блоке.

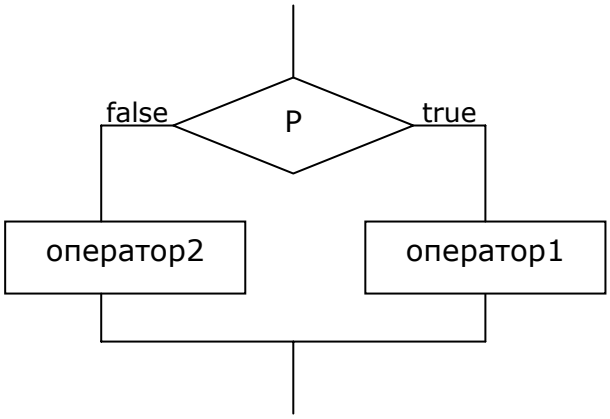
2. Ветвление

if (логическое выражение P)

оператор1

else

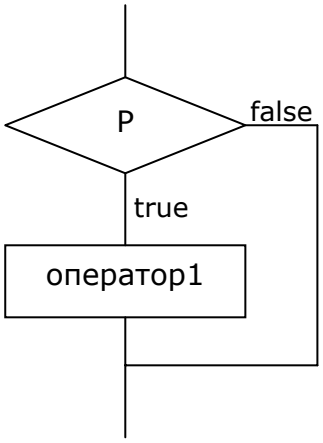
оператор2



В качестве оператора может использоваться любая из управляющих конструкций в виде строго одного оператора, но этот оператор может быть блоком. Это правило справедливо для всех управляющих конструкций.

Допускается сокращенное ветвление.

if (P) оператор;



Пример:

Вычислить значение переменной, заданной соотношением

$$y=\begin{cases} x^2, \text{ если } x<0 \\ x+10, \text{ если } x\geq 0 \end{cases}$$

Фрагмент программы с использованием ветвления:

```
double x, y;
x = double.Parse(Console.ReadLine());
if (x < 0)
    y = x * x;
else
    y = x + 10;
```

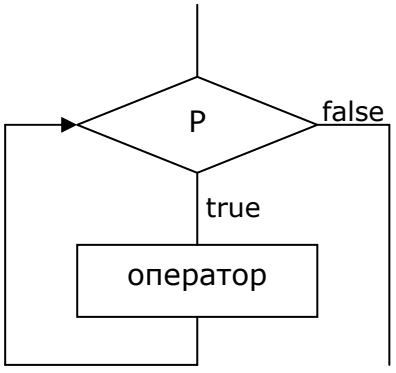
Фрагмент программы с использованием ветвления:

```
double x, y;
x = double.Parse(Console.ReadLine());
y = 0.0;
if (x < 0)
    y = x * x;
if (x>=0)
    y = x + 10;
```

3. Циклы

<u>Цикл с предусловием</u>

while (P) оператор

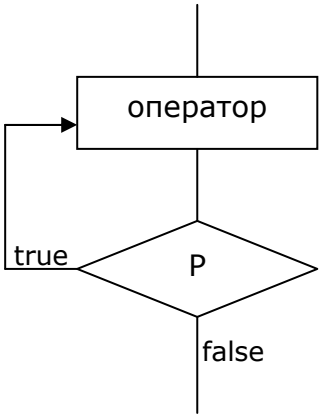


<u>Цикл с постусловием</u>

do

оператор

while (P);



Универсальный цикл

for(выражение1; выражение2; выражение3) оператор

Схема выполнения универсального цикла:

1. вычисляется выражение1, которое играет роль начальной установки параметров цикла
2. вычисляется выражение2, которое играет роль условия повтора цикла. Если значение этого выражения false, выполнение цикла завершается, в противном случае переходим к шагу 3.
3. Выполняется оператор, образующий тело цикла
4. вычисляется выражение3, которое играет роль модификатора параметров цикла
5. переход на шаг 2

Любое из выражений в заголовке цикла может отсутствовать, при этом разделяющие их скобки должны сохраняться. Отсутствующее выражение2 означает, что условие повтора цикла имеет значение **true**.

Примеры использования циклов

```
int i=0; //Счетчик
while (i<2)
{
   Console.WriteLine ("Снег"); //1
   i++;                        //2
}
```

Шаг	**i в начале шага**	**Экран**	**i в конце шага**
1	0	Снег	1
2	1	Снег	2
3	2		

Если не образовать блок операторов, заключив строки 1 b 2 в фигурные скобки, то цикл будет выполняться до бесконечности, так как изменение значения счетчика не будет.

```
int i=0;
do
{
   Console.WriteLine ("Снег");
   i++;
}
```

while (i<2);

Шаг	**i в начале шага**	**Экран**	**i в конце шага**
1	0	Снег	1
2	1	Снег	2

int i;
for (i=0;i<2;i++)
Console.WriteLine ("Снег");

Шаг	**i в начале шага**	**Экран**	**i в конце шага**
1	0	Снег	1
2	1	Снег	2
3	2		

for (;;)
Console.WriteLine ("Снег");.//Бесконечный вывод этого сообщения

Выражение инициализации может содержать объявление локальных переменных. Локальные переменные, объявленные в заголовке цикла, видны только внутри цикла.

Выражение инициализации может содержать несколько выражений, разделенных "**,**", а объявлений – не более одного

Пример

Вычислить сумму n элементов следующего ряда. Сумму выводить после прибавления каждого слагаемого.

$$S=\sum_{i=1}^{N} i/(i^2+1) = 1/2 + 2/5 + 3/10 + \ldots + N/N^2+1$$

int N;
string str; // рабочая строка для вывода на экран
int i;
double s; // переменная, в которой будет накапливается сумма
N=5;
for (i=1, s=0; i<=N; i++, Console.WriteLine(str))
{
*s+=(double)i/(i*i+1.0);*
str=s.ToString();
}

Второй вариант решения задачи
int N;
string str;
double s;
N=5;
s=0;
for (int i=0; i<=N; i++, Console.WriteLine(str))

```
{
   s+=(double)i/(i*i+1.0);
   str=s.ToString();\
}
```

4. Дополнительные операторы управления

- Принудительное завершение цикла **break;**
- Принудительное завершение текущего шага цикла и переход на следующий шаг цикла **continue;**
- Безусловный переход в заданную точку программы

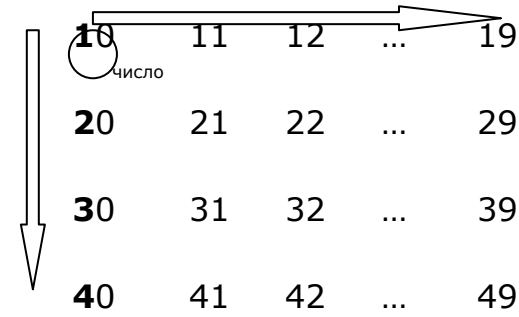
goto_метка;

метка: оператор;

- Возврат из метода **return;**

Пример

Сгенерируем сочетание из двух десятичных цифр, первая – от 1 до 4, вторая – от 0 до 9. Первая цифра рассматривается как старшая цифра числа, вторая – как младшая. Вывести на экран числа кратные 5.



10	11	12	…	19
20	21	22	…	29
30	31	32	…	39
40	41	42	…	49

```
public static void Main()
{
  int a, b, c;// старшая цифра, младшая цифра, число
  for (a=1; a<=4; a++)
    for (b=0; b<=9; b++)
    {
      c=a*10+b;
      if (c%5 !=0) continue;
        Console.Write (c.ToString()+" ");
        //break;
        //goto_EXIT;
        //return;
    }
  EXIT:  Console.WriteLine ("\nВсе!");
}
```

Результат вывода в исходном варианте программы

10 15 20…25 30 35 40 45

ВСЕ!

Убираем комментарий около break;

Выполнение оператора break приведет к прекращению перебора младшей цифры при обнаружении первого же числа, кратного 5.

Результат вывода

10 20 30 40

ВСЕ!

Комментируем оператор break и снимаем комментарий с оператора goto_EXIT. При обнаружении первого же числа, кратного 5, произойдет выход на метку EXIT:

Результат вывода

10

ВСЕ!

Оператор безусловного перехода целесообразно применять в единственном случае - для выхода из вложенного цикла за пределы внешнего цикла.

Убираем комментарий с оператора return и восстанавливаем комментарий около оператора goto EXIT. При обнаружении первого же числа, кратного 5, будет выполнен оператор return. Поскольку в данном случае оператор возврата применен к методу Main(), это приведет к завершению программы.

Результат вывода

10

5. Множественное ветвление

Конструкция предназначена для выбора одной из нескольких ветвей.

switch (выражение – переключатель)

целочисленное значение, строка, char

{

case_Маркер: операторы; break;

константа

…

case_Маркер: операторы; break;

<default: операторы; break;> //необязательная часть

}

Схема выполнения:

1.Вычисляется значение переключателя.

2.Значение переключателя последовательно сравнивается с маркером ветвей.

3.Выполняется та ветвь, маркер которой совпал со значением переключателя.

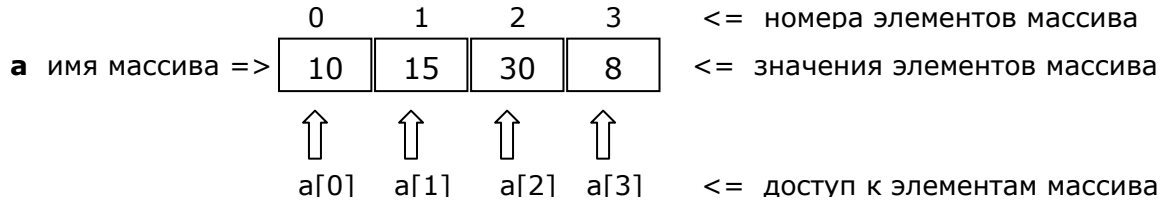
Одна из ветвей может иметь стандартный маркер *default.* При наличии такого маркера, отмеченная им ветвь будет выполняться в том случае, когда переключатель не совпал ни с одним маркером.

ВВЕДЕНИЕ В МАССИВЫ И СТРОКИ

1. Понятие массива

В общем случае массив представляет собой набор элементов одного типа, расположенных в непрерывной области памяти.

Массив имеет тип, каждый элемент имеет уникальный номер. Доступ к элементу массива выполняется путем указания порядкового номера и имени массива.



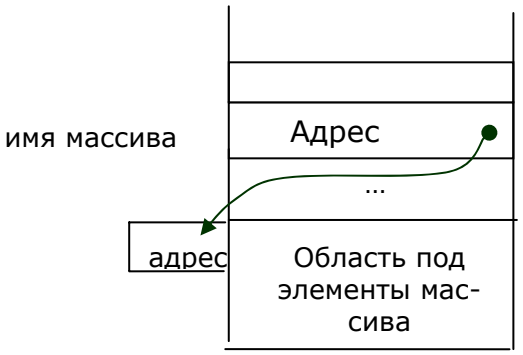
В языке C# массив определен как класс с именем *Array*. Следовательно, массив относится к ссылочным типам данных. Различают:

- Одномерный массив
- Многомерный массив
- Массив массивов

2. Объявление массива и его инициализация

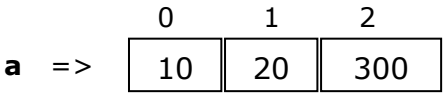
```
Тип элементов[]  ИмяМассива;
ИмяМассива = new Тип элементов[кол-во элементов];
```

```
int[] a;
a=new int[3];
```



При выделении памяти значения элементов могут быть определены явным образом, при этом количество элементов указывать не обязательно.

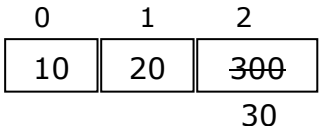
```
int[] a;
a=new int[]{10, 20, 300};
```



3. Операции

Операции с элементами – любые операции, определенные для того типа, к которому относятся элементы массива.

a[2]=a[0]+a[1]



Операции с массивом:

- присваивание (копируется значение ссылки)
- отношение (сравниваются значения ссылок)

int[] a, b, c;

bool f1, f2;

a=new int[]{10, 20};

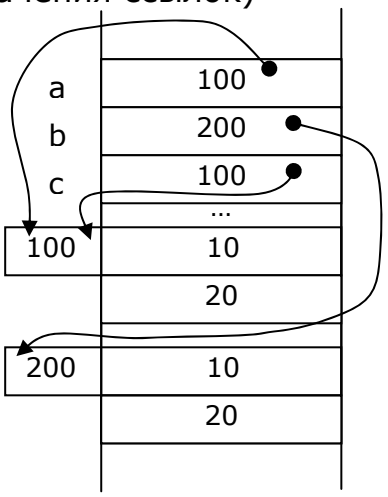
b=new int[]{10, 20};

f1= a==b; //false 100≠200

c=a;

f2= c==a; //true 100=100

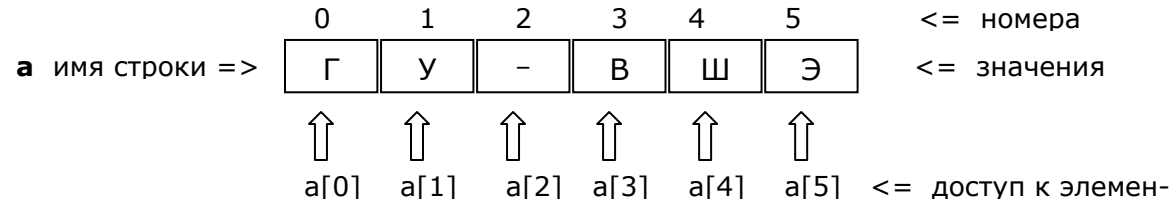
b=a;



Область памяти, на которую потеряна ссылка (в данном случае область памяти по адресу 200), ставится в очередь на освобождение. Эта память будет автоматически освобождена программой MSDN «Сборщик мусора».

4. Понятие строки

Строка предназначена для хранения текста в виде множества символов (кодировка Unicode: 1 символ – 2 байта).



Отличия от массива

Хотя строка структурно подобна массиву, но формально строка не является массивом и имеет собственный тип данных string (String – синоним в MSDN) со своим набором свойств и методов.

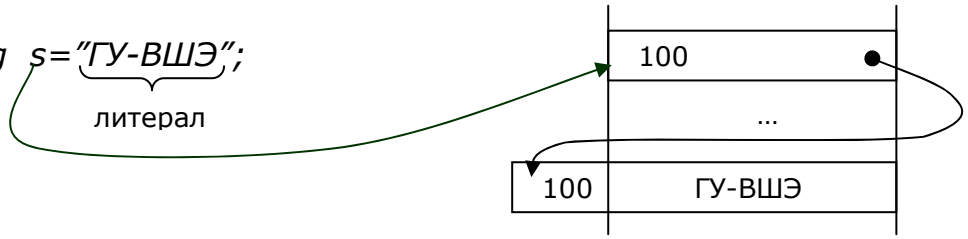
Тип строка объявлен как класс, т.е. строка – это ссылочный тип. Тип строка относится к стандартным типам данных, т.е. все свойства заранее определены.

Операции с объектами типа строка содержательно выполняются не так, как операции с массивами.

5. Объявление и инициализация строк

string ИмяСтроки;

Размер строки указывать не требуется, он будет автоматически определен при инициализации или присавивании.



6. Операции со строками

Операции над элементами строки – это все операции, определенные для типа *char*. Исключение: над элементом строки запрещена операция присваивания, т.е. значение элемента строки изменить непосредственно невозможно.

int Код; string s="ABBA";

Код=s[0]; // Код => код символа 'A'=65

~~*s[0]='Z';*~~ *// ошибка!*

Альтернативой непосредственному изменению элемента строки является формирование новой строки с нужным значением элемента.

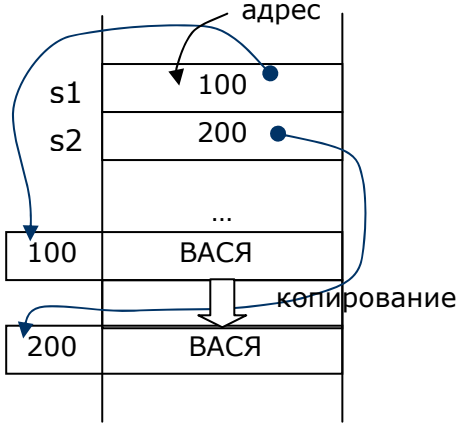
Операции над строками

Присваивание:

В отличие от массива копируется не ссылка, а значение строки.

string s1="ВАСЯ", s2;

s2=s1;



Отношения:

Операции равно(==) и неравно(!=) равносильны сравнению значений строк.

s2==s1; //true ВАСЯ=ВАСЯ

В этом заключается отличие строки от массива - для массивов сравниваются ссылки.

Операции больше(>), больше или равно(>=), меньше(<), меньше или равно(<=) равносильны сравнению ссылок.

s2>=s1; //false 100>=200

При выполнении указанных операций отличий между массивом и строкой нет – в обоих случаях сравниваются ссылки.

Для того, чтобы сравнить строки на предмет больше/меньше по значению необходимо использовать метод *CompareTo*.

Сцепление строк:

строка1+строка2=>новая строка - объединение исходных

В операции сцепления могут использоваться помимо строки операнды других типов. При этом старшим считается тип строки.

int a=10, b=20;

string s1, s2, s3;

s1="Итого:"+a; // Итого:10

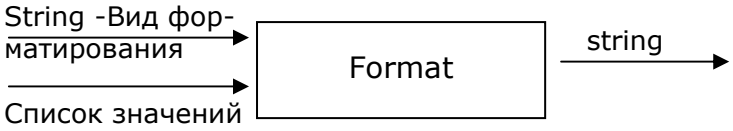
s2=" Итого:"+a+b; // Итого:1020

s3=a+b+"= Итого"; //30= Итого

7. Форматирование строки

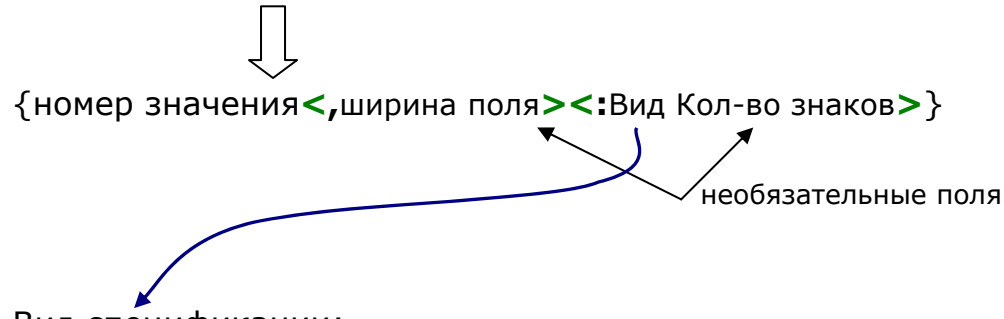
Класс *string* содержит статический метод *Format()*, с помощью которого можно сформировать строку на основе заданных значений в заданной программистом форме.

Метод *Format* имеет две группы параметрова:



Вид форматирования:

- обычные символы (копируются в строку)
- спецификаторы



Вид спецификации:

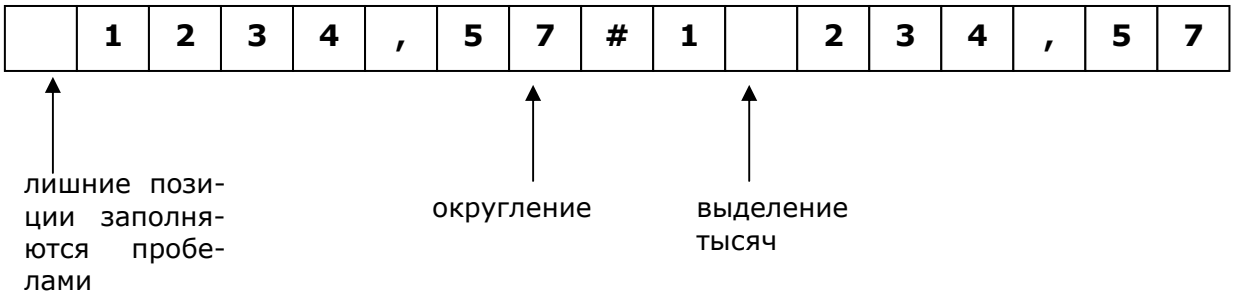
- **D** или **d** - целое число

- **F** или **f** – вещественное число в форме F (ЦЧ и ДЧ)
- **E** или **e** – веществ. число в форме E (мантисса и порядок)
- **N** или **n** – вещественное число с выделением тысячи
- **X** или **x** – целое 16-ричное число

Номера значений отсчитываются с нуля, выравнивание в заданном поле производится по правой границе, если ширина поля указана отрицательным числом, то выравнивание производится по левой границе.

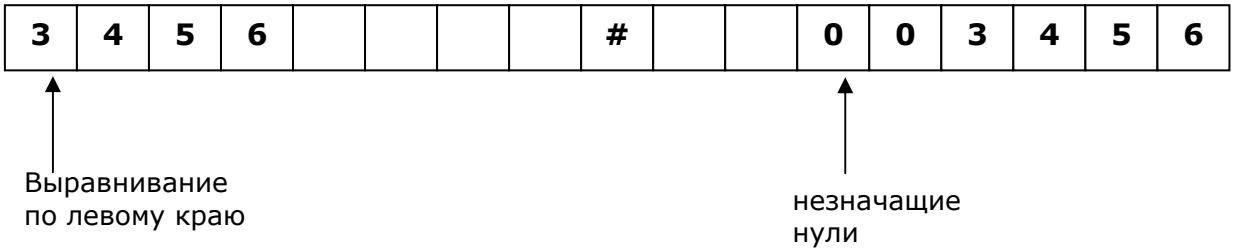
Вещественные числа округляются по правилам арифметики. Если вид спецификации не указан, он определяется по типу выводимого значения. Количество знаков для вещественных чисел определяет количество знаков после запятой, а для целых – необходимость выводить незначащие нули.

```
double a=1234.5678;
long b=3456;
short c=65;
string s1, s2, s3;
s1=string.Format("{0,8:f2}#{1,8:n2},a,a);
```



Если поля для вывода недостаточно, ширина будет автоматически увеличена до нужного размера.

```
s1=string.Format("{0,-8:d}#{1,8:d6},b,b);
```

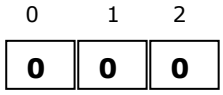


МАССИВЫ И СТРОКИ

1. Одномерные массивы

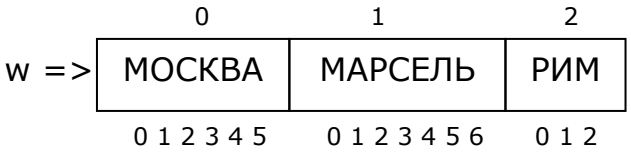
Одномерный массив - набор элементов одного типа, тип элементов стандартный или определен в разрабатываемой программе.

double[] a;
a=new double[3];



Тип может быть любой, например строковый:

string[] w;
w=new string[]{“МОСКВА”, “МАРСЕЛЬ”, “РИМ”};



string s=“”; // пустая строка
s=s+w[0][0]+w[2][1]+w[1][2]; // s => МИР

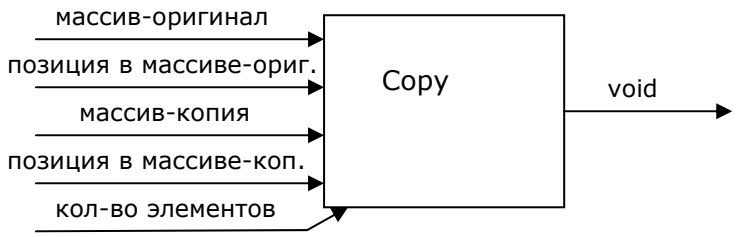
Тип массива определен как класс **Array** (т.е. ссылочный тип) в пространстве имен **System**.

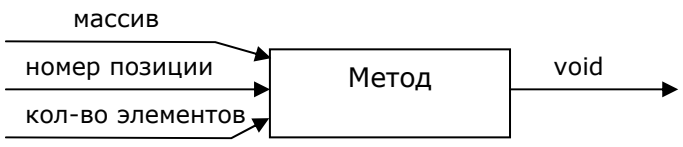
Обработка массива производится поэлементно, выход за границы массива фиксируется как исключение.

Дополнительные возможности определяются полями и методами.

Для массива определено поле объекта – **Length**, определяющее количество элементов в массиве.

Статические методы:





- **Sort**
- **Reverse**

Метод **Copy** выполняет копирование элементов из массива оригинала в массив копию. Метод **Sort** сортирует заданную часть массива по возрастанию значения элементов. Метод **Reverse** выполняет переворот заданной части массива.

```
int[] a,b; //массивы a и b
a=new int[]{10,20,30,40};
b=new int[a.Length]; // 2-ой массив той же длины
Array.Copy (a, 1, b, 0, 2); // копирование
Array.Sort (b, 0, b.Length); // отсортировали все элементы из b
Array.Reverse (b, 1, 2);
//Массив a:                          10  20   30  40
//Массив b после копирования:     20  30    0    0
//Массив b после сортировки:       0    0   20  30
//Массив b после переворота:       0  20    0   30
```

2. Динамические массивы

Размер обычного массива фиксируется при его создании и в процессе выполнения программы напрямую его изменить невозможно. На практике встречаются задачи, когда количество элементов в массиве должно меняться в процессе выполнения программы.

Альтернативой массиву в этом случае является динамический массив. Динамический массив определен как класс **ArrayList** в пространстве имен **System.Collections**.

Принципиальное отличие от обычного массива: размер динамического массива автоматически изменяется при добавлении и удалении элементов, которые выполняются специальными методами.

Поле объекта: **Count** – количество элементов в массиве

Методы объекта для добавления элементов в массив:

Add (значение**)** // добавление элемента в конец массива

AddRange (массив**)** // добавление массива в конец массива

Insert (номер позиции, значение**)** //добавление элемента в заданную позицию

InsertRange (номер позиции, массив**)** // добавление массива с заданной позиции

Методы объекта для удаления элементов из массива:

Remove (значение**)**

RemoveAt (номер позиции**)**

RemoveRange (номер позиции, количествово элементов**)**

Clear () // удаление всех элементов

Пример:

```
ArrayList_a;

int[] b,c;

b=new int[]{10, 20};

c=new int[100];

a=new ArrayList();                    a          Count
```

	пусто	0
a.Add(5);	5	1
a.Insert(0,77); // в позицию 0 значение 77	77 5	2
a.AddRange(b);	77 5 10 20	4
a.CopyTo(1,c,0,3);	4	
a.InsertRange(0,b);	10 20 77 5 10 20	6
a.Remove(20);	10 77 5 10 20	5
a.RemoveAt(3);	10 77 5 20	4
a.RemoveRange(1,2);	10 20	2
a.Clear();	пусто	0

Если действие выполнить невозможно, действие не выполняется. Если выполняемое действие приводит к выходу за границы массива, фиксируется исключительная ситуация.

Выбор, каким массивом пользоваться, зависит от ситуации. Обычный массив структурно проще, доступ к элементу быстрее, методов больше.

У динамического массива можно без проблем удалять/добавлять элементы, легко изменять размер.

На практике используется сочетание динамического и обычного массивов. Это порождает проблему создания динамического массива на основе обычного и наоборот.

*динамический массив(**a**) → обычный массив(**b**)*

a.CopyTo (позиция в дин.мас-ве ,обычный массив, позиция в об.мас-ве, кол-во элементов)

*обычный массив(**b**) → динамический массив(**a**)*

a.AddRange (b) //добавление обычного массива в пустой динамический

3. Строки

Понятие строки и операции со строками были рассмотрены ранее. Остановимся на *дополнительных возможностях, которые предоставляются методами класса string.*

Поле объекта: **Length**

Методы объекта:

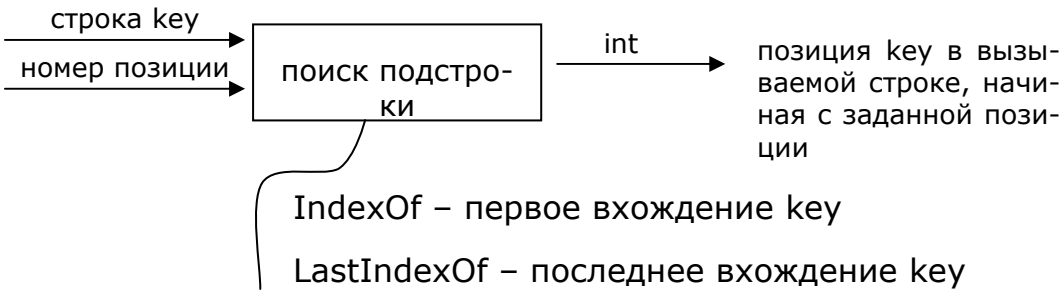
сравнение строк на предмет больше-меньше

ИмяСтроки.**CompareTo (**строка, с которой сравниваем**)**

Сравнение выполняется лексикографичекси (по алфавиту), метод возвращает целое число:

-1, если строка < той, с которой сравниваем
0, если строка = той, с которой сравниваем
+1, если строка > той, с которой сравниваем

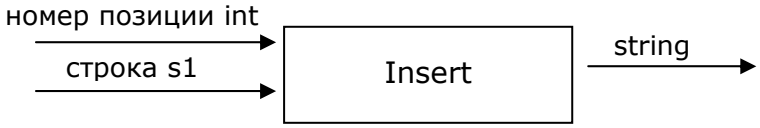
поиск подстроки



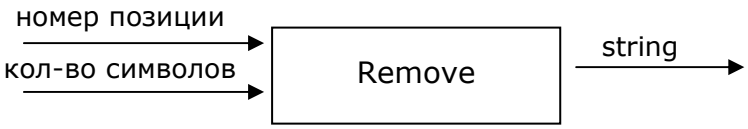
Возвращает -1, если key отсутствует

Замена в вызывающей строке всех подстрок s1 на новую подстроку s2.

Вставка в вызывающую строку строки s1 с заданной позиции.



Удаление в строке заданного количества символов, начиная с заданной позиции.



Разбиение вызывающей строки на слова. Возможные разделители между словами указываются в массиве символов.

Если несколько разделителей идут подряд, первый из них считается разделителем, а на месте остальных формируется пустая строка.

Формирование строки путем соединения строк, указанных в массиве. Слова в объединенной строке разделяются строкой-разделителем.

Метод Join - статический

string s1="око , за";

string[] word;

char[] sep; // массив разделителей

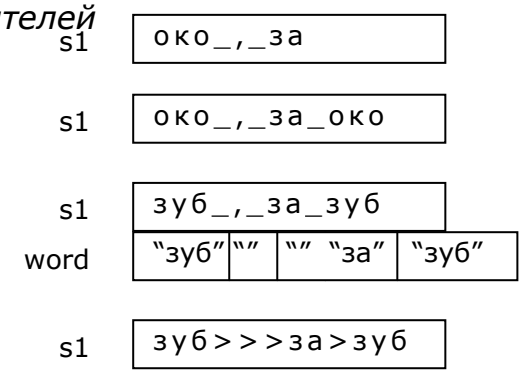
sep = new char[]{' ', ',', '.'};

s1=s1.Insert(8, "_око");

s1=s1.Replace("око", "зуб");

word=s1.Split(sep);

s1=string.Join (">", word);



4. Динамические строки

Тип динамической строки определен как класс – **StringBuilder**, пространство имен **System.Text**.

Создание динамической строки
StringBuilder ИмяСтроки;
ИмяСтроки=new StringBuilder();
Отличия от обычной строки:

1. Элементы динамической строки можно изменять напрямую, путем присваивания.
2. В операциях ==, !=, = участвуют не элементы строки, а адреса (аналогично массиву).

Поле объекта: **Length**
Методы:
ИмяСтроки.**Append (**символ**)** //добавление символа в конец строки
ИмяСтроки.**Append (**символ, кол-во**)** //добавление заданного количества символов в конец строки
ИмяСтроки.**Remove (**номер позиции, кол-во символов**)** //удаление заданного кол-ва символов с заданной позиции

По сравнению со строками динамические строки имеют меньший набор методов, поэтому при решении задач используется комбинация строк и динамических строк, что порождает проблему их преобразования.

Схема выполнения преобразований:

string s => char[] a

s.ToCharArray;

char[] a => string s

s=new_string(a);

string s => StringBuilder b

b=new_ StringBuilder(s);

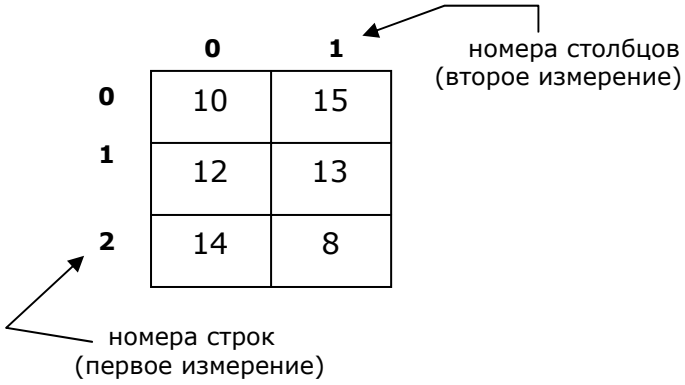
StringBuilder b => string s

s=b.ToString();

5.Двумерные массивы

Предназначены для хранения данных, организованных в виде прямоугольной таблицы (матрицы). Таблица имеет определенное количество строк и столбцов, которые нумеруются, начиная с нуля.

Обработка массива производится поэлементно. Для доступа к элементу матрицы необходимо указать номер строки и столбца, на пересечении которых находится этот элемент.



Объявление двумерного массива:

Тип элементов [,] ИмяМассива;

ИмяМассива=new Тип элементов [кол-во строк, кол-во столбцов];

При выделении памяти можно выполнять инициализацию массива, указывая значения, относящиеся к каждой строке. Размерность массива можно не указывать, она будет определена по количеству инициализирующих значений.

int[,] a, b;

a=new int[3,2];

b=new int[,] {{1,2}, {3,4}, {5,6}};

a

0	0
0	0
0	0

b

1	2
3	4
5	6

Свойство Length применительно к двумерному массиву соответствует общему количеству элементов: b.Length => 6

Пример:

Сформировать и выдать на экран целочисленную единичную матрицу размером m×n.

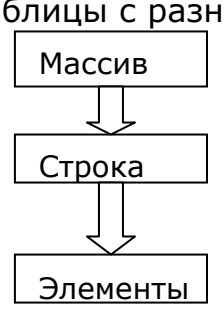
```
int [,]  a;
int i,j; //номера строк и столбцов
string s; //строка, формирующая символьное значение строки матрицы
int n=5;
a=new int [n,n];
for (i=0; i<n; i++)
  a [i,i]=1;
for (i=0; i<n; i++)
  { s="";
    for (j=0; j<n; j++)
          s=s + a[i,j] + "_";
    Console.WriteLine (s);
```

6. Массив массивов

Для двумерных массивов невозможно выделить в качестве отдельного семантического понятия строку. Следовательно, нет возможности применять стандартные методы обработки массивов применительно к строке.

Массив массивов позволяет рассматривать строку матрицы как одномерный массив, следовательно, имеется возможность применять к отдельной строке методы обработки одномерных массивов.

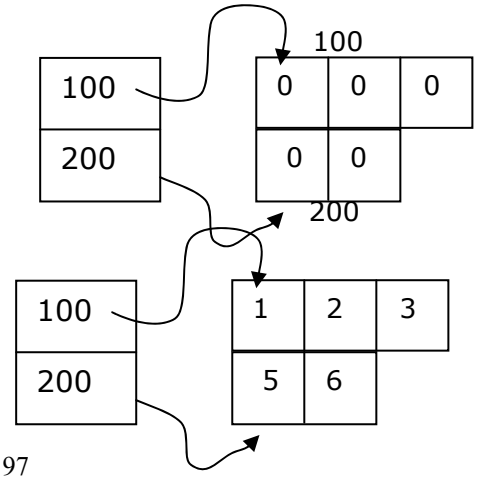
Кроме того, имеется возможность назначить каждой строке таблицы индивидуальную длину, следовательно имеется возможность создавать таблицы с разной длиной строки.



Объявление массива массивов

```
Тип элементов [][] ИмяМассива;
//память под ссылки
ИмяМассива=new_ Тип элементов [кол-во строк][];
//память под каждую строку
ИмяМассива[№строки]=new_Тип[кол-во элементов в строке]
```

```
int[][] a;
a=new int[2][];
a[0]=new int[3];
a[1]=new int[2];
```




```
//инициализация
a[0]=new int[] {1, 2, 3};
```

a[1]=new int[] {5,6};

Свойство Length определено как для всего массива, так и для каждой строки отдельно:

a.Length => 2 (2 строки)

a[0].Length => 3 (3 элемента в строке)

Для каждой строки можно использовать методы обработки одномерных массивов. При поэлементной обработке массива для доступа к элементу необходимо указать номер строки и номер элемента внутри строки. Каждый номер указывается в своей паре скобок:

a[0][1]=88;

МЕТОДЫ

1. Определение метода

Метод определяет некоторую процедуру обработки данных. Выполнение объектно-ориентированной программы можно рассматривать как взаимодействие объектов путем передачи сообщений. Сообщение равносильно вызову метода. Т.е. выполнение программы можно рассматривать как взаимодействие методов с точки зрения двух аспектов:

- по управлению (вызов/возврат)
- по обмену данных

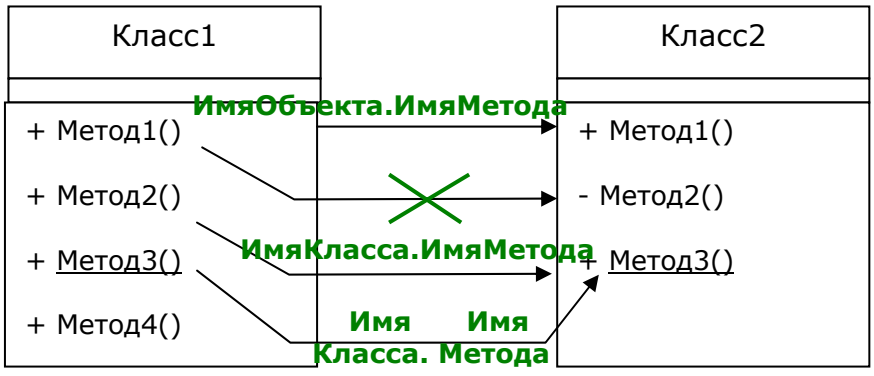
Определение метода может быть выполнено только в составе класса.
Определение метода должно содержать:

1. Имя метода
2. Доступ к данному методу со стороны других
3. Принадлежность методу конкретному объекту или классу
4. Параметры, которые принимаются методом для обработки и тип возвращаемого результата
5. Собственно процедуру обработки параметров

```
<модификаторы>_ТипВозвращаемогоОбъекта Имя (параметры)
{
    <локальные переменные> + процедура
}
```

2. Взаимодействие методов по управлению

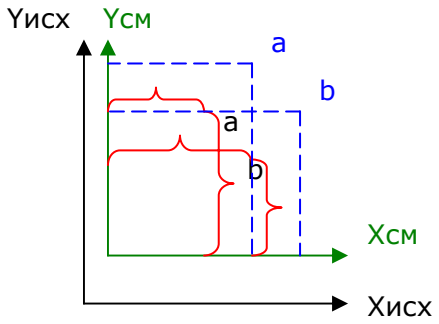
Методы имеют непосредственный доступ к полям своего класса. Статические методы имеют доступ только к статическим полям и могут вызывать только статические методы.



Вызов метода сопровождается выполнением процедуры, определенной методом и возвратом результата в точку вызова.

Пример

Определить класс, описывающий точку на плоскости, точка определяется в системе координат, которая смещена относительно исходной системы координат.



В классе реализовать операции:

1. Сдвиг смещенной системы координат со всеми точками на единицу по обеим осям. Эту операцию назовем СдвигСистемы.
2. Сдвиг конкретной точки в смещенной системе координат (СдвигТочки). Сдвиг точки выполняется на единицу по обеим осям
3. Вывод координат точки в смещенной системе координат (КоорСм)
4. Вывод координат точки в исходной системе координат (КоорИсх)

Точка
+x // поля, которые определяют координаты в смещенной **+y** //системе **+x0** // насколько сдвинута система **+y0** // поля статические, т.к. эта характеристика для всех точек (система сдвигается вместе со всеми точками)
+СдвигСистемы() // применяем для всего класса точек, а не для конкретной точки , поэтому метод статический **+СдвигТочки()** **+КоорСм()** **+КоорИсх()**

```
class_Точка
{
     public_int x,y;
     public_static_int x0,y0;
      public_static_void_СдвигСистемы(){x0++; y0++};
      public_void_СдвигТочки(){x++;y++}
      public_void_КоорСм()
      { Console.WriteLine (x+","+y);}
      public_void_КоорИсх()
      {Console.WriteLine ((x+x0)+","+(y+y0));}
}
class_Пример
{
public_static_void_Main()
{Точка a,b;
a=new_Точка();
b=new_ Точка();
a.x=3; a.y=4;
b.x=4; b.y=3;
```

```
//установка начального положения смещенной системы координат
Точка.x0=1;
Точка.y0=2;
//сдвиг координат всех точек
Точка.СдвигСистемы();
//посмотреть координаты в смещенной системе
a.КоорСм(); // (3,4)
b.КоорСм(); // (4,3)
a.КоорИсх(); // (5,7)
b.КоорИсх(); // (6,6)
// сдвинуть точку A
a.СдвигТочки();
a.КоорСм(); // (4,5)
b.КоорСм(); // (4,3)
a.КоорИсх(); // (6,8)
b.КоорИсх(); // (6,6)
}
}
```

3. Обмен данными между методами

При рассмотрении данного вопроса взаимодействие между методами упрощено – рассматриваем вариант, когда все методы расположены в одном классе.

Существует 3 способа обмена данными:

1. Использование общих полей класса (рассмотренный ранее вариант)
2. Обмен данными через параметры методов
3. Возврат данных из метода

Обмен данными через параметры методов

При определении метода указываются параметры, для каждого параметра указывается тип и имя. При вызове метода указываются аргументы. Значения аргументов будут подставлены вместо параметров (попарно, слева направо). Аргументом может быть любое выражение, тип которого совпадает с типом параметра или может быть автоматически преобразован к типу параметра.

Виды параметров:

1. *Входные* (прием данных) – при вызове снимается копия аргумента и передается в метод на место параметра. Метод не имеет возможности изменить аргумент. По умолчанию все параметры считаются входными.

2. *Выходные* – при вызове в метод передается ссылка на область памяти, где хранится аргумент. Метод обрабатывает данные по этой ссылке, т.е. работает с оригиналом аргумента. Выходные параметры при определении метода и вызове метода должны иметь модификатор *out*.

3. *Входные - выходные* – по механизму реализации аналогичны выходным параметрам, но вызывающий метод в обязательном порядке должен определить значение метода. Выходные параметры при определении метода и вызове метода должны иметь модификатор *ref.*

При использовании выходных параметров аргумент должен быть именем объекта.

Возврат объекта из метода

При определении метода указывается тип возвращаемого из метода объекта. Возврат значения выполняется оператором *return Объект;*

Оператор прекращает выполнение метода и производит возврат в точку выова метод и возвращает в вызвавший метод на место вызова – выражение.

Выражение может определить выражение любого типа, т.е. из метода можно вернуть:

1. Значение стандартного типа (целые, вещественные и т.д.)
2. Массив
3. Объект определенного класса

Последние два варианта позволяют добиться возврата из метода множества значений.

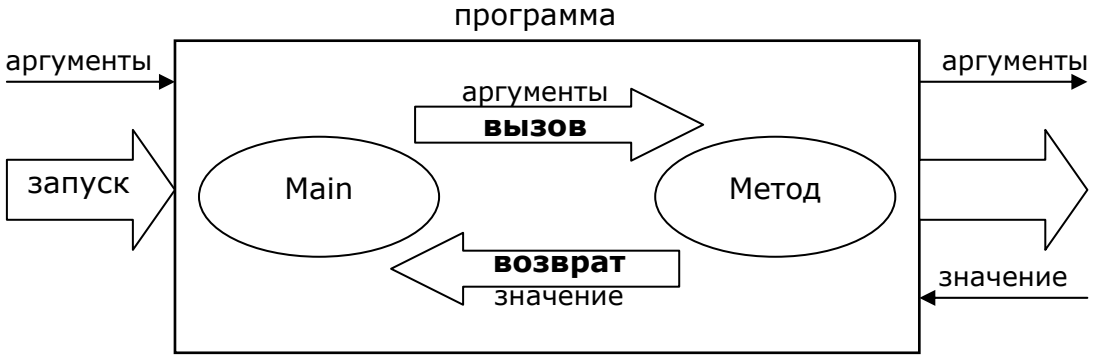
Пример:

Определить метод, выполняющий сдвиг координат точки на заданное расстояние.

```
class_Example
{
  public_static_void_Mov (ref int x, ref int y, int d)
  {
     x=x+d;
     y=y+d;
}

  public_static_void_Main()
  {
     int a=4, b=3, d=10;
     Move (ref a, ref b, d);
  }
}
```

4 Параметры метода Main()

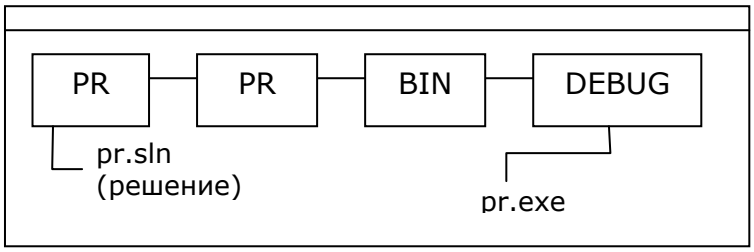
Выполнение программы сводится к вызову методов в определенной последовательности. Методу могут передаваться аргументы, а из метода – возвращаться значение.



С другой стороны сама программа может быть запущена из другой программы и может запускать другую программу.

В рамках данного вопроса рассмотрим вариант, когда программа запускается из среды исполнения. Поскольку метод Main() выполняется первым при запуске программы, то именно через этот метод производится прием аргументов.

Для запуска программы в среде исполнения формируется команда. Команда должна содержать спецификацию исполнимого файла и возможно аргументы, которые в команде указываются через пробел.



C:\PR\PR\BIN\DEBUG\pr.exe_аргумент1_ аргумент2_... _аргументN

Состав параметров метода Main фиксирован: Main (string[] args)

Метод принимает параметры из команды в виде массива строк. Каждая строка хранит символьное представление одного аргумента.

Пример

Разработать программу, предназначенную для хранения в виде массива цен на товары и их изменения на заданную сумму. Изменение выполняется методами

- Plus – увеличение
- Minus – уменьшение

Исполняемый файл хранится в файле pr.exe.

Соглашение об аргументах в команде

1. Сумма
2. Вид операции
   - **Плюс** (в любом регистре) или **+**
   - **Минус** (в любом регистре) или **–**
3. Фамилия оператора (ключ)

Пример команды

C:\PR\PR\BIN\DEBUG\pr.exe_50_Плюс_Иванов

Массив ***args*** в Main()

0	«50»
1	«Плюс»
2	«Иванов»

```
public static void Plus(double[] p, double s)
 // массив цен, сумма, на которую увеличиваем
 {
     for (int i = 0; i < p.Length; i++)
         p[i] += s;
 }
 public static void Minus(double[] p, double s)
```

```
{
    for (int i = 0; i < p.Length; i++)
        p[i] -= s;
}
public static Main(string[] args)
{
    double[] pr; //массив с ценами
    double delta; //сумма
    string key = "Иванов"; //ключ
    pr = new double[] { 100.0, 200.0, 300.0 };

    //проверяем сколько аргументов нам передали
    if (args.Length != 3) return;
    //завершение программы - ошибка1

    //проверка фамилии
    if (args[2] != key) return;
    //завершение программы - ошибка2

    //пересчет цен
    delta = double.Parse(args[0]);

    //чтение операции
    switch (args[1].ToUpper())
    {
        case "ПЛЮС":
        case "+": Plus(pr, delta); break;
        case "МИНУС":
        case "-": Minus(pr, delta); break;
    }

    //вывод новых цен

    return;
}
```

5.Перегруженные методы

В одном классе может быть определено несколько методов с одним именем, но с разным составом параметров по количеству, типам, способу передачи аргументов. Такие методы называются перегруженными.

Перегруженные методы используются в составе классов для определения однотипных по функциональному назначению операций, реализуемых по-разному.

Выяснение конкретного метода, который будет реально вызван, производится на этапе трансляции по следующей схеме:

1. Вызывающий метод – это тот метод, который совпадает с вызовом по имени, количеству и типу параметров.
2. Если такого метода нет, вызывается метод, который совпадает с вызовом по имени и количеству параметров, при этом аргументы могут быть преобразованы к типу параметров без потери информации
3. Если такого метода нет, фиксируется ошибка.

Пример

Определить класс, описывающий понятие точка, методы, реализующие операции:

- Установка точки на числовой оси
- Установка точки на плоскости
- Установка точки в трехмерном пространстве
- Вычисление расстояния от точки до начала координат

```
class Точка
{
    private double x, y, h;

    //установка точки на прямой
    public void Set(double xp)
    {
        x = xp; y = 0; h = 0;
    }

    //установка точки на плоскости
    public void Set(double xp, double yp)
    {
        x = xp; y = yp; h = 0;
    }

    //установка точки в пространстве
    public void Set(double xp, double yp, double hp)
    {
        x = xp; y = yp; h = hp;
    }

    //вычисление расстояния
    public double Dist()
    {
        return Math.Sqrt(x * x + y * y + h * h);
    }
}

 class Program
 {
    public static void Main()
    {
        double d; //расстояние
        Точка тчк = new Точка();

        тчк.Set(2.0); //будет вызван первый Set()
        d=тчк.Dist(); // d=2.0

        тчк.Set(3.0,4.0); //будет вызван второй Set()
        d=тчк.Dist(); // d=5.0

        тчк.Set(2,3,6); //будет вызван третий Set
```

```
            d=тчк.Dist(); // d=7.0
        }
    }
```

Перегруженные методы не различаются по типу возвращаемого значения.

6. Конструкторы

Представляют собой специфический метод класса. Специфика заключается:

в назначении – конструктор автоматически вызывается при создании объекта и в силу этого используется, как правило, для инициализации полей объекта.

в оформлении (т.е. в синтаксисе) – конструктор должен иметь имя, совпадающее с именем класса, конструктор не имеет возвращаемого значения.

Конструкторы могут быть перегружены, вызов соответствующего конструктора выполняется при создании объекта

МАССИВЫ ОБЪЕКТОВ СОБСТВЕННЫХ КЛАССОВ

1. Способы описания структуры предметной области

Любая программа – модель предметной области, некоторой части реальности. Предметная область состоит из элементов, которые в свою очередь состоят из более простых элементов.

Современные языки программирования содержат механизмы, позволяющие отразить структуру элементов. Два самых распространенных механизма:

1. Массив
2. Объект

Взаимодействие этих механизмов приведено на рис.1.

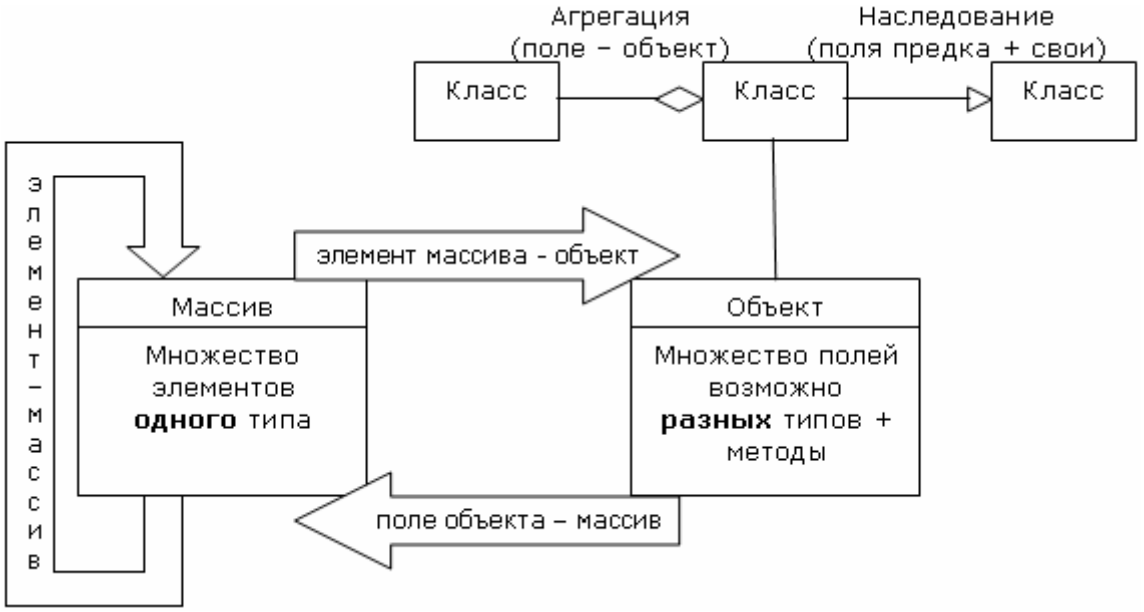


рис.1

Массив может содержать элемент, представляющий собой объект какого-то класса. В свою очередь, полем класса может быть массив. Комбинируя эти механизмы, можно определять сложные типы данных.

2. Массив объектов одного класса

Механизм формирования массива объектов и работы с ним в целом идентичен массиву значимых типов. Принципиальное отличие массива объектов от массива значимых типов заключается в том, что объект определенного класса представляет собой ссылку, и, следовательно, массив объектов хранит не собственно объекты, а ссылки.

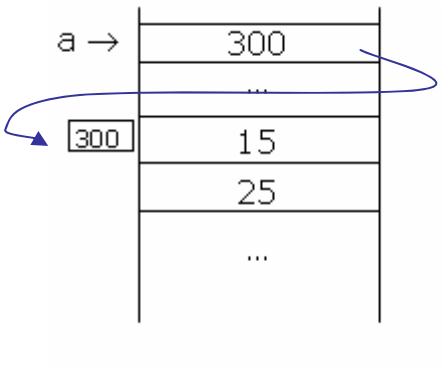
Следствие:

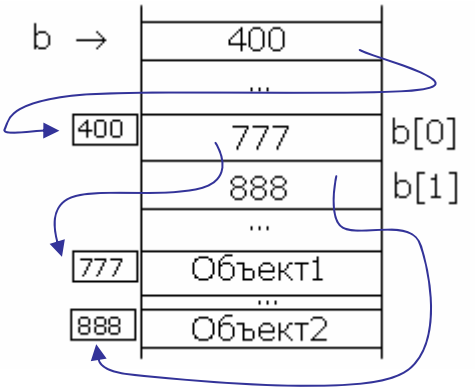
Объект, на который указывает элемент массива, должен быть созданным явным образом с использованием операции new.

Массив значимых типов Массив объектов

```
int[] a;                                    Тип[] b;
a=new int[2];                               b=new Тип [2];
```

```
a[0]= 15;
a[1]= 25;
```





Объект создаем в явном виде:
b[0]=new Тип();
b[1]= new Тип();

В каждом классе по умолчанию определен конструктор без параметров. При создании объекта вызывается конструктор, в задачу которого входит начальная установка полей объектов: поля значимого типа – 0, поля ссылочного типа – null.

Пример:

Определить класс «Студент» с полями: фамилия, Номер зачетной книжки, оценки за последнюю сессию и методом вывода данных о студенте на экран.

```
class Студент
{
   public string Фам;
   public int Зач;
   public int[] Оц;

   public void Показать()
   {
      int i;
      Console.Write(Фам + Зач);
      for (i = 0; i < Оц.Length; i++)
         Console.Write(Оц[i]);
      Console.WriteLine();
   }
         }
```

Создадим объект класса Студент:

```
Студент ст;
ст = new Студент();
```

Фам	Зач	Оц
«»	0	0

Конструктор может быть переопределен, в этом случае конструтор по умолчанию становится недоступным. Если есть необходимость в конструкторе без параметров, конструктор должен быть определен в явном виде.

Как правило, с помощью конструктора задаются начальные значения полей, эти значения могут быть приняты за счет параметров конструктора. В качестве параметра конструктора может быть выбран объект того же типа – конструктор копированияю Его задача – снять копию полей принятого объекта.

Различают два вида конструкторов копирования:

1. Поверхностное копирование (с полей значений снимается копия значений, с полей ссылок – копия ссылок)
2. Конструктор полного копирования (с полей значений снимается копия значений, для полей ссылок создается объект и в него копируются данные, на которые указывает ссылочное поле оригинала)

Конструктор с параметрами

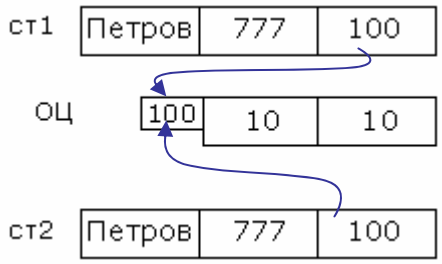
```
public Студент (string Фам, int зач, int[] оц)
{       this.Фам = Фам; //this – ссылка на активный объект
        this.Зач=Зач;
        this.Оц=Оц;
}
```

Конструктор поверхностного копирования

```
public Студент (Студент ст)
{       Фам = ст.Фам;
        Зач = ст.Зач;
        Оц= ст.Оц;
}
```

Основная программа

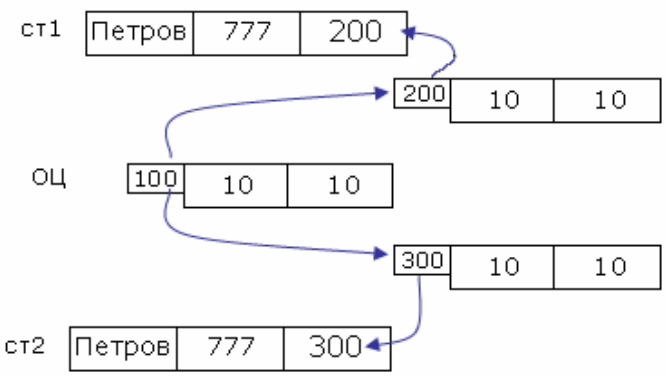
```
Студент ст1; Студент ст2;
int[] Оц;
Оц=new int[]{10,10};
ст1=new Студент («Петров», 777, Оц);
ст2=new Студент (ст1);
```



При изменении массива оценок изменятся данные и в объекте ст1, и в объекте ст2.

Конструктор полного копирования

```
public Студент (string Фам, int зач, int[] оц)
{       this.Фам = Фам;
        this.Зач=Зач;
        this.Оц=new int[Оц.Length];
        Array.Copy(Оц,0,this. Оц,0, Оц.Length);
}

public Студент (Студент ст)
{       Фам = ст.Фам;
        Зач = ст.Зач;
        Оц= new int[Оц.Length];
        Array.Copy(Оц,0,this. Оц,0, Оц.Length);
}
```



У каждого объекта при копировании создается свой собственный массив оценок.

3. Массив объектов разных типов

Часто возникает необходимость хранить в одном массиве объекты разных типов.

Пример:

Сформировать массив сотрудников факультета: студенты и преподаватели. Класс Студент аналогичен рассмотренному. Класс Преподаватель имеет поля: фамилия, название кафедры. Метод – вывод фамилии и названия кафедры.

Проблема:

1. Требуется массив объектов разных типов
2. Разрешается массив объектов одного типа

Решение проблемы:

Все классы (типы) являются наследниками класса Object. Объект этого класса представляет собой ссылку, которой может быть присвоена ссылка любого типа.

```
Object ob;
Студент ст;
Преподаватель пр;
…
ob = ст;
ob = пр;
```

Таким образом, массив должен формироваться как массив типа Object (или массив ArrayList, который по определению содержит ссылки типа Object).

Проблема: при извлечении элементов из такого массива необходимо выполнить обратное преобразование. Преобразование выполняется в явном виде.

Студент

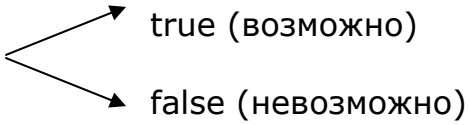
Object ⇒

Преподаватель

```
Object ob;
Студент ст;
Преподаватель пр;
…
ob = пр;
пр = (Преподаватель) ob;
```

Порядок заполнения элементов массива в общем случае неизвестен, поэтому возникает проблема определения типа объекта, на который указывает сслыка.

Операция проверки возможности преобразования ссылки к целевому типу:



ob is Преподаватель ⇒ true
ob is Студент ⇒ false

Другой способ:

Воспользоваться методом объекта GetType(). Свойство Name – имя в виде строки символов:

ob. GetType().Name ⇒ «Преподаватель»

4. Упаковка и распаковка значений

Ссылке типа Object можно присвоить значение значимого типа, при этом хранящееся значение преобразуется к ссылочному типу (упаковка). Ссылка типа Object может быть преобразована к значимому типу (распаковка). Упаковка выполняется автоматически, распаковка выполняется путем приведения к заданному типу.

```
Object ob;
int x=10, y;
// упаковка
ob=x;
// распаковка
y=(int)ob;
```

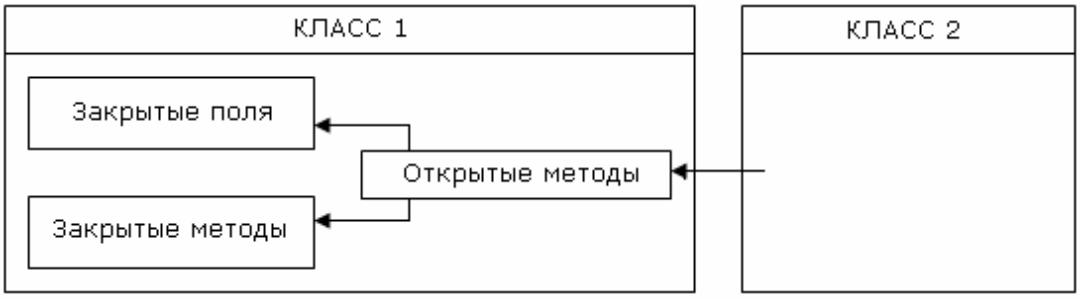
Вывод:

Массивы типа Object и массивы ArrayList могут хранить как ссылочные данные, так и значимые данные.

ИНКАПСУЛЯЦИЯ

1. Понятие инкапсуляции и механизм ее реализации

Под инкапсуляцией понимается скрытие деталей реализации класса (объекта) от других классов.



Поле класса закрывается для прямого доступа извне. Доступ к ним предоставляется через открытые методы. Эти методы определяют интерфейс класса (в данном случае имеется в виду общий смысл понятия «интерфейс», а не класс C# Интерфейс).

Механизмом реализации инкапсуляции являются модификаторы доступа:

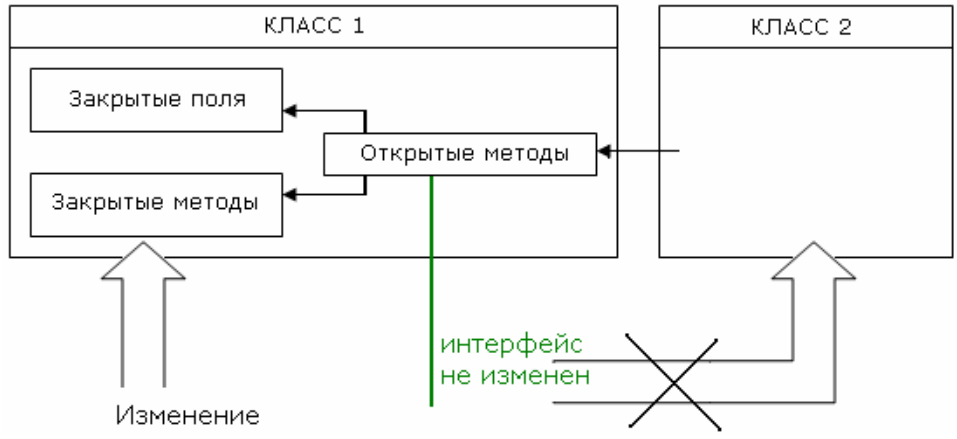
private ➔ доступ извне запрещен
public ➔ доступ разрешен

Инкапсуляция предназначена для решения следующих проблем:

1. Локализация изменения программного кода
2. Защита полей класса от нежелательного воздействия извне

2. Локализация изменений программного кода

Изменение в способе реализации полей и методов класса не должно приводить к необходимости изменения основного кода.

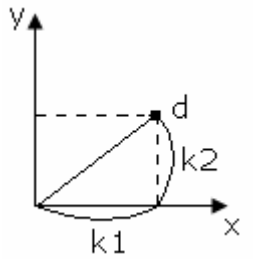


Пример

Определить класс, описывающий положение точки на плоскости. Предусмотреть следующие операции:

- Изменение координат точки
- Выдача расстояния от точки до начала координат

При разработке класса координаты точки заданы в прямоугольной системе координат.




```
class Точка
  {
    private double k1, k2; //координаты точки

    //открытый метод для изменения координат
    public void Установить(double x, double y)
    {
      k1 = x;
      k2 = y;
    }

    //открытый метод для вычисления расстояния
    public double Вычислить()
    {
      return Math.Sqrt(k1 * k1 + k2 * k2);
    }

    class Program
    {
      public static void Main()
      {
        Точка t; double d;
        t = new Точка();
        t.Установить(3.0, 4.0); // вместо t.k1 =3.0; t.k2=4.0
        d = t.Вычислить(); //=> 5
        t.Установить(4.0, 3.0);
        d = t.Вычислить();
      }
          }
```

В процессе эксплуатации программы выяснилось, что установка коодинат точки производится намного реже, чем запрос на вычисление расстояния. Разработчики класса Точка для повышения эффективности программы приняли решение хранить координаты точкив полярной системе координат.

Как быть разработчикам класса Program?

Если доступ к полям класса Точка выполянется через специальные методы, то разработчики класса Точка должны оставить параметры неизменными, но изменить их реализацию. В этом случае код класса Program не требует никаких изменений.

```
class Точка
```

```
{
   private double k1, k2; //координаты точки

   //открытый метод для изменения координат
   public void Установить(double x, double y)
   {
      k2 = Math.Sqrt(x*x+y*y);
      k1 = Math.Atan ((y,x));
   }

   //открытый метод для вычисления расстояния
   public double Вычислить()
   {
      return Math.Sqrt(k1 * k1 + k2 * k2);
         }
```

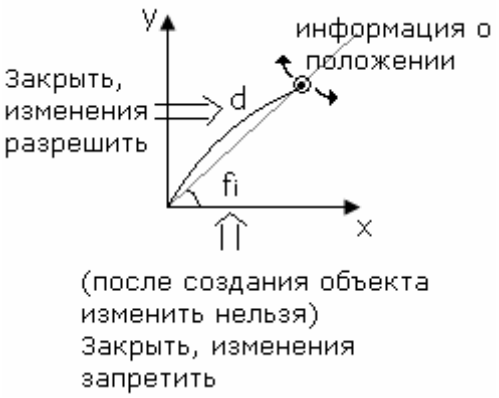
3. Защита полей от вмешательства извне

Поля, запрещенные для доступа извне, объявляются закрытыми. Методов доступа к ним не существует. Никаким способо изменить или получить значение невозмонжо. Как вариант, поле закрывается, открывается метод доступа по чтению.

Пример

Определить класс Точка, описывающий точки, расположенные на луче, выходящем из начала координат. Координаты точки полярны как для разработчика, так и для пользователя.

Объект должен иметь возможность передвигаться по лучу и выдавать информацию о своем теккущем положении.




```
class Точка
      {
         private double fi, d;

         public Точка(double fi)
         { this.fi = fi; }

         public void Установить(double d)
         { this.d=d;}

         public double ВычислитьFI()
         {return fi;}
```

```
    public double ВыдатьD()
    { return d; }
}

    class Program
    {
      public static void Main()
      {
        Точка t; double fi;
        t = new Точка(Math.PI/2);
        t.Установить(5.0);
        fi=t.ВычислитьFI();
      }
    }
}
```

Альтернативой инкапсуляции с точки зрения защитных полей являются открытые поля доступные для чтения. Поле, открытое для чтения, может быть установлено конструктором. Дальнейшие изменения запрещены.

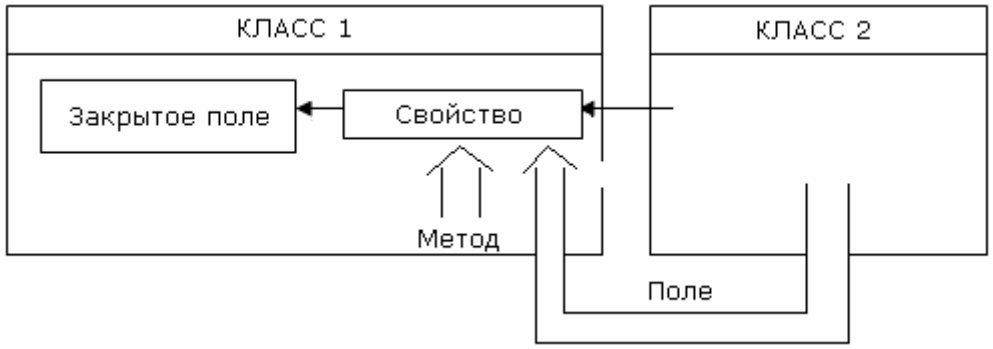
Пример аналогичен предыдущему, реализован через открытые поля для чтения.

```
class Точка
  {
    public readonly fi;
    private double d;

    public Точка(double fi)
    { this.fi = fi; }

    public void Установить(double d)
    { this.d=d;}

    public double ВычислитьFI()
    {return fi;}

    public double ВыдатьD()
    { return d; }
}

    class Program
    {
      public static void Main()
      {
        Точка t; double fi;
        t = new Точка(Math.PI/2);
        t.Установить(5.0);
        fi=t.fi; //45 градусов
      }
    }
}
```

Возможно использование открытых статических полей только для чтения. Такие поля должны иметь инициализатор. Значение поля устанавливается до создания объекта и изменение его (даже с помощью конструктора) невозможно.

4. Свойства

Свойство – это метод для установки и-или получения закрытого поля. С этой точки зрения свойство относится к интерфейсной части класса.



Определение свойства:

```
public_Тип возвращаемого значения_Имя свойства
// параметров нет, поэтому скобок после имени нет и быть не может

{       set {код для установки поля}
                                неявный параметр (value)
        get {код для чтения поля}
}
```

Свойства используются для поддержки инкапсуляции, создания виртуальных полей (т.е. реально полей нет, но у пользователя создается впечатление, что они есть; на самом деле значение вычисляется).

Пример

Определить класс Точка на плоскости в прямоугольных координатах, точка может располагаться только выше оси x или на ней.

Предусмотреть возможность изменения координат точки и вычисление расстояния до нее. Операции рассматривать, как свойства.

```
class Точка
{
   private double x, y;

   public double xp
   {
      set { x = value; }
      get { return x; }
   }

   public double yp
   {
      set
      {
         if (value >= 0)
            y = value;
         else y = 0;
      }
      get { return y; }
   }

   public double Dp
   {
      get { return Math.Sqrt(x * x + y * y); }
   }
}

class Program
{
   public static void Main()
   {
      Точка t;
      t = new Точка();
      t.xp = 3.0;
      t.yp = 4.0;
      t.Dp;// => 5.0
   }
}
```

АГРЕГАЦИЯ

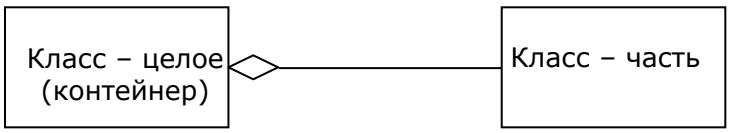
1. Понятие агрегации и ее разновидности

С точки зрения моделирования предметной области, агрегация предназначена для моделирования сложной системы, которая включает в себя более простые подсистемы или неделимые части. Другими словами, агрегация раскрывает организацию системы по принципу «целое → часть» и показывает, из каких частей состоит система и как они взаимосвязаны.

Две основных разновидности агрегации:

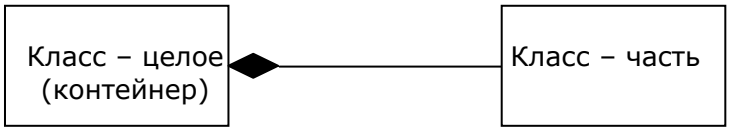
- **собственно агрегация** (агрегация)

Части представляют собой самостоятельные объекты, которые создаются независимо от целого и затем объединяются в единую систему. Уничтожение системы не сопровождается уничтожением частей. Они продолжают самостоятельное существование.

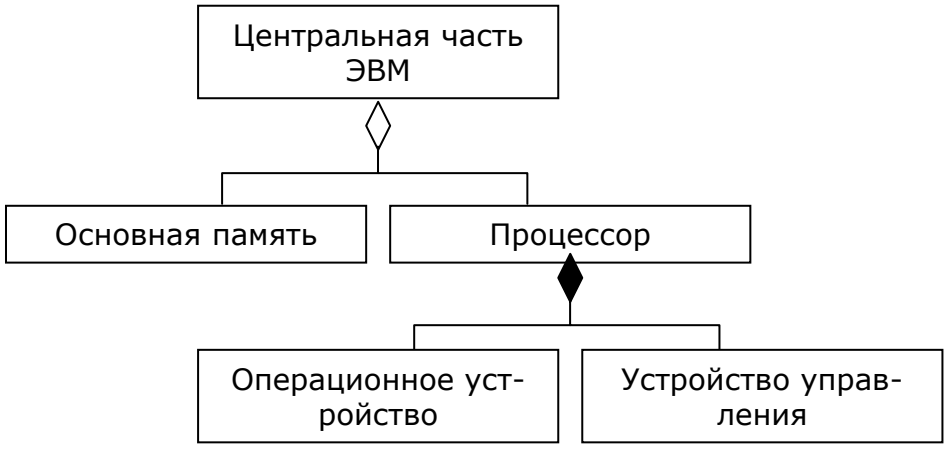


- **композиция**

Части представляют собой объекты, которые создаются при создании системы. Вне рамок целого части самостоятельно не существуют, при уничтожении системы, части уничтожаются.



Деление системы на составные части представляет собой иерархию: на верхнем уровне – целое, на нижних – части.

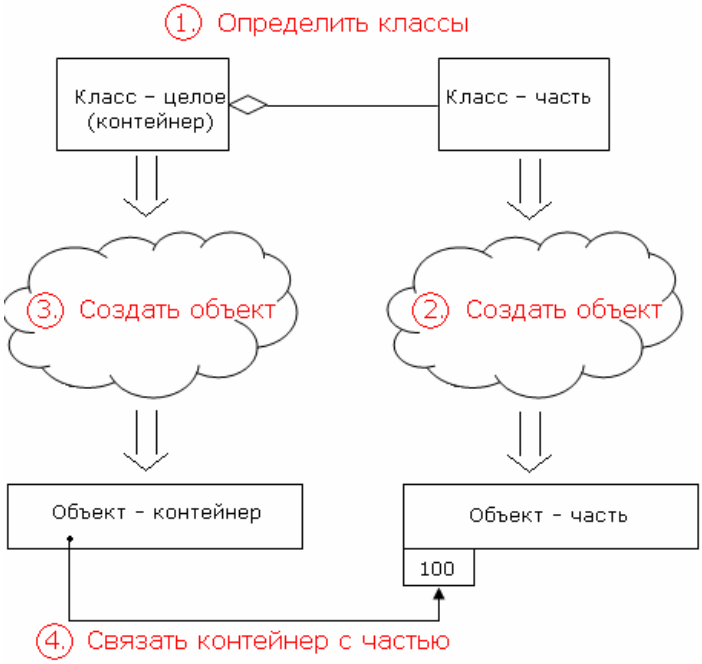


С точки зрения технологии программирования, агрегация представляет собой реализацию идеи повторного использования кода. Код, реализующий класс-часть, в готовом виде, без изменений, может быть использован в классе-контейнере.

Механизм реализации: полю класса-контейнера назначается тип класса-части.

2. Агрегация на основе классов

При реализации агрегации на основе классов необходимо исходить из того факта, что класс относится к ссылочным типам, т.е. имя объекта – ссылка на область памяти, где хранится объект. Следовательно типичная схема будет иметь вид:



Изменение состояния объекта-части вне рамок объекта-контейнера немедленно изменяет состояние контейнера.

Базовый пример

Определить сущность «осветительный прибор», частями которого являются «корпус» и «лампочки». Осветительный прибор собирается из отдельно купленных частей и должен допускать их замену.

```
class Корпус
{
   private string тип; // тип корпуса
   private double цена;

   public string Тип
   {
      set { тип = value; }
      get { return тип; }
   }

   private double Цена
   {
      set { цена = value; }
      get { return цена; }
   }

}

class Лампочка
```

```
        {
            private double мощность;
            private double цена;

            public double Мощность
            {
                set { мощность = value; }
                get { return мощность; }
            }

            public double Цена
            {
                set { цена = value; }
                get { return цена; }
            }
        }

        class Прибор
        {
            private Корпус кор;
            private Лампочка лам;
            private string состояние; //включено или выключено

            //конструктор
            public Прибор(Корпус кор, Лампочка лам, string состояние)
            {
                this.кор = кор;
                this.лам = лам;
                this.состояние = состояние;
            }

            //свойство
            public string Состояние
            {
                set { Состояние = value; }
            }

            //вывести данные
            public void Показать()
            {
                Console.WriteLine("{0} {1} {2} {3} {4} {5}", состояние, кор.Тип, кор.Цена,
лам.Цена, лам.Мощность);
            }
        }
```

В основной программе соберем осветительный прибор из корпуса типа «Тюльпан» ценой 800 рублей и лампочки мощностью 60 Вт, ценой 30 рублей.

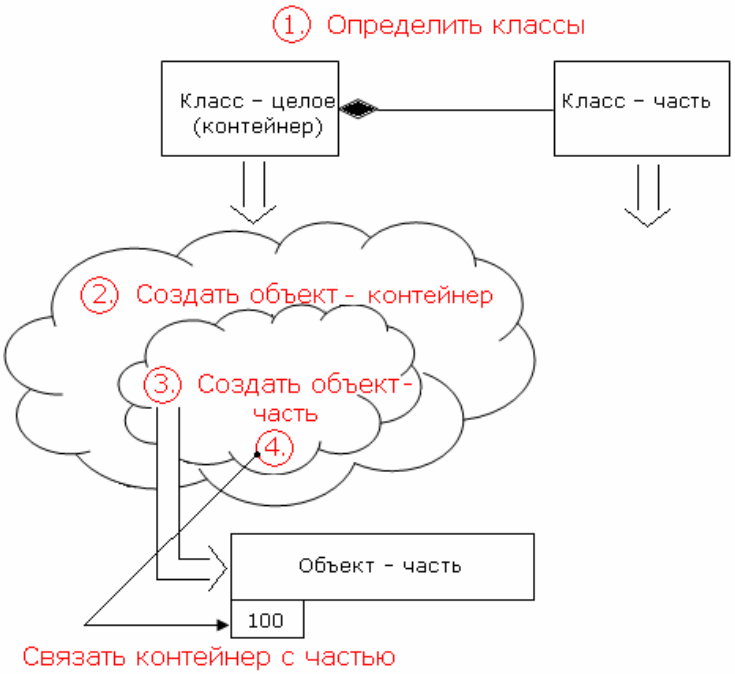
```
  class Корпус
  {
    private string тип; // тип корпуса
    private double цена;

    public string Тип
```

```
    {
        set { тип = value; }
        get { return тип; }
    }

    public double Цена
    {
        set { цена = value; }
        get { return цена; }
    }

}

class Лампочка
{
    private double мощность;
    private double цена;

    public double Мощность
    {
        set { мощность = value; }
        get { return мощность; }
    }

    public double Цена
    {
        set { цена = value; }
        get { return цена; }
    }
}

class Прибор
{
    private Корпус кор;
    private Лампочка лам;
    private string состояние; //включено или выключено

    //конструктор
    public Прибор(Корпус кор, Лампочка лам, string состояние)
    {
        this.кор = кор;
        this.лам = лам;
        this.состояние = состояние;
    }

    //свойство
    public string Состояние
    {
        set { состояние = value; }
    }

    //вывести данные
    public void Показать()
    {
```

```
            Console.WriteLine("{0}, {1}, {2}, {3}, {4}, {5}", состояние, кор.Тип, кор.Цена,
лам.Цена, лам.Мощность);
        }
    }
    class Program
    {
        static void Main()
        {//1. объявляем ссылки
            Корпус к;
            Лампочка л;
            Прибор п;

            //2. создаем объкты
            к = new Корпус();
            к.Тип = "Тюльпан";
            к.Цена = 800.0;

            л = new Лампочка();
            л.Мощность = 60.0;
            л.Цена = 30.0;

            //3., 4. контейнер
            п = new Прибор(к, л, "Включено");
            п.Показать();

            //меняем лампочку
            л.Мощность = 200.0;
            п.Показать();
        }
```

3. Композиция на основе классов

В этом случае объект-часть должен создаваться в процессе создания объекта-контейнера.



Изменить состояние объекта-части можно только в рамках самого контейнера, извне это сделать нельзя.

Пример

Сущности аналогичны базовым. Отличия в предметной области: осветительный прибор собирается на заводе изготовителе и представляет собой единое целое. Замена частей невозможна.

Отличия в программном коде заключаются в изменении конструктора класса Прибор. В этом конструкторе создаются объекты-части (лампочка и корпус изготавливаются на заводе на основе их технических характеристик, принятых в качестве параметров).

```
public Прибор(Корпус кор, Лампочка лам, string состояние)
{
this.кор = new Корпус();
this.кор.Тип = кор.Тип;
this.кор.Цена = кор.Цена;
this.лам = new Лампочка();
this.лам.Цена = лам.Цена;
this.лам.Мощность = лам.Мощность;
this.состояние = состояние;
  }
```

Весь остальной код остается неизменным. В обоих случаях будут выведены одни те же характеристики прибора.

```
//смена эталонной лампочки
п = new Прибор(к, л, "Включено");
```

Если поле, определяющее лампочку в классе Прибор сделать открытым, лампочку можно будет заменить, не создавая нового прибора.

```
п.лам.Мощность = 200.0;
```

Но это противоречит описанию предметной области.

4. Композиция на основе структур

Структура – тип данных, который позволяет определять в рамках единого целого поля разных типов и методы их обработки. В этом смысле структура аналогична классу.

Отличия от класса (основные)

1. Структуры относятся к типам значения, т.е. при присваивании одной структуры другой происходит копирование не ссылки, а всех полей структуры.

   Копирование выполняется поверхностное (с полей значения снимается копия значений, с полей ссылок снимается копия ссылки).

   Таким образом, задача создания осветительного прибора, не допускающего замены корпуса и лампочки, может быть решена с помощью кода абсолютно идентичного коду базового примера пу-

тем замены классов Корпус, Лампочка, Прибор, путем замены классов на структуры.

class → struct

2. Для структур не поддерживается механизм наследования.

Прочие отличия

1. Конструктор по умолчанию для структур невозможно переопределить, он всегда есть и всегда доступен. Можно добавить конструктор с параметрами.
2. Создания объекта типа stuct не требует операции new. Но в этом случае поля структуры будут неопределенными по значению.
3. При определении структуры запрещена инициализация полей.

НАСЛЕДОВАНИЕ

1 Понятие наследования

С точки зрения моделирования предметной области наследование предназначено для описания некоторой сущности (**потомка**) путем указания отличий от другой сущности (**предка**).

В ООП наследование рассматривается как отношение между классами, при котором класс-потомок представляет собой разновидность класса-предка.

Отношение наследования раскрывает предметную область по типу «общее - специфическое». Представление предметной области в виде наследования представляет собой иерархию, на вершине которой находится класс-предок (**базовый класс**), а на нижних уровнях – классы-наследники (**наследуемые классы**).

Предок может иметь множество наследников. Наследник может иметь не более одного прямого предка. Другими словами, в C# явным образом не поддерживается множественное наследование. Потомок не может унаследовать поля и методы от нескольких прямых предков. Пример наследования приведен на Рис.1.

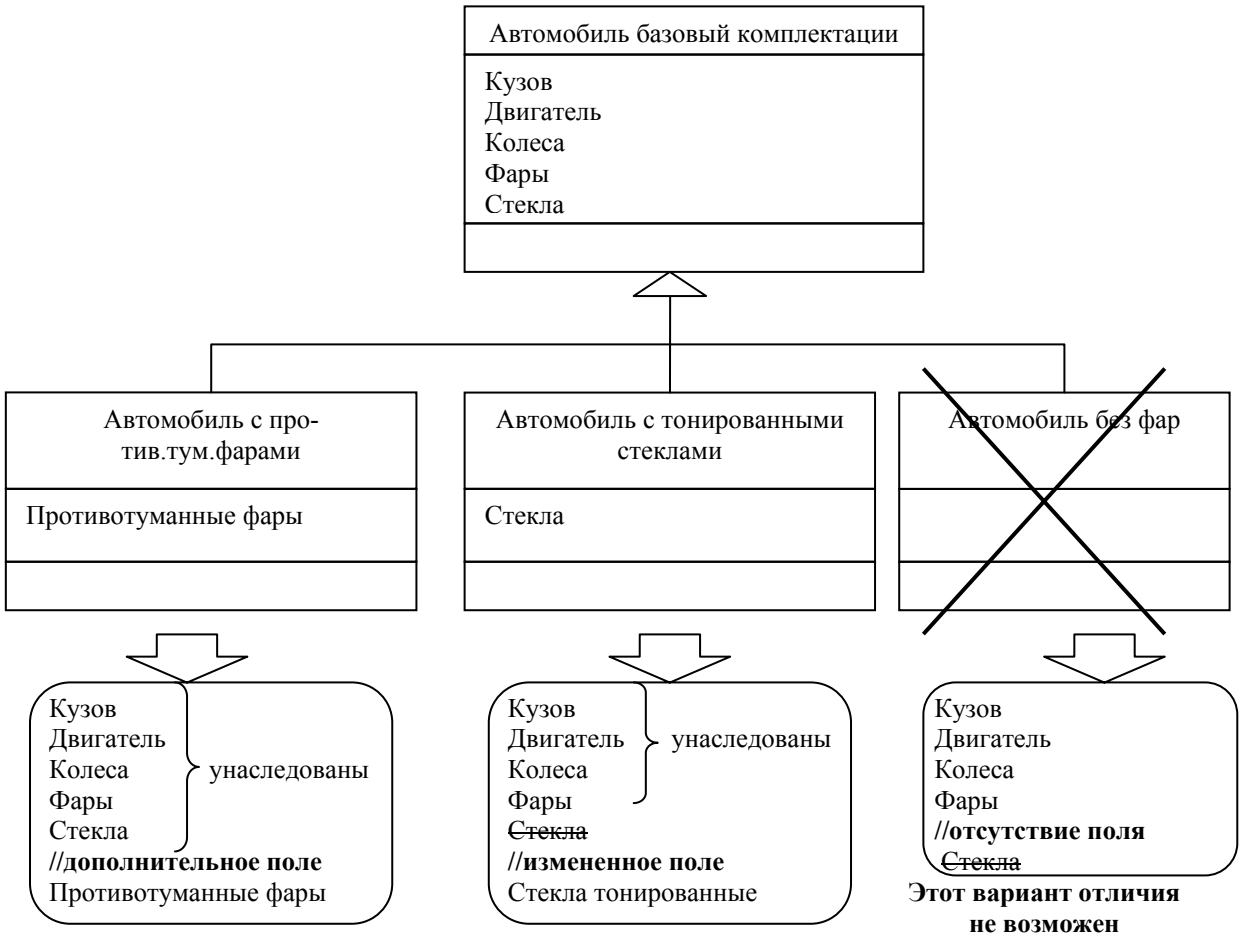


Рис.1

Специфические черты классу-потомку могут быть приданы двумя способами:

- Дополнение унаследованных от предка полей и методов новыми полями и методами. В результате потомок становится сложнее по структуре и поведению.

- Изменение унаследованных от предка полей и методов. В результате потомок не становится сложнее, меняется лишь способ реализации полей и методов, то есть имеет место отношение: то же самое, но сделано по-другому.

Механизм реализации:

- При определении класса-потомка указывается базовый класс: class ИмяКласса: ИмяБазовогоКласса;
- В классе определяются дополнительные поля и методы или переопределяются поля и методы базового класса.

С точки зрения технологии программирования наследование позволяет решить проблемы:

• Повторное использование кода (создание новых классов на основе существующих)

• Модификация существующего кода (изменение в базовом классе немедленно приводит к изменению в классах-потомках)

2. Наследование как средство усложнения базового класса

Реализуется путем определения в классе-потомке дополнительных полей, методов и свойств. Класс-потомок будет иметь:

- Унаследованные поля, методы и свойства
- Дополнительные поля, методы и свойства

Конструкторы не наследуются, поэтому класс-потомок должен иметь собственный конструктор. При создании объекта класса-потомка конструкторы должны вызываться, начиная с базового класса. В начале инициализируются унаследованные поля, потом – специфические поля, как показано на Рис.2.

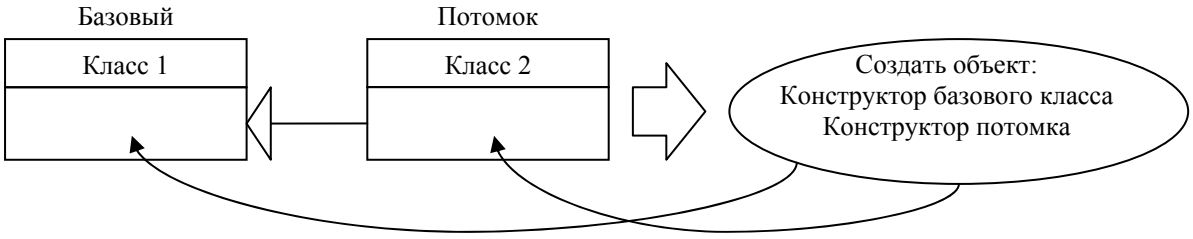


Рис.2

Указанная очередность вызовов реализуется одним из способов:

1. Конструктор потомка не имеет вызова конструктора-предка. Автоматически вызывается конструктор предка без параметров.
2. Конструктор потомка явно вызывает конструктор предка.

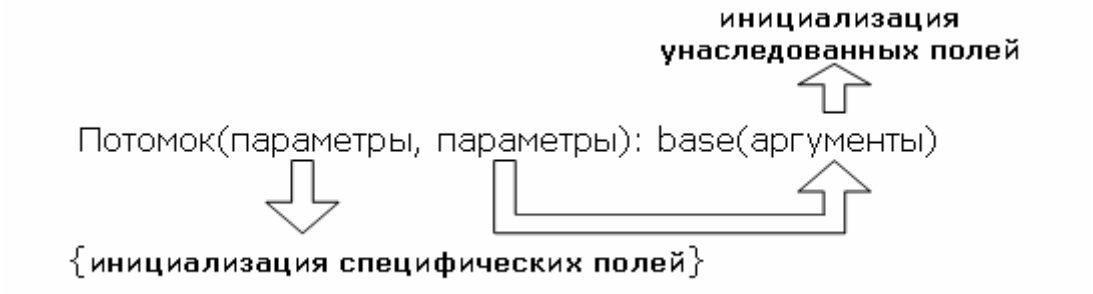


Рис.3

Пример: определить базовый класс «Человек». Человек характеризуется фамилией и умением сообщать свою фамилию. Потомками класса «Человек» являются:

-класс «Владелец», дополнительно характеризующийся номером автомобиля и умением сообщать его

-класс “Служащий”, дополнительно характеризующийся названием фирмы и умением сообщать его.

-класс “Студент”, дополнительно характеризующийся названием ВУЗа

```
class Человек
{
   protected string фам;
   public Человек(string фам)
   {
     this.фам = фам;
   }
   public void Показать()
   {
      Console.WriteLine("Я - человек: " + фам);
   }
}


class Студент:Человек
{
   private string вуз;
   public Студент(string фам, string вуз):base(фам)
   {
        this.вуз = вуз;
   }
}


class Владелец: Человек
{
   private string ном;
   public Владелец(string фам, string ном): base(фам)
   {
      this.ном = ном;
   }
   public void Инфо()
   {
       Console.WriteLine("Я - владелец: "+фам+" --> " + ном);
   }
}


class Служащий: Человек
{
   private string фирма;
   public Служащий(string фам, string фирма): base(фам)
   {
       this.фирма = фирма;
   }
```

```
  public void Инфо()
  {
    Console.WriteLine("Я - служащий: "+фам + "____" + фирма);
  }
}

class Program
{
   static void Main(string[] args)
   {
      Студент ст = new Студент("Иванов","ВШЭ");
      Владелец вл = new Владелец("Петров", "А777АА-99RUS");
      Служащий сл = new Служащий("Сидоров", "Рога и копыта");

      ст.Показать(); //Унаследованный метод
      //Я - человек: Иванов

      вл.Показать(); //Унаследованный метод
      //Я - человек: Петров

      сл.Инфо(); // Дополнительно определенный метод
     //Я - служащий: Сидоров ______ Рога и копыта

      вл.Инфо();     //Дополнительно определенный метод
      //Я - владелец: Петров --> А777АА-99RUS
   }
}
```

Для работы с объектом-потомком могут быть использованы ссылки типа Предок и Потомок. Ссылке базового класса можно присвоить ссылку на класс-потомок. Ссылки типа Предок можно использовать только для доступа к унаследованным полям, методам и свойствам (предок ничего не знает о том, что «привнесли» потомки).

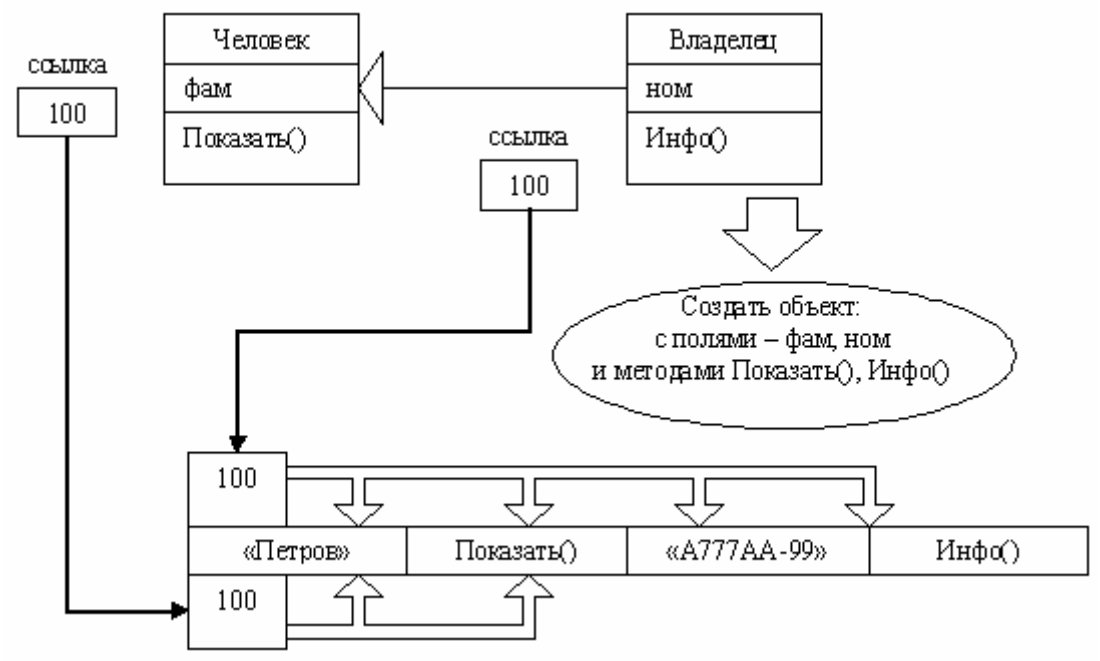


Рис.4

```
class Человек
{
   protected string фам;
   public Человек(string фам)
   {
     this.фам = фам;
   }
   public void Показать()
   {
      Console.WriteLine("Я - человек: " + фам);
   }
}

class Владелец: Человек
{
   private string ном;
   public Владелец(string фам, string ном): base(фам)
   {
      this.ном = ном;
   }
   public void Инфо()
   {
       Console.WriteLine("Я - владелец: "+фам+" --> " + ном);
   }
}


class Program
{
   static void Main(string[] args)
   {
     Человек ч;
     Владелец вл = new Владелец("Петров", "A777AA-99RUS");

     ч = вл;
     ч.Показать(); //Унаследованный метод
     //Я - человек: Петров

     вл.Показать(); //Унаследованный метод
     //Я - человек: Петров

     ч.Инфо();     //Ошибка

     вл.Инфо();     //Дополнительно определенный метод
     //Я - владелец: Петров --> A777AA-99RUS
   }
}
```

Возможность усложнять базовый класс может рассматриваться как альтернатива агрегации. При этом необходимо помнить, что механизмы наследования и агрегации реализуются по-разному.

Наследование – это аналог производства изделия «с нуля». Базовый чертеж (класс) дополняется и полученный чертеж изделия (класс-наследник) используется для изготовления изделия (объекта). *На этапе*

выполнения нет деления объекта на предка и потомка – это единый объект.

Агрегация – аналог производства изделия методом «отверточной сборки». Изготовленные по отдельным чертежам детали (объекты) собираются в более сложное изделие (объект). Класс-контейнер – это сборочный чертеж изделия, определяющий его составные части. *На этапе выполнения сохраняется четкое деление объекта на контейнер и часть. Это самостоятельные объекты.*

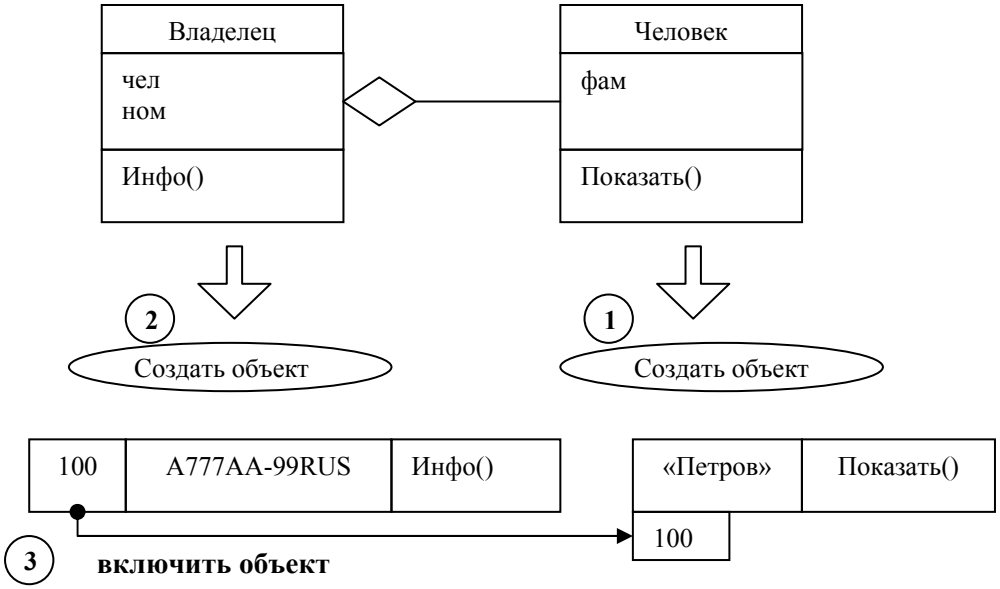


Рис.5

```
class Человек
{
   public string фам;
   public Человек(string фам)
   {
     this.фам = фам;
   }
   public void Показать()
   {
     Console.WriteLine("Я - человек: " + фам);
   }
}

class Владелец
{
   public Человек чел;
   private string ном;
   public Владелец(Человек чел, string ном)
   {
     this.чел = чел;
     this.ном = ном;
   }
```

```
    public void Инфо()
    {
      Console.WriteLine("Я - владелец: "+чел.фам+" --> " + ном);
    }
}

class Program
{
    static void Main(string[] args)
    {
       Человек ч = new Человек("Петров");
        Владелец вл = new Владелец(ч,"A777AA-99RUS");

        вл.чел.Показать();
       //Я - человек: Петров

        вл.Инфо();
        //Я - владелец: Петров --> A777AA-99RUS
    }
}
```

3. Наследование как средство изменения базового класса

Реализуется через переопределение полей, методов и свойств базового класса в классе-потомке. На практике чаще всего используется изменение поведения – т.е. переопределение методов.

Синтаксически это реализуется как определение в классе-потомке метода с тем же именем, составом параметров и возвращаемым значением, что и в базовом классе.

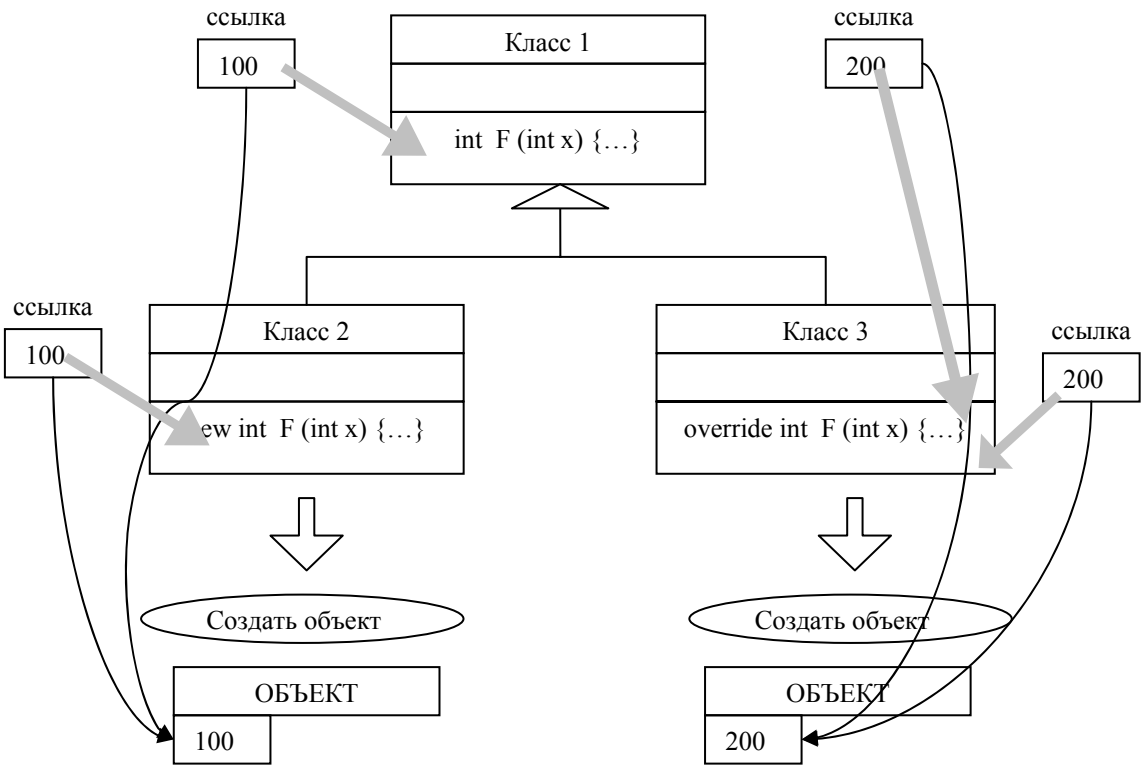


Рис.6

C# поддерживает два способа переопределения метода, определенного в базовом классе на метод, определенный в классе-потомке (Рис.6):

1. new – новая версия метода. Решение о вызываемом методе принимается по типу ссылки. Через ссылку класса-предка вызывается метод, определенный в классе предке, а через ссылку класса-потомка вызывается метод, определенный в классе-потомке.

2. override – метод, заменяющий метод предка. Метод может быть вызван через ссылку класса-потомка или ссылку базового класса. Решение о вызываемом методе принимается по типу объекта, на который указывает ссылка. Для реализации этого способа метод в классе-предке должен быть объявлен как виртуальный (virtual).

```
class Человек
{
   protected string фам;
   public Человек(string фам)
   {
     this.фам = фам;
   }
   public virtual void Показать()
   {
      Console.WriteLine("Я - человек: " + фам);
   }
}
class Студент:Человек
{
   private string вуз;
   public Студент(string фам, string вуз):base(фам)
   {
      this.вуз = вуз;
   }
}

class Владелец: Человек
{
   private string ном;
   public Владелец(string фам, string ном): base(фам)
   {
     this.ном = ном;
   }
   public new void Показать()
   {
     Console.WriteLine("Я - владелец: " + фам + " --> " + ном);
   }
}

class Служащий: Человек
{
   private string фирма;
   public Служащий(string фам, string фирма): base(фам)
   {
     this.фирма = фирма;
   }
```

```
   public override void Показать()
   {
     Console.WriteLine("Я - служащий: "+фам +" ____ "+ фирма);
   }
}

class Program
{
   static void Main(string[] args)
   {
     Человек ч;
     Студент ст = new Студент("Иванов","ВШЭ");
     Владелец вл = new Владелец("Петров", "А777АА-99RUS");
     Служащий сл = new Служащий("Сидоров", "Рога и копыта");

     ч = ст;
     ч.Показать();  //Вызов метода предка
     //Я - человек: Иванов
     ст.Показать(); //Вызов метода предка
     //Я - человек: Иванов

     ч = вл;
     ч.Показать();   //Вызов метода предка
     //Я - человек: Петров
     вл.Показать();  //Вызов метода наследника
     //Я - владелец: Петров --> А777АА-99RUS

     ч = сл;
     ч.Показать();  //Вызов метода наследника
     //Я - служащий: Сидоров ____ Рога и копыта
     сл.Показать(); //Вызов метода наследника
     //Я - служащий: Сидоров ____ Рога и копыта
  }
}
```

Применение вызова переопределенного метода override позволяет просто решить проблему определения вызываемого метода при хранении в массиве объектов разного типа. Массив организуется как массив ссылок базового типа. Вызываемый метод определяется автоматически по типу объекта, на который указывает ссылка.

Метод в базовом классе может быть объявлен как абстрактный. Такой метод не содержит реализации – тела. Это только заголовок. Другими словами, базовый класс только декларирует общее поведение, а реализовать его обязаны потомки. В этом отличие абстрактных методов от виртуальных. Виртуальный метод может, но не обязан быть переопределен в потомках. Абстрактный метод должен быть переопределен в классе-потомке в обязательном порядке.

Класс, содержащий хотя бы один абстрактный метод, является абстрактным, о чем должно быть указано в определении класса

```
abstract class ИмяКласса
{
……
}
```

Потомки обязаны переопределять абстрактный метод с модификатором override (new невозможно, т.к. у предка нет никакой версии реализации метода).

```
abstract class Человек
{
   protected string фам;
   public Человек(string фам)
   {
     this.фам = фам;
   }
   public  abstract  void Показать();
}

class Студент:Человек
{
   private string вуз;
   public Студент(string фам, string вуз):base(фам)
   {
     this.вуз = вуз;
   }
   public override void Показать()
   {
     Console.WriteLine("Я - студент: " + фам + " " + вуз);
   }
}

class Владелец: Человек
{
  private string ном;
  public Владелец(string фам, string ном): base(фам)
  {
    this.ном = ном;
  }
  public override void Показать()
  {
    Console.WriteLine("Я - владелец: " + фам + " --> " + ном);
  }
}

class Служащий: Человек
{
  private string фирма;
  public Служащий(string фам, string фирма): base(фам)
  {
    this.фирма = фирма;
  }
  public override void Показать()
  {
    Console.WriteLine("Я - служащий: "+фам + " ____ " + фирма);
  }
}
```

```
class Program
{
  static void Main(string[] args)
  {
    Человек ч;
    Студент ст = new Студент("Иванов","ВШЭ");
    Владелец вл = new Владелец("Петров", "A777AA-99RUS");
    Служащий сл = new Служащий("Сидоров", "Рога и копыта");
    ч = ст;
    ч.Показать();  //Вызов метода наследника
    //Я - студент: Иванов ВШЭ
    ст.Показать(); //Вызов метода наследника
    //Я - студент: Иванов ВШЭ

    ч = вл;
    ч.Показать();   //Вызов метода наследника
    //Я - владелец: Петров --> A777AA-99RUS
    вл.Показать();  //Вызов метода наследника
    //Я - владелец: Петров --> A777AA-99RUS

    ч = сл;
    ч.Показать();  //Вызов метода наследника
    //Я - служащий: Сидоров ____ Рога и копыта
    сл.Показать(); //Вызов метода наследника
    //Я - служащий: Сидоров ____ Рога и копыта
  }
}
```

Объекты абстрактного класса создать невозможно.

ИСКЛЮЧЕНИЯ И ОТЛАДКА

1. Традиционные способы обработки ошибок. Понятие «Структурная обработка исключений».

Многие системные функции возвращают значение, указывающее на успешное или безуспешное выполнение функции. Однако такой способ извещения программы имеет ряд недостатков:

- Во-первых, программист обязан выполнять все проверки возвращаемого значения и либо реагировать на ошибки, либо передавать их на более высокий уровень программы. Если на одном из уровней проверка не проводится, то ошибки могут повлиять на другие части программы.
- Во-вторых, текст программы загромождается операторами if...else, обрабатывающими нетипичные случаи.
- В-третьих, информация о причине возникновения ошибки не всегда легко доступна коду, который должен обработать ошибку. К тому же функция не всегда способна предугадать появление всех ошибок, связанных с внешней средой.

Для доступа прикладных программ к информации о возникшем *исключении* в Windows (начиная с NT) разработан специальный механизм, называемый *структурной обработкой исключений*.

Исключительная ситуация (или исключение) — это ошибка, которая возникает во время выполнения программы.

Структурная обработка исключений — это метод, применяемый Windows для обработки как программных, так и внутренних аппаратных исключений.

Средство обработки исключений Windows не зависит от используемого языка программирования: один и тот же механизм используется для всех языков.

Каждый язык определяет, каким образом в нем реализуется этот механизм.

2. Организация обработки исключений в C#

Основная идея обработки исключений состоит в том, что в программе можно определить блок кода, именуемый **обработчиком исключений**, который автоматически будет получать управление при возникновении определенной ошибки. Обработчик исключений оформляется в виде **catch-блока**.

В этом случае не обязательно проверять результат выполнения каждой конкретной операции или метода вручную.

Программные инструкции, которые нужно проконтролировать на предмет исключений, помещаются в try-блок.

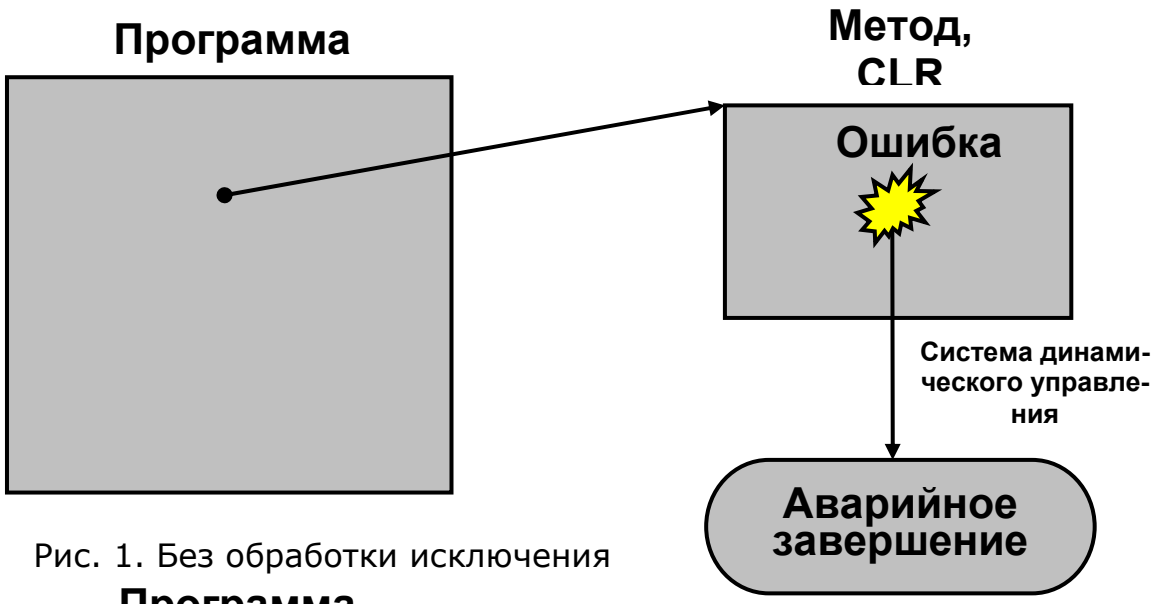


Рис. 1. Без обработки исключения

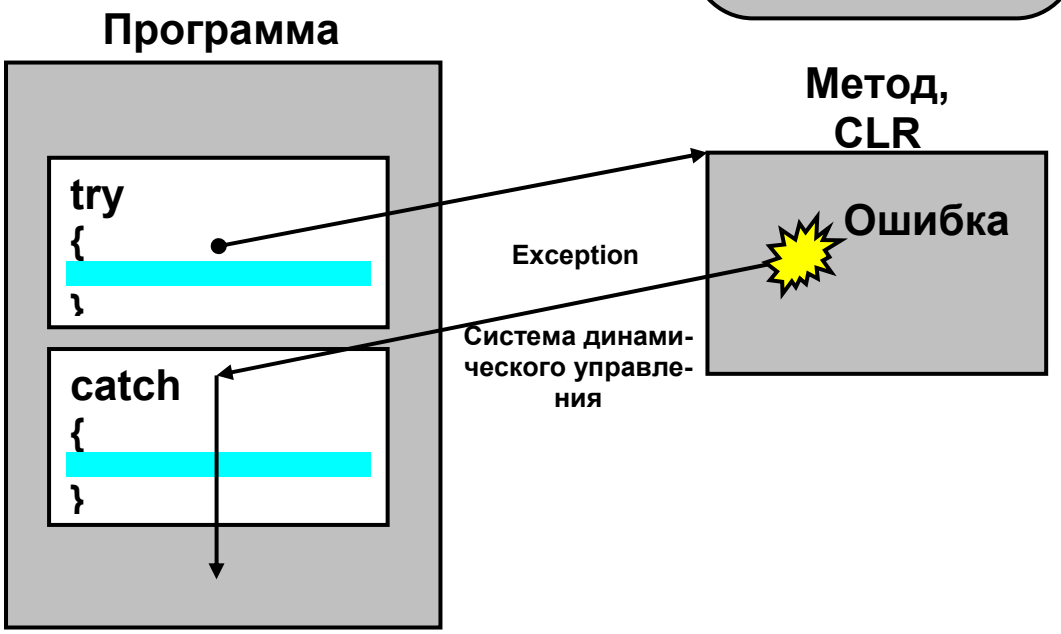


Рис.2. Обработка исключения

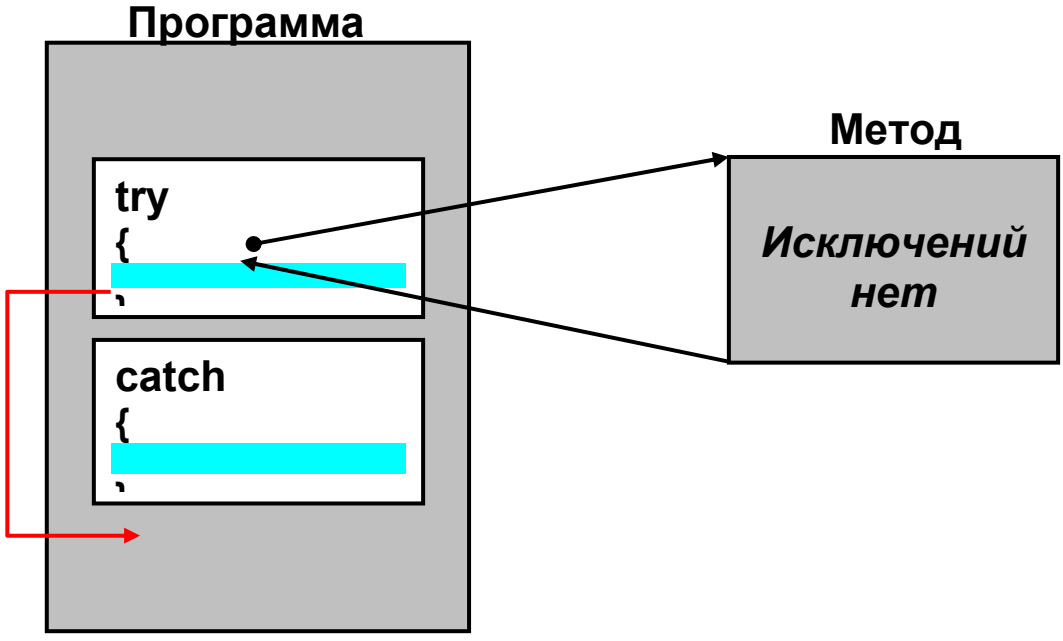
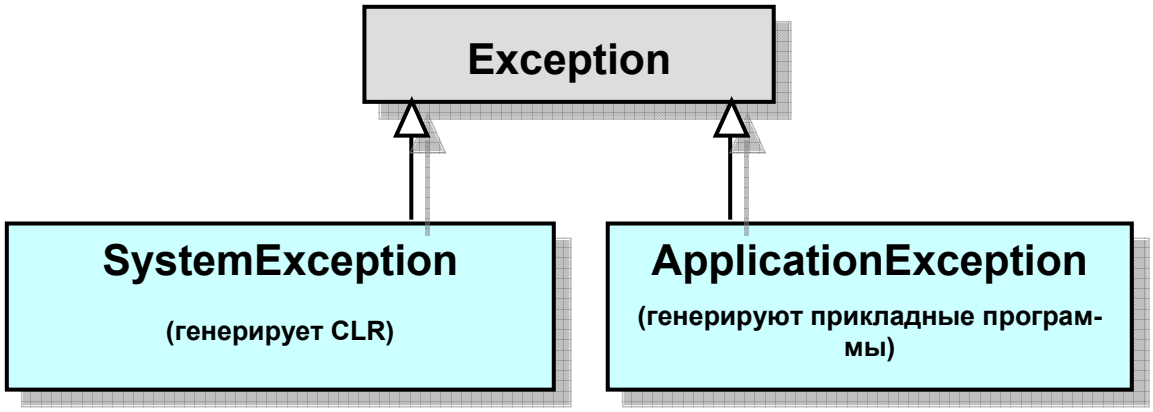


Рис.3 Безошибочное выполнение

В C# исключения представляются классами, выведенными из встроенного класса исключений **Exception**.

Наследниками класса Exception являются классы SystemException и ApplicationException, которые делят исходные исключения на две группы.



Формат записи *try/catch-блоков*:

```
try
{
   // Блок кода, подлежащий проверке на наличие ошибок.
}

catch   ( ExcepType1   e )
{
    //  Обработчик  исключения типа ExcepTypel.
}

catch ( ExcepType2  e )
{
    // Обработчик  исключения типа ExcepType2.
}
```

ExcepType — это тип сгенерированного исключения.
e – экземпляр типа ExcepType.

Варианты catch-блоков:

```
catch   ( ExcepType )
{
   //  Обработчик исключения типа ExcepTypel.
}

catch ( IOException  e )
{
   //  Обработчик всех исключений ввода-вывода
}

catch ( SystemException  e )
{
```

```
    // Обработчик всех системных исключений
}

catch ( Exception  e )
{
    // Обработчик всех исключений.
    // Передается информация об исключении
}

catch ()
{
   // Обработчик всех исключений.
   // Информация об исключении не передается
}
```

Системные исключения класса SystemException

Производный класс	**Описание**
ArithmeticException	Ошибка в арифметических операциях или операциях преобразования (предок DivideByZeroException и OverFlowException)
ArrayTypeMismatchException	Тип сохраняемого значения несовместим с типом массива
DivideByZeroException	Попытка деления на нуль
FormatException	Попытка передать в метод аргумент неверного формата
IndexOutOfRangeException	Индекс массива оказался вне диапазона
InvalidCastException	Неверно выполнено динамическое приведение типов
OutOfMemoryException	Обращение к оператору new оказалось неудачным из-за недостаточного объема свободной памяти
OverflowException	Имеет место арифметическое переполнение и используется оператор checked
StackOverflowException	Переполнение стека
NullReferenceException	Была сделана попытка использовать нулевую ссылку, т.е. ссылку, которая не указывает ни на какой объект
IOException	Ошибка ввода-вывода
TypeInitializationExeption	Отсутствует блок catch для обработки исключения, сгенерированного статическим конструктором

3. Свойства и методы класса Exception

Конструкторы (2 из 4):

1. Инициализирует новый экземпляр класса Exception.

public Exception();

2. Инициализирует новый экземпляр класса Exception с заданным сообщением об ошибке.

public Exception (string);
[C++] public: Exception(String*);
[JScript] public function Exception(String);

В классе Exception определен ряд **свойств**:

Message	Содержит строку, которая описывает причину ошибки
StackTrace	Содержит имя класса и метода, вызвавшего исключение
TargetSite	Содержит имя метода, из которого было вызвано исключение
Source	Содержит имя программы, вызвавшей исключение
HelpLink	Строка с любой дополнительной информацией

Методы:
Метод ToString() возвращает строку с описанием исключения.

Пример 1:

```
try
{
   double d = double.Parse(Console.ReadLine());
}
catch (Exception e)
{
   Console.WriteLine ("Полное описание: "          + e);  //вызов ToString
   Console.WriteLine ("Сообщение об ошибке: "   + e.Message);
   Console.WriteLine ("Имя класса и метода: "     + e.StackTrace);
   Console.WriteLine ("Метод: "                                + e.TargetSite);
}
```

Пример 2:

```
public static void Main()
{
   int[ ] a = new int[4];

   try
   {
      for ( int i = 0;   i < 10;   i++)
         a[i] = i;
   }
```

```
    catch (IndexOutOfRangeException)
    {
      Console.WriteLine ("Индекс вне диапазона!");
      return;
    }
  }
```

Пример 3:

```
  public static void Main()
  {
    try
    {
      Class1.Divide(a, b);
    }

    catch (DivideByZeroException)
    {
      Console.WriteLine("Делитель равен 0");
      b = 1;
    }
  }
```

Пример 4.

```
  public static void Main()
  {
    try
    {
      double c = double.Parse(Console.ReadLine());
    }

    catch (Exception e)
    {
      Console.WriteLine("Error: " + e.Message);
      return;
    }
  }
```

4. Использование нескольких catch-блоков

С try-блоком можно связать не один, а несколько catch-блоков.

Однако все catch- блоки должны перехватывать исключения различного типа.

```
using System;
class Demo
{
  public static void Main()
  {
    // Здесь массив a длиннее массива b.
    int[] a = { 4, 8, 10, 32, 64, 128, 256, 512 };
    int[] b = { 2, 0, 1, 4, 0, 8 };
```

```
        for (int i = 0; i < a.Length; i++)
        {
          try
          {
            Console.WriteLine(a[i] + " / " + b[i] + " равно " + a[i] / b[i]);
          }

          catch (DivideByZeroException)
          {
            Console.WriteLine("Деление на ноль!");
          }

          catch (IndexOutOfRangeException)
          {
            Console.WriteLine("Выход за границы массива.");
            return;
          }
        }
      }
    }
```

В общем случае catch-выражения проверяются в том порядке, в котором они встречаются в программе.

Выполняется только инструкция, тип исключения которой совпадает со сгенерированным исключением. Все остальные catch-блоки игнорируются.

catch-блок, предназначенный для "глобального перехвата" должен быть последним в последовательности catch-блоков.

5. Вложение try-блоков

Один try-блок можно вложить в другой. Исключение, сгенерированное во **внутреннем** try-блоке и не перехваченное catch-инструкцией, которая связана с этим try-блоком, передается во **внешний** try-блок.

Пример.

В следующей программе исключение типа IndexOutOfRangeException перехватывается не внутренним try-блоком, а внешним.

```
using System;
class Demo
{
  public static void Main()
  {
    // Здесь массив a длиннее массива b.
    int[] a = { 4, 8, 16, 32, 64, 128, 256, 512 };
    int[] b = { 2, 0,  4,  4,    0,    8 };


    try
```

```
        {       // Внешний try-блок.
          for (int i = 0; i < a.Length; i++)
          {
            try
            {          // Вложенный try-блок.
              Console.WriteLine (a[i] + " / " + b[i] + " = " + a[i] / b[i]);
            }
            catch (DivideByZeroException)
            {
              Console.WriteLine ("На ноль делить нельзя!");
            }
          }
        }

        catch (IndexOutOfRangeException)
        {
          Console.WriteLine ("Нет соответствующего элемента.");
          Console.WriteLine ("Неисправимая ошибка. "  +
                                "Программа завершена.");
          return;
        }
    . . . . . .
      }
}
```

Исключение, которое может быть обработано внутренним try-блоком (в данном случае это деление на нуль), позволяет программе продолжать работу.

Однако нарушение границ массива перехватывается внешним try-блоком и заставляет программу завершиться.

Внешние try-блоки можно также использовать в качестве механизма **"глобального перехвата"** для обработки тех ошибок, которые не перехватываются внутренними блоками.

6. Использование блока finally

Возникновение и обработка исключения, как правило, приводит к нетипичному завершению программы. При этом в программе могут остаться **неосвобожденные ресурсы**, например, открытые файлы.

Для освобождения ресурсов используют блок *finally*.

Этот блок должен следовать за блоком try или catch. Последний является необязательным.

Формат:

```
try
{
  //  Блок кода, предназначенный для обработки ошибок.
}
catch (ExcepTypel exOb)
{
  //  Обработчик для исключения  типа  ExcepTypel.
```

```
}
catch (ExcepType2 exOb)
{
   //  Обработчик для исключения типа  ExcepType2.
}
. . . . . .
finally
{
   // Код завершения обработки исключений.
}
```

Блок finally будет выполнен в любом случае – было сгенерировано исключение или нет.

Блок finally не может быть определен без блока try. Эта пара (как правило, с блоками catch) может находиться и внутри вызываемого метода.

В одной программе (методе) может быть несколько блоков finally. Все эти блоки будут выполнены после своих try-блоков.

Пример.

```
public static void Main()
{
   try
   {
      double c = double.Parse(Console.ReadLine());
   }
   catch (Exception e)
   {
      Console.WriteLine("Error: " + e.Message);
      return;
   }
   finally
   {
      Console.WriteLine("Здесь освобождаются ресурсы." +
                                    "Блок выполняется всегда");
   }
}
```

7. Генерирование исключений вручную

Если пользователь обнаружил в своем методе ошибку, то он может сгенерировать исключение вручную, используя инструкцию **throw**.

Вызывающая метод программа должна быть готова обработать это исключение.

Формат ее записи таков:

throw exceptOb;

Элемент *exceptOb* — это объект класса исключений, производного от класса Exception.

throw передает управления в CLR.

Пример 1. Генерирование стандартного исключения.

```
using System;

class Class1
{
   public static int Divide (int a, int b)
   {
      if (b == 0)
          throw  new  DivideByZeroException();
      else
         return a / b;
   }
}

class Demo
{
   public static void Main()
   {

      try
      {
         Console.WriteLine("a/b=" + Class1.Divide(4, 0));
      }
      catch (DivideByZeroException)
      {
         Console.WriteLine("На ноль делить нельзя!");
      }
   }
}
```

Пример 2. Генерирование пользовательского исключения.

```
// Деление пополам числа, которое должно быть четным.
using System;

class Class1
{
   public static int Divide2(int a)
   {
      if ( (float)(a / 2) > 0 )
          throw  new Exception ("Число должно быть четным.");
      else
         return a / 2;
   }
}

class Demo
{
   public static void Main()
   {
```

```
        try
        {
           Class1.Divide2 (3);
        }
        catch (Exception e)
        {
           Console.WriteLine("ERROR: " + e.Message);
        }
     }
  }
```

8. Повторное генерирование исключений

Исключение, перехваченное одной catch-инструкцией, можно перегенерировать, чтобы обеспечить возможность его перехвата внешней catch-инструкцией.

Самая распространенная причина для повторного генерирования исключения — позволить нескольким обработчикам получить доступ к исключению.

Чтобы повторно сгенерировать исключение, достаточно использовать ключевое слово **throw**, не указывая исключения.

Пример.

```
using System;

class Class1
{
   public static void genException()
   {
      int[ ] a = { 4, 8, 16, 32, 64, 128, 256, 512 };
      int[ ] b = { 2, 0, 4,   4,   0,    8 };

      for (int i = 0; i < a.Length; i++)
      {
         try
         {
            Console.WriteLine(a[i] + " * " + b[i] + "=" + a[i]*b[i]);
         }
         catch (IndexOutOfRangeException)
         {
            Console.WriteLine("Массивы имеют разную длину.");
             throw;    // Генерируем исключение повторно.
         }
      }
   }
}

class Demo
{
   public static void Main()
   {
      try
      {
```

```
            Class1.genException();
        }
        catch (IndexOutOfRangeException)
        {
            // Перехватываем повторно сгенерированное исключение.
            Console.WriteLine ("Неисправимая ошибка.  "
                                    + "Программа завершена.");
            return;
        }
    }
}
```

9. Пользовательские исключения

В C# имеется возможность обрабатывать исключения, создаваемые программистом.

Для этого достаточно определить класс как производный от класса Exception или от класса ApplicationException, "родоначальника" иерархии, зарезервированной для исключений, связанных с прикладными программами.

Пример 1.

```
class MyArrayException1 : ApplicationException
{
   HelpLink = "Смотри файл Readme.txt";
   Source    = "Программа PGM1";

   public MyArrayException1 (string Message) : base(Message)
   {  }
}
```

Пример 2.

```
class MyArrayException2 : ApplicationException
{
   // Реализуем стандартные  конструкторы.
   public MyArrayException2() : base() { }
   public MyArrayException2 (string Message) : base(Message)
   { }

   // Переопределяем метод  ToString ()
   public override string ToString()
   {
      return Message;
   }
}
```

Пример 3.

```
class MyArrayException3 : ApplicationException
{
   public MyArrayException3()
                              : base ("Выход за границу массива")
```

```
  { }
}
```

Третий пример практически ничего оригинального не вносит, поэтому вместо создания нового класса достаточно использовать существующий, например:

```
Exception myExc = new Exception ("Выход за границу массива");
throw  myExc;
```

или

```
throw  new Exception ("Выход за границу массива");
```

или

```
throw  new  IndexOutOfRangeException ("Выход за границу массива");
```

10. Перехват исключений производных классов

Если нужно перехватывать исключения и базового, и производного класса, поместите первой в catch-последовательности инструкцию с заданием производного класса. В противном случае catch-инструкция с заданием базового класса будет перехватывать все исключения производных классов.

```
// Инструкции перехвата исключений производных классов
// должны стоять перед инструкциями перехвата
// исключений базовых классов.
using System;

class ExceptA : ApplicationException
{
   public ExceptA() : base() { }
   public ExceptA (string str) : base(str) { }
}

class ExceptB : ExceptA
{
   public ExceptB() : base() { }
   public ExceptB (string str) : base(str) { }
}

class OrderMatters
{
   public static void Main()
   {
      for (int x = 0; x < 3; x++)
      {
         try
         {
            if (x == 0) throw new ExceptA (
                                    "Исключение типа ExceptA.");
            else if (x == 1) throw new ExceptB (
```

```
                                    " Исключение типа ExceptB.");
            else throw new Exception();
        }

        catch (ExceptB exc)
        {
            Console.WriteLine(exc);
        }

        catch (ExceptA exc)
        {
            Console.WriteLine(exc);
        }

        catch (Exception exc)
        {
            Console.WriteLine(exc);
        }
    }
  }
}
```

11. Использование ключевых слов checked и unchecked

По умолчанию исключение OverflowException не генерируется. C# позволяет управлять генерированием исключений при возникновении переполнения с помощью ключевых слов **checked** и **unchecked**, которые должны находиться в try-блоке.

Чтобы указать, что некоторое выражение должно быть проконтролировано на предмет переполнения, используйте ключевое слово checked. А чтобы проигнорировать переполнение, используйте ключевое слово unchecked. В последнем случае результат будет усечен так, чтобы его тип соответствовал типу результата выражения. Режим unchecked действует в настройке среды VS по умолчанию.

Ключевое слово checked имеет две формы. Одна проверяет конкретное выражение и называется операторной checked-формой. Другая же проверяет блок инструкций.

Формат:

```
checked (выражение)
   или
checked
{
     // Инструкции, подлежащие проверке.
}
```

Если значение контролируемого выражения переполнилось, генерируется исключение типа OverflowException.

Ключевое слово unchecked имеет две формы. Одна из них — операторная форма, которая позволяет игнорировать переполнение для за-

данного выражения. Вторая игнорирует переполнение, которое возможно в блоке инструкций.

Формат:

```
unchecked (выражение)
   или
unchecked
{
  // Инструкции, для которых переполнение игнорируется.
}
```

В случае переполнения выражение усекается:

```
 byte a=127, b=127, result;

 try {
     result = checked ( (byte)(a*b) );  // инструкция вызовет
  }                                     // исключение.
```

Примеры. Определить в программе несколько вещественных переменных. Ввести с клавиатуры их значения. В случае ввода неверных данных, используя блоки try и catch, реализовать повторный ввод.

Вариант 1. Для большого количества вводимых данных. Один блок try и один блок catch на все операции ввода. После обработки исключения повторяется весь выполненный ввод, включая правильный, заново. Этот вариант хорош для отладки, но не для итоговой программы.

```
using System;
class TestTryCatch
{
 public static void Main()
 {
   double  a, b, c;

   while (true)
   {
     try
     {
         Console.Write("Введи первое вещественное число:");
         a = double.Parse(Console.ReadLine());

         Console.Write("Введи второе вещественное число:");
         b = double.Parse(Console.ReadLine());

         Console.Write("Введи третье вещественное число:");
         c = double.Parse(Console.ReadLine());

         break;
     }

     catch
     {
```

```
            Console.WriteLine("\nВведено неверное данное! " +
                        " Повторите весь ввод заново.\n");
            // continue;  //нужен, если после catch в этом
            // цикле следует какая-либо обработка
          }
        } // while-end


      Console.WriteLine("Ввод закончен!\n");
      // далее основная часть программы
        . . .
    }
  }
```

Вариант 2. Блоки try и catch на каждый оператор ввода. Самый простой вариант, но при большом количестве вводимых данных – самый длинный.

```
using System;
class TestTryCatch
{
   public static void Main()
   {
      double a, b;

      while (true)
      {
         Console.Write("Введи первое вещественное число:");
         try
         {
            a = double.Parse(Console.ReadLine());
            break;
         }
          catch
         {
            Console.WriteLine("\nВведено неверное данное!\n");
         }
      }


      while (true)
      {
         Console.Write("Введи второе вещественное число:");
         try
         {
            b = double.Parse(Console.ReadLine());
            break;
         }
         catch
         {
            Console.WriteLine("\nВведено неверное данное!\n");
         }
      }
```

```
        Console.WriteLine("Ввод закончен!\n");
        // далее основная часть программы
        . . .
    }
}
```

Вариант 3. Для большого количества вводимых данных. Один блок try и один блок catch на все операции ввода. Запоминается номер ошибочного ввода. После обработки исключения повторяется только последний ошибочный ввод.

```
using System;
class TestTryCatch
{
   public static void Main()
   {
     double  a, b, c;
     bool    rep=true;
     int     n=1;              // номер неудачного ввода

     while (rep==true)
     {
       try
       {
         switch (n)
         {
         case 1:
           Console.Write("Введи первое вещественное число:");
           a = double.Parse(Console.ReadLine());
           n = 2;                        // N следующего ввода
           break;
        case 2:
           Console.Write("Введи второе вещественное число:");
           b = double.Parse(Console.ReadLine());
           n = 3;
           break;
        case 3:
           Console.Write("Введи третье вещественное число:");
           c = double.Parse(Console.ReadLine());
           n = 1;             // для повтора программы
           rep = false;       // для выхода из цикла while
           break;
           }
      }

      catch
      {
        Console.WriteLine("\nВведено неверное данное!\n");
        //continue; //если после catch в этом цикле следует обработка
      }
    } // while-end
```

```
    rep=true;                                   // для повтора программы

    Console.WriteLine("Ввод закончен!\n");
    // далее основная часть программы
     . . .
  }
}
```

БАЙТОВЫЕ И СИМВОЛЬНЫЕ ПОТОКИ

1. Организация C#-системы ввода-вывода

Понятие потока

Внешние устройства делятся на:

- Устройства ввода-вывода (дисплей, клавиатура, принтер, последовательный порт, мышь, сканер и т.д.)
- Запоминающие устройства последовательного доступа (стример)
- Запоминающие устройства прямого доступа (МД, CD-ROM, DVD-ROM);

C#-программы выполняют операции ввода-вывода посредством потоков. С точки зрения концепции ввода/вывода, **поток (stream)** — это абстракция, которая определяется как последовательность байтов, участвующих в операции ввода/вывода и независящая от устройства.

С точки зрения реализации, **поток** – это объект, используемый для передачи данных.

Концепция передачи данных отделена от конкретного источника, поэтому источники можно заменять. Внешним источником может быть даже переменная программы. Характер поведения всех потоков одинаков, поэтому создано множество обобщенных методов, предназначенных для перемещения данных между внешними источниками и переменными программы. Нюансы различных физических устройств потоковыми методами во внимание не принимаются.

Следовательно, классы и методы, работающие с потоками, можно применить ко многим типам устройств. Например, методы, используемые для записи данных на консольное устройство, также можно использовать для записи в дисковый файл.

рис.1

На самом низком уровне все C#-системы ввода-вывода оперируют байтами. На физическом уровне Windows используется буферизация и кэширование записей. Их целью является:

- согласование ввода/вывода с характеристиками внешнего устройства;
- повышение эффективности системы, за счет уменьшения количества операций ввода/вывода с внешними устройствами, например, с диском.

Буферизованный ввод-вывод:

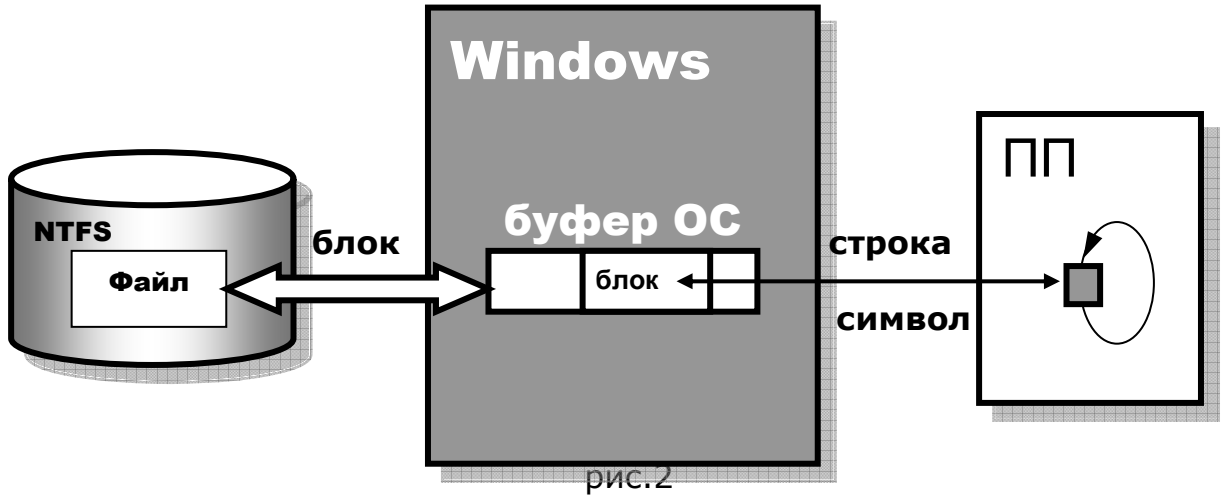
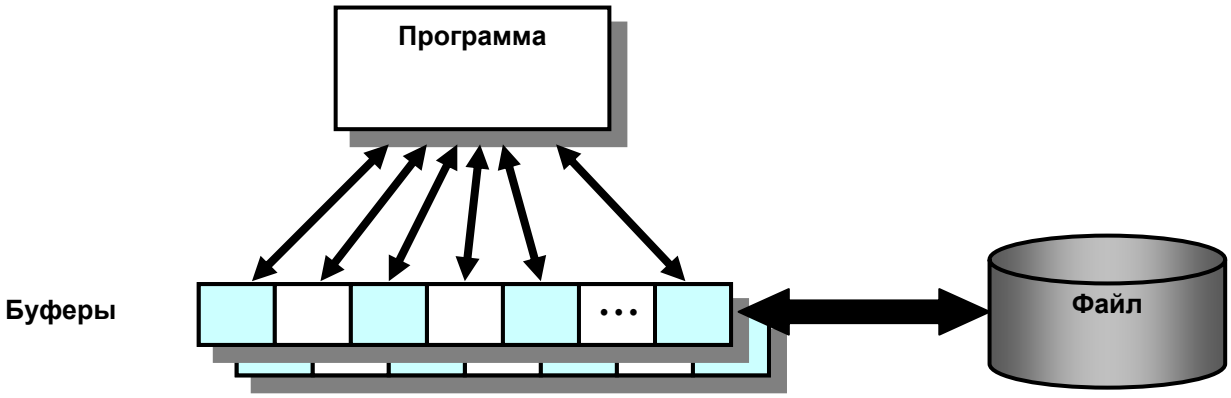


рис.2

Буфер заполняется блоками (кластерами): первый раз при открытии файла для чтения.

Буферов может быть несколько.



см. Flush

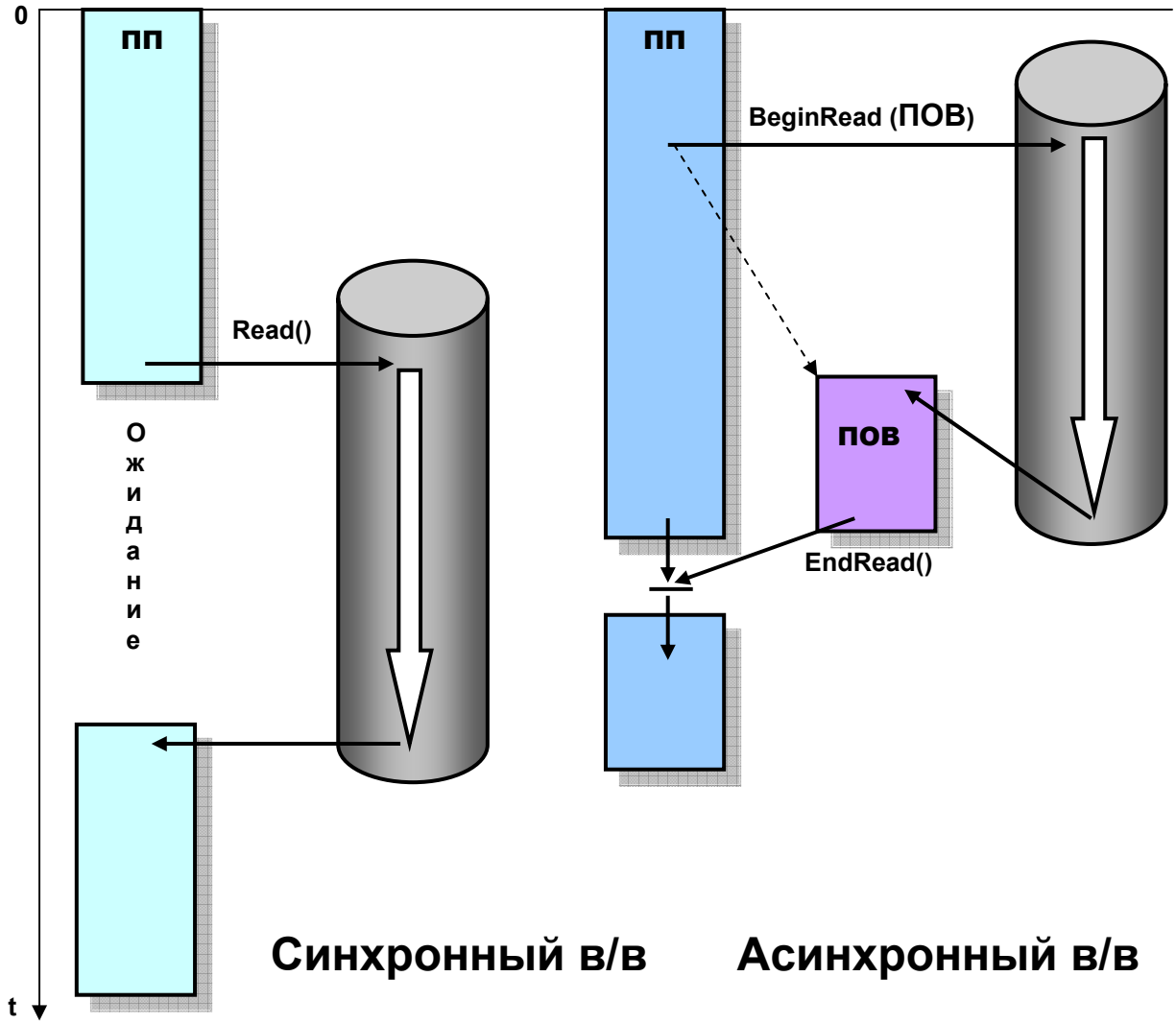
рис. 3

Синхронный и асинхронный ввод/вывод

Как правило, после запуска операции ввода-вывода программа переходит в *состояние ожидания* и выходит из него только после завершения обмена. Такой способ выполнения операций ввода-вывода называется **синхронным**.

Однако продолжительный ввод-вывод целесообразно выполнять в **асинхронном** режиме, когда программа не блокируется, а продолжает выполняться. Асинхронный режим реализуется с помощью *подпрограммы обратного вызова* (на рис.4 – ПОВ). Об этом пойдет речь в других лекциях.

рис.4



Стандартные потоки

В C# определен ряд классов, которые преобразуют байтовый поток в символьный, и наоборот, выполняя byte-char- и char-byte-перевод автоматически.

Наряду с байтовыми и символьными потоками существуют двоичные потоки.

Кроме потоков, создаваемых программами, в Windows существуют несколько встроенных (стандартных) потоков.

Эти потоки не надо создавать, они всегда доступны программам.

2. Классы потоков

Все классы потоков (кроме стандартных) определены в пространстве имен System.IO.

Центральную часть байтовой потоковой C#-системы занимает класс System.IO.Stream.

```
public abstract class  Stream     : MarshalByRefObject, IDisposable
public    class  FileStream       : Stream
public    class  MemoryStream     : Stream
```

```
public abstract      class  TextReader : MarshalByRefObject, Idisposable
public abstract      class  TextWriter  : MarshalByRefObject, Idisposable

public    class  StreamReader : TextReader
public    class  StreamWriter  : TextWriter

public    class  StringReader   : TextReader
public    class  StringWriter    : TextWriter

public    class  BinaryReader : Idisposable
public    class  BinaryWriter  : Idisposable
```

Основные классы.

потоковые: FileStream, StreamReader, StreamWriter, BinaryReader, BinaryWriter, Console.

непотоковые: File.

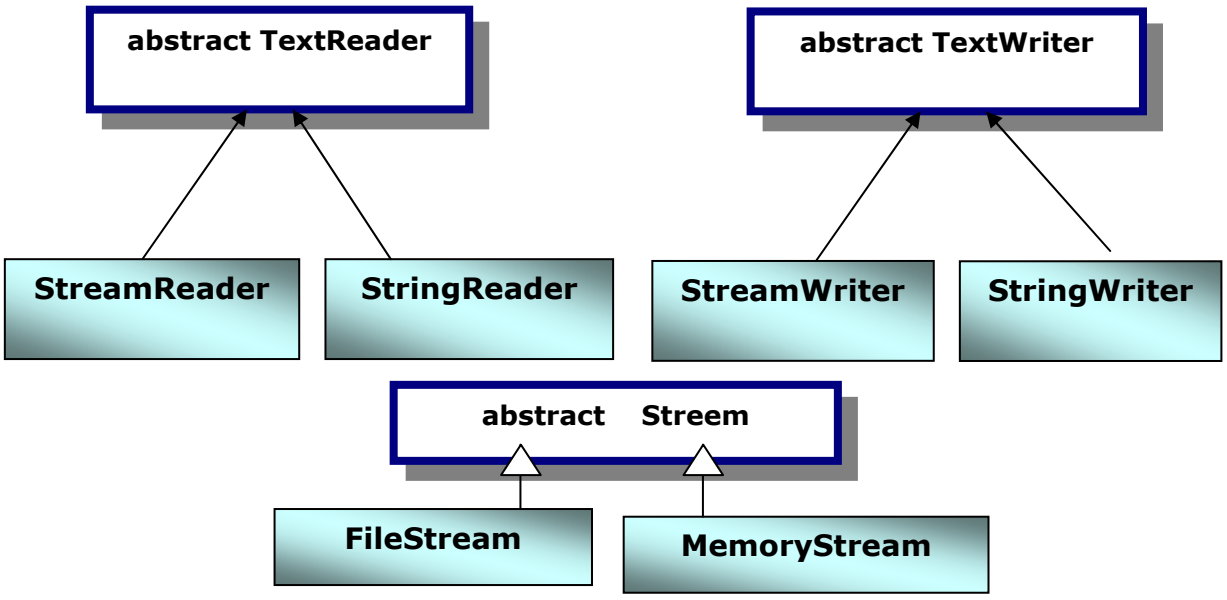


рис. 5

Другие классы: MemoryStream, NetworkStream, BufferedStream, CryptoStream.

3. Консольный ввод-вывод данных

C#-программы в пространстве имен System могут использовать:

- стандартный входной поток Console.In, в который вводится информация с клавиатуры;
- стандартный выходной поток Console.Out, в который выводится информация, направляемая в консоль (на экран);
- стандартный выходной поток Console.Error сообщений об ошибках, в который выводится информация, направляемая в консоль (на экран).

Console.In, Console.Out и Console.Error – это свойства класса Console, значением которых являются соответствующие символьные потоки (объекты).

Примеры ввода-вывода:

Console.ReadLine(…) и Console.WriteLine(…).

Console.Out и Console.Error — объекты типа TextWriter. Поток Console.In - объект типа TextReader.

Для доступа к этим потокам можно использовать методы и свойства, определенные в классах **TextWriter** и **TextReader** соответственно. Однако это целесообразно делать только для потока Console.Error:

```
public static void Main()
{
   int a, b=7, c=0;

   try
   {
         a = b / c;              // Деление на нуль: генерируем исключение.
   }

   catch (DivideByZeroException  exc)
   {
         Console.Error.WriteLine (exc.Message);
   }
}
```

У стандартных потоков можно, не переделывая программу, менять источник и приемник информации (т.е. перенаправлять поток).

Перенаправить стандартный поток можно двумя способами.

1) средствами Windows (внешними);

2) из программы (внутренними средствами) с помощью методов SetIn(), SetOut() и SetError(), которые являются членами класса Console.

Способ 1. Средствами Windows все стандартные потоки, кроме потока ошибок, могут быть перенаправлены при запуске программы (то есть временно) на любое совместимое устройство ввода-вывода.

Для этого при запуске программы из командной строки можно использовать оператор "<", чтобы перенаправить поток Console.In и операторы ">" и ">>" для перенаправления потока Console.Out.

Примеры.
```
example.exe  >      NewFile.txt
example.exe  >> OldFile.txt
example.exe  <      OldFile.txt
example.exe  >      NewFile.txt  <  OldFile.txt
```

Способ 2. Перенаправление осуществляется с помощью методов SetIn(), SetOut() и SetError(), которые являются членами класса Console:

Формат:
```
static void SetIn      (TextReader input)
static void SetOut   (TextWriter output)
static void SetError (TextWriter output)
```

Такое перенаправление имеет постоянное действие и его нельзя отменить или повторно перенаправить при запуске программы.

Пример.

```
using System;
using System.IO;

class Redirect
{
   public static void Main()
   {
      StreamWriter log_out = new StreamWriter("C:\\logfile.txt");

      // Направляем стандартный выходной поток в системный журнал.
      Console.SetOut(log_out);

      Console.WriteLine ("Это начало системного журнала.");

      for (int i = 0; i < 10; i++)  Console.WriteLine(i);

      Console.WriteLine ("Это конец системного журнала.");
      log_out.Close();
   }
}
```

Можно указывать любой поток, если он является производным от класса TextReader или TextWriter.

4. Чтение и запись двоичных файлов

(потоковый файловый ввод-вывод)

На уровне операционной системы все файлы обрабатываются на побайтовой основе.

рис.6

Открытие файла (создание файлового потока)

Чтобы создать байтовый поток, связанный с файлом, можно создать объект класса FileStream.

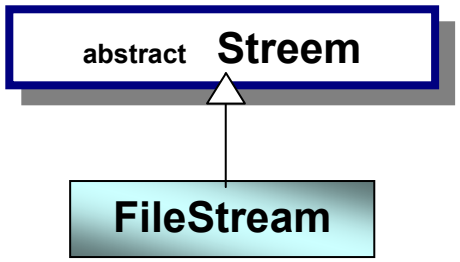


рис.7

В классе FileStream определено несколько конструкторов. Чаще всего из них используется следующий:

FileStream (string filename, FileMode mode)

filename - путь к файлу, который необходимо открыть.
mode - режим открытия файла.

Этот конструктор открывает файл для доступа с разрешением чтения и записи.

Примеры путей:	*Буквальные литералы:*
"c:\\MyDir\\MyFile.txt"	@"c:\MyDir\MyFile.txt"
"c:\\MyDir"	@"c:\MyDir"
"MyDir\\MySubdir"	@"MyDir\MySubdir"
"\\\\MyServer\\MyShare"	@"\\MyServer\MyShare"

Таблица. Значения перечисления **FileMode**

Значение	Описание
FileMode.Append	Добавляет выходные данные в конец файла
FileMode.Create	Создает новый выходной файл. Существующий файл с таким же именем будет разрушен
FileMode.CreateNew	Создает новый выходной файл. Файл с таким же именем не должен существовать
FileMode.Open	Открывает существующий файл

FileMode.OpenOrCreate	Открывает файл, если он существует. В противном случае создает новый
FileMode.Truncate	Открывает существующий файл, но усекает его длину до нуля

Исключения:

IOException	файл невозможно открыть из-за ошибки ввода-вывода
FileNotFoundException	файл невозможно открыть по причине его отсутствия
ArgumentNullException	имя файла представляет собой null-значение
ArgumentException	некорректен параметр mode
SecurityException	пользователь не обладает правами доступа
DirectoryNotFoundException	некорректно задан каталог

Пример.

```
FileStream   fin;
fin = new FileStream("test.dat",  FileMode.Open);
```

Если необходимо ограничить доступ только чтением или только записью, используйте следующий конструктор:

```
FileStream (string filename, FileMode mode, FileAccess how)
```

Значения перечисления FileAccess

FileAccess.Read	только читать
FileAccess.Write	только писать
FileAccess.ReadWrite	читать и писать

Пример.

```
FileStream  fin;
fin = new FileStream("test.dat", FileMode.Open, FileAccess.Read);
```

Закрытие файла

void Close() – объектный метод

Пока файл не закрыт, его не могут использовать другие программы.
Любые данные, оставшиеся в дисковом буфере, будут переписаны на диск, ресурсы освободятся.

Открытое свойство: Length – длина файла.

Чтение/запись байтов

Открытые методы класса FileStream

Flush	Переопределен. Инициирует запись всех буферизованных данных на основное устройство и очищает все

	буферы для данного потока.
Read	Переопределен. Чтение блока байтов из потока и запись данных в заданный буфер.
ReadByte	Переопределен. Считывание байта из файла и перемещение положение чтения на один байт.
Write	Переопределен. Запись блока байтов в данный поток с использованием данных из буфера.
WriteByte	Переопределен. Запись байта в текущую позицию в потоке файла.

ЧТЕНИЕ:
Можно прочитать любой файл. Чтобы прочитать из файла *один байт*:

```
int  ReadByte()
```

Каждый прочитанный байт преобразуется в int. При обнаружении конца файла метод возвращает —1.

Чтобы считать *массив байтов*:

```
int   Read (byte[ ] buf,  int offset,  int numBytes)
```

Метод Read() пытается считать numBytes байтов в массив buf, начиная с элемента buf[offset]. Он возвращает количество успешно считанных байтов.

ЗАПИСЬ:
Чтобы записать 1 *байт в файл*:

```
void  WriteByte (byte val)
```

Чтобы записать в файл *массив байтов*:

```
void   Write (byte[ ]   buf,   int  offset,   int  numBytes)
```

Метод Write() записывает в файл numBytes байтов из массива buf, начиная с элемента buf [offset].

Исключения:

IOException	ошибка ввода-вывода
NotSupportedException	поток не открыт для ввода/вывода данных
ObjectDisposedException	поток закрыт

Если вы хотите записать данные на физическое устройство вне зависимости от того, полон буфер или нет, вызовите следующий метод:

```
void   Flush ()
```

Пример. Копирование файла (без обработки исключений).

Программа CopyFile копирует файл любого типа, включая выполняемые файлы. Имена исходного и приемного файлов указываются в программе.

```
using System;
using System.IO;

class CopyFile
{
   public static void Main(void)
   {
      int i;
      FileStream   fin,  fout;

      // Открываем входной файл.
      fin = new FileStream ("c:\\source.txt", FileMode.Open);

      // Открываем выходной файл.
      fout = new FileStream ("c:\\target.txt", FileMode.Create);

      // Копируем файл.
      while (true)
      {
         i = fin.ReadByte();
          if (i == -1) break;
         fout.WriteByte ( (byte)i );
      }

      fin.Close();   fout.Close();
   }
}
```

Пример. Копирование файла с обработкой исключений.
Имена исходного и приемного файлов указываются в командной строке.

```
/*
```
Для использования этой программы укажите при запуске имя исходного и приемного файлов.
Например, чтобы скопировать файл FIRST.DAT в файл SECOND.DAT, используйте следующую командную строку:

```
CopyFile   FIRST.DAT   SECOND.DAT

*/

using System;
using System.IO;

class CopyFile
{
```

```
public static void Main(string[] args)
{
   int i ;
   FileStream fin ;
   FileStream fout ;

   try
   {
      // Открываем входной файл.
      try
      {
          i = 1;
         fin = new FileStream (args[0], FileMode.Open);
      }

      catch (FileNotFoundException exc)
      {
           Console.WriteLine(exc.Message + "\nВходной файл не найден.");
         return;
      }

      // Открываем выходной файл.
      try
      {
          i = 2;
         fout = new FileStream (args[1], FileMode.Create);
      }

      catch (IOException exc)
      {
         Console.WriteLine ( exc.Message +
                        "\nОшибка при открытии выходного файла.");
         return;
      }
   }
   catch (IndexOutOfRangeException exc)     // ошибка извлечения строки из массива
args[]
   {
      Console.WriteLine(exc.Message +
                  "\nНе указано имя файла {0} в командной строке", i );
      return;
   }

   // Копируем файл.
   try
   {
      do
      {
         i = fin.ReadByte();
         if (i != -1) fout.WriteByte ( (byte)i );
      } while (i != -1);
   }

   catch (IOException exc)
```

```
        {
          Console.WriteLine(exc.Message +
                                                "Ошибка при чтении/записи файла.");
        }

        fin.Close();
        fout.Close();
    }
}
```

5. Чтение и запись текста (символьные потоки)

Символьные потоки работают с любой кодировкой.

Чтобы выполнять файловые операции на символьной основе, поместите объект класса **FileStream** внутрь объекта класса **StreamReader** или класса **StreamWriter**.

Эти классы автоматически преобразуют байтовый поток в строки и наоборот.

ОСОБЕННОСТИ:

Метод **StreamReader.ReadLine()** автоматически определяет конец строки и прекращает чтение в этом месте.

Метод **StreamWriter.Writer()** автоматически добавляет комбинацию конца строки (\r\n – возврат каретки, перевод строки).

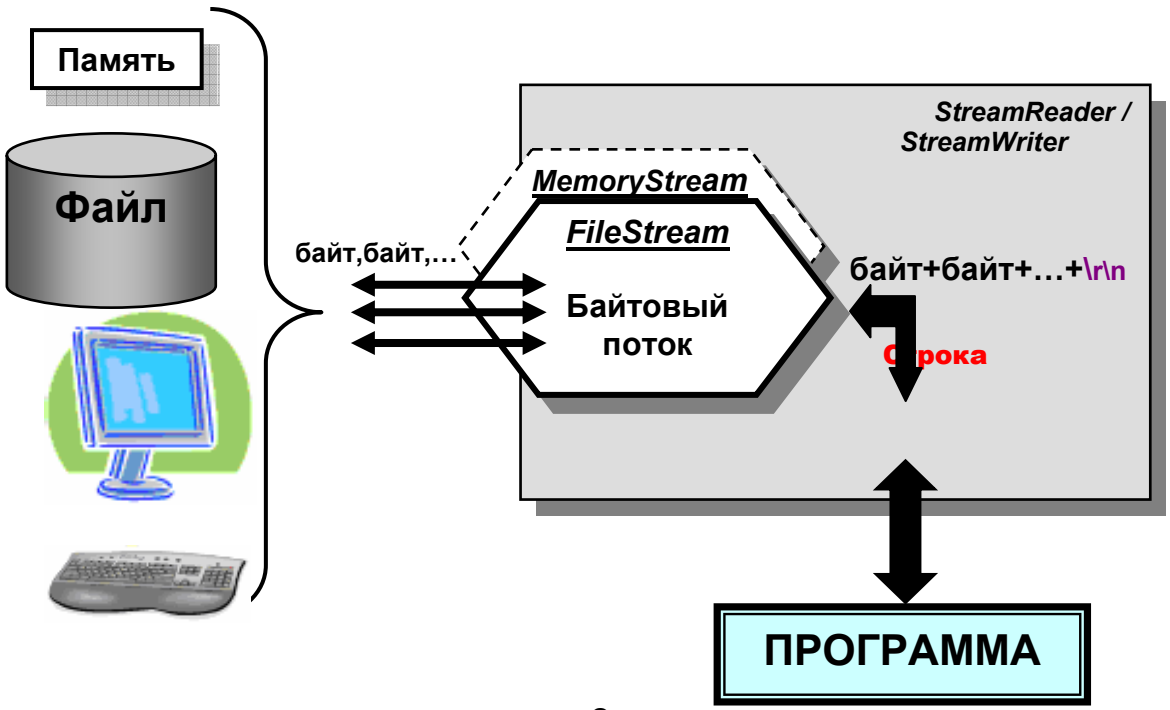


рис.8

Присоединение файла или потока к классам StreamReader или StreamWriter

Класс **StreamReader** можно использовать для *чтения* данных из любого источника. Для этого вместо применения имени файла в конструкторе можно применять ссылку на другой поток. Например, в классе FileInfo существуют методы OpenText() и CreateText(), которые возвращают ссылки на StreamReader.

Класс **StreamWriter** можно использовать для *записи* данных в любой приемник.

Класс *StreamReader* — производный от *TextReader*.
Класс *StreamWriter* — производный от класса *TextWriter*.

Следовательно, классы StreamWriter и StreamReader имеют доступ к методам и свойствам, определенным их базовыми классами.

StreamReader

В классе StreamReader определено несколько конструкторов. Чаще всего используются следующие конструкторы, которые проверяют маркеры кодировки в начале файла (2-3 символа):

```
StreamReader (string  path)        //прямое присоединение StreamReader к файлу
```

path – полный путь (имя файла). Файл должен существовать.

```
StreamReader (Stream stream) // присоединение StreamReader к потоку
```

stream - имя открытого потока.

Примеры.

```
1) StreamReader  sr = new StreamReader(@"c:\Temp\ReadMy.txt");

2) FileStrim fs = new FileStrim (@"c:\Temp\ReadMy.txt",
                    FileMode.Open, FileAccess.Read, FileShare.None);
   StreamReader  sr = new StreamReader(fs);
```

Методы чтения:

```
string str = sr.ReadLine();          // читать строку,       null - EOF

string str2 = sr.ReadToEnd();  // прочитать весь остаток файла

int nextChar = sr.Read ();  // прочитать один символ (-1 = EOStrim)

char[ ] chr = new char[100];
int n = sr.Read(chr, 0, 100);   // 100 символов со смещением 0
                                                  // n – реально прочитанное кол-во
```

Возвращаемая строка не содержит завершающий знак возврата каретки или перевода строки.

Если ссылка, возвращаемая методом ReadLine(), равна значению null, значит, достигнут конец файла.

StreamWriter

Самые популярные конструкторы класса StreamWriter:

```
StreamWriter (string  path)

StreamWriter (Stream stream)

StreamWriter (string  filename,    bool  appendFlag)
```

Если appendFlag равен значению true, выводимые данные добавляются в конец существующего файла. В противном случае заданный файл перезаписывается. В последних двух случаях, если файл не существует, он создается.

Примечание. Применяется кодировка UTF8, принятая в .NET по умолчанию. Для смены – другие конструкторы.

Методы записи:

```
string str;
sr.WriteLine(str);                    // записать строку

char chr1 = 'a';
sr.Write(chr1);                       // записать один символ

char[ ] chr = new char[100];
sr.Write (chr);                       // записать 100 символов

char[ ] chr = new char[100];
sr.Write (chr, 25, 50);               // записать 50 символов с 25-ого
```

Если вы сами формируете строку из последовательности символов (для Write), то не забудьте ее оформить как строку:

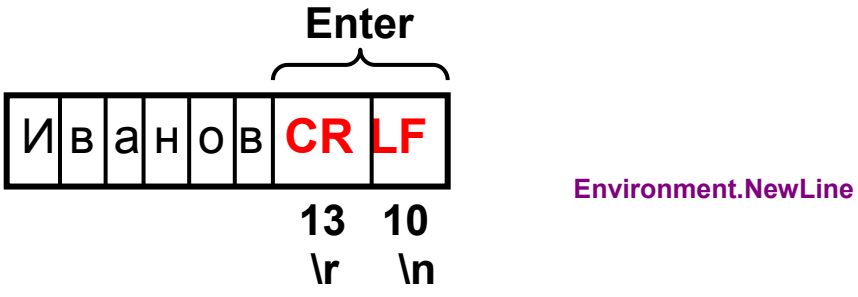


рис. 9

Пример чтения/записи. (Без обработки исключений.)

```
/*  Простая утилита  "клавиатура-диск"
    демонстрирует использование класса StreamWriter. */

using System;
using System.IO;

class KtoD
{
   public static void Main()
   {
      string str;
      FileStream  fout;

      fout = new FileStream ( "test.txt",   FileMode.Create );
      StreamWriter   sout = new StreamWriter ( fout );

      Console.WriteLine ("Введите текст ('стоп' для завершения).");
      while (true);
      {
         str = Console.ReadLine();

         if (str = "стоп") break;
         {
            str = str + "\r\n";          // Добавляем символы новой строки.
          sout.Write(str);
         }
      }

      sout.Close();
   }
}
```

Исключения:

ArgumentException	поток stream не открыт для ввода/вывода
ArgumentNullException	поток stream имеет null-значение
FileNotFoundException	Не удается найти файл (для path).
DirectoryNotFoundException	Указанный путь (path) недопустим; возможно, он отсылает к неотображенному диску.
IOException	Параметр path включает неправильный или недопустимый синтаксис имени файла, имени папки или метки тома.

Пример. (С обработкой исключений.)

```
/*  Простая утилита  "клавиатура-диск", которая
    демонстрирует использование класса StreamWriter. */
using System;
using System.IO;
```

```
class KtoD
{
   public static void Main()
   {
      string str;
      FileStream fout;

      try
      {
         fout = new FileStream("test.txt", FileMode.Create);
      }
      catch (IOException exc)
      {
         Console.WriteLine(exc.Message +  " Не удается открыть файл.");
         return;
      }

      StreamWriter fstr_out = new StreamWriter(fout);

      Console.WriteLine("Введите текст ('стоп' для завершения).");
      do
      {
         Console.Write(": ");
         str = Console.ReadLine();

         if (str != "стоп")
         {
            str = str + "\r\n";          // Добавляем символ новой строки.

            try
            {
               fstr_out.Write(str);
            }
            catch (IOException exc)
            {
               Console.WriteLine(exc.Message + " Ошибка при работе с файлом.");
               return;
            }
         }
      } while (str != "стоп");

      fstr_out.Close();
   }
}
```

Следующая программа создает простую утилиту "диск - клавиатура", которая считывает текстовый файл test.txt и отображает его содержимое на экране. Таким образом, эта программа представляет собой дополнение к утилите, представленной ранее.

Пример. (Без обработки исключений.)

```
//  Простая утилита "диск - клавиатура" демонстрирует
```

```
//      использование класса StreamReader.
using System;
using System.IO;

class DtoK
{
   public static void Main()
   {
      string s;

      FileStream fin = new FileStream ("test.txt", FileMode.Open);

      StreamReader sin = new StreamReader(fin);

      // Считываем файл построчно.
      while ((s = sin.ReadLine()) != null)
      {
         Console.WriteLine(s);
      }

      sin.Close();
   }
}
```

Пример. (С обработкой исключений.)

```
using System;
using System.IO;

class DtoS
{
   public static void Main()
   {
      string s;
      FileStream fin;

      try
      {
         fin = new FileStream ("test.txt", FileMode.Open);
      }
      catch (FileNotFoundException exc)
      {
         Console.WriteLine(exc.Message + "Не удается открыть файл.");
         return;
      }

      StreamReader fstr_in = new StreamReader(fin);

      // Считываем файл построчно.
      while ((s = fstr_in.ReadLine()) != null)
      {
         Console.WriteLine(s);
      }
```

```
        fstr_in.Close();
    }
}
```

Пример открытия выходного файла классом StreamWriter. (Без обработки исключений.) Новая версия предыдущей утилиты "клавиатура-диск".

```
// Открытие файла с использованием класса StreamWriter.

using System;
using System.IO;

class KtoD
{
   public static void Main()
   {
      string str;

      // Открываем файл напрямую, используя класс StreamWriter.
      StreamWriter  sout  =  new StreamWriter ( "test.txt" );

      Console.WriteLine ("Введите текст ('стоп' для завершения).");
      do
      {
         Console.Write (": ");
         str = Console.ReadLine();

         if (str != "стоп")
         {
            str = str + "\r\n";        // Добавляем символ новой строки
            sout.Write(str);
         }
      }  while (str != "стоп");

      sout.Close();
   }
}
```

Если файл не существует, то он создается. У конструктора класса FileStreem возможностей больше.

Кодировка текстовых потоков

В C# используются следующие статические свойства класса System.Text.Encoding для кодировки текстовых потоков

Кодировка	Описание
ASCII	Кодировка ASCII без символов кириллицы, в которой для представления текстовых символов используются младшие 7 бит байта
Unicode	Кодировка UNICODE. Для представления символов используется 16 бит (т. е. 2 байта). Сигнатура = FF FE в нач.файла

	для StreamWriter
UTF7	Кодировка UCS Transformation Format. Применяется для представления символов UNICODE. В ней используются младшие 7 бит данных
UTF8	То же, но для представления символов UNICODE в ней используется 8 бит данных
Default	Системная кодировка ANSI (не путайте ее с кодировкой ASCII). В этой кодировке для представления каждого символа используется 8 бит данных

Свойства класса кодировки System.Text.Encoding:
ASCII – 1 байт (старший бит = 0);
Default – по умолчанию (UTF8);
Unicode – 2 байта;
UTF32 – 4 байта;
UTF7 – 1 байт, старший бит не используется;
UTF8 – 1 байт (по умолчанию в .NET).

ФАЙЛЫ В ОП, ДВОИЧНЫЕ ПОТОКИ, НЕПОТОКОВЫЙ ВВОД-ВЫВОД, ПРЯМОЙ ДОСТУП К ФАЙЛУ.

1. Файлы в оперативной памяти

Присоединение массива байтов к классам StreamReader или StreamWriter.

Класс MemoryStream

Иногда удобно создать модель файла в ОП. В этом случае нужно заменить *открытие файла* на *создание потока в памяти*.

Класс MemoryStream определяет поток байтов в оперативной памяти (см. рис.1). Класс MemoryStream — это реализация класса Stream.

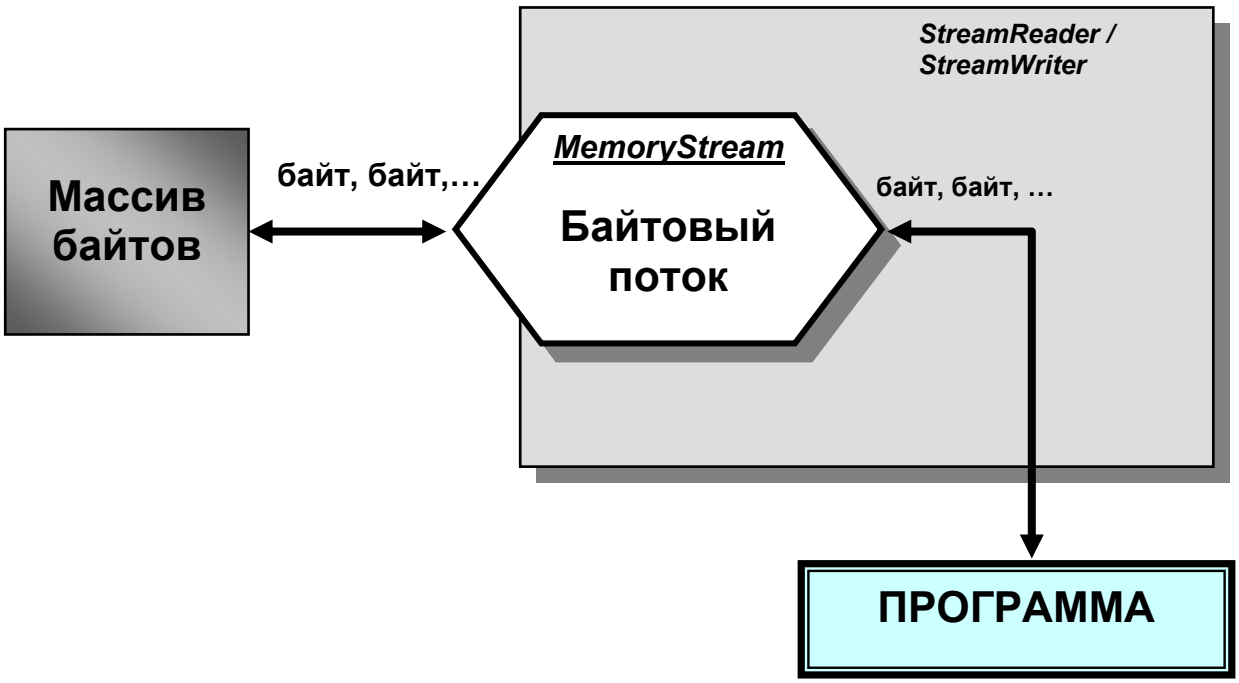


рис.1

Для работы с этим потоком могут использоваться классы StreamReader и StreamWriter. С помощью их методов можно считывать входные данные из массива байтов или записывать их в массив.

Один из конструкторов класса MemoryStream:

```
MemoryStream (byte[ ] buf)
```

buf — это массив байтов, который предполагается использовать в операциях ввода-вывода в качестве источника и/или приемника информации.

- Поток доступен как на чтение, так и на запись.
- В поток, создаваемый этим конструктором, можно записывать данные или считывать их из него.
- Перед использованием этого конструктора необходимо позаботиться о достаточном размере массива buf, чтобы он позволил сохранить все направляемые в него данные.
- Этот поток поддерживает метод Seek(), который позволяет перед выполнением чтения/записи сместиться к любому байту массива.

Пример:

```
// Демонстрация использования класса MemoryStream.
using System;
using System.IO;
class MemStrDemo
{
   public static void Main()
   {
      byte[ ] storage = new byte[255];              // Модель файла

      // Создаем поток с ориентацией на память.
      MemoryStream mem = new MemoryStream (storage);

      // Помещаем объект mem в оболочку StreamWriter.
      StreamWriter  memWtr = new StreamWriter (mem);

      // Записываем данные в память с помощью объекта memWtr.
      for (int i = 0; i < 10; i++)
            memWtr.WriteLine ("byte [{0}]: {0}", i);

      memWtr.Write ('.');       // Ставим в конце точку.
      memWtr.Flush ();              // Форсируем вывод


       Console.WriteLine ("Считываем данные прямо из массива storage: ");

      // Отображаем напрямую содержимое памяти.
      foreach (char ch in storage)
      {
         if (ch == '.') break;     // После точки будет EOF, который является концом стро-
ки.
         Console.Write (ch);
      }

     Console.WriteLine("\nСчитываем данные посредством объекта memRdr: ");

      // Помещаем объект mem в оболочку StreamReader.
      StreamReader memRdr = new StreamReader (mem);

      // Считываем данные из объекта mem, используя средство чтения потоков.

      mem.Seek (0, SeekOrigin.Begin);  // Установка указателя позиции
                                                                  // в начало потока.
      string str = memRdr.ReadLine();

      while (str != null)
      {
         str = memRdr.ReadLine();
         if (str.CompareTo(".") == 0) break;  // После точки следует EOF,
                                                                  // поэтому точка считыва-
ется как строка.
         Console.WriteLine(str);
      }
```

```
    }
}
```

Вот как выглядят результаты выполнения этой программы:

```
Считываем данные прямо из массива storage:
byte    [0] : 0
byte    [1] : 1
byte    [2] : 2
byte    [3] : 3
byte    [4] : 4
byte    [5] : 5
byte    [6] : 6
byte    [7] : 7
byte    [8] : 8
byte    [9] : 9

Считываем данные посредством объекта memRdr:
byte    [0] : 0
byte    [1] : 1
byte    [2] : 2
byte    [3] : 3
byte    [4] : 4
byte    [5] : 5
byte    [6] : 6
byte    [7] : 7
byte    [8] : 8
byte    [9] : 9
```

Вместо

```
mem.Seek (0, SeekOrigin.Begin);
```

можно применить

```
memWtr.Close();
mem = new MemoryStream (storage);
```

Классы StringReader и StringWriter

Если удобнее работать не с массивом байтов в ОП (класс MemoryStream), а с массивом строк, то целесообразно использовать классы *StringReader* и *StringWriter* (наследники классов TextReader и TextWriter).

Конструкторы:

```
StringReader (string str);

StringWriter();
```

Строка особого формата автоматически создается конструктором StringWriter().

```
using System;
```

```
using System.IO;
class StrRdrDemo
{
   public static void Main()
   {

                    // Создаем объект класса StringWriter.
      StringWriter strWtr = new StringWriter();
                    // Записываем данные в StringWriter-объект.
      for ( int i = 0;  i < 10;  i++ )
          strWtr.WriteLine ("Значение i равно: " + i);

                    // Создаем объект класса StringReader.
      StringReader strRdr = new StringReader (strWtr.ToString());

      string str;
      do
      {
          str = strRdr.ReadLine();          // Считываем данные из StringReader-
объекта.
          Console.WriteLine(str);
      }  while (str != null);
   }
}
```

2. Двоичные потоки

Для выполнения операций ввода/вывода данных различных типов без их преобразований предназначены классы **BinaryReader** и **BinaryWriter**. Важно понимать, что эти данные считываются и записываются с использованием внутреннего двоичного формата, а не в текстовой форме, понятной человеку.

Классы BinaryReader и BinaryWriter представляют собой оболочку для байтового потока, которая управляет чтением\записью двоичных данных.

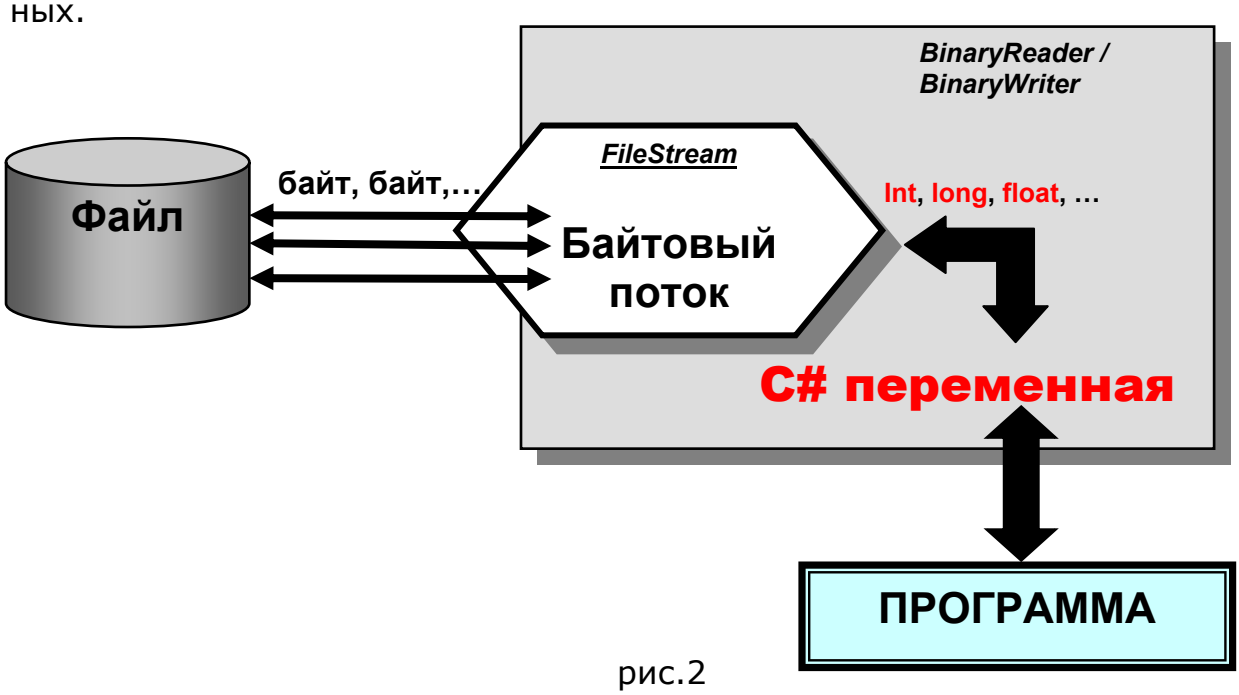


рис.2

Наиболее употребительный *конструктор* класса *BinaryWriter* имеет вид:

BinaryWriter (Stream stream [, Encoding])

Наиболее употребительный конструктор класса BinaryReader имеет вид:

BinaryReader (Stream stream [, Encoding])

stream - поток, в который будут записываться/ считываться данные.

Чтобы записать двоичные данные в файл или прочитать из файла, можно использовать для этого параметра объект, созданный классом FileStream.

Методы вывода данных класса BinaryWriter

Метод	Описание
void Write (sbyte val)	Записывает byte-значение (со знаком)
void Write (byte val)	Записывает byte-значение (без знака)
void Write (byte [ ] buf)	Записывает массив byte-значений
void Write (short val)	Записывает целочисленное значение типа short короткое целое)
void Write (ushort val)	Записывает целочисленное ushort - значение (короткое целое без знака)
void Write (int val)	Записывает целочисленное значение типа int
void Write (uint val)	Записывает целочисленное uint-значение (целое без знака)
void Write (long val)	Записывает целочисленное значение типа long (длинное целое)
void Write (ulong val)	Записывает целочисленное ulong-значение (длинное целое без знака)
void Write (float val)	Записывает float-значение
void Write (double val)	Записывает double-значение
void Write (char val)	Записывает символ
void Write (char [ ] buf)	Записывает массив символов
void Write (string val)	Записывает string-значение с использованием его внутреннего представления, которое включает спецификатор длины

Методы ввода данных, определенные в классе BinaryReader

Метод	Описание
bool ReadBoolean()	Считывает bool-значение
byte ReadByte()	Считывает byte-значение
sbyte ReadSByte()	Считывает sbyte-значение
byte[ ] ReadBytes(int num)	Считывает num байтов и возвращает их в виде массива
char ReadChar()	Считывает char-значение
char[ ] ReadChars(int num)	Считывает num символов и возвращает их в виде массива
double ReadDouble()	Считывает double-значение

float ReadSingle()	Считывает float-значение
short ReadInt16()	Считывает short-значение
int ReadInt32()	Считывает int-значение
long ReadInt64()	Считывает long-значение
ushort ReadUInt16()	Считывает ushort -значение
uint ReadUInt32()	Считывает uint-значение
ulong ReadUInt64()	Считывает ulong-значение
string ReadString()	Считывает string-значение, представленное во внутреннем двоичном формате, который включает спецификатор длины. Этот метод следует использовать для считывания строки, которая была записана с помощью объекта класса BinaryWriter
int Read ()	Возвращает целочисленное представление следующего доступного символа из вызывающего входного потока. При обнаружении конца файла возвращает значение -1
int Read (byte [ ] buf, int offset, int num)	Делает попытку прочитать num байтов в массив buf, начиная с элемента buf[offset], и возвращает количество успешно считанных байтов
int Read (char [ ] buf, int offset, int num)	Делает попытку прочитать num символов в массив buf, начиная с элемента buf[offset], и возвращает количество успешно считанных символов

Исключения, генерируемые классами BinaryReader и BinaryWriter:

ArgumentException	поток stream не открыт для ввода/вывода
ArgumentNullException	поток stream имеет null-значение
IOException	ошибка ввода/вывода
EndOfStreamException	Обнаружен конец потока (для Read...)

Пример. (Без обработки исключений.)

```
// Запись в файл двоичных данных с последующим их считыванием.
using System;
using System.IO;

class RWData
{
   public static void Main()
   {
      BinaryWriter          dataOut;
      BinaryReader dataIn;
      FileStream    fs;

      int             i = 10;
      double          d = 1023.56;
```

```
      bool  b = true;

      fs = new FileStream ("testdata", FileMode.Create);
      dataOut = new BinaryWriter (fs);

      dataOut.Write (i);
      dataOut.Write (d);
      dataOut.Write (b);
      dataOut.Write (12.2 * 7.4);

      dataOut.Close();

      // Теперь попробуем прочитать эти данные.
      fs = new FileStream ("testdata", FileMode.Open);
      dataIn = new BinaryReader(fs);

      i  = dataIn.ReadInt32();
      d = dataIn.ReadDouble();
      b = dataIn.ReadBoolean();
      d = dataIn.ReadDouble();

      dataIn.Close();

      Console.WriteLine ("Прочитали " + i);
      Console.WriteLine ("Прочитали " + d);
      Console.WriteLine ("Прочитали " + b);
      Console.WriteLine ("Прочитали " + d);
   }
}
```

Пример. С обработкой исключений.

```
// Запись в файл двоичных данных с последующим их считыванием.
using System;
using System.IO;

class RWData
{
   public static void Main()
   {
      BinaryWriter dataOut;
      BinaryReader dataIn;
      FileStream fs;

      int i = 10;
      double d = 1023.56;
      bool b = true;

      try
      {
         fs = new FileStream("testdata", FileMode.Create);
         dataOut = new BinaryWriter(fs);
      }
      catch (IOException exc)
      {
```

```
            Console.WriteLine(exc.Message + "\nНе удается открыть файл.");
            return;
        }

        try
        {
            dataOut.Write(i);
            dataOut.Write(d);
            dataOut.Write(b);
            dataOut.Write(12.2 * 7.4);
        }
        catch (IOException exc)
        {
            Console.WriteLine(exc.Message +
            "\nОшибка при записи.");
        }

        dataOut.Close();

        // Теперь попробуем прочитать эти данные.
        try
        {
            fs = new FileStream("testdata", FileMode.Open);
            dataIn = new BinaryReader(fs);
        }
        catch (FileNotFoundException exc)
        {
            Console.WriteLine(exc.Message + "\nНе удается открыть файл.");
            return;
        }

        try
        {
            i = dataIn.ReadInt32();
            d = dataIn.ReadDouble();
            b = dataIn.ReadBoolean();
            d = dataIn.ReadDouble();
        }
        catch (IOException exc)
        {
            Console.WriteLine(exc.Message + "Ошибка при считывании.");
        }
        dataIn.Close();
        Console.WriteLine("Прочитали " + i);
        Console.WriteLine("Прочитали " + d);
        Console.WriteLine("Прочитали " + b);
        Console.WriteLine("Прочитали " + d);
    }
}
```

3. Непотоковый файловый ввод-вывод (2005)

Предоставляется *статическим* классом **File**. Этот класс не является наследником потоковых классов.

public static class File

Методы ввода/вывода:

S	ReadAllBytes	Opens a binary file, reads the contents of the file into a byte array, and then closes the file.
S	ReadAllLines (String)	Opens a text file, reads all lines of the file, and then closes the file.
S	ReadAllLines (String, Encoding)	Opens a file, reads all lines of the file with the specified encoding, and then closes the file.
S	ReadAllText (String)	Opens a text file, reads all lines of the file, and then closes the file.
S	ReadAllText (String, Encoding)	Opens a file, reads all lines of the file with the specified encoding, and then closes the file.
S	WriteAllBytes	Creates a new file, writes the specified byte array to the file, and then closes the file. If the target file already exists, it is overwritten.
S	WriteAllLines (String, String[ ])	Creates a new file, write the specified string array to the file, and then closes the file. If the target file already exists, it is overwritten.
S	WriteAllLines (String, String[], Encoding)	Creates a new file, writes the specified string array to the file using the specified encoding, and then closes the file. If the target file already exists, it is overwritten.
S	WriteAllText (String, String)	Creates a new file, writes the specified string to the file, and then closes the file. If the target file already exists, it is overwritten.
S	WriteAllText (String, String, Encoding)	Creates a new file, writes the specified string to the file using the specified encoding, and then closes the file. If the target file already exists, it is overwritten.

Другие методы:

S	Exists	Determines whether the specified file exists.
S	AppendAllText	Overloaded. Appends the specified stringto the file, creating the file if it does not already exist.

Форматы методов:

У всех перечисленных ниже методов параметр path определяет путь к файлу

```
public static  byte[ ]      ReadAllBytes ( string path )
public static  string[ ]    ReadAllLines ( string path )
public static  string       ReadAllText  ( string path )

public static  void         WriteAllBytes ( string path, byte[ ] bytes )
public static  void         WriteAllLines ( string path, string[ ] contents )
public static  void         WriteAllText  ( string path, string contents )
```

Методы открывают файл, выполняют чтение/запись всего файла, а затем закрывают файл. И это все – одним вызовом.

Методы **Write**… перезаписывают старые файлы и создают их, если они не существовали.

Для методов **Read**… файлы должны существовать.

Метод **WriteAllLines** дописывает символы конца строки. При записи другими методами необходимо добавить символы конца строки вручную: *"\r\n"* или *Environment.NewLine*.

Метод **ReadAllLines** разделяет строки по символам их конца.

Методы **ReadAllLines** и **ReadAllBytes** автоматически устанавливают размерность массива по фактическому количеству считанных элементов (строк, байтов).

Методы **ReadAllText** и **WriteAllText** работают с содержимым файла как с одной строкой, не реагируя на символы конца строк.

Таблица исключений для метода ReadAllLines:

Exception type	Condition
ArgumentException	path is a zero-length string, contains only white space, or contains one or more invalid characters as defined by InvalidPathChars.
ArgumentNullException	path is a null reference.
PathTooLongException	The specified path, file name, or both exceed the system-defined maximum length. For example, on Windows-based platforms, paths must be less than 248 characters, and file names must be less than 260 characters.
DirectoryNotFoundException	The specified path is invalid (for example, it is on an unmapped drive).
IOException	An I/O error occurred while opening the file.

UnauthorizedAccessException	path specified a file that is read-only. -or- This operation is not supported on the current platform. -or- path specified a directory. -or- The caller does not have the required permission.
FileNotFoundException	The file specified in path was not found.
NotSupportedException	path is in an invalid format.
SecurityException	The caller does not have the required permission.

Пример использования методов WriteAllLines, ReadAllLines, AppendAll-Text и Exists:

```
using System;
using System.IO;
class Test
{
   public static void Main()
   {
      string path = @"c:\temp\MyTest.txt";

      // Текст добавляется в файл только один раз
      if (  ! File.Exists (path) )
      {
         // Создать файл и записать строки. Закрыть файл.
         string[ ] createText = { "Hello", "And", "Welcome" };
         File.WriteAllLines (path, createText);
      }

      // Этот текст будет добавляться при каждом запуске программы
      string appendText = "This is extra text" + Environment.NewLine;
      File.AppendAllText (path, appendText);

      // Открыть файл, прочитать и закрыть.
      string[] readText = File.ReadAllLines (path);

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

```
Hello
And
Welcome
This is extra text
This is extra text
```

This is extra text

Пример использования методов WriteAllText и ReadAllText.

```
using System;
using System.IO;
using System.Text;

class Test
{
   public static void Main()
   {
      string path = @"c:\temp\MyTest.txt";

      if (!File.Exists(path))
      {
         // Создать файл и записать текст. Закрыть файл.
         string createText = "Hello and Welcome" + Environment.NewLine;
        File.WriteAllText (path, createText);
      }

      string appendText = "This is extra text" + Environment.NewLine;
      File.AppendAllText(path, appendText);

      // Открыть файл, прочитать и закрыть.
      string readText = File.ReadAllText(path);
      Console.WriteLine(readText);
   }
}
```

Hello and Welcome
This is extra text
This is extra text

4. Файлы с последовательным и прямым доступом

Файлы с последовательным доступом:

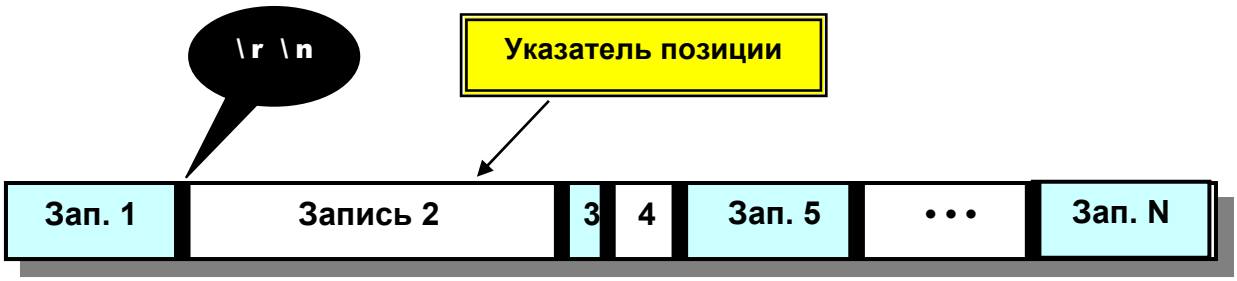


рис.3

Адр(запись i) = ???

Файлы с прямым доступом:

Способ 1.

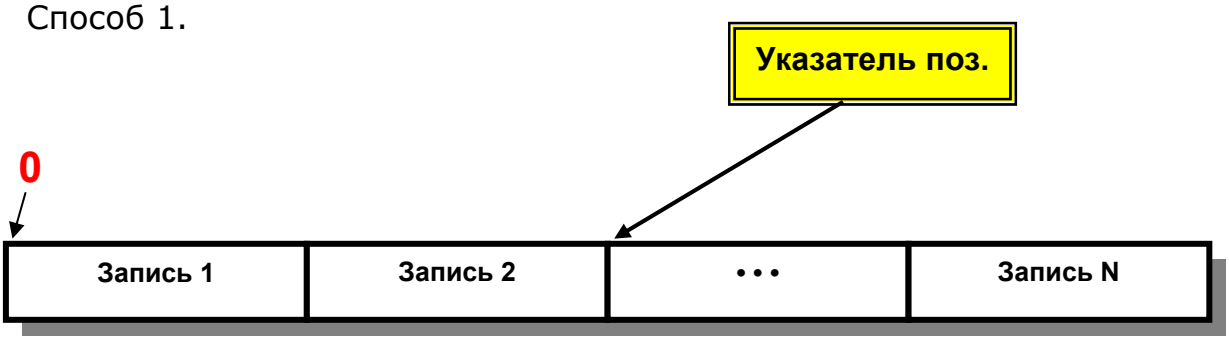


рис.4

Length(Запись 1) = Length(Запись 2) = Length(Запись N)=const

Адр(Запись i) = Length(Запись) * (i-1) байт

Способ 2. Файл адресов записей: Адр(запись i) = Адр[i-1]

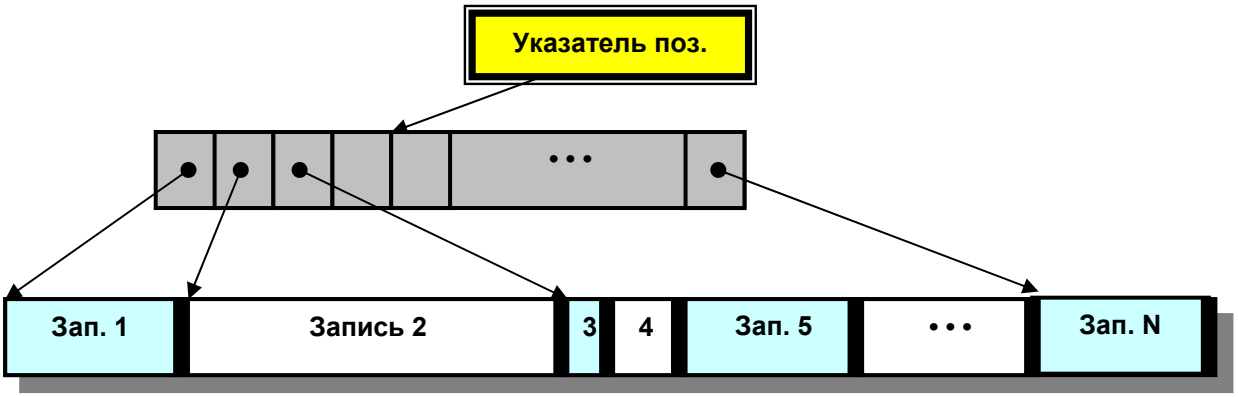


рис.5

Для доступа к требуемой записи необходимо использовать метод **Seek()**, определенный в классе FileStream. Этот метод позволяет установить указатель позиции (файловый указатель) в любое место файла.

Формат:

```
long   Seek (long newPos,  SeekOrigin origin)
```

newPos - новая позиция файлового указателя, выраженная в байтах, относительно позиции, заданной элементом *origin*. Перемещаться можно как вперед (+), так и назад (-).

Элемент *origin* может принимать одно из значений, определенных перечислением *SeekOrigin*:

Значение	Описание
SeekOrigin.Begin	Поиск от начала файла
SeekOrigin.Current	Поиск от текущей позиции
SeekOrigin.End	Поиск от конца файла

После обращению к методу Seek() следующая операция чтения или записи данных будет выполняться на новой позиции в файле.

Исключение	Описание
IOException	Ошибка ввода-вывода
NotSuppotedException	Базовый поток не поддерживает эту функцию

```
f = new FileStream ("random.dat", FileMode.Create);

f.Seek(0, SeekOrigin.Begin);      // Переход в начало файла.
f.Seek(4, SeekOrigin.Begin);      // Поиск пятого байта (№ б. = 4).

f.Seek(-10, SeekOrigin.End);      // Поиск одиннадцатого байта от конца.
// Демонстрация произвольного доступа к файлу.
// Записать в файл алфавит прописными буквами,
// а затем выборочно считать его.
using System;
using System.IO;
class RandomAccessDemo
{
   public static void Main()
   {
      FileStream f;
      char ch;
      try
      {
         f = new FileStream ("random.dat", FileMode.Create);
      }
      catch (IOException exc)
      {
         Console.WriteLine (exc.Message);
         return;
      }
      // Записываем в файл алфавит.
      for ( int i = 0;  i < 26;  i++)
      {
         try
         {
            f.WriteByte ((byte)('A' + i));
         }
         catch (IOException exc)
         {
            Console.WriteLine (exc.Message);
            return;
         }
      }
      try
      {
         // Теперь считываем отдельные значения.
         f.Seek (0, SeekOrigin.Begin); // Поиск первого байта.
         ch = (char)f.ReadByte();
         Console.WriteLine("Первое значение равно " + ch);

         f.Seek (1, SeekOrigin.Begin); // Поиск второго байта.
```

```
            ch = (char)f.ReadByte();
            Console.WriteLine("Второе значение равно " + ch);
            f.Seek (4, SeekOrigin.Begin); // Поиск пятого байта.
            ch = (char)f.ReadByte();
            Console.WriteLine ("Пятое значение равно " + ch);
            Console.WriteLine ();

            // Теперь считываем значения через одно.
            Console.WriteLine ("Выборка значений через одно: ");
            for (int i = 0; i < 26; i += 2)
            {
               f.Seek (i, SeekOrigin.Begin);          // Переход к i-му байту.
               ch = (char)f.ReadByte();
               Console.Write (ch + " ");
            }
         }
         catch (IOException exc)
         {
            Console.WriteLine (exc.Message);
         }
         Console.WriteLine();
         f.Close();
      }
   }
```

При выполнении этой программы получены такие результаты:
Первое значение равно A
Второе значение равно B
Пятое значение равно E
Выборка значений через одно: ACEGIKMOQSUWY

Проблемы.

Строки имеют переменную длину (по фактическому содержимому). Решением проблемы м.б.: 1) добаление к строкам пробелов до одинаковой длины методом PadRicht(); 2) преобразование строки в массив символов методом ToCharArray().

В кодировке, используемой по умолчанию (UTF8), русские буквы кодируются двумя байтами, а английские – одним. Решение проблемы: указывать кодировку явно, например, System.Text.Encoding.Unicode.

Кодировка текстовых потоков

В C# используются следующие статические свойства класса System.Text.Encoding для кодировки текстовых потоков

Кодировка	Описание
ASCII	Кодировка ASCII без символов кириллицы, в которой для представления текстовых символов используются младшие 7 бит байта
Unicode	Кодировка UNICODE. Для представления символов используется 16 бит (т. е. 2 байта). Сигнатура = FF FE в

	нач.файла для StreamWriter
UTF7	Кодировка UCS Transformation Format. Применяется для представления символов UNICODE. В ней используются младшие 7 бит данных
UTF8	То же, но для представления символов UNICODE в ней используется 8 бит данных
Default	Системная кодировка ANSI (не путайте ее с кодировкой ASCII). В этой кодировке для представления каждого символа используется 8 бит данных

Свойства класса кодировки System.Text.Encoding:
ASCII – 1 байт (старший бит = 0);
Default – по умолчанию (UTF8);
Unicode – 2 байта;
UTF32 – 4 байта;
UTF7 – 1 байт, старший бит не используется;
UTF8 – 1 байт (по умолчанию в ,NET).

Примеры произвольного доступа к записям.
Задача 1.

/*
Разработать класс «Студент». Каждый студент определяется полями:
Фамилия, имя (тип string);
Год рождения (тип int);
Средний балл за предыдущий год обучения или за вступ. экзамен (тип float).

Создать программу учета студентов группы (в виде массива). Информацию о студентах ввести с клавиатуры. Признаком конца списка является ввод пробела в качестве фамилии студента.

Вывести на экран информацию обо всех студентах.

Информацию обо всех студентах сохранить в файле C:\Temp\GroupDirect.bin с произвольным доступом. Обеспечить постоянную длину всех записей. */

```
using System;
using System.IO;
using System.Collections;

class Student
{
   public string    fio;
   public uint      yar;
   public float     ball;

   public Student(string f, uint y, float b)
   {
      fio = f;
      yar = y;
      ball = b;
   }
```

```
   public void Show()
   {
      Console.WriteLine("Студент {0}, год рождения {1}, ср.балл = {2}",
                  fio, yar, ball);
   }
}

class Example1
{
   public static void Main()
   {
      string         fio = " ";
      uint           god = 0, n = 1;
      float          ball = 0;
      Student        std;
      BinaryWriter dataOut;
      FileStream     fs;


      ArrayList tableStd = new ArrayList();

      while (true)
      {
        Console.WriteLine(
         "\n_____Студент {0}_____",
                                                    n);
         Console.Write(
           "Введите фамилию (не > 15 символов, пробел - конец списка): ");
         fio = Console.ReadLine();
         if (fio == " " || fio.Length == 0)
            break;
         try
         {
            Console.Write("Введите год рождения: ");
            god = uint.Parse(Console.ReadLine());

            Console.Write("Введите средний балл: ");
            ball = float.Parse(Console.ReadLine());
         }
         catch
         {
        Console.WriteLine("==ОШИБКА== Неверный формат ввода. Повторите!");
           continue;
         }

         std = new Student(fio, god, ball);
         tableStd.Add(std);
         n++;
      }

      // Таблица создана. выведем ее на экран и сохраним в файле
      Console.WriteLine("\n");

      try
```

```
        {
          fs = new FileStream(@"c:\Temp\GroupDirect.bin", FileMode.Create,
                                                                    Fil-
eAccess.Write);
          dataOut = new BinaryWriter(fs, System.Text.Encoding.Unicode);


          for (int i = 0; i < tableStd.Count; i++)
          {
            std = (Student)tableStd[i];
            std.Show();

            dataOut.Write("Студент");
            fio = std.fio.PadRight(15);
            dataOut.Write(fio);
            dataOut.Write(std.yar);
            dataOut.Write(std.ball);
          }
        }

        catch (Exception e)
        {
          Console.WriteLine("Error: " + e.Message);
          return;
        }
        fs.Close();
      }
    }
```

Задача 2.

```
/*
Вывести на экран информацию из файла C:\Temp\GroupDirect.bin
о студентах, заданных порядковым номером.
Номера студентов ввести с клавиатуры.
Первый студент имеет номер 1.
Признаком конца списка является ввод 0 в качестве номера студента.
*/

using System;
using System.IO;
using System.Collections;
using System.Text;

class Example
{
  public static void Main()
  {
    string  fio;
    uint    god;
    float   ball;

    uint    N;      // порядковый номер студента в файле

    string str;
```

```
            string path = @"c:\Temp\GroupDirect.bin";
            uint   kRec;                        // количество записей
            const uint lRec = (7*2+1)+(15*2+1)+4+4;   // длина записи
                                                                                    //                    (Сту-
дент,fio,god,ball)

           if (!File.Exists(path))
           {
              Console.WriteLine("Файл отсутствует.");
              return;
           }

           FileStream fs = new FileStream(path, FileMode.Open, FileAccess.Read);
           BinaryReader dataIn = new BinaryReader(fs, Encoding.Unicode);

           kRec = (uint)fs.Length / lRec;  //кол.записей = длина файла /

          длина записи
           Console.WriteLine("В файле студентов: {0}", kRec);

           while (true)
           {

               Console.WriteLine("\n_____");
               Console.Write("Введите номер студента (0 - конец поиска): ");
              try
              {
                  N = uint.Parse(Console.ReadLine());
              }
              catch
              {
                  Console.WriteLine("==ОШИБКА== Повторите ввод целого числа.");
                  continue;
              }
              if (N == 0) break;

              if (N > kRec)
              {
                Console.WriteLine("\n==ОШИБКА== Студента с таким номером нет.");
                Console.WriteLine("       Всего в файле студентов: {0}", kRec);
                continue;
              }

             // Позиционируем указатель записи
             fs.Seek((N - 1) * lRec, SeekOrigin.Begin);
             try
             {
              str = dataIn.ReadString();
              fio = dataIn.ReadString();
              god = dataIn.ReadUInt32();
              ball = dataIn.ReadSingle();

              Console.WriteLine("Студент {0}, год рождения {1}, ср.балл = {2}\n",
                                fio,          god,        ball);
```

```
        }
        catch (Exception e)
        {
            Console.WriteLine("Error: " + e.Message);
            return;
        }
    }
    fs.Close();
  }
}
```

СЕРИАЛИЗАЦИЯ, ОПЕРАЦИИ С ФАЙЛАМИ И КАТАЛОГАМИ

1. Сохранение и восстановление объектов

Сохранение объектов представляет собой определенную проблему. Связано это с тем, что обычно переменные объекта являются недоступными в программе. Для решения этой проблемы создано пространство имен **System.Runtime.Serialization** с классами, используемыми для *сериализации* и *десериализации* объектов.

Сериализация — это процесс преобразования объекта или графа объектов в линейную последовательность байтов для сохранения с помощью потока.

Десериализация — это процесс изъятия сохраненных сведений и создания из них объектов.

Для того чтобы объекты можно было сериализовать, его класс должен быть помечен атрибутом [Serializable].

По умолчанию CLR не предполагает, что объекты будут сохраняться в каком-либо хранилище (например, в файле на диске).

Для сериализации объекта нужно выполнить следующие шаги:

- Пометить сериализуемый класс атрибутом [Serializable].
- Создать байтовый поток.
- Создать объект класса BinaryFormatter, управляющий сериализацией.
- Вызвать метод Serialize() этого объекта для выполнения сериализации.

Для десериализации объектов нужно вместо метода Serialize() выполнить метод Deserialize() и привести воостанавливаемые объекты к требуемому типу.

```
// Сохранение объектов
using System;
using System.IO;
using System.Runtime.Serialization;
using System.Runtime.Serialization.Formatters.Binary;

[Serializable]
class Customer
{
   private int    CustomerNumber;          // N покупателя
   private string CustomerName;        // Имя покупателя
   private string CustomerCountry;             // Страна

   public Customer (int Number, string Name, string Country)
   {
      this.CustomerNumber = Number;
      this.CustomerName   = Name;
      this.CustomerCountry = Country;
   }
```

```
    public void WriteCustomer()
    {
      Console.WriteLine("Customer Number: " + CustomerNumber);
      Console.WriteLine("Customer Name: "    + CustomerName);
      Console.WriteLine("Customer Country: " + CustomerCountry);
    }
}
class SaveObj
{
    public static void Main()
    {
      FileStream  fs;

      Customer Customer1 = new Customer (1, "Ален Дилон", "Франция");
      Customer Customer2 = new Customer (2, "Иванов", "Россия");

      Customer1.WriteCustomer();
      Customer2.WriteCustomer();

      // Создание потока
      fs = new FileStream ("c:\\Temp\\Customer.dat", FileMode.Create);

      // Воспользуемся поддержкой двоичного форматирования в CLR
      BinaryFormatter sf = new BinaryFormatter();

      // Сохраним объект в файле в двоичном виде
      sf.Serialize (fs, Customer1);
      sf.Serialize (fs, Customer2);
      fs.Close();

      // Восстановим из файла сериализованный объект
      fs = new FileStream ("c:\\Temp\\Customer.dat", FileMode.Open);

      Customer NewCustomer1 = (Customer)sf.Deserialize (fs);
      NewCustomer1.WriteCustomer();

      Customer NewCustomer2 = (Customer)sf.Deserialize (fs);
      NewCustomer2.WriteCustomer();
    }
}
```

Если объект должен использоваться в другом приложении, лучшим выбором может оказаться не двоичный формат, а формат протокола SOAP (в виде XML-файла). SOAP – простой протокол доступа к объектам.

Для реализации этой задачи нужно:

- Вместо пространства имен System.Runtime.Serialization.Formatters.Binary использовать System.Runtime.Serialization.Formatters.Soap.
- Вместо класса BinaryFormatter использовать класс SoapFormatter.
- Файл создавать с расширением xml (необязательно).

```
// Сохранение объектов при помощи протокола SOAP

using System;
using System.IO;
using System.Runtime.Serialization;
                // using System.Runtime.Serialization.Formatters.Binary;
using System.Runtime.Serialization.Formatters.Soap;

[Serializable]
class Customer
{
   private int         CustomerNumber;          // N покупателя
   private string CustomerName;          // Имя покупателя
   private string CustomerCountry;          // Страна

   public Customer (int Number, string Name, string Country)
   {
      this.CustomerNumber = Number;
      this.CustomerName    = Name;
      this.CustomerCountry = Country;
   }

   public void WriteCustomer()
   {
      Console.WriteLine("Customer Number: " + CustomerNumber);
      Console.WriteLine("Customer Name: "    + CustomerName);
      Console.WriteLine("Customer Country: " + CustomerCountry);
   }
}

class SaveObj
{
   public static void Main()
   {
      FileStream fs;

      Customer Customer1 = new Customer(1, "Ален Дилон", "Франция");
      Customer Customer2 = new Customer(2, "Иванов", "Россия");

      Customer1.WriteCustomer();
      Customer2.WriteCustomer();

      // Создание потока
      // fs = new FileStream ("c:\\Temp\\Customer.dat", FileMode.Create);
      fs = new FileStream ("c:\\Temp\\Customer.xml", FileMode.Create);

      // Воспользуемся поддержкой SOAP форматирования
      // BinaryFormatter sf = new BinaryFormatter();
      SoapFormatter sf = new SoapFormatter();

      // Сохраним объект в файле в двоичном виде
      sf.Serialize (fs, Customer1);
      sf.Serialize (fs, Customer2);
      fs.Close();
```

```
        // Восстановим из файла сериализованный объект
        fs = new FileStream ("c:\\Temp\\Customer.xml", FileMode.Open);

        Customer NewCustomer1 = (Customer)sf.Deserialize (fs);
        NewCustomer1.WriteCustomer();

        Customer NewCustomer2 = (Customer)sf.Deserialize (fs);
        NewCustomer2.WriteCustomer();
    }
}
```

Классы **SoapFormatter** и **BinaryFormatter** предоставляют возможности для сериализации не только отдельных объектов, но и более сложных структур, например хеш-таблиц.

Такие структуры должны реализовывать **интерфейс ISerializable**. Реализация интерфейса ISerializable необходима классу в том случае, если он хочет управлять обработкой своей сериализации. Все коллекции FCL реализуют указанный интерфейс.

Пример сериализации хеш-таблицы.

```
using System;
using System.IO;
using System.Collections;
using System.Runtime.Serialization;
using System.Runtime.Serialization.Formatters.Binary;

public class App
{
   [STAThread]
   static void Main()
   {
      Serialize();
      Deserialize();
   }

   static void Serialize()
   {
    FileStream fs;

      // Создать хеш-таблицу с ключем и полем данных.
      Hashtable addresses = new Hashtable();

      addresses.Add ("Сергей",   "Москва, ул Пирогова, д.17, кв.25");
      addresses.Add ("Юлия",      "Новгород, ул.Космонавтов,  д. 24");
      addresses.Add ("Николай", "Тула, пр-т Ленина, д.134, кв. 76");

      fs = new FileStream ("C:\\Temp\\DataFile.dat", FileMode.Create);

      BinaryFormatter bf = new BinaryFormatter();
      try
      {
         bf.Serialize (fs, addresses);
```

```
        }
        catch (SerializationException e)  //  в пространстве System.Runtime.Serialization
        {
            Console.WriteLine ("Ошибка сериализации: " + e.Message);
            throw;
        }
        finally
        {
            fs.Close();
        }
    }

    static void Deserialize()
    {
     FileStream fs;
        Hashtable addresses = null;

        // Open the file containing the data that you want to deserialize.
        fs = new FileStream ("C:\\Temp\\DataFile.dat", FileMode. Open);

        try
        {
            BinaryFormatter bf = new BinaryFormatter();

            addresses = (Hashtable)bf.Deserialize(fs);
        }
        catch (SerializationException e)
        {
            Console.WriteLine("Ошибка десериализации: " + e.Message);
            throw;
        }
        finally
        {
            fs.Close();
        }

        foreach (DictionaryEntry de in addresses)
        {
            Console.WriteLine("{0} живет в городе {1}.", de.Key, de.Value);
        }
    }
}
```

Другой способ сохранения объектов (без сериализации).

С помощью открытых методов Get…() получать значения полей объекта, записывать их в файл. При восстановлении объекта считывать поля из файла и подавать их значения в конструктор. Для полей объекта, которые не инициализируются конструктором, использовать методы Set…().

Сериализация/десериализация графа объектов

Если сериализуемый объект содержит ссылки на другие сериализуемые объекты, которые в свою очередь так же содержат ссылки на сериализуемые объекты, то такая структура образует граф объектов.

При сериализации объекта, находящегося в вершине графа, будет сериализован весь граф.

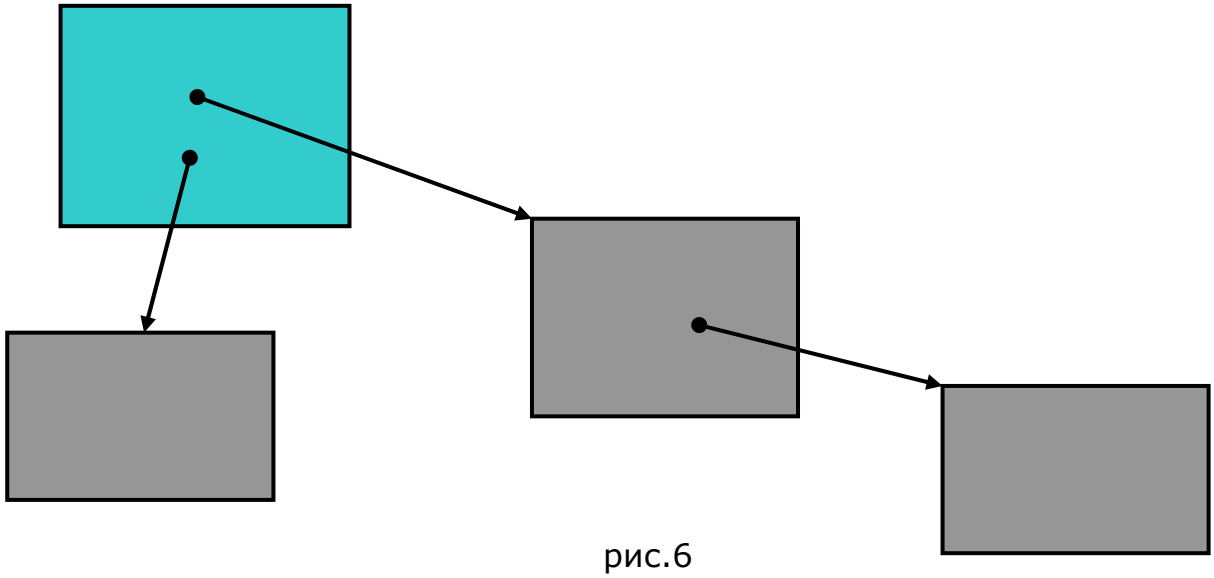


рис.6

Произвольный доступ к сериализованным объектам.

Осуществляется так же, как и в случае хранения двоичных данных в файле.

Способ 1

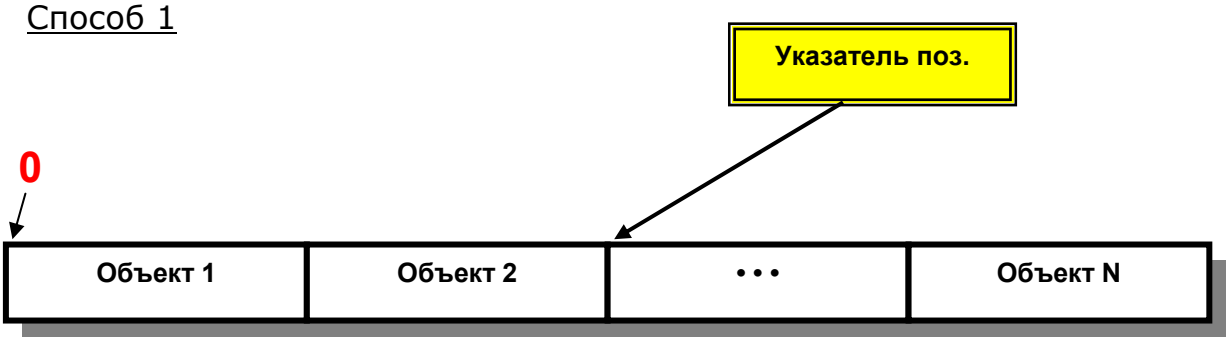


рис.7

Length(Объект) = const

Способ 2

Использование файла указателей (см. лекцию 3).

Пример произвольного доступа к объектам

```
/*    У заданных студентов из файла C:\Temp\GroupSerial.bin заменить средние баллы.
Фамилии студентов и их новые средние баллы ввести с клавиатуры.
Признаком конца списка является ввод Enter в качестве фамилии студента.
Вывести файл на экран
*/

using System;
using System.IO;
using System.Collections;
using System.Runtime.Serialization;
using System.Runtime.Serialization.Formatters.Binary;
```

```
[Serializable]
class Student
{
   public string fio;
   public uint        yar;
   public float  ball;

   public Student(string f, uint y, float b)
   {
      fio  = f;
      yar  = y;
      ball = b;
   }
   public void Show()
   {
    Console.WriteLine("Студент {0}, год рождения {1}, ср.балл = {2}",
                      fio,          yar,         ball);
   }
}

class Example1
{
   public static void Main()
   {
      string          fio = " ";
      Student        std;
      float  newBall = 0;
      uint   k;
      long  pos;


      string path = @"c:\Temp\GroupSerial.bin";
      FileStream fs;
      BinaryFormatter frm;

      ArrayList tableStd = new ArrayList();

      frm = new BinaryFormatter();

      // Выборочная модификация объектов
      fs = new FileStream(path, FileMode.Open);

      while (true)
      {
         Console.WriteLine("_____" +
                    "_____");
         Console.Write(
              "Введите фамилию (пробел - конец поиска): ");
         fio = Console.ReadLine();

         if (fio == " " || fio.Length == 0)
            break;
```

```
            try
            {
               Console.Write("Введите новый средний балл : ");
               newBall = float.Parse(Console.ReadLine());
            }
            catch
            {
              Console.WriteLine("ОШИБКА. Повторите ввод сначала");
               continue;
            }

            try
            {
               k = 0;
               while (fs.Position < fs.Length)
               {
                  // Запомнить позицию читаемой записи
                  pos = fs.Seek(0, SeekOrigin.Current);

                  std = (Student)frm.Deserialize(fs);

                  if (std.fio == fio)
                  {
                     Console.WriteLine("---Найден---");
                     std.ball = newBall;
                     fs.Seek(pos, SeekOrigin.Begin);
                     frm.Serialize(fs, std);
                     k = 1;
                  }
               }
               if (k == 0)
                  Console.WriteLine("Студент не найден");
               fs.Seek(0, SeekOrigin.Begin);
            }

            catch (Exception e)
            {
               Console.WriteLine("Error: " + e.Message);
               return;
            }
         }
         fs.Close();
      }
   }
```

2. OpenFileDialog и SaveFileDialog для SDI-приложений

Добавляем на форму элементы **TextBox** и **MainMenu**. В MainMenu будет всего три пункта — File, Open и Save .В TextBox устанавливаем свойство Multiple в true.

Добавляем на форму элемент управления OpenFileDialog из окна панели инструментов ToolBox. Подобно элементу MainMenu, он будет располагаться на панели невидимых компонент:

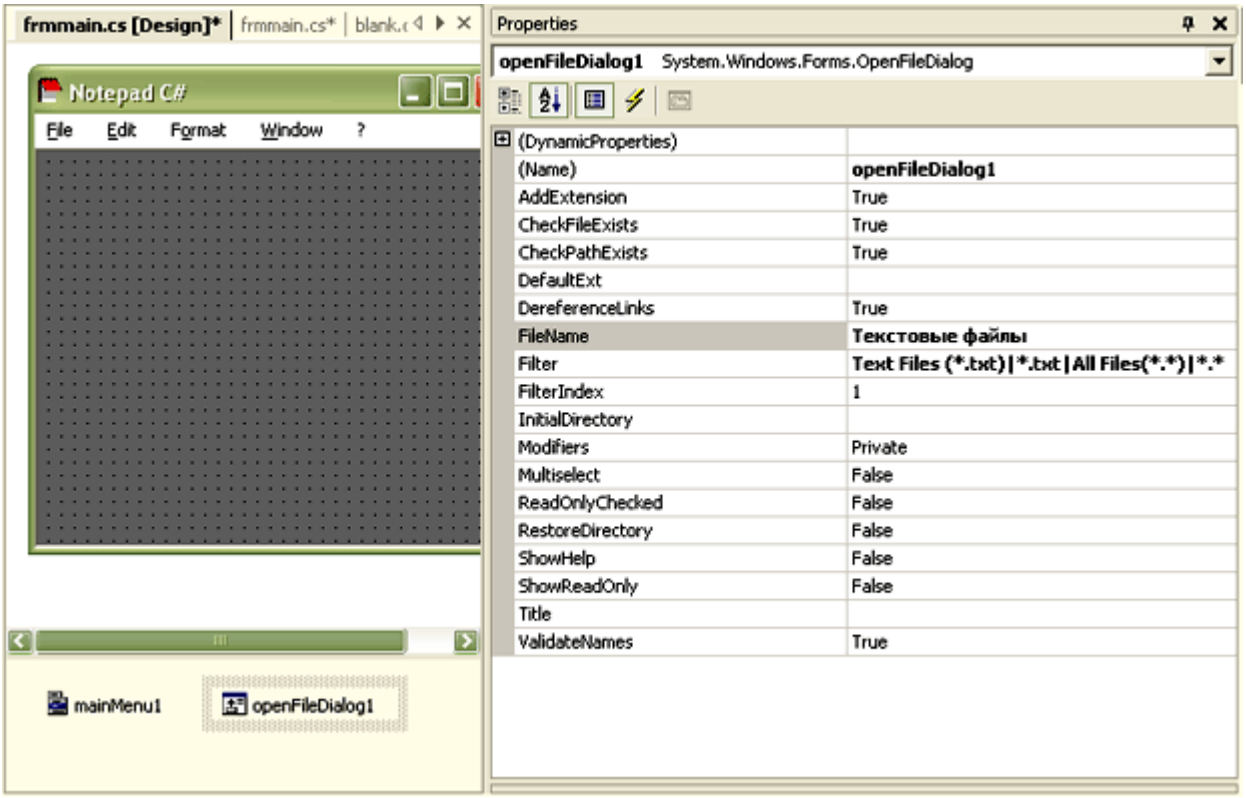


рис.8

Свойство **FileName** задает название файла, которое будет находиться в поле "Имя файла:" при появлении диалога. На рис.8 название в этом поле — "Текстовые файлы".

Свойство **Filter** задает ограничение файлов, которые могут быть выбраны для открытия — в окне будут показываться только файлы с заданным расширением. Через вертикальную разделительную линию можно задать смену типа расширения, отображаемого в выпадающем списке "Тип файлов". Здесь введено Text Files (*.txt)|*.txt|All Files(*.*)|*.* что означает обзор либо текстовых файлов, либо всех.

Свойство **InitialDirectory** позволяет задать директорию, откуда будет начинаться обзор. Если это свойство не установлено, исходной директорией будет рабочий стол.

Из окна **ToolBox** перетаскиваем так же и элемент SaveFileDialog — свойства этого элемента в точности такие же, как и у диалога OpenFileDialog.

Переходим в код формы. Подключаем пространство имен для работы с файловыми потоками:

```
using System.IO;
```

Добавляем обработчик для пункта меню Open:

```
private void menuOpen_Click(object sender, System.EventArgs e)
  {
    //Показываем диалог выбора файла
    openFileDialog1.ShowDialog();
```

```
    // Переменной fileName присваиваем имя открываемого файла
    string  fileName = openFileDialog1.FileName;

    //Создаем поток fs и открываем файл для чтения.
    FileStream filestream = File.Open(fileName, FileMode.Open, FileAccess.Read);

    //Проверяем,  открыт ли поток,  и если открыт, выполняем условие
    if ( filestream != null )
    {
      //Создаем объект streamreader и связываем его с потоком filestream
      StreamReader streamreader = new StreamReader(filestream);

      //Считываем весь файл и записываем его в TextBox
      txtBox.Text = streamreader.ReadToEnd();

      //Закрываем поток.
      filestream.Close();
    }
  }
```

Добавляем обработчик для пункта меню Save:

```
private void menuSave_Click(object sender, System.EventArgs e)
{
 //Показываем диалог выбора файла
 saveFileDialog1.ShowDialog();

 // В качестве имени сохраняемого файла устанавливаем переменную fileName
 string fileName = saveFileDialog1.FileName;

 //Создаем поток fs и открываем файл для записи.
 FileStream filestream = File.Open(fileName, FileMode.Create, FileAccess.Write);

 //Проверяем,  открыт ли поток,  и если открыт, выполняем условие
 if ( filestream != null )
 {
    //Создаем объект streamwriter и связываем его с потоком filestream
    StreamWriter streamwriter = new StreamWriter(filestream);

    //Записываем данные из TextBox в файл
    streamwriter.Write(txtBox.Text);

    //Переносим данные из потока в файл
    streamwriter.Flush();

    //Закрываем поток
    filestream.Close();
 }
}
```

В консольном приложении:

```
 // Открыть диалоговое окно для выбора файла
OpenFileDialog dlgOpen = new OpenFileDialog();
```

```
dlgOpen.Title = "Найдите требуемый файл";                    // заголовок окна

 // Показать окно
if (dlgOpen.ShowDialog() == DialogResult.OK)
{
  FileStream fs = new FileStream(dlgOpen.FileName,FileMode.Open);
  . . .
```

3. Работа с каталогами и файлами

В пространстве имен System.IO есть четыре класса, предназначенные для работы с физическими файлами и структурой каталогов на диске: **Directory**, **File**, **DirectoryInfo** и **FileInfo**. С их помощью можно выполнять создание, удаление, перемещение файлов и каталогов, а также получение их свойств.

Классы Directory и File реализуют свои функции через статические методы.

DirectoryInfo и FileInfo обладают схожими возможностями, но они реализуются путем создания объектов соответствующих классов.

DirectoryInfo - класс

Предоставляет методы экземпляра класса для создания, перемещения, копирования и перечисления объектов файловой системы в папках и вложенных папках.

```
[Serializable]
public sealed class DirectoryInfo : FileSystemInfo
```

Открытые конструкторы

DirectoryInfo - конструктор	Инициализирует новый экземпляр класса DirectoryInfo для указанного пути.

Открытые свойства

Attributes (унаследовано от FileSystemInfo)	Возвращает или устанавливает FileAttributes для текущего класса FileSystemInfo.
CreationTime (унаследовано от FileSystemInfo)	Возвращает или устанавливает время создания текущего объекта FileSystemInfo.
CreationTimeUtc (унаследовано от FileSystemInfo)	Возвращает или устанавливает время создания текущего объекта FileSystemInfo в формате всеобщего скоординированного времени (UTC).
Exists	Переопределен. Возвращает значение, определяющее, создана ли папка.
Extension (унаследовано от FileSystemInfo)	Возврат строки, содержащей расширение файла.
FullName (унаследовано от FileSystemInfo)	Возвращает полный путь к папке или файлу.

LastAccessTime (унаследовано от FileSystemInfo)	Возвращает или устанавливает время последнего доступа к текущему файлу или папке.
LastAccessTimeUtc (унаследовано от FileSystemInfo)	Возвращает или устанавливает дату и время последнего доступа к заданному файлу или папке в формате всеобщего скоординированного времени (UTC).
LastWriteTime (унаследовано от FileSystemInfo)	Возвращает или устанавливает время последней операции записи в текущий файл или папку.
LastWriteTimeUtc (унаследовано от FileSystemInfo)	Возвращает или устанавливает время последней операции записи в текущий файл или папку в формате всеобщего ско-ординированного времени (UTC).
Name	Переопределен. Возвращает имя данного экземпляра DirectoryInfo.
Parent	Возвращает родительскую папку указанной вложенной папки.
Root	Возвращает корневой элемент пути.

Открытые методы

Create	Создает папку.
CreateObjRef (унаследовано от MarshalByRefObject)	Создает объект, который содержит всю необходимую информацию для создания прокси-сервера, используемого для коммуникации с удаленными объектами.
CreateSubdirectory	Создает вложенную папку или вложенные папки по указанному пути. Указанный путь должен относиться к текущему экземпляру класса DirectoryInfo.
Delete	Переопределен. Удаляет из пути DirectoryInfo и его содержимое. Перегружен.
Equals (унаследовано от Object)	Перегружен. Определяет, равны ли два экземпляра Object.
GetDirectories	Возвращает вложенные папки текущей папки. Перегружен.
GetFiles	Возвращает список файлов текущей папки. Перегружен.
GetFileSystemInfos	Перегружен. Извлекает массив объектов со строгим типом FileSystemInfo.
GetHashCode (унаследовано от Object)	Служит хеш-функцией для конкретного типа, пригоден для использования в алгоритмах хеширования и структурах данных, например в хеш-таблице.

GetLifetimeService (унаследовано от MarshalByRefObject)	Извлекает служебный объект текущего срока действия, который управляет средствами срока действия данного экземпляра.
GetObjectData (унаследовано от FileSystemInfo)	Устанавливает объект SerializationInfo с именем файла и дополнительными сведениями об исключении.
GetType (унаследовано от Object)	Возвращает Type текущего экземпляра.
InitializeLifetimeService (унаследовано от MarshalByRefObject)	Получает служебный объект срока действия, для управления средствами срока действия данного экземпляра.
MoveTo	Перемещает экземпляр DirectoryInfo и его содержимое в другое место.
Refresh (унаследовано от FileSystemInfo)	Обновление состояния объекта.
ToString	Переопределен. Возвращает исходный путь, указанный пользователем.

Пример использования класса DirectoryInfo.

```
// Создаются два каталога, выводится информация о них
// и предпринимается попытка удаления каталога.
using System;
using System.IO;
class Class1
{
      static void DirInfo (DirectoryInfo di)
      {
         // Вывод информации о каталоге
         Console.WriteLine ("===== Информация о папке =====");
         Console.WriteLine ("Полное имя: " + di.FullName);
         Console.WriteLine ("Имя: " + di.Name);
         Console.WriteLine ("Родительский каталог: " + di.Parent);
         Console.WriteLine ("Создан: " + di.CreationTime);
         Console.WriteLine ("Атрибуты: " + di.Attributes);
         Console.WriteLine ("Корневой каталог: " + di.Root);
         Console.WriteLine ("============================");
      }

      static void Main ()
      {
         DirectoryInfo di1 = new DirectoryInfo (@"c:\MyDir");
         DirectoryInfo di2 = new DirectoryInfo (@"c:\MyDir\temp");
         try
         {   // Создать каталоги
            di1.Create ();
            di2.Create ();

            // Вывести информацию о каталогах
```

```
            DirInfo (di1);
            DirInfo (di2);

            // Попытаться удалить каталог
            Console.WriteLine ("Попытка удалить {0}.", di1.Name);
            di1.Delete ();
         }
         catch  (Exception)
         {
            Console.WriteLine ("Попытка не удалась ");
         }
      }
   }
```

Примечание:
di1.Delete (true); - можно удалить и непустой каталог.

FileInfo - класс

Предоставляет методы экземпляра для создания, копирования, удаления, перемещения и открытия файлов, а также помогает при создании объектов FileStream.

```
[Serializable]
public   sealed   class   FileInfo : FileSystemInfo
```

Открытые конструкторы

FileInfo - конструктор	Инициализирует новый экземпляр класса FileInfo, действующего в качестве обертки для пути файла.

Открытые свойства

Attributes (унаследовано от FileSystemInfo)	Возвращает или устанавливает FileAttributes для текущего класса FileSystemInfo.
CreationTime (унаследовано от FileSystemInfo)	Возвращает или устанавливает время создания текущего объекта FileSystemInfo.
CreationTimeUtc (унаследовано от FileSystemInfo)	Возвращает или устанавливает временя создания текущего объекта FileSystemInfo в формате всеобщего скоординированного времени (UTC).
Directory	Возвращает экземпляр родительской папки.
DirectoryName	Возвращает строку, описывающую полный путь к папке.
Exists	Возвращает значение, показывающее, существует ли файл. Переопределен.
Extension (унаследовано от FileSystemInfo)	Возврат строки, содержащей расширение файла.
FullName (унаследовано	Возвращает полный путь к папке или файлу.

от FileSystemInfo)	
LastAccessTime (унаследовано от FileSystemInfo)	Возвращает или устанавливает время последнего доступа к текущему файлу или папке.
LastAccessTimeUtc (унаследовано от FileSystemInfo)	Возвращает или устанавливает дату и время последнего доступа к заданному файлу или папке в формате всеобщего скоординированного времени (UTC).
LastWriteTime (унаследовано от FileSystemInfo)	Возвращает или устанавливает время последней операции записи в текущий файл или папку.
LastWriteTimeUtc (унаследовано от FileSystemInfo)	Возвращает или устанавливает время последней операции записи в текущий файл или папку в формате всеобщего скоординированного времени (UTC).
Length	Возвращает размер текущего файла.
Name	Возвращает имя файла. Переопределен.

Открытые методы

AppendText	Создает объект StreamWriter, добавляющий текст в файл, описываемый этим экземпляром FileInfo.
CopyTo	Перегружен. Копирование существующего файла в новый файл.
Create	Создание файла.
CreateObjRef (унаследовано от MarshalByRefObject)	Создает объект, который содержит всю необходимую информацию для создания прокси-сервера, используемого для коммуникации с удаленными объектами.
CreateText	Создание объекта StreamWriter, который записывает новый текстовый файл.
Delete	Удаление файла без возможности восстановления. Переопределен.
Equals (унаследовано от Object)	Перегружен. Определяет, равны ли два экземпляра Object.
GetHashCode (унаследовано от Object)	Служит хеш-функцией для конкретного типа, пригоден для использования в алгоритмах хеширования и структурах данных, например в хеш-таблице.
GetLifetimeService (унаследовано от MarshalByRefObject)	Извлекает служебный объект текущего срока действия, который управляет средствами срока действия данного экземпляра.
GetObjectData (унаследовано от FileSystemInfo)	Устанавливает объект SerializationInfo с именем файла и дополнительными сведениями об исключении.
GetType (унаследовано от	Возвращает Type текущего экземпляра.

Object)	
InitializeLifetimeService (унаследовано от MarshalByRefObject)	Получает служебный объект срока действия, для управления средствами срока действия данного экземпляра.
MoveTo	Перемещение заданного файла в новое положение с возможностью задания нового имени файла.
Open	Открывается файл с различными правами доступа на чтение-запись и совместное использование. Перегружен.
OpenRead	Создает разрешенный только для чтения FileStream.
OpenText	Создает StreamReader с кодировкой UTF-8, который выполняет считывание из существующего текстового файла.
OpenWrite	Создает разрешенный только для записи объект FileStream.
Refresh (унаследовано от FileSystemInfo)	Обновление состояния объекта.
ToString	Переопределен. Возвращает полный путь как строку.

Пример.

```
using System;
using System.IO;

class Test
{
   public static void Main ()
   {
      string  path = @"c:\temp\MyTest.txt";

      if  ( !File.Exists (path) )
      {
            // Создать файл для записи и освободить ресурсы
                  // с помощью Dispose () после выхода из блока using
         using  (StreamWriter sw = File.CreateText  (path))
         {
            sw.WriteLine ("Hello");
            sw.WriteLine ("And");
            sw.WriteLine ("Welcome");
         }
      }

                  // Открыть файл для чтения.
      using  (StreamReader sr = File.OpenText  (path))
      {
         string s = "";
         while  ( (s = sr.ReadLine ()) != null)
```

```
        {
           Console.WriteLine (s);
        }
      }

      try
      {
            string path2 = path + "temp";
          // Путь к несуществующему файлу. Исключение не возбуждается.
            File.Delete  (path2);

                  // Копировать файл.
            File.Copy  (path, path2);
        Console.WriteLine ("{0} был скопирован в {1}.", path, path2);

                  // Удалить только что созданный файл.
            File.Delete  (path2);
        Console.WriteLine ("{0} был успешно удален.", path2);
      }
      catch  (Exception e)
      {
        Console.WriteLine ("ОШИБКА: {0}", e);
      }
   }
}
```

Тот же пример без оператора using

```
using System;
using System.IO;

class Test
{
   public static void Main ()
   {
      string path = @"c:\temp\MyTest.txt";

      if  (!File.Exists (path))
      {
         // Создать файл для записи
         StreamWriter sw = File.CreateText (path);
         sw.WriteLine ("Hello");
         sw.WriteLine ("And");
         sw.WriteLine ("Welcome");
        sw.Close ();
      }

      // Open the file to read from.
      StreamReader sr = File.OpenText (path);
      string s = "";
      while  ( (s = sr.ReadLine ()) != null)
      {
         Console.WriteLine (s);
      }
     sr.Close ();
```

```
        try
        {
           string path2 = path + "temp";
           // Путь к несуществующему файлу.
           File.Delete (path2);

           // Копировать файл.
           File.Copy (path, path2);
           Console.WriteLine ("{0} был скопирован в {1}.", path, path2);

           // Удалить только что созданный файл.
           File.Delete (path2);
           Console.WriteLine ("{0} был успешно удален.", path2);
        }
        catch  (Exception e)
        {
           Console.WriteLine ("ОШИБКА: {0}", e);
        }
     }
  }
```

4. Что осталось за рамками лекций по вводу-выводу.

Класс NetworkStream – получает и посылает байты через сеть процессору, работающему на другом конце сетевого соединения.

Класс BufferedStream – выполняет буферизацию данных для повышения эффективности ввода-вывода.

Класс System.Security.Cryptography.CryptoStream – потоковая реализация криптографии.

Класс DriveInfo – получить список доступных устройств и информацио о любом из них.

Класс System.Security.AccessControl.FileSecurity – хранит список управления доступом (ACL – access control lists) к файлу (папке) на который ссылается.

Класс System.Security.AccessControl.FileSystemAccessRule – отдельное право доступа.

Класс Microsoft.Win32.Registry – предоставляет ключи для класса RegistryKey.

Класс Microsoft.Win32.RegistryKey – работает с ключами реестра (создание, чтение, модификация).

Методические рекомендации по использованию средств ввода-вывода

Открытие файла

1. Если требуется указать режим открытия и доступа:

```
fs = new FileStream ("testdata", FileMode.Create);  //Read и Write
fs = new FileStream ("testdata", FileMode.Open,
                                                    FileAccess.Read);          //  только
Read
fs = new FileStream ("testdata", FileMode.Append,
```

```
                                                FileAccess.Write);      // только Write
```

Связать поток с обработчиком потока:

```
StreamWriter   sw = new StreamWriter  (fs);              // для записи
StreamReader sw = new StreamReader (fs);       // для чтения
```

2. Более простой вариант открытия на запись. Если файл не существует, он создается:

```
StreamWriter sw = new StreamWriter ("C:\\Temp\\Text.txt");
StreamWriter sw = new StreamWriter ("Text.txt", true);  // Append
```

3. Более простой вариант открытия на чтение. Файл должен существовать:

```
StreamReader sw = new StreamReader ("C:\\Temp\\Text.txt");
```

4. В случае использования методов WriteAllLines или ReadAllLines открытие файла осуществляется автоматически перед выполнением операции записи/чтения. После чтения/записи файл закрывается.

Текстовые файлы для просмотра редактором
Файлы с последовательным доступом

- Для просмотра файла текстовым редактором.
- Без последующей разборки записей на данные.
- Самое дружественное представление информации на экране.

Примечание. Записи не структурированы, поэтому не предполагается их разбор на переменные. Каждая запись – это одна переменная типа string.

Форматирование:
Метод Format() для каждой записи, то есть одна запись – это одна строка. В записи, кроме переменных, предполагается наличие связующих слов.

```
 str = Format("Это {0}-ая строка", i);
```

Вывод в файл:
Способ 1. В цикле методом WriteLine класса StreamWriter:

```
sw. WriteLine ("Группа 171-1");
sw. WriteLine (Format("Студент {0}, курс {1}", fio, kurs) );
```

Способ 2. Накапливать строки в массиве строк и выводить их одним вызовом статического метода WriteAllLines класса File:

```
string[ ] text = { "Hello", "And", "Welcome" };
File.WriteAllLines (@"c:\temp\Text.txt", text );
```

Проблема: не всегда при создании массива строк изначально известен его размер.

Чтение:
только последовательное методом ReadLine в цикле или методом ReadAllLines за один вызов без последующей разборки записи на данные.

Достоинство метода WriteAllLines: размерность массива строк устанавливается методом.

Найти требуемую строку можно методами класса String, например, StartsWith.

Загрузка в базу данных:
не предполагается, так как все строки разные.

- Для просмотра файла текстовым редактором.
- Для редактирования файла вручную из текстового редактора.
- Для загрузки файла в программу и разбиения записей на составляющие данные.

1. Выводимые данные могут содержать внутри себя произвольное количество пробелов (разделителей).

Примечание. Объединить все данные в одну строку можно, но из-за наличия неконтролирумых пробелов разобрать такую запись после ее чтения из файла на данные не представляется возможным (если использовать пробел как разделитель).

Поэтому предлагается способ вывода каждой переменной в виде отдельной строки.

Форматирование:
Каждая нестроковая переменная преобразуется в строку методом ToString. В каждой записи в строго фиксированном месте могут присутствовать связующие слова.

```
int   kurs ;
str = ToString(kurs) ;
```

Вывод в файл:
Способ 1. Каждая переменная выводится в виде отдельной строки методом WriteLine() класса StreamWriter:

```
sw. WriteLine (fio);
sw. WriteLine (kurs.ToString());
```

Способ 2. Все выводимые строки накапливаются в массиве строк и этот массив выводится одним вызовом статического метода WriteAllLines класса File:

```
string[ ] text = new string[200] ;
text[0] = fio;  text[1] = kurs.ToString();   // и т.д. для всех переменных
        . . .        // и т.д. для всех записей
text[i] = fio;  text[i+1] = kurs.ToString(); // и т.д. для всех переменных

File.WriteAllLines ("Text.txt", Text );
```

Проблема: не всегда при создании массива строк известен его размер. Следует использовать класс ArrayList.

Чтение:
только последовательное методом ReadLine в цикле или методом ReadAllLines за один вызов.

Загрузка в базу данных:
Каждая прочитанная строка (т.е. переменная) преобразуется в исходный тип:

```
kurs = int.Parse ( text );          // после ReadLine
-или-
kurs = int.Parse ( text[i] );       // после ReadAllLines
```

Связующие слова пропускаются

2. Выводимые данные не содержат внутри себя пробелов (разделителей) или содержат их фиксированное количество.

Примечание. Можно объединить все данные в одну строку, отделяя их друг от друга пробелом (разделителями). Разобрать такую запись после ее чтения из файла на данные можно методом Split класса String.

Форматирование:
Формируется одна строка на запись, состоящая из данных, преобразованных в подстроки методами Format() и/или ToString.
Данные друг от друга должны отделяться одним или несколькими пробелами (уникальными разделителями), на которые при разборке будет реагировать метод Split.
В записях, кроме переменных, могут находиться связующие слова, которые при разборке строки пропускаются.

```
str = Format ("Запись-1 " + fio + " " + kurs);
 -или-
str = "Запись-1 " + fio + " " + ToString (kurs);
```

Вывод в файл:
Способ 1. Методом WriteLine класса StreamWriter:

```
sw. WriteLine (str);
```

Способ 2. Накапливать строки в массиве строк и выводить их одним вызовом статического метода WriteAllLines класса File:

```
string[ ] text ;
    . . .
File.WriteAllLines (@"c:\temp\Text.txt", text );
```

Чтение:
только последовательное методом ReadLine в цикле или методом ReadAllLines за один вызов.

Загрузка в базу данных:
Каждая прочитанная строка разбивается на массив подстрок, каждая из которых потом преобразуется с помощью метода Parse:

```
char [ ]   delimiter = { ',  ' };  // запятая и пробел - разделители
string     record = "Иванов, курс 2";
string [ ] data = null;

data = record.Split (delimiter);          // data[0] – fio,  data[2] – kurs
fio = data[0] ;    kurs = int.Parse (data[2]);
```

Связующие слова пропускаются (в примере - data[1]).

Текстовый файл с прямым доступом

- Для просмотра файла текстовым редактором.
- Для редактирования файла вручную из текстового редактора.
- Для загрузки файла в программу с использованием прямого доступа к записям.

Создавать текстовый файл с прямым доступом к записям не целесообразно, так как обеспечить одинаковую длину всех записей достаточно не просто. Например, целые числа 12 и 12345 занимают в памяти по 4 байта (для int), в то время как их символьное представление – 2 и 5 байт. Кроме того, такие записи практически нельзя редактировать текстовым редактором, так как при редактировании записей наверняка нарушится их исходная длина. Есть и еще одна проблема: русские и английские символы в файле по умолчанию кодируются разным количеством байт. Следовательно, если в одной записи будут английские буквы, а в другой – русские, то у этих записей будут разные размеры.

Вывод: в подавляющем большинстве случаев текстовые файлы предназначены только для последовательного чтения записей.

Двоичные файлы для загрузки в программу
Дв. файлы с последовательным доступом к записям

Для сохранения данных в файле и последующей загрузки файла в программу. Просмотр текстовым редактором бессмысленен.

1. Запись представляет собой совокупность отдельных переменных, которые могут составлять информационную часть объекта.

Примечание.

А. Для вывода и ввода записей рекомендуется использовать двоичный поток, создаваемый классами BinaryWriter и BinaryReader.

Б. Понятие записи в этом случае – это логическое объединение данных, например, одного объекта. В отличие от строк, двоичные данные друг от друга ничем не отделяются.

Форматирование:

не требуется, так как данные выводятся в их внутреннем (двоичном) формате.

Вывод данных в файл:

```
FileStream  fs;

int            i   = 10;
double         d   = 1023.56;
string                 str = "Привет студентам!";

fs = new FileStream ("testdata", FileMode.Create);
BinaryWriter  dataOut = new BinaryWriter (fs);
dataOut.Write (i);
dataOut.Write (d);
dataOut.Write (str);
```

Чтение и загрузка в базу данных:

только последовательное методами Read...() в цикле:

```
fs = new FileStream ("testdata", FileMode.Open);
BinaryReader dataIn = new BinaryReader(fs);

i    = dataIn.ReadInt32();
d    = dataIn.ReadDouble();
str = dataIn.ReadString();
```

2. Сохранение и восстановление объектов.

Примечание. Для вывода и ввода объектов рекомендуется использовать сериализацию (сохранение) и десериализацию (восстановление) объектов.

```
using System;
using System.IO;
using System.Runtime.Serialization;
using System.Runtime.Serialization.Formatters.Binary;

[Serializable]
class Student
{
```

```
    private int     kurs ;     // номер курса
    private string fio;                  // Фамилия студента

    public Student (int k, string f)
    {   kurs = k;    fio   = f;   }
  }

  class SaveObj
  {
    public static void Main()
    {
      FileStream  fs;

      Student st1 = new Student (1, "Тенишева");
      Student st2 = new Student (2, "Иванов");

      // Создание потока
      fs = new FileStream ("Student.dat", FileMode.Create);

      // Воспользуемся двоичным форматированием
      BinaryFormatter bf = new BinaryFormatter();

      // Сохраним объект в файле в двоичном виде
      bf.Serialize (fs, st1);      bf.Serialize (fs, st2);
      fs.Close();

      // Восстановим из файла сериализованный объект
      fs = new FileStream ("Student.dat", FileMode.Open);

      Student newSt1 = (Student) bf.Deserialize (fs);
      Student newSt2 = (Student) bf.Deserialize (fs);
    }
  }
```

Двоичные файлы с прямым доступом к записям

Для сохранения данных в файле и выборочного доступа к записям (на ввод и вывод).

Примечание. Для реализации прямого доступа к записям необходимо иметь способ определения адреса записи в файле. Адресом является смещение до первого байта требуемой записи от начала файла. Первый байт файла имеет смещение 0.

Существует два основных способа решения этой задачи.

1. Все записи файла имеют одинаковую (известную) длину.

В этом случае нет смысла использовать какую-либо дополнительную информацию о месторасположении записей.

1) Для обеспечения одинаковых размеров всех записей необходимо:

- чтобы значения строковой переменной во всех записях имели одинаковую длину. Для этого необходимо каждое значение дополнить пробелами до требуемого размера (см. метод PadRight (int);).

- во-вторых, чтобы все символы – русские и английские – кодировались одинаковым количеством байтов (например, по 2 байта), необходимо в классе кодировки выбрать Unicode.

2) При определении адреса требуемой записи, необходимо учесть, что размер каждой строки в записи файла вдвое больше числа ее символов (Unicode – 2 байта), плюс один байт на каждую строку записи. В этом байте хранится длина строки.

Для ввода-вывода рекомендуется использовать двоичный поток. Поиск требуемых записей может быть организован как на диске, так и в оперативной памяти.

1.1 Поиск требуемых записей, выполняется на диске

Примечание. Это случай, когда файл невозможно целиком прочитать в память или когда количество операций ввода-вывода невелико.

```
int       kurs = 1;                                  //4 байта
string    fio = "Иванов       ";  //15 символов -> 31 б.

fs = new FileStream ("testdata", FileMode.Create);
dataOut = new BinaryWriter (fs, Encoding.Unicode);
dataOut.Write (kurs);
dataOut.Write (fio);

kurs = 2;
fio = "Smit          ";
dataOut.Write (kurs);
dataOut.Write (fio);

kurs = 3;
fio = "Григорьева    ";
dataOut.Write (kurs);
dataOut.Write (fio);

dataOut.Close();

fs = new FileStream ("testdata", FileMode.Open);
dataIn = new BinaryReader(fs, Encoding.Unicode);
fs.Seek(35, SeekOrigin.Begin); //указатель на 2-ю запись
                                  // Длина записи = 4+(15*2)+1 = 35

kurs = dataIn.ReadInt32();        // читаем вторую запись
fio  = dataIn.ReadString();
```

1.2 Все операции с файлом выполняются в оперативной памяти

Примечание. Если необходимо осуществлять выборочное чтение большого количества записей из небольшого двоичного файла (Size < 2 Гб), то предпочтительнее будет прочитать весь файл последовательно в оперативную память, в массивы переменных, входящих в состав запи-

сей. Совокупность всех элементов массивов с одинаковым индексом относятся к одному объекту.

Постоянная длина записей позволяет определить их количество:

```
int n = (int)fs.Length / 35;            // количество записей
```

Прямой файл всегда может быть прочитан последовательно:

```
int[ ]    kursAr          = new int[n] ;
string[ ] fioAr   = new string[n];
          . . .
for (int i = 0;  i < n;  i++)
{
    kursAr[i] = dataIn.ReadInt32();      //  (i+1) - я
    fioAr[i]    = dataIn.ReadString();   //                  запись
}
```

2. Все записи двоичного файла имеют разную длину.

Такая ситуация, как правило, возникает из-за наличия строковых полей, имеющих разную длину.

Вместо того чтобы обеспечивать постоянство длины записей, при их выводе в файл запоминаются адреса записей в дополнительном (индексном) файле.

Если все адреса записывать в файл индексов в двоичном формате как переменную одного из целых типов (byte, ushort, uint или ulong), то такой файл будет файлом с прямым доступом. Техника работы с файлом прямого доступа была описана выше.

После определения адреса требуемой записи выполняется позиционирование в файле данных и выполнение операций считывания двоичных данных записи.

Данный подход можно использовать для прямого доступа к сериализованным объектам.

Недостатком такого подхода является то, что при модификации строковых полей записи, как правило, изменяется их длина, поэтому модифицировать можно только нестроковые поля (или обеспечивать постоянство длины строк в сериализованных объектах).

Читать двоичный файл в байтовый массив методом ReadAllBytes() или другим байтовым методом, а затем организовывать выборку данных из этого массива, кажется простой задачей. Но отображение байтов на типы данных – это непростые алгоритмы.

Таким образом, для прямого доступа к записям файл должен быть, как правило, двоичным.

Если прямой доступ не требуется, файл может быть любым – и текстовым, и двоичным.

Если файл должен просматриваться в текстовом редакторе файл должен быть текстовым.

ИНТЕРФЕЙСЫ

1. Определение и реализация интерфейса

Интерфейс определяет набор методов, которые должны быть реализованы классом, наследующим этот интерфейс. Сам интерфейс не реализует методы.

Интерфейсы объявляются с помощью ключевого слова **interface**. Вот как выглядит упрощенная форма объявления интерфейса:

```
interface имя
{
тип_возврата   имя_метода_1 (список_параметров);
тип_возврата   имя_метода_2 (список_параметров);
 . . . . . .
тип_возврата   имя_метода_N (список_параметров);
}
```

Интерфейсы синтаксически подобны абстрактным классам.

Однако в интерфейсе ни один метод не может включать тело, т.е. интерфейс, в принципе, не предусматривает какой бы то ни было реализации.

Он определяет, что должно быть сделано, но не уточняет, как.

Какой-либо интерфейс может унаследовать и реализовать любое количество классов. И каждый по-своему. При этом один класс может реализовать любое число разных интерфейсов.

Для реализации интерфейса класс должен обеспечить тела (реализацию) методов, описанных в интерфейсе. Каждый класс может определить собственную реализацию. Таким образом, два класса могут реализовать один и тот же интерфейс различными способами, но все классы поддерживают одинаковый набор методов.

Поскольку интерфейс для всех объектов одинаков, это позволяет программе, “осведомленной” о наличии интерфейса, использовать объекты любого класса.

Рассмотрим пример интерфейса для класса, который генерирует ряд чисел.

```
public interface ISeries
{
   int    getNext ();              // Возвращает следующее число ряда.
   void reset ();                  // Выполняет перезапуск.
   void setStart (int x);    // Устанавливает начальное значение.
}
```

Этот интерфейс имеет имя ISeries. Хотя префикс "I" необязателен, многие программисты его используют, чтобы отличать интерфейсы от классов. Интерфейс ISeries объявлен открытым, поэтому он может быть реализован любым классом в любой программе.

Помимо сигнатур методов интерфейсы могут объявлять сигнатуры свойств, индексаторов и событий.

Так как в интерфейсе методы не содержат реализации, то:

- нельзя создать экземпляр интерфейса;
- интерфейсы не могут иметь полей, так как это подразумевает некоторую внутреннюю реализацию;
- они не могут определять конструкторы, деструкторы или операторные методы;
- кроме того, ни один член интерфейса не может быть объявлен статическим;
- все методы в классе-интерфейсе по умолчанию являются открытыми и виртуальными, однако при их реализации писать override не нужно.

Реализация интерфейсов

Формат записи класса, который реализует интерфейс:

```
class имя_класса : имя_интерфейса
{
   // тело класса
}
```

Если класс реализует интерфейс, он должен это сделать в полном объеме, т.е. реализация интерфейса не может быть выполнена частично.

Классы могут реализовать несколько интерфейсов. В этом случае имена интерфейсов отделяются запятыми.

Класс может наследовать базовый класс и реализовать один или несколько интерфейсов. В этом случае список интерфейсов должен возглавлять имя базового класса.

Рассмотрим пример реализации интерфейса ISeries, объявление которого было приведено выше. Здесь создается класс с именем ByTwos, генерирующий ряд чисел, в котором каждое следующее число больше предыдущего на два.

```
class ByTwos : ISeries
{
   int start;
   int val;

   public ByTwos()
   {
      start = 0;
      val = 0;
   }

   public int getNext()
   {
      val += 2;
      return val;
   }
```

```
    public void reset()
    {
      val = start;
    }

    public void setStart(int x)
    {
      start = x;
      val = start;
    }

    // Здесь могут быть дополнительные члены
}
```

Рассмотрим пример, демонстрирующий использование интерфейса, реализованного классом ByTwos.

```
using System;

class SeriesDemo
{
  public static void Main()
  {
    ByTwos ob = new ByTwos();

    if (ob is ISeries)
     Console.WriteLine("Объект реализует интерфейс ISeries");
    else throw new Exception ("Объект НЕ реализует ISeries");


    for (int i = 0; i < 5; i++)
      Console.WriteLine("Следующее значение = " + ob.getNext());

    Console.WriteLine("\nПереход в исходное состояние.");
    ob.reset();

    for (int i = 0; i < 5; i++)
      Console.WriteLine("Следующее значение = " + ob.getNext());

    Console.WriteLine("\nНачинаем с числа 100.");
    ob.setStart(100);
    for (int i = 0; i < 5; i++)
      Console.WriteLine("Следующее значение = " + ob.getNext());
  }
}
}
```

Рассмотрим пример другой реализации интерфейса. Класс Primes генерирует ряд простых чисел. Обратите внимание на то, что его способ реализации интерфейса ISeries в корне отличается от используемого классом ByTwos.

```
class Primes : ISeries
{
   int start;
   int val;

   public Primes()
   {
      start = 2;
      val = 2;
   }

   public int getNext()
   {
      int i, j;
      bool isprime;

      val++;
      for (i = val; i < 1000000; i++)
      {
         isprime = true;
         for (j = 2; j < (i / j + 1); j++)
         {
            if ((i % j) == 0)
            {
               isprime = false;
               break;
            }
         }
         if (isprime)
         {
            val = i;
            break;
         }
      }
      return val;
   }

   public void reset()
   {
      val = start;
   }

   public void setStart(int x)
   {
      start = x;
      val = start;
   }
}
```

Здесь важно понимать, что, хотя классы Primes и ByTwos генерируют разные ряды чисел, оба они реализуют один и тот же интерфейс ISeries.

Использование интерфейсных ссылок

Ссылочная переменная интерфейсного типа может ссылаться на любой объект, который реализует ее интерфейс.

В этом случае интерфейс выступает в роли базового класса, а его наследники являются производными классами. Поэтому в случае наследования интерфейса работает тот же механизм, что и для классов.

Важно понимать, что интерфейсная ссылочная переменная "осведомлена" только о методах, объявленных "под сенью" ключевого слова interface. Следовательно, интерфейсную ссылочную переменную нельзя использовать для доступа к другим переменным или методам, которые может определить объект, реализующий этот интерфейс.

Интерфейсные свойства

Формат объявления:

```
тип   имя
{
  get;
  set;
}
```

Свойства, предназначенные только для чтения или только для записи, содержат только get- или set-элемент, соответственно.

Интерфейсные индексаторы

Формат объявления:

```
тип_элемента this [int индекс]
{
  get;
  set;
}
```

Индексаторы, предназначенные только для чтения или только для записи, содержат только get- или set- элемент, соответственно.

Наследование интерфейсами интерфейсов

Один интерфейс может унаследовать "богатство" другого. Синтаксис этого механизма аналогичен синтаксису, используемому для наследования классов.

Если класс реализует интерфейс, который наследует другой интерфейс, этот класс должен обеспечить способы реализации для всех членов, определенных внутри цепочки наследования интерфейсов.

Пример:

```
public interface IA { . . . }
public interface IB : IA { . . . }
class MyClass : IB { . . . }   // Класс должен реализовать члены IA и IB
```

В производном интерфейсе можно объявить член, который скрывает член, определенный в базовом интерфейсе (член д.б. помечен как new).

Явная реализация членов интерфейса

При реализации члена интерфейса можно квалифицировать его имя с использованием имени интерфейса (явная реализация).

В этом случае такой член будет недоступен вне класса.

```
interface IA
{
   int   Method (int x);
}

// Явная реализация интерфейсного метода
class Class : IA
{
   int IA.Method(int x)
   {
      return x / 3;
   }
}
```

Причина: класс может реализовать два интерфейса, которые объявляют методы с одинаковыми именами и типами. Полная квалификация имен позволяет избежать неопределенности.

Для чего нужны интерфейсы

Интерфейс не несет в себе никакой функциональности, а объявленные в интерфейсе методы можно реализовать в классе и без существования интерфейса. Почему же он бывает необходим?

Интерфейс может быть базовым классом для нескольких производных для реализации динамического связывания (вызова перегруженных методов).

Если Вы решили реализовать класс с интерфейсом, который будет использовать какая-либо программа (клиент), то компилятор не позволит Вам сделать ошибку в количестве методов, которые необходимо реализовать, сигнатуре каждого метода интерфейса и возвращаемом им значении.

Программа, использующая класс, в котором должен быть реализован интерфейс, может удостовериться в этом, прежде чем вызывать его интерфейсные методы, с помощью рефлексии или операторов as или is:

```
if (!(ob is IA))
    throw new Exception ("Объект НЕ реализует интерфейс IA");
```

Ряд средств языка во время выполнения программы так же осуществляют проверку наличия интерфейса.

Например, оператор foreach опрашивает объект, на предмет того, реализует ли он интерфейс System.Collection.IEnumerable.

Если Вы подготовили объект с этим интерфейсом, то оператор foreach будет успешно использовать Ваш метод GetEnumerator(). Этот метод возвратит объект с методом MoveNext() и свойством Current для доступа к очередному элементу коллекции.

Если окажется, что объект не реализует требуемый интерфейс, foreach сгенерирует исключение.

2. Интерфейсы коллекций

В C# под коллекцией понимается группа объектов. Пространство имен System. Collections содержит множество интерфейсов и классов, которые определяют и реализуют коллекции различных типов. Все эти коллекции разработаны на основе набора четко определенных интерфейсов.

Ряд встроенных реализаций интерфейсов в таких коллекциях как ArrayList, Hashtable, Stack и Queue, вы можете использовать "как есть".

У каждого программиста также есть возможность реализовать собственную коллекцию, но в большинстве случаев достаточно встроенных.

Среда .NET Framework поддерживает три основных типа коллекций:

- общего назначения,
- специализированные,
- ориентированные на побитовую организацию данных.

Коллекции общего назначения реализуют ряд основных структур данных, включая динамический массив, стек и очередь. Сюда также относятся словари, предназначенные для хранения пар ключ/значение. Коллекции общего назначения работают с данными типа object, поэтому их можно использовать для хранения данных любого типа.

Мощь коллекций состоит в том, что они могут хранить не только встроенные типы, но и объекты любого типа, включая объекты классов, создаваемых программистами.

Коллекции специального назначения ориентированы на обработку данных конкретного типа или на обработку уникальным способом. Например, существуют специализированные коллекции, предназначенные только для обработки строк или однонаправленного списка.

Классы коллекций, ориентированных на побитовую организацию данных, служат для хранения групп битов. Коллекции этой категории поддерживают такой набор операций, который не характерен для коллекций других типов. Например, в известной многим биториентированной коллекции BitArray определены такие побитовые операции, как И и исключающее ИЛИ.

Таблица. Интерфейсы коллекций

Интерфейс	Описание
IEnumerable	Определяет метод GetEnumerator(), который поддерживает перечислитель для любого класса коллекции
IEnumerator	Содержит методы, которые позволяют поэлементно получать содержимое коллекции
ICollection	Определяет элементы, которые должны иметь все коллекции
IList	Определяет коллекцию, к которой можно получить доступ посредством индексатора
IDictionary	Определяет коллекцию, которая состоит из пар ключ/значение
IDictionaryEnumerator	Определяет перечислитель для коллекции, которая реализует интерфейс IDictionary
IComparer	Определяет метод compare(), который выполняет сравнение объектов, хранимых в коллекции
IHashCodeProvider	Определяет хеш-функцию

Интерфейсы IEnumerable, IEnumerator и IDictionaryEnumerator

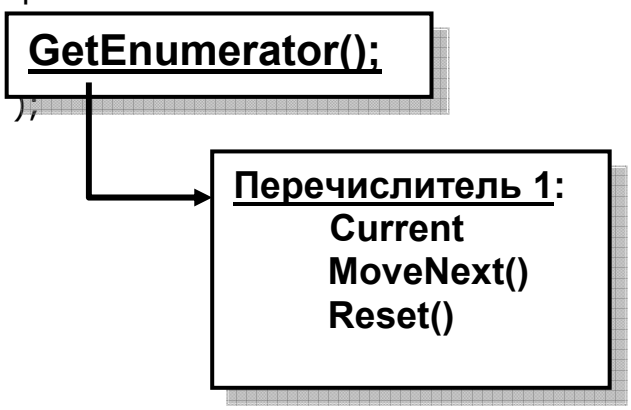
Основополагающим для всех коллекций является реализация перечислителя (нумератора), который поддерживается интерфейсами IEnumerator и IEnumerable.

Перечислитель обеспечивает стандартизованный способ поэлементного доступа к содержимому коллекции. Поскольку каждая коллекция должна реализовать интерфейс IEnumerable, к элементам любого класса коллекции можно получить доступ с помощью методов, определенных в интерфейсе IEnumerator.

Следовательно, после внесения небольших изменений код, который позволяет циклически опрашивать коллекцию одного типа, можно успешно использовать для циклического опроса коллекции другого типа (например в цикле foreach).

Все коллекции C# реализуют перечислитель.

```
interface IEnumerable
{
   IEnumerator  GetEnumerator( );
}
interface IEnumerator
{
   object Current { get };
   bool    MoveNext();
   void    Reset();
}
```



Метод GetEnumerator возвращает объект-перечислитель для коллекции.

Исключение. Для коллекций, в которых хранятся пары ключ/значение (т.е. словари), метод GetEnumerator() возвращает объект типа IDictionaryEnumerator, а не типа IEnumerator.

Класс IDictionaryEnumerator является производным от класса IEnumerator и распространяет свои функциональные возможности перечислителя на область словарей.

Почему перечислитель является объектом?
Ответ: у одной коллекции может быть несколько перечислителей.

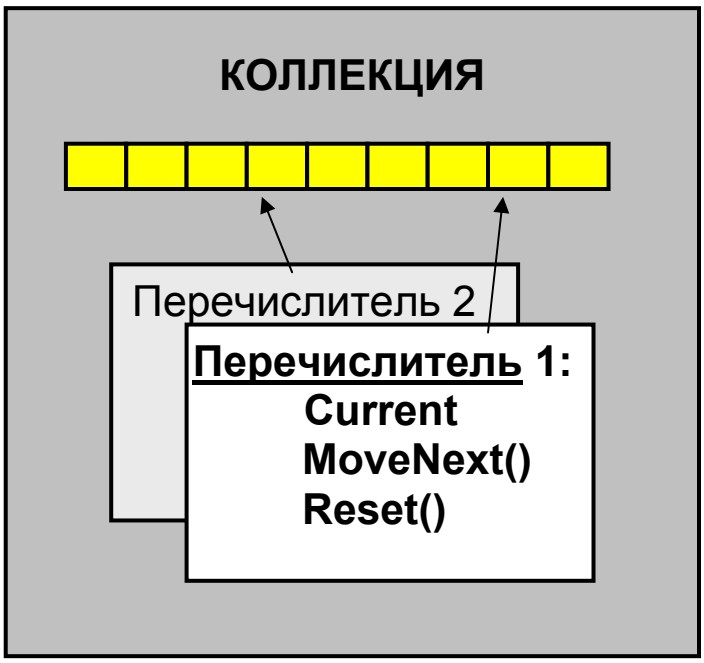


рис.1

Пример. На основе структуры Vector создать коллекцию. Коллекция должна перечислять трехмерные координаты вектора (X, Y, Z).

```
using System;
using System.Collections;
using System.Text;

class MainEntryPoint
{
   static void Main(string[] args)
   {
      Vector Vect1 = new Vector(1.0, 2.0, 5.0);
      foreach (double Next in Vect1)
         Console.Write("  " + Next);
      Console.ReadLine();
   }
}

struct Vector : IEnumerable
```

```
{
   private double x, y, z;

   public Vector(double x, double y, double z)
   {
      this.x = x;   this.y = y;   this.z = z;
   }

   public IEnumerator GetEnumerator()
   {
      return new VectorEnumerator(this);
   }

   private class VectorEnumerator : IEnumerator
   {
      Vector theVector;
      int       location;

      public VectorEnumerator(Vector Vect)
      {
         theVector = Vect;
         location = -1;
      }

      public bool MoveNext()
      {
         ++location;
         return (location > 2) ? false : true;
      }

      public object Current
      {
         get
         {
            switch (location)
            {
               case 0:
                  return theVector.x;
               case 1:
                  return theVector.y;
               case 2:
                  return theVector.z;
               default:
                  throw new IndexOutOfRangeException(
                              "Вышли за границу вектора: " + location);
            }
         }
      }
      public void Reset()
      {
         location = -1;
      }
   }
}
```

Интерфейс ICollection

Интерфейс ICollection является фундаментом, на котором построены все коллекции. Он наследует интерфейс IEnumerable.

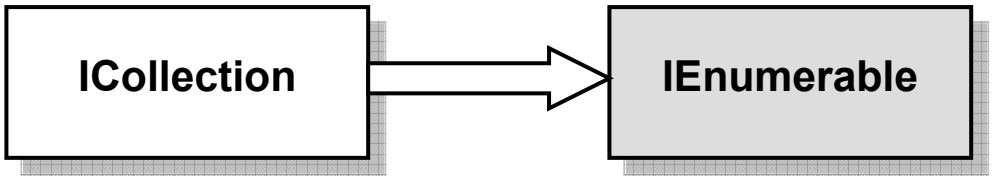


рис.2

В интерфейсе ICollection определены следующие свойства:

Свойство	Описание
int Count { get; }	Количество элементов коллекции в данный момент
bool IsSynchronized { get; }	Принимает значение true, если коллекция синхронизирована, и значение false в противном случае. По умолчанию коллекции не синхронизированы. Но для большинства коллекций можно получить синхронизированную версию
object SyncRoot { get; }	Объект, для которого коллекция может быть синхронизирована

Методы интерфейса ICollection:

Метод	Описание
void CopyTo (Array target, int startIdx);	Метод копирует содержимое коллекции в массив target, начиная с индекса startIdx.
IEnumerator GetEnumerator();	Метод возвращает перечислитель коллекции (задан интерфейсом IEnumerable).

Интерфейс IList

рис.3

Интерфейс IList наследует интерфейс ICollection и определяет поведение коллекции, доступ к элементам которой разрешен посредством индекса с отсчетом от нуля.

Помимо методов, определенных в интерфейсе ICollection, интерфейс IList определяет и собственные методы (они сведены в табл.).

Некоторые из этих методов служат для модификации коллекции. Если же коллекция предназначена только для чтения или имеет фиксированный размер, вызов этих методов приведет к генерированию исключения типа NotSupportedException.

Методы, определенные в интерфейсе IList

Метод	Описание
int Add (object obj)	Добавляет объект obj в вызывающую коллекцию. Возвращает индекс, по которому этот объект сохранен.
void Clear ( )	Удаляет все элементы из вызывающей коллекции.
bool Contains (object obj)	Возвращает значение true, если вызывающая коллекция содержит объект, переданный в параметре obj, и значение false в противном случае.
int IndexOf (object obj)	Возвращает индекс объекта obj, если он (объект) содержится в вызывающей коллекции. Если объект obj не обнаружен, метод возвращает -1.
void Insert (int idx, object obj)	Вставляет в вызывающую коллекцию объект obj по индексу, заданному параметром idx. Элементы, находившиеся до этого по индексу idx и далее, смещаются вперед, чтобы освободить место для вставляемого объекта obj.
void Remove (object obj)	Удаляет первое вхождение объекта obj из вызывающей коллекции. Элементы, находившиеся до этого за удаленным элементом, смещаются назад, чтобы ликвидировать образовавшуюся "брешь".
void RemoveAt (int idx)	Удаляет из вызывающей коллекции объект, расположенный по индексу, заданному параметром idx. Элементы, находившиеся до этого за удаленным элементом, смещаются назад, чтобы ликвидировать образовавшуюся "брешь".

Свойства и индексатор интерфейса IList

Свойства и индексатор	Описание
bool IsFixedSize { get; }	true - означает, что в такую коллекцию нельзя вставлять элементы и удалять их из нее.

bool IsReadOnly { get; }	true - содержимое коллекции изменению не подлежит.
object this[int idx] { get; set; }	Считывание или запись значения элемента с индексом idx. Нельзя добавлять новые элементы

Интерфейс IDictionary

Коллекция, которая реализует интерфейс IDictionary, служит для хранения пар ключ/значение.

Сохраненную однажды пару можно затем извлечь по заданному ключу. Интерфейс IDictionary наследует интерфейс ICollection.

рис.4

Методы, определенные в интерфейсе IDictionary

Метод	Описание
void Add (object key, object v)	Добавляет в вызывающую коллекцию пару ключ/значение, заданную параметрами key и v. Ключ key не должен быть нулевым. Если окажется, что ключ key уже хранится в коллекции, генерируется исключение типа ArgumentException.
void Clear()	Удаляет все пары ключ/значение из вызывающей коллекции.
bool Contains (object key)	Возвращает значение true, если вызывающая коллекция содержит объект key в качестве ключа. В противном случае возвращает значение false.
iDictionaryEnumerator GetEnumerator( )	Возвращает перечислитель для вызывающей коллекции.
void Remove (object key)	Удаляет элемент, ключ которого равен значению key

В интерфейсе IDictionary определены следующие свойства:

Свойство	Описание
bool isFixedsize { get; }	Равно значению true, если словарь имеет фиксированный размер.
bool isReadOniy { get; }	Равно значению true, если словарь предназначен только для чтения.
iCollection Keys { get; }	Получает коллекцию ключей
iCollection Values { get; }	Получает коллекцию значений.

С помощью свойств Keys и Values ключи и значения, хранимые в словарной коллекции, можно получить в виде отдельных списков.

В интерфейсе IDictionary определен следующий индексатор:
object this [object key] { get; set; }

Этот индексатор можно использовать для получения или установки значения элемента. Его можно также использовать для добавления в коллекцию нового элемента. "Индекс" в данном случае не является обычным индексом, а ключом элемента.

Интерфейс IDictionaryEnumerator

Класс коллекции, который реализует интерфейс IDictionary, предназначен для хранения пар ключ/значение. Для опроса элементов в такой коллекции используется интерфейс IDictionaryEnumerator, а не IEnumerator.

Класс IDictionaryEnumerator является производным от класса IEnumerator и дополнительно определяет "свои" три свойства. Первое объявляется так:

DictionaryEntry Entry { get; }

Два остальные свойства интерфейса IDictionaryEnumerator:

object Key { get; }
object Value { get; }

Свойство Entry с помощью перечислителя позволяет получить следующую пару ключ/значение в форме структуры типа DictionaryEntry.

Пример.

```
Hashtable ht = new Hashtable();

ht.Add ("Анатолий", "555-3456");
ht.Add ("Александр", "555-3452");

IDictionaryEnumerator etr = ht.GetEnumerator();
while (etr.MoveNext())
  Console.WriteLine(etr.Entry.Key + ": " + etr.Entry.Value);
```

Структура DictionaryEntry

Коллекции, в которых хранятся пары ключ/значение, используют для их хранения объект типа DictionaryEntry.

В этой структуре определены следующие свойства:

public object Key { get; set; }
public object Value { get; set; }

Эти свойства используются для получения доступа к ключу или к соответствующему ему значению.

Объект типа DictionaryEntry можно создать с помощью следующего конструктора:

```
public   DictionaryEntry (object key, object value)
```

Здесь параметр key принимает ключ,
а параметр value — значение.

Интерфейс IComparer

Этот интерфейс можно использовать для задания способа сортировки элементов коллекции. В интерфейсе IComparer определен метод Compare(), который позволяет сравнивать два объекта:

```
int Compare (object v1,  object v2)
```

Метод Compare() возвращает положительное число, если значение v1>v2, отрицательное, если v1<v2, и нуль, если v1=v2.

Интерфейс IHashCodeProvider

Интерфейс IHashCodeProvider должен быть реализован коллекцией, если программисту необходимо определить собственную версию метода GetHashCode(). По умолчанию используется метод Object.GetHashCode().

КОЛЛЕКЦИИ

1. Классы коллекций общего назначения

Класс	Описание
ArrayList	Динамический массив, т.е. массив который при необходимости может увеличивать свой размер.
Hashtable	Хеш-таблица (словарь) для пар ключ/значение.
Queue	Очередь, или список, действующий по принципу: первым прибыл — первым обслужен.
SortedList	Отсортированный список пар ключ/значение.
Stack	Стек, или список, действующий по принципу: первым прибыл — последним обслужен.

Класс ArrayList

Класс ArrayList предназначен для поддержки динамических массивов, которые при необходимости могут увеличиваться или сокращаться.

Класс ArrayList реализует интерфейсы:

- ICollection,
- IList,
- lEnumerable,
- ICloneable.

// Демонстрация использования ArrayList-массива.

```
using System;
using System.Collections;

class ArrayListDemp
{
   public static void Main()
   {
      ArrayList al = new ArrayList();    // Создаем динамический массив.

      // Добавляем элементы в динамический массив.
      al.Add('C');
      al.Add('A');
      al.Add('E');

      // Отображаем массив, используя индексацию.
      for (int i = 0; i < al.Count; i++)
         Console.Write(al[i] + " ");

      al.Remove('A');              // Удаляем элемент

      al[0] = 'Y';                 // Изменяем элементы
      al[1] = 'X';
```

```
        al.Sort();                    // Сортировка массива

        foreach (char ch in al)
          Console.Write(ch + " ");

        Console.WriteLine("Индекс элемента 'Y' равен " +
                    al.BinarySearch('Y'));  // поиск элемента

        // Создаем обычный массив из динамического.
        char[] ia = (char[])al.ToArray (typeof(char));
      }
    }
```

Класс Queue

Добавление элементов в очередь и удаление их из нее осуществляется по принципу "первым пришел — первым обслужен" (first-in, first-out— FIFO).

Очередь — это динамическая коллекция, которая при необходимости увеличивается, чтобы принять для хранения новые элементы.

Класс Queue реализует интерфейсы:

- ICollection (здесь определено свойство int Count {get;}),
- lEnumerable,
- ICloneable.

Методы, определенные в классе Queue

Метод	Описание
public virtual bool Contains(object v)	Возвращает значение true, если объект v содержится в вызывающей очереди. В противном случае возвращает значение false
Public virtual void Clear()	Устанавливает свойство Count равным нулю, тем самым эффективно очищая очередь
public virtual object Dequeue()	Возвращает объект из начала вызывающей очереди, Возвращаемый объект из очереди удаляется
public virtual void Enqueue(object v)	Добавляет объект v в конец очереди
public virtual object Peek ( )	Возвращает объект из начала вызывающей очереди, но не удаляет его
public static Queue Synchronized(Queue q)	Возвращает синхронизированную версию очереди, заданной параметром q
public virtual object[ ] ToArray ( )	Возвращает массив, который содержит копии элементов из вызывающей очереди
public virtual void TrimToSlze( )	Устанавливает свойство capacity равным значению свойства Count

```
// Демонстрация класса Queue.
using System;
using System.Collections;

class QueueDemo
{
   public static void Main()
   {
      int a;
      Queue q = new Queue();

      q.Enqueue(22);
      q.Enqueue(65);
      q.Enqueue(91);

      foreach (int i in q)
         Console.Write(i + " ");
      Console.WriteLine();

      try
      {
         a = (int)q.Dequeue(); Console.WriteLine(a);
         a = (int)q.Dequeue(); Console.WriteLine(a);
         a = (int)q.Dequeue(); Console.WriteLine(a);
         a = (int)q.Dequeue(); Console.WriteLine(a);
      }
      catch (InvalidOperationException)
      {
         Console.WriteLine("Очередь пуста.");
      }
   }
}
```

Класс Hashtable

Класс Hashtable предназначен для создания коллекции, в которой для хранения объектов используется хеш-таблица. В хеш-таблице для хранения информации используется механизм, именуемый хешированием (hashing), Суть хеширования состоит в том, что для определения уникального значения, которое, называется хеш-кодом, используется информационное содержимое соответствующего ему ключа. Хеш-код затем используется в качестве индекса, по которому в таблице отыскиваются данные, соответствующие этому ключу.

Преимущество хеширования — в том, что оно позволяет сохранять постоянным время выполнения таких операций, как поиск, считывание и запись данных, даже для больших объемов информации.

Hashtable-коллекция не гарантирует сохранения порядка элементов.

Класс Hashtable реализует интерфейсы:

- IDictionary,
- ICollection,
- lEnumerable,
- ISerializable,

- IDeserializationCallback,
- ICloneable.

В классе Hashtable определено множество конструкторов, но чаще всего используется следующий:

public Hashtable();

Наиболее употребимые методы класса Hashtable

Метод	Описание
public virtual bool ContainsKey(object k)	Возвращает значение true, если в вызывающей Hashtable-коллекции содержится ключ, заданный параметром k. В противном случае возвращает значение false.
public virtual bool ContainsValue(object v)	Возвращает значение true, если в вызывающей Hashtable-коллекции содержится значение, заданное параметром v. В противном случае возвращает значение false
public virtual IDictionaryEnumerator GetEnumerator()	Возвращает для вызывающей Hashtable-коллекции нумератор типа IDictionaryEnumerator
public static Hashtable Synchronized(Hashtable ht)	Возвращает синхронизированную версию вызывающей Hashtabie-коллекции, переданной в параметре ht.

В классе Hashtable, помимо свойств, определенных в реализованных им интерфейсах, также определены два собственных public-свойства. Используя следующие свойства, можно из Hashtable-коллекции получить коллекцию ключей или значений:

```
public  virtual   ICollection Keys { get ; }
public  virtual   ICollection Values { get ; }
```

В классе определен так же и индексатор
this[ключ] {get; set;}

В классе Hashtable пары ключ/значение хранятся в форме структуры типа DictionaryEntry, но по большей части вас это не будет касаться, поскольку свойства и методы обрабатывают ключи и значения отдельно.

```
// Демонстрация использования Hashtable-коллекции.
using System;
using System.Collections;

class HashtableDemo
```

```
{
   public static void Main()
   {
      Hashtable ht = new Hashtable();    // Создаем хеш-таблицу.

      // Добавляем элементы в хеш-таблицу.
      ht.Add("здание",              "жилое помещение");
      ht.Add("книга",               "набор печатных слов");
      ht.Add("яблоко",              "съедобный фрукт");
      ht.Add("автомобиль",          "транспортное средство");

      //Добавляем элементы с помощью индексатора.
      ht ["трактор"] = "сельскохозяйственная машина";

      // Извлекаем элемент по ключу.
      string value = (string)ht ["автомобиль"];
      Console.WriteLine ("---автомобиль: " + value);

      // Получаем коллекцию ключей.
      // Используем ключи для получения значений.
      Console.WriteLine("\n---Первый способ---");
      ICollection c = ht.Keys;

      foreach(string str in c)
         Console.WriteLine(str + ": " + ht[str]);

      ht.Remove("трактор");                 // Удалить элемент

      Console.WriteLine("\n---Второй способ---");
      // Используем структуру DictionaryEntry для получения ключей и значений.
      foreach (DictionaryEntry de in ht)
      {
         Console.WriteLine(de.Key + ": " + de.Value);
      }
   }
}
```

Результаты выполнения этой программы таковы
(порядок - другой):

```
---автомобиль: транспортное средство

---Первый способ---
здание: жилое помещение
автомобиль: транспортное средство
яблоко: съедобный фрукт
книга: набор печатных слов
трактор: сельскохозяйственная машина

---Второй способ---
здание: жилое помещение
автомобиль: транспортное средство
яблоко: съедобный фрукт
книга: набор печатных слов
```

Класс SortedList

Класс SortedList предназначен для создания коллекции, которая хранит пары ключ/значение в упорядоченном виде, а именно отсортированы по ключу.

Класс SortedList реализует интерфейсы:

- ICollection,
- IEnumerable,
- ICloneable.

Конструктор:
public SortedList( )

Наиболее употребимые методы класса SortedList

Метод	Описание
public virtual bool ContainsKey (object key)	Возвращает значение true, если в вызывающей SortedList-коллекции содержится ключ, заданный параметром key. В противном случае возвратит значение false
public virtual bool ContainsValue(object value)	Возвращает значение true, если в вызывающей SortedList-коллекции содержится значение, заданное параметром value. Иначе – false.
public virtual object GetByIndex(int idx)	Возвращает значение, индекс которого задан параметром idx
public virtual IDictionaryEnumerator GetEnumerator ()	Возвращает нумератор типа IDictionaryEnumerator для вызывающей SortedList-коллекции
public virtual object GetKey (int idx)	Возвращает ключ, индекс которого задан параметром idx
public virtual IList GetKeyList()	Возвращает IList- коллекцию ключей, хранимых в вызывающей SortedList-коллекции
public virtual IList GetValueList()	Возвращает IList-коллекцию значений, хранимых в вызывающей SortedList-коллекции
public virtual int IndexOfKey (object key)	Возвращает индекс ключа, заданного параметром key. Возвращает значение -1, если в списке нет заданного ключа
public virtual int IndexOfValue (object value)	Возвращает индекс первого вхождения значения, заданного параметром value. Возвращает -1, если в списке нет заданного ключа
public virtual void SetByIndex(int idx, object value)	Устанавливает значение по индексу, заданному параметром idx, равным значению, переданному в параметре value

public static SortedList Synchronized(SortedList sl)	Возвращает синхронизированную версию SortedList-коллекции, переданной в параметре sl
public virtual void TrimToSize()	Устанавливает свойство capacity равным значению свойства Count

В классе определены индексатор
this[ключ] {get; set;}

Объекты коллекции нумеруются, начиная с 0.

Получить предназначенную только для чтения коллекцию ключей или значений, хранимых в SortedList-коллекции, можно с помощью таких свойств:

public virtual ICollection Keys { get; }
public virtual ICollection Values { get; }

Порядок следования ключей и значений в полученных коллекциях отражает порядок SortedList-коллекции.

Подобно Hashtable-коллекции, SortedList-список хранит пары ключ/значение в форме структуры типа DictionaryEntry, но с помощью методов и свойств, определенных в классе SortedList, программисты обычно получают отдельный доступ к ключам и значениям.

```
// Демонстрация SortedList-коллекции.
using System ;
using System.Collections ;

class SLDemo
{
   public static void Main()
   {
      // Создаем упорядоченную коллекцию типа SortedList.
      SortedList sl = new SortedList();

      // Добавляем в список элементы.
      sl.Add ("здание",          "жилое помещение");
      sl.Add ("книга",                  "набор печатных слов");
      sl.Add ("яблоко",          "съедобный фрукт");
      sl.Add ("автомобиль",      "транспортное средство");

      // Добавляем элементы с помощью индексатора.
      sl ["трактор"] = "сельскохозяйственная машина";

      // Извлекаем элемент по ключу.
      string value = (string) sl ["автомобиль"];
      Console.WriteLine ("---по ключу: " + value);
      Console.WriteLine ("---по индексу: " + sl.GetByIndex(3));
```

```
        // Получаем коллекцию ключей.
        ICollection c = sl.Keys;

        // Используем ключи для получения значений.
        Console.WriteLine("\n--Содержимое списка, полученное "
                                              + "с помощью индексатора.");
        foreach (string str in c)
          Console.WriteLine(str + ": " + sl[str]);

        sl.Remove("трактор");                 // Удалить элемент

        // Отображаем список, используя целочисленные индексы.
        Console.WriteLine("\n--Содержимое списка, полученное "
                                + "с помощью целочисленных индексов.");
        for (int i = 0; i < sl.Count; i++)
          Console.WriteLine(sl.GetByIndex(i));

        // Отображаем целочисленные индексы элементов списка.
        Console.WriteLine("\n --Целочисленные индексы" +
                                " элементов списка.");
        foreach (string str in c)
          Console.WriteLine(str + ": " + sl.IndexOfKey(str));
      }
    }
```

Результаты выполнения этой программы таковы (отсортировано):

```
---по ключу: транспортное средство
---по индексу: сельскохозяйственная машина

--Содержимое списка, полученное с помощью индексатора.
автомобиль: транспортное средство
здание: жилое помещение
книга: набор печатных слов
трактор: сельскохозяйственная машина
яблоко: съедобный фрукт

--Содержимое списка, полученное с помощью целочисленных индексов.
транспортное средство
жилое помещение
набор печатных слов
съедобный фрукт

--Целочисленные индексы элементов списка.
автомобиль: 0
здание: 1
книга: 2
яблоко: 3
```

Класс Stack

Стек представляет собой список, добавление и удаление элементов к которому осуществляется по принципу "последним пришел — первым обслужен" (last-in, first-out — LIFO).

Стек — это динамическая коллекция, которая при необходимости увеличивается, чтобы принять для хранения новые элементы.

Стек реализует интерфейсы:

- ICollection,
- IEnumerable,
- ICloneable.

Методы, определенные в классе Stack

Метод	Описание
public virtual bool Contains(object value)	Возвращает значение true, если объект value содержится в вызывающем стеке. В противном случае возвращает значение false
public virtual void Clear()	Устанавливает свойство Count равным нулю, тем самым эффективно очищая стек
public virtual object Peek()	Возвращает элемент, расположенный в вершине стека, но не удаляет его
public virtual object Pop()	Возвращает элемент, расположенный в вершине стека, и удаляет его
public virtual void Push(object value)	Помещает объект value в стек
public static Stack Synchronized(Stack stk)	Возвращает синхронизированную версию stack-списка, переданного в параметре stk
public virtual object[ ] ToArray()	Возвращает массив, который содержит копии элементов вызывающего стека

// Демонстрация использования класса Stack.

```
using System;
using System. Collections;

class StackDemo
{
   public static void Main()
   {
      int a;
      Stack st = new Stack();

      st.Push(22);
      st.Push(65);
```

```
        st.Push(91);

        foreach (int i in st) Console.Write(i + " ");

        Console.WriteLine();
        try
        {
           a = (int)st.Pop();   Console.WriteLine(a);
           a = (int)st.Pop();   Console.WriteLine(a);
           a = (int)st.Pop();   Console.WriteLine(a);
           a = (int)st.Pop();   Console.WriteLine(a);
        }
        catch (InvalidOperationException)
        {
           Console.WriteLine("Стек пуст.");
        }
     }
  }
```

Хранение битов с помощью класса BitArray

Класс BitArray предназначен для поддержки коллекции битов. Поскольку его назначение состоит в хранении битов, а не объектов, то и его возможности отличаются от возможностей других коллекций. Тем не менее, класс BitArray поддерживает базовый набор средств коллекции посредством реализации интерфейсов iCollection, lEnumerable и ICloneable.

С помощью этого конструктора можно создать BitArray-коллекцию заданного размера:

```
public   BitArray (int size)
```

Здесь параметр size задает количество битов в коллекции, причем все они инициализируются значением false.

```
public  BitArray (bool[] bits)
```

В этом случае каждый элемент массива bits становится битом BitArray-коллекции. При этом каждый бит в коллекции соответствует элементу массива bits. Более того, порядок элементов массива bits аналогичен порядку битов в коллекции.

BitArray-коллекцию можно также создать из массива байтов. Для этого используйте следующий конструктор:

```
   public  BitArray (byte[] bits)
```

Здесь битами коллекции становится набор битов, содержащийся в массиве bits, причем элемент bits[0] определяет первые восемь битов, элемент bits[1] — вторые восемь битов и т.д.

Подобным образом можно создать Bit Array-коллекцию из массива других типов.

BitArray-коллекций могут быть индексированными. Каждый индекс соответствует определенному биту, причем нулевой индекс соответствует младшему биту.

В коллекции созданы методы для выполнения логических операций с битами.

```
// Демонстрация использования класса BitArray.
using System;
using System.Collections;

class BADemo
{
   public static void showBits(string rem, BitArray bits)
   {
      Console.WriteLine(rem);
      for (int i = 0; i < bits.Count; i++)
         Console.Write("{0, -6} ", bits[i]);
      Console.WriteLine("\n");
   }
   public static void Main()
   {
     BitArray ba = new BitArray(8);                  // все биты = false

     byte[] b = { 67 };
     BitArray ba2 = new BitArray(b);
     Console.Write("Бит 5 = {0, -6} ", ba2[5]);

      showBits("Исходное содержимое бит.коллекции ba:", ba);

     ba = ba.Not();
      showBits ("Содержимое коллек. ba после вызова Not():", ba);
     showBits("Содержимое коллекции ba2:", ba2);

     BitArray ba3 = ba.Xor(ba2);
     showBits("Результат операции ba XOR ba2:", ba3);
   }
}
```

2. Специализированные коллекции

В среде .NET Framework предусмотрена возможность создания специализированных коллекций, которые оптимизированы для работы с конкретными типами данных или для особого вида обработки.

Эти классы коллекций (они определены в пространстве имен System.Collections.Specialized):

Специализированная коллекция	Описание
CollectionsUtil	Коллекция, в которой игнорируется различия между строчным и прописным написанием символов в строках

HybridDictionary	Коллекция, в которой для хранения небольшого числа пар ключ/значение используется класс ListDictionary. Но при превышении коллекцией определеного размера для хранения: элементов автоматически используется класс Hashtable
ListDictionary	Коллекция, в которой для хранения пар ключ/значение используется связный список. Такую коллекцию рекомендуется использовать лишь при небольшом количестве элементов
NameValueCollection	Отсортированная коллекция пар ключ/значение, в которой как ключ, так и значение имеют тип string
StringCollection	Коллекция, оптимизированная для хранения строк
StringDictionary	Хеш-таблица, предназначенная для хранения пар ключ/значение, в которой как ключ, так и значение имеют тип string

В пространстве имен System.Collections также определены три абстрактных базовых класса, CollectionBase, ReadOnlyCollectionBase и DictionaryBase, которые предполагают создание производных классов и предназначены для использования в качестве отправной точки при разработке программистом собственных специализированных классов.

3. Способы сортировки объектов коллекции

Реализация интерфейса IComparable

Если необходимо отсортировать динамический массив (типа ArrayList) объектов, определенных пользователем (или, если вам понадобится сохранить эти объекты в коллекции типа SortedList), то вы должны сообщить коллекции информацию о том, как сравнивать эти объекты.

Один из способов — реализовать интерфейс IComparable.

В этом интерфейсе определен только один метод CompareTo(), который позволяет определить, как должно выполняться сравнение объектов соответствующего типа. Общий формат использования метода CompareTo() таков:

```
int   CompareTo (object obj)
```

Метод CompareTo() сравнивает вызывающий объект с объектом, заданным параметром obj. Чтобы отсортировать объекты коллекции в возрастающем порядке, этот метод (в вашей реализации) должен возвращать нуль, если сравниваемые объекты равны положительное значение, если вызывающий объект больше объекта obj, и отрицательное число, если вызывающий объект меньше объекта obj.

Для сортировки в убывающем порядке достаточно инвертировать результат описанного сравнения. Метод CompareTo() может сгенерировать

исключение типа ArgumentException, если тип объекта obj несовместим с вызывающим объектом.

```
//Реализация интерфейса IComparable.
using System;
using System.Collections;

class Inventory : IComparable
{
   string  name;
   double  cost;
   int     onhand;

   public Inventory(string n, double c, int h)
   {
      name    = n;
      cost    = c;
      onhand  = h;
   }

   public override string ToString()
   {
      return String.Format("{0, -16}Цена: {1,8:C} В наличии: {2}",
                           name,          cost,          onhand);
   }

   // Реализуем интерфейс IComparable.
   public int CompareTo(object obj)
   {
      Inventory b;
      b = (Inventory)obj;
      return this.name.CompareTo(b.name);
   }
}

class IcomparableDemo
{
   public static void Main()
   {
      ArrayList inv = new ArrayList();

      inv.Add(new Inventory("Плоскогубцы", 5.95, 3));
      inv.Add(new Inventory("Гаечные ключи", 8.29, 2));
      inv.Add(new Inventory("Молотки", 3.50, 4));

      Console.WriteLine("---Информация до сортировки:");
      foreach (Inventory i in inv)
         Console.WriteLine(" " + i);

      // Сортируем список.
      inv.Sort();

      Console.WriteLine("---Информация после сортировки:");
      foreach (Inventory i in inv)
```

```
        Console.WriteLine(" " + i);
    }
}
```

Здесь метод сортировки предоставил исходный класс Inventory.

А как быть, если класс не реализует интерфейс IComparable?
В этом случае можно воспользоваться интерфейсом IComparer.

Использование интерфейса IComparer

Интерфейс IComparer определяет метод сортировки, у которого ссылка на объект, реализующий метод сравнения, используется в качестве параметра. В интерфейсе IComparer определен только один метод Compare():

```
int   Compare (object obj1, object obj2)
```

Метод Compare() сравнивает объект obj1 с объектом obj2. Возвращаемое значение то же.

// Использование интерфейса Icomparer.

```
using System;
using System.Collections;

class CompInv : IComparer
{
   public int Compare (object obj1, object obj2)
   {
     return
      ((Inventory)obj1).name.CompareTo(((Inventory)obj2).name);
   }
}

class Inventory
{
   public string     name;
             double cost;
             int                onhand;

   public Inventory(string n, double c, int h)
   {
      name = n;  cost = c;  onhand = h;
   }

   public override string ToString()
   {
      return String.Format(" {0,-16}Цена: {1,8:C} В наличии: {2}",
                                    name,          cost,          onhand);
   }
}

class MailList
```

```
{
   public static void Main()
   {
      ArrayList inv = new ArrayList();

      // Добавляем элементы в список.
      inv.Add(new Inventory("Плоскогубцы", 5.95, 3));
      inv.Add(new Inventory("Гаечные ключи", 8.29, 2));
      inv.Add(new Inventory("Молотки", 3.50, 4));

      Console.WriteLine("Информация до сортировки:");
      foreach (Inventory i in inv)
         Console.WriteLine("   " + i);
      Console.WriteLine();

      // Сортируем список, используя интерфейс IComparer.
      inv.Sort(new CompInv());

      Console.WriteLine("Информация после сортировки:");
      foreach (Inventory i in inv)
         Console.WriteLine("   " + i);
   }
}
```

4. Сериализация коллекций

Все классы коллекций общего назначения и специализированные коллекции имеют атрибут [Serializable]. Следовательно, такие коллекции можно сохранять на диске с помощью одного обращения к методу Serialize(). Затем так же одним вызовом метода Deserialize() можно восстановить коллекцию в памяти.

Таблица коллекций и часто используемых методов

Коллекция	Добавить элемент	Извлечь с удалением	Прочитать элемент	Удалить элемент
ArrayList	al.Add(ob)		ob=al [i]	al.Remove(key)
Queue	q.Enqueu(ob)	q.Dequeu()	ob=q.Peek()	
Hashtable	ht.Add(ob) ht [key]=ob		ob=ht [key]	ht.Remove(key)
SortedList	sl.Add(ob) sl [key]=ob		ob=sl [key] sl.GetByIndex(i)	sl.Remove(key)
Stack	st.Push(ob)	st.Pop()	ob=st.Peek()	

РАБОТА С КАТАЛОГАМИ И ФАЙЛАМИ

В пространстве имен System. IO есть четыре класса, предназначенные для работы с физическими файлами и структурой каталогов на диске: Directory, File, Directorylnfo и Filelnfo. С их помощью можно выполнять создание, удаление, перемещение файлов и каталогов, а также получение их свойств.

Классы Directory и File реализуют свои функции через статические методы. Directorylnfo и Filelnfo обладают схожими возможностями, но они реализуются путем создания объектов соответствующих классов. Классы Directorylnfo и Filelnfo происходят от абстрактного класса FileSystemlnfo, который снабжает их базовыми свойствами, описанными в табл.1.

Таблица 1. Свойства класса FileSystemlnfo

Свойство	Описание
Attributes	Получить или установить атрибуты для данного объекта файловой системы. Для этого свойства используются значения перечисления FileAttributes
CreationTime	Получить или установить время создания объекта файловой системы
Exists	Определить, существует ли данный объект файловой системы
Extension	Получить расширение файла
Full Name	Возвратить имя файла или каталога с указанием полного пути
LastAccessTime	Получить или установить время последнего обращения к объекту файловой системы
LastWriteTime	Получить или установить время последнего внесения изменении и обьект файловой системы
Name	Возвратить имя файла. Это свойство доступно только для чтения. Для каталогов возвращает имя последнего каталога в иерархии, если это возможно. Если пет, возвращает полностью определенное имя

Класс Directory Info содержит элементы, позволяющие выполнять необходимые действия с каталогами файловой системы. Эти элементы перечислены в табл. 2.

Таблица 2. Элементы класса Directorylnfo

Элемент	Описание
Create, CreateSubDIrectory	Создать каталог или подкаталог по указанному пути в файловой системе
Delete	Удалить катапог со всем его содержимым
GetDirectories	Возвратить массив строк, представляющих все подкаталоги
GetFiIes	Получить файлы в текущем каталоге в виде массива объектов
класса Filelnfo	
MoveTo	Переместить каталог и все его содержимое на

	новый адрес в файловой системе
Parent	Возвратить родительский каталог

В листинге 1 приведен пример, в котором создаются два каталога, выводится информация о них и предпринимается попытка удаления каталога.

Листинг 1. Использование класса DirectoryInfo

```
using System;
using System.IO;
namespace ConsoleApplication1
{
   class Classl
   {
      static void DirInfo(DirectoryInfo di)
      {
         // Вывод информации о каталоге
         Console.WriteLine("===== Directory Info =====");
         Console.WriteLine("FullName: " + di.FullName);
         Console.WriteLine("Name: " + di.Name);
         Console.WriteLine("Parent: " + di.Parent);
         Console.WriteLine("Creation: " + di.CreationTime);
         Console.WriteLine("Attributes: " + di.Attributes);
         Console.WriteLine("Root: " + di.Root);
         Console.WriteLine("==========================");
      }

      static void Main()
      {
         DirectoryInfo di1 = new DirectoryInfo(@"c:\MyDir");
         DirectoryInfo di2 = new DirectoryInfo(@"c-\MyDir\temp");
         try
         {
            // Создать каталоги
            di1.Create();
            di2.Create();

            // Вывести информацию о каталогах
            DirInfo(di1);
            DirInfo(di2);
            // Попытаться удалить каталог
            Console.WriteLine("Попытка удалить {0}.", di1.Name);
            di1.Delete();
         }
         catch (Exception)
         {
            Console.WriteLine("Попытка не удалась ");
         }
      }
   }
}
```

Результат работы программы:
===== Directory Info =====
Full Name: c:\MyDir
Name: MyDir
Parent:
Creation: 30.04.2006 17:14:44
Attributes: Directory
Root: c:\
===== Directory Info =====
Full Name: c:\MyDir\temp
Name: temp
Parent: MyDir
Creation: 30.04.2006 17:14:44
Attributes: Directory
Root: c:\
Попытка удалить MyDir.
Попытка не удалась

Каталог не пуст, поэтому попытка его удаления не удалась. Впрочем, если использовать перегруженный вариант метода Delete с одним параметром, задающим режим удаления, можно удалить и непустой каталог:

di1.Delete(true);

Обратите внимание на свойство Attributes. Некоторые его возможные значения, заданные в перечислении FileAttributes приведены в табл. 3.

Таблица 3. Некоторые значения перечисления FileAttributes

Значение	Описание
Archi ve	Используется приложениями при выполнении резервного копирования, а в некоторых случаях - при удалении старых файлов
Compressed	Файл является сжатым
Di rectory	Объект файловой системы является каталогом
Encrypted	Файл является зашифрованным
Hidden	Файл является скрытым
Normal	Файл находится в обычном состоянии, и для него установлены любые другие атрибуты. Этот атрибут не может использоваться с другими атрибутами
OffIine	Файл, расположенный на сервере, кэширован в хранилище на клиентском компьютере. Возможно, что данные этого файла уже устарели
ReadOnly	Файл доступен только для чтения
System	Файл является системным

Листинг 2 демонстрирует использование класса FileInfo для копирования всех файлов с расширением jpg из каталога d:\foto в каталог d:\temp. Метод Exists позволяет проверить, существует ли исходный каталог.

Листинг 2. Копирование файлов

```
using System;
```

```
using System.IO;
namespace ConsoleApplicationi
{
   class Classl
   {
      static void Main()
      {
         try
         {
            string DestName = @"d:\temp\";
            DirectoryInfo dest = new DirectoryInfo(DestName);
            dest.Create();   // создание целевого каталога

            DirectoryInfo dir = new DirectoryInfo(@"d:\foto");
            if (!dir.Exists)    // проверка существования каталога
            {
               Console.WriteLine("Каталог " + dir.Name + " не существует");
               return;
            }
            FileInfo[] files = dir.GetFiles("*.jpg");  //список файлов
            foreach (FileInfo f in files)
               f.CopyTo(dest + f.Name);   // копирование файлов

            Console.WriteLine("Скопировано " +
            files.Length + " jpg-файлов");
         }
         catch (Exception e)
         {
            Console.WriteLine("Error: " + e.Message);
         }
      }
   }
}
```

Использование классов File и Directory аналогично, за исключением того, что их методы являются статическими и, следовательно, не требуют создания объектов.

Класс FileStream реализует эти элементы для работы с дисковыми файлами. Для определения режимов работы с файлом используются стандартные перечисления FileMode, FileAccess и FileShare. Значения этих перечислений приведены в табл. 11.2—11.4.

В листинге 3 представлен пример работы с файлом. В примере демонстрируются чтение и запись одного байта и массива байтов, а также позиционирование в потоке.

Листинг 3. Пример использования потока байтов

```
using System;
using System.IO;
class Class1
{
   static void Main()
```

```
    {
      FileStream f = new FileStream("test.txt",
                                    FileMode.Create, FileAccess.ReadWrite);

      f.WriteByte(100);    // в начало файла записывается число 100
      byte[] x = new byte[10];
      for (byte i = 0; i < 10; ++i)
      {
        x[i] = (byte)(10 - i);
        f.WriteByte(i);       // записывается 10 чисел от 0 до 9
      }

      f.Write(x, 0, 5);                    // записывается 5 элементов массива

      byte[] y = new byte[20];

      f.Seek(0, SeekOrigin.Begin);         // текущий указатель - на начало
      f.Read(y, 0, 20);                    // чтение из файла в массив

      foreach (byte elem in y)
        Console.Write(" " + elem);

      Console.WriteLine();

      f.Seek(5, SeekOrigin.Begin);         // текущий указатель - на 5-й элемент
      int a = f.ReadByte();                // чтение 5-го элемента
      Console.WriteLine(a);

      a = f.ReadByte();              // чтение 6-го элемента
      Console.WriteLine(a);

      Console.WriteLine("Текущая позиция в потоке" + f.Position);
      f.Close();
    }
}
```

В листинге 4 создается текстовый файл, в который записываются две строки. Вторая строка формируется из преобразованных численных значений переменных и поясняющего текста. Содержимое файла можно посмотреть в любом текстовом редакторе. Файл создается в том же каталоге, куда среда записывает исполняемый файл. По умолчанию это каталог ...\ConsoleApplication1\bin\Debug.

Листинг 4. Вывод в текстовый файл

```
using System;
using System.IO;
class Class2
{
  static void Main()
  {
    try
    {
```

```
            StreamWriter f = new StreamWriter("text.txt");
            f.WriteLine("Вывод в текстовый файл:");
            double a = 12.234;
            int b = 29;
            f.WriteLine(" a = {0.6:C}    b = {1.2:X}", a, b);
            f.Close();
        }
        catch (Exception e)
        {
            Console.WriteLine("Error: " + e.Message);
            return;
        }
    }
}
```

В листинге 5 файл, созданный в предыдущем листинге, выводится на экран.

Листинг 5. Чтение текстового файла

```
using System;
using System.IO;
class Class4
{
    static void Main()
    {
        try
        {
            StreamReader f = new StreamReader("text.txt");
            string s = f.ReadToEnd();
            Console.WriteLine(s);
            f.Close();
        }
        catch (FileNotFoundException e)
        {
            Console.WriteLine(e.Message);
            Console.WriteLine(" Проверьте правильность имени файла! ");
            return;
        }
        catch (Exception e)
        {
            Console.WriteLine("Error: " + e.Message);
            return;
        }
    }
}
```

В этой программе весь файл считывается за один прием с помощью метода ReadToEnd(). Чаще возникает необходимость считывать файл построчно, такой пример приведен в листинге 6. Каждая строка при выводе предваряется номером.

Листинг 6. Построчное чтение текстового файла

```
using System;
using System.IO;
class Class5
{
   static void Main()
   {
      try
      {
         StreamReader f = new StreamReader("text.txt");
         string s;
         long i = 0;

         while ((s = f.ReadLine()) != null)
            Console.WriteLine("{0}:  {1}", ++i, s);
         f.Close();
      }
      catch (FileNotFoundException e)
      {
         Console.WriteLine(e.Message);
         Console.WriteLine("Проверьте правильность имени файла!");
         return;
      }
      catch (Exception e)
      {
         Console.WriteLine("Error: " + e.Message);
         return;
      }
   }
}
```

Пример преобразования чисел, содержащихся в текстовом файле, в их внутреннюю форму представления приведен в листинге 7. В программе вычисляется сумма чисел в каждой строке.

На содержимое файла накладываются весьма строгие ограничения: числа должны быть разделены ровно одним пробелом, после последнего числа в строке пробела быть не должно, файл не должен заканчиваться символом перевода строки. Методы разбиения строки и преобразования в целочисленное представление рассматривались ранее.

Листинг 7. Преобразования строк в числа

```
using System;
using System.IO;
class Class6
{
   static void Main()
   {
      try
      {
         StreamReader f = new StreamReader("numbers.txt");
         string s;
         const int n = 20;
```

```
            int[] a = new int[n];
            string[] buf;

            while ((s = f.ReadLine()) != null)
            {
               buf = s.Split(' ');
               long sum = 0;

               for (int i = 0; i < buf.Length; ++i)
               {
                  a[i] = Convert.ToInt32(buf[i]);
                  sum += a[i];
               }
               Console.WriteLine("{0} сумма: {1}", s, sum);
            }
            f.Close();
         }
         catch (FileNotFoundException e)
         {
            Console.WriteLine(e.Message);
            Console.WriteLine(" Проверьте правильность имени файла!");
            return;
         }
         catch (Exception e)
         {
            Console.WriteLine("Error: " + e.Message);
            return;
         }
      }
   }
```

Результат работы программы:

```
12 4  сумма: 7
3 44 -3 6  сумма: 50
8 1 1  сумма: 10
```

В листинге 8 приведен пример формирования двоичного файла. В файл записывается последовательность вещественных чисел, а затем для демонстрации произвольного доступа третье число заменяется числом 8888.

Листинг 8. Формирование двоичного файла

```
using System;
using System.IO;
class Class7
{
   static void Main()
   {
      try
      {
         BinaryWriter fout = new BinaryWriter(
         new FileStream(@"D:\C#\binary", FileMode.Create));
```

```
            double d = 0;

            while (d < 4)
            {
               fout.Write(d);
               d += 0.33;
            }
            fout.Seek(16, SeekOrigin.Begin);  // второй элемент файла
            fout.Write(8888d);
            fout.Close();
         }
         catch (Exception e)
         {
            Console.WriteLine("Error: " + e.Message);
            return;
         }
      }
   }
```

При создании двоичного потока в него передается объект базового потока. При установке указателя текущей позиции в файле учитывается длина каждого значения типа doubl e — 8 байт.

Попытка просмотра сформированного программой файла в текстовом редакторе весьма медитативная, но не информативная, поэтому в листинге 9 приводится программа, которая с помощью экземпляра BinaryReader считывает содержимое файла в массив вещественных чисел, а затем выводит этот массив на экран. При чтении принимается во внимание тот факт, что метод ReadDouble при обнаружении конца файла генерирует исключение EndOfStreamException. Поскольку в данном случае это не ошибка, тело обработчика исключений пустое.

Листинг 9. Считывание двоичного файла

```
using System;
using System.IO;
class Class8
{
   static void Main()
   {
      try
      {
         FileStream f = new FileStream(@"D:\C#\binary", FileMode.Open);

         BinaryReader fin = new BinaryReader(f);
         long n = f.Length / 8;       // количество чисел в файле
         double[] x = new double[n];
         long i = 0;

         try
         {
            while (true) x[i++] = fin.ReadDouble();          // чтение
         }
         catch (EndOfStreamException e) { }
```

```
            foreach (double d in x)
               Console.Write("    " + d); // вывод

            fin.Close();
            f.Close();
        }
        catch (FileNotFoundException e)
        {
            Console.WriteLine(e.Message);
            Console.WriteLine("Проверьте правильность имени файла!");
            return;
        }
        catch (Exception e)
        {
            Console.WriteLine("Error:  " + e.Message);
            return;
        }
    }
}
```

Результат работы программы:

```
0   0,33   8888   0,99   1,32   1,65   1,98  2,31  2,64  2,97  3,3  3,63  3.96
```

ЭЛЕМЕНТЫ УПРАВЛЕНИЯ WINDOWS FORMS ОБЩИЕ СВЕДЕНИЯ. УПРАВЛЕНИЕ ХОДОМ ВЫПОЛНЕНИЯ ПРОГРАММЫ

- Кнопки (Button),
- флажки (CheckBox),
- переключатели (RadioButton),
- меню,
- инструменты

Элементы управления — это классы, обеспечивающие взаимодействие между пользователем и программой.

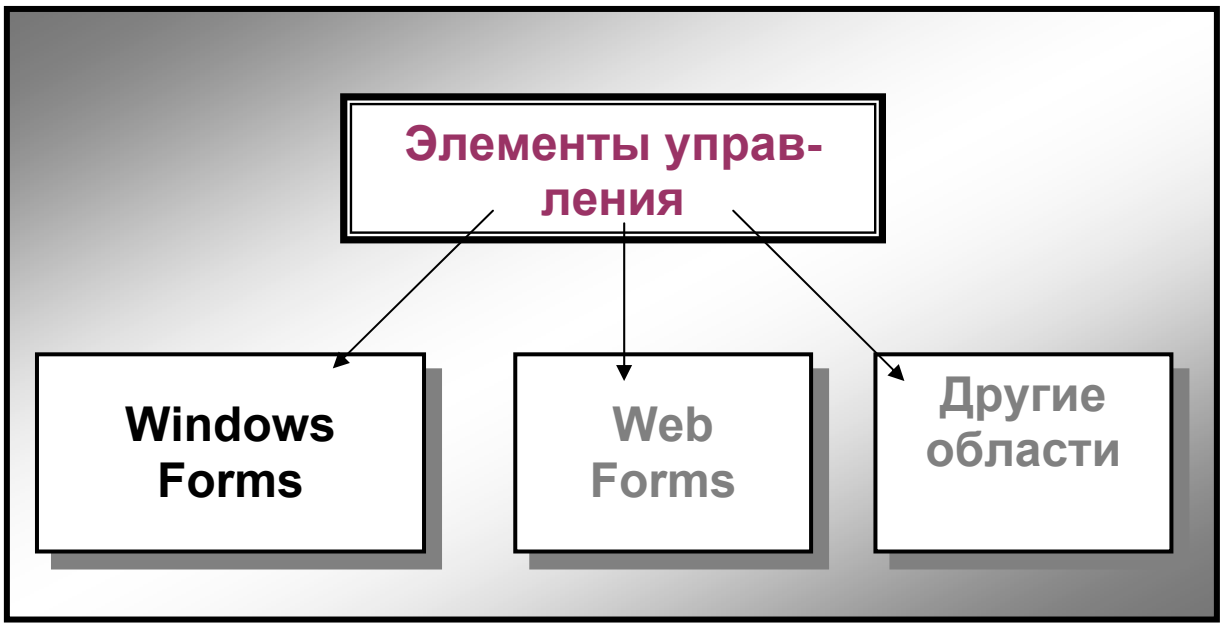


рис.1

1. Элементы управления Windows Forms

В режиме конструирования программы можно проектировать интерфейс, используя различные окна.

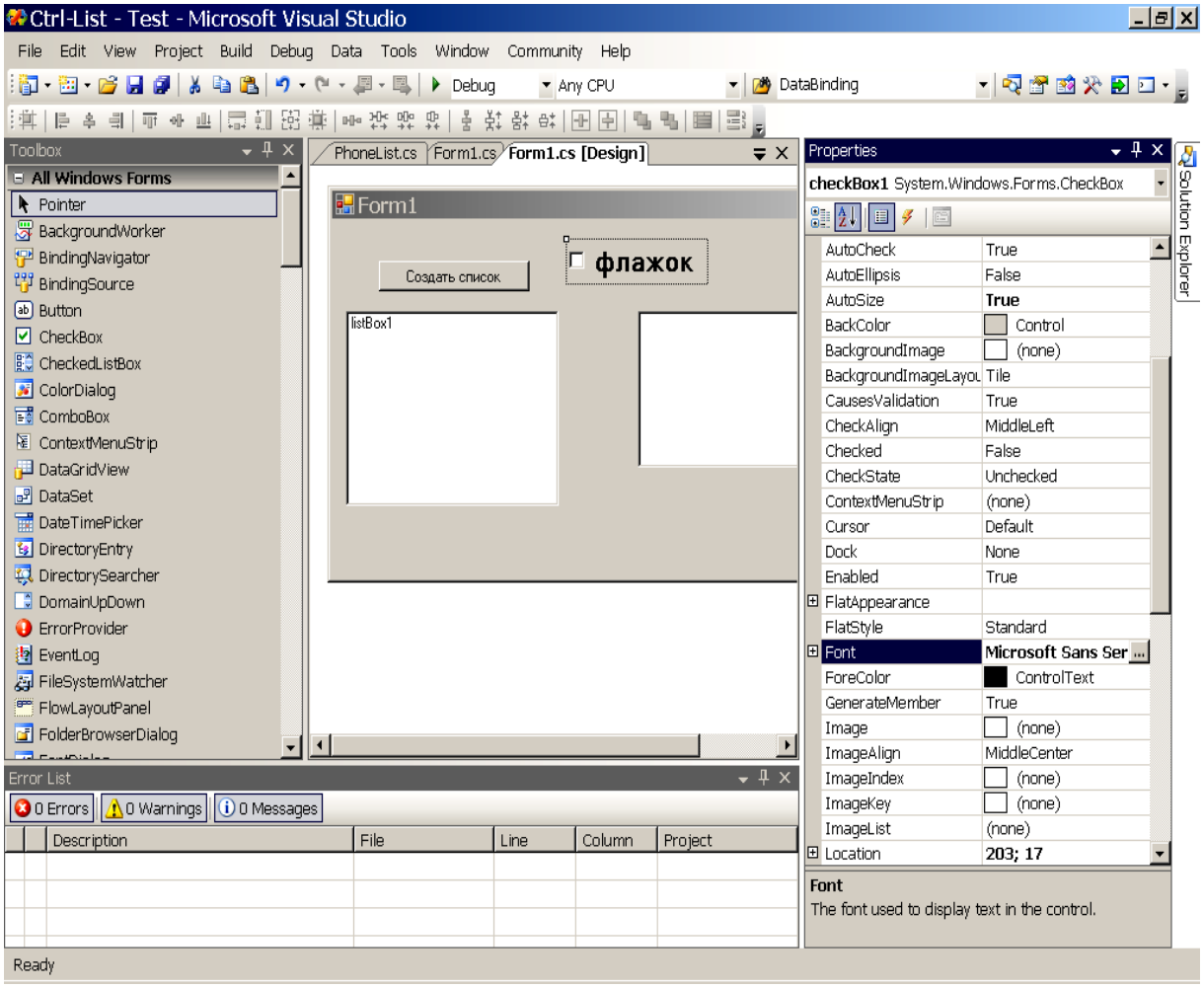


рис.2

Почти все элементы управления наследуют класс Component, поэтому они являются компонентами. Но отображаются в форме только те ЭУ, которые прямо или косвенно наследуют класс Control. Этот класс реализует основную функциональность для отображения и манипулирования элементами. Такие ЭУ будем называть **общими ЭУ**.

Элементы управления, не наследующие класс Control, отображаются только на специальной панели компонентов. Такие ЭУ будем называть **компонентами**. Примером компонентов являются диалоговые окна.

На панели компонентов отображаются так же и некоторые элементы управления, которые наследуют класс Control. Это контейнеры пунктов меню (главного и контекстно-зависимого), инструментов и элементов строки состояния. В форме отображаются их коллекции.

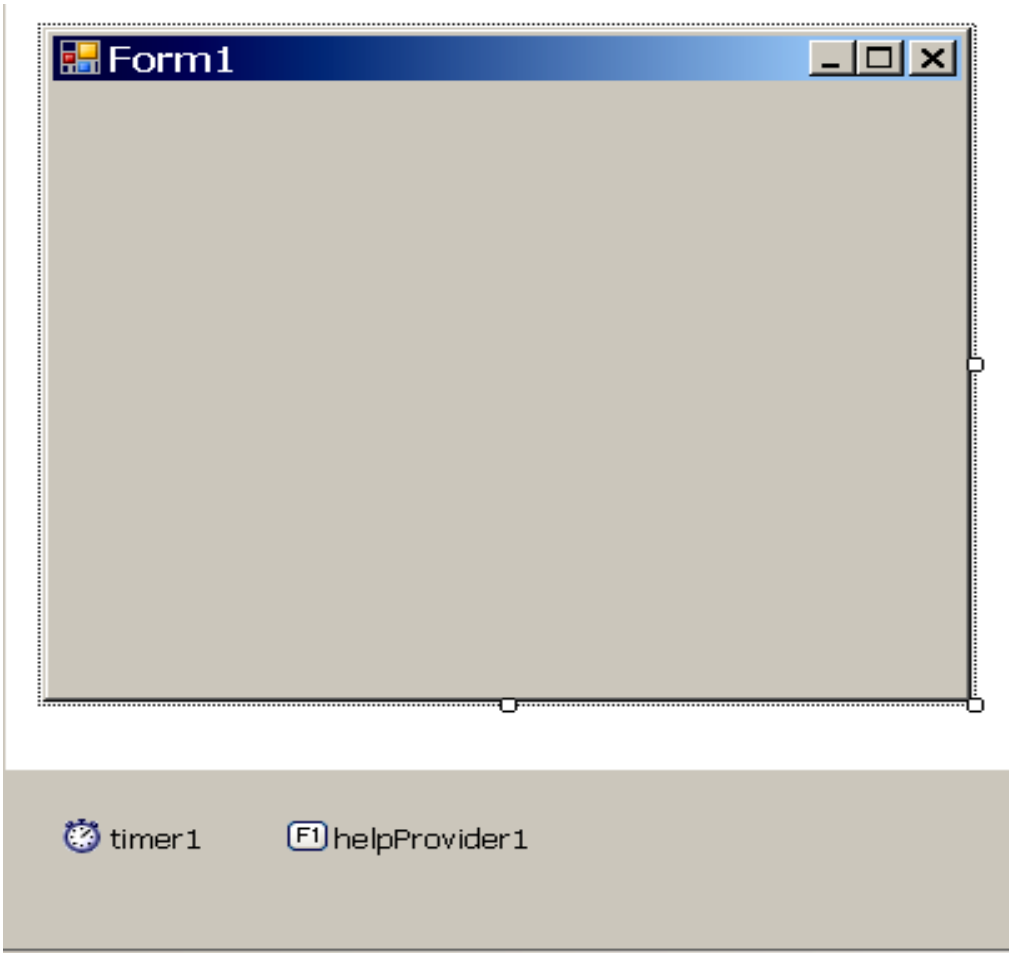


рис.3

Ниже рассмотрены элементы управления согласно общему функциональному назначению.

Функция	Элемент управления	Описание
Редактирование текста	TextBox	Отображает текст, введенный во время разработки, который может редактироваться пользователями во время выполнения, а также может быть изменен программными средствами.
	RichTextBox	Позволяет представлять текст в простом текстовом формате или в формате RTF.
	MaskedTextBox - 2005	Позволяет автоматически форматировать введенные данные. Проверяет допустимость формата пользовательского ввода.

	DataGridView - 2005 (DataGrid-2003)	Вывод данных в таблицу и их редактирование.
Отображение текста только для чтения	Label	Отображает текст, недоступный для непосредственного редактирования пользователем.
	StatusStrip – 2005 (StatusBar – 2003)	Отображает сведения о текущем состоянии приложения в окне, заключенном в рамку, обычно в нижней части родительской формы.
Выбор из списка	ListBox	Отображает список текстовых и графических элементов (значков).
	CheckedListBox	Отображает список с полосой прокрутки, состоящий из элементов с флажками.
	ComboBox	Отображает раскрывающийся список.
	DomainUpDown	Отображает список текстовых элементов, который можно прокручивать с помощью кнопок со стрелками.
	NumericUpDown	Отображает список чисел, который можно прокручивать с помощью кнопок со стрелками.
	ListView	Отображает элементы в одном из четырех представлений: только текст, текст с маленькими значками, текст с большими значками и подробности.
	TreeView	Отображает иерархическую структуру объектов с узлами, которые кроме текста могут включать флажки и значки.
Вывод и хране-	PictureBox	Отображает в рамке

ние графики		графические файлы, например точечные рисунки или значки.
	ImageList	Компонент. Служит местом хранения изображений. Элементы управления ImageList и хранящиеся в них рисунки могут повторно использоваться в других приложениях.
Задание значений	CheckBox	Отображает флажок и надпись для текста. В основном используется для задания параметров.
	RadioButton	Выводит кнопку, которая может быть включена или выключена.
	Trackbar	Позволяет задавать значения на шкале, перемещая по ней ползунок.
	HScrollBar	Горизонтальная линейка прокрутки.
	VScrollBar	Вертикальная линейка прокрутки.
Установка даты	DateTimePicker	Выводит графический календарь, позволяющий пользователю выбрать дату или время.
	MonthCalendar	Выводит графический календарь, позволяющий пользователю выбрать диапазон дат.
Диалоговые окна (компоненты)	ColorDialog	Отображает диалоговое окно выбора цвета, позволяющее задать цвет элемента интерфейса.
	FontDialog	Отображает диалоговое окно для задания шрифта и его атрибутов.
	OpenFileDialog	Отображает диалоговое окно для поиска и выбора файла.

	SaveFileDialog	Отображает диалоговое окно для сохранения файла.
	PrintDialog	Отображает диалоговое окно для выбора принтера и задания его атрибутов.
	PrintPreviewDialog	Отображает диалоговое окно, показывающее, как будет выглядеть напечатанный объект PrintDocument.
	PageSetupDialog	Предоставляет диалоговое окно, которое позволяет пользователям изменять параметры страницы, в том числе поля и ориентацию листа.
	FolderBrowserDialog	Предоставляет обычное диалоговое окно, которое позволяет пользователю выбирать папку.
Элементы управления меню	MenuStrip - 2005 (MainMenu-2003)	Контейнер для структур меню типа ToolStripMenuItem.
	ContextMenuStrip - 2005 (ContextMenu-2003)	Реализует контекстно-зависимое меню, появляющееся при щелчке объекта правой кнопкой мыши. Контейнер объектов типа ToolStripMenuItem.
Команды	Button	Используется для запуска, остановки или прерывания процесса.
	LinkLabel	Отображает текст как веб-ссылку и вызывает событие, когда пользователь щелкает этот текст. Обычно такой текст является ссылкой на другое окно или на веб-узел.
	NotifyIcon	Отображает значок в области уведомлений панели задач, соответствующий приложе-

		нию, выполняемому в фоновом режиме.
	ToolStrip - 2005 (ToolBar - 2003)	Контейнер. Используется для создания коллекции кнопок панели инструментов, коллекции пунктов меню и строк состояния. Является базовым для MenuStrip и StatusStrip.
Группировка других элементов управления	Panel	Группирует набор элементов управления в прокручиваемую рамку без надписи.
	GroupBox	Группирует набор элементов управления (например, переключателей) в непрокручиваемую рамку с надписью.
	TabControl	Страница с вкладками для эффективной организации доступа к сгруппированным объектам.
	SplitContainer - 2005	Три элемента в одном. Состоит из двух панелей с линейкой между ними, представленной двумя прямыми линиями. Линейку можно перемещать влево и вправо (или вверх-вниз). Пример: Explorer.
Вывод подсказки	HelpProvider	Компонент. Связывает элементы управления с темами подсказки.
	ToolTip	Предоставляет небольшое прямоугольное всплывающее окно, которое содержит краткое описание назначения элемента управления; окно отображается, когда указатель мыши располагается в области, за-

		нимаемой элементом управления.
	ErrorProvider	Компонент. Связывает элемент управления с индикатором, и высвечивает индикатор в случае обнаружения ошибки.
Печать документа	PrintDocument	Определяет повторно используемый объект, отправляемый для вывода на принтер.
	PrintPreviewControl	Предоставляет начальную часть предварительного просмотра печати без каких бы то ни было диалоговых окон или кнопок. Большинство объектов PrintPreviewControl находятся в объектах PrintPreviewDialog, но не обязательно.
	ProgressBar	Полоса, растущая по длине в соответствии с длительностью какого-либо процесса.
	Splitter	Позволяет во время выполнения программы менять размеры элементов управления, присоединенных к краям элемента управления Splitter. Когда пользователь помещает указатель мыши на элемент управления Splitter, указатель меняет свой вид, что служит признаком того, что размеры элементов управления, прикрепленных к элементу управления Splitter, могут быть изменены.

	Timer	Компонент. Генерирует событие через определенные интервалы времени.

Другие элементы управления, дополнительно включенные в VS 2005.

BackgroundWorker	Компонент. Executes an operation on a separate thread.
BindingNavigator	Компонент. Наследник ToolStrip. Создает панель инструментов для навигации с кнопками: влево, вправо, в конец, в начало, удалить и др.
BindingSource	Компонент. Инкапсулирует источник данных для формы.
DataSet	Источник данных для DataGridView. Таблица БД.
DirectoryEntry	Компонент. Представляет узел или объект в хранилище Active Directory. Используется для связывания объекта.
DirectorySearcher	Компонент. Поиск в Active Directory объекта с заданными атрибутами.
EventLog	Компонент. Предоставляет доступ к журналу событий Windows.
FileSystemWatcher	Компонент. Listens to the file system change notifications and raises events when a directory, or file in a directory, changes.
FlowLayoutPanel	Контейнер. Управляет расположением (раскладкой) содержащихся на панели (в контейнере) элементов управления (форм).
TableLayoutPanel	Контейнер. Использует сеточную структуру (таблицу) для управления раскладкой элементов управления панели.
MessageQueue	Очередь сообщений. Используется в коммуникациях (Интернет,...).
PerfomanceCounter	Компонент. Представляет счетчик оборудования Windows NT.
Process	Компонент. Обслуживает доступ к локальным и удаленным процессам, а так же разрешает пользователю запускать и останавливать локальные системные процессы.
PropertyGrid	Обеспечивает пользователя интерфейсом (окном как в VS) для просмотра свойств объекта.
SerialPort	Компонент. Предоставляет доступ к свойствам последовательного порта.
ServiceController	Компонент. Представляет сервисы Windows и позволяет запускать и останавливать их, манипулировать ими или дает информацию о сервисах.
ToolStripContainer	Используется для стыковки элементов

	управления, основанных на ToolStrip
WebBrowser	Разрешает использование навигации Web-страниц внутри формы.
ReportViewer	Управляющий элемент. Инкапсулирует методы и свойства, используемые для управления просмотром отчета. Содержит панель инструментов навигации.

Итого: 68 элементов + ЭУ ADO.NET и Web.

Следует помнить, что помимо элементов управления форм Windows в формы Forms можно добавлять элементы ActiveX, а также пользовательские элементы.Если в списке перечисленных элементов управления отсутствует нужный, его можно создать самостоятельно.

Общие параметры элементов управления

Класс Control является базовым для всех общих элементов управления, форм и контейнеров. Его свойства:

Размер и размещение:

Width и **Hight** - ширина и высота ЭУ. Определяют размер ЭУ.

Size – структура. Возвращает или задает высоту и ширину элемента управления. Поля: Width и Hight (ширина и высота).

Location – задает или возвращает значение структуры Point с координатами X и Y левого верхнего угла элемента управления относительно левого верхнего угла контейнера.

Bounds – свойство возвращает объект Rectangle, представляющий экранную область (включая заголовок и полосы прокрутки), занятую элементом управления.

ClientSize – структура Size – клиентская область без заголовка и полос прокрутки; меню и строки инструментов включаются.

Также используются свойства, возвращающие отд.значения:

Left (возвращает или задает координату по оси X левого края элемента управления)

Top (координата по оси Y верхнего края элемента управления)

Right (возвращает расстояние от правого края элемента управления до левого края контейнера)

Bottom (возвращает расстояние между нижним краем элемента управления и верхним краем клиентской области контейнера)

Dock – определяет к какой грани родительского элемента должен пристыковываться данный элемент.

```
richTextBox1.Dock = DockStyle.Fill;   // размер ЭУ = размеру родительского
```

Все значения перечисления DockStyle: Top, Botton, Richt, Left, None, Fill.

Anchor (якорь) – прикрепляет на постоянном расстоянии.

Другие свойства:

Name – возвращает или задает имя элемента (ссылка на объект).

Text – возвращает или задает текст, как правило, отображаемый в ЭУ.

BackColor, **ForeColor** – цвет фона и переднего плана ЭУ.

BackGroundImage – графический образ для фона ЭУ.

BackGroundImageLayout – способ отображения графического образа в ЭУ.

Font – шрифт и его параметры.

Visible – сделать элемент видимым или невидимым.

TabIndex – порядковый номер ОЭУ в последовательности перемещения по клавише Tab.

ContextMenuStrip – связывает ОЭУ с контекстно-зависимым меню.

Члены класса Control:

Открытые конструкторы

Control - конструктор	Перегружен. Инициализирует новый экземпляр класса Control.

Открытые свойства

AccessibilityObject	Возвращает AccessibleObject, назначенный элементу управления.
AccessibleDefaultActionDescription	Возвращает или задает описание выполняемого по умолчанию действия элемента управления для использования клиентскими приложениями со специальными возможностями.
AccessibleDescription	Возвращает или задает описание элемента управления, используемого клиентскими приложениями со специальными возможностями.
AccessibleName	Возвращает или задает имя элемента управления, используемого клиентскими приложениями со специальными возможностями.
AccessibleRole	Возвращает или задает доступную роль элемента управления.
AllowDrop	Возвращает или задает значение, указывающее, может ли элемент управления принимать данные, перемещенные на него пользователем.
Anchor	Возвращает или задает значение, указывающее, какие края элемента управления будут привязаны к краям контейнера.
BackColor	Возвращает или задает цвет фона элемента управления.
BackgroundImage	Возвращает или задает фоновое

	изображение, выводимое на элементе управления.
BindingContext	Возвращает или задает BindingContext для элемента управления.
Bottom	Возвращает расстояние между нижним краем элемента управления и верхним краем клиентской области контейнера.
Bounds	Возвращает или задает размер и местоположение элемента управления, включая неклиентские элементы.
CanFocus	Возвращает значение, показывающее, может ли элемент управления получать фокус.
CanSelect	Возвращает значение, показывающее, доступен ли элемент управления для выделения.
Capture	Возвращает или задает значение, определяющее, была ли мышь захвачена элементом управления.
CausesValidation	Возвращает или задает значение, показывающее, вызывает ли элемент управления проверку любого элемента управления, требующего проверки при получении фокуса.
ClientRectangle	Возвращает прямоугольник, задающий клиентскую область элемента управления.
ClientSize	Возвращает или задает высоту и ширину клиентской области элемента управления.
CompanyName	Возвращает название организации или имя создателя приложения, содержащего элемент управления.
Container (унаследовано от Component)	Возвращает IContainer, содержащий Component.
ContainsFocus	Возвращает значение, указывающее, имеет ли элемент управления или один из его дочерних элементов фокус ввода.
ContextMenu	Возвращает или задает меню быстрого вызова, связанное с элементом управления.
Controls	Возвращает коллекцию элементов управления, содержащихся в элементе управления.
Created	Возвращает значение, показывающее, был ли создан элемент

	управления.
Cursor	Возвращает или задает курсор, отображаемый, когда указатель мыши находится на элементе управления.
DataBindings	Возвращает привязки данных для этого элемента управления.
S DefaultBackColor	Возвращает используемый по умолчанию цвет фона элемента управления.
S DefaultFont	Возвращает шрифт элемента управления, используемый по умолчанию.
S DefaultForeColor	Возвращает цвет изображения элемента управления, используемый по умолчанию.
DisplayRectangle	Возвращает прямоугольник, предоставляющий отображаемую область элемента управления.
Disposing	Возвращает значение, показывающее, находится ли элемент управления в процессе удаления.
Dock	Возвращает или задает край родительского контейнера, к которому прикрепляется элемент управления.
Enabled	Возвращает или задает значение, показывающее, имеет ли элемент управления возможность отвечать на действия пользователя.
Focused	Возвращает значение, показывающее, имеет ли элемент управления фокус ввода.
Font	Возвращает или задает шрифт текста, отображаемого элементом управления.
ForeColor	Возвращает или задает основной цвет элемента управления.
Handle	Возвращает дескриптор окна, к которому привязан элемент управления.
HasChildren	Возвращает значение, определяющее, содержит ли элемент управления один или несколько дочерних элементов.
Height	Возвращает или задает высоту элемента управления .
ImeMode	Возвращает или задает режим редактора методов ввода (IME) элемента управления.
InvokeRequired	Возвращает значение, показы-

	вающее, следует ли вызывающему оператору обращаться к методу invoke во время вызовов метода из элемента управления, так как вызывающий оператор находится не в том потоке в котором был создан элемент управления.
IsAccessible	Возвращает или задает значение, показывающее, является ли элемент управления видимым для приложений со специальными возможностями.
IsDisposed	Возвращает значение, показывающее, был ли удален элемент управления.
IsHandleCreated	Возвращает значение, показывающее, имеется ли у элемента управления связанный с ним дескриптор.
Left	Возвращает или задает координату по оси X левого края элемента управления (в точках).
Location	Возвращает или задает координаты левого верхнего угла элемента управления относительно левого верхнего угла контейнера.
S ModifierKeys	Возвращает значение, показывающее, какие из управляющих клавиш (SHIFT, CTRL и ALT) нажаты в данный момент.
S MouseButtons	Возвращает значение, показывающее, какая из кнопок мыши нажата в данный момент.
S MousePosition	Возвращает позицию указателя мыши в координатах экрана.
Name	Возвращает или задает имя элемента управления.
Parent	Возвращает или задает родительский контейнер элемента управления.
ProductName	Возвращает имя продукта сборки, содержащей элемент управления.
ProductVersion	Возвращает версию сборки, содержащей элемент управления.
RecreatingHandle	Возвращает значение, показывающее, происходит ли в данный момент повторное создание дескриптора элементом управления.
Region	Возвращает или задает область

	окна, связанную с элементом управления.
Right	Возвращает расстояние от правого края элемента управления до левого края контейнера.
RightToLeft	Возвращает или задает значение, показывающее, выровнены ли записи элемента управления для поддержки языков, использующих шрифты с написанием справа налево.
Site	Переопределен. Возвращает или задает подложку элемента управления.
Size	Возвращает или задает высоту и ширину элемента управления.
TabIndex	Возвращает или задает последовательность перехода элемента управления внутри контейнера.
TabStop	Возвращает или задает значение, показывающее, можно ли передать фокус данному элементу управления при помощи клавиши TAB.
Tag	Возвращает или задает объект, содержащий данные элемента управления.
Text	Возвращает или задает текст, связанный с данным элементом управления.
Top	Возвращает или задает координату по оси Y верхнего края элемента управления (в точках).
TopLevelControl	Возвращает родительский элемент управления, не имеющий другого родительского элемента Windows Forms. Как правило, это самая внешняя Form, в которой содержится элемент управления.
Visible	Возвращает или задает значение, определяющее, отображается ли элемент управления.
Width	Возвращает или задает ширину элемента управления.

Итого: 67 свойств

Открытые методы

BeginInvoke	Перегружен. Выполняет делегат асинхронно на том потоке, на котором был создан основной дескриптор элемента управле-

	ния.
BringToFront	Помещает элемент управления в начало z-последовательности.
Contains	Извлекает значение, показывающее, является ли указанный элемент управления дочерним элементом.
CreateControl	Вызывает принудительное создание элемента управления, включая создание дескриптора и дочерних элементов.
CreateGraphics	Создает объект Graphics для элемента управления.
CreateObjRef (унаследовано от MarshalByRefObject)	Создает объект, который содержит всю необходимую информацию для создания прокси-сервера, используемого для коммуникации с удаленными объектами.
Dispose (унаследовано от Component)	Перегружен. Освобождает ресурсы, используемые объектом Component.
DoDragDrop	Начинает операцию перетаскивания.
EndInvoke	Извлекает возвращаемое значение асинхронной операции, предоставленное переданным объектом IAsyncResult.
Equals (унаследовано от Object)	Перегружен. Определяет, равны ли два экземпляра Object.
FindForm	Извлекает форму, на которой находится элемент управления.
Focus	Задает фокус ввода элементу управления.
S FromChildHandle	Извлекает элемент управления, содержащий указанный дескриптор.
S FromHandle	Возвращает элемент управления, связанный в данный момент с указанным дескриптором.
GetChildAtPoint	Извлекает дочерний элемент управления, имеющий указанные координаты.
GetContainerControl	Возвращает следующий ContainerControl в цепочке родительских элементов управления данного элемента.
GetHashCode (унаследовано от Object)	Служит хеш-функцией для конкретного типа, пригоден для использования в алгоритмах хеширования и структурах данных, например в хеш-таблице.
GetLifetimeService (унаследовано от MarshalByRefObject)	Извлекает служебный объект текущего срока действия, который управляет средствами срока действия данного экземпляра.
GetNextControl	Извлекает следующий или предыдущий элемент управления в последовательности перехода дочерних элементов.

GetType (унаследовано от Object)	Возвращает Type текущего экземпляра.
Hide	Скрывает элемент управления.
InitializeLifetimeService (унаследовано от MarshalByRefObject)	Получает служебный объект срока действия, для управления средствами срока действия данного экземпляра.
Invalidate	Перегружен. Объявляет недопустимой конкретную область элемента управления и вызывает отправку сообщения изображения элементу управления.
Invoke	Перегружен. Выполняет делегат в том потоке, которому принадлежит основной дескриптор окна элемента управления.
IsMnemonic	Определяет, является ли указанный знак назначенным знаком, присвоенным элементу управления в заданной строке.
PerformLayout	Перегружен. Заставляет элемент управления применять логику макета к дочерним элементам управления.
PointToClient	Вычисляет расположение указанной точки экрана в координатах клиента.
PointToScreen	Вычисляет расположение указанной клиентской точки в координатах экрана.
PreProcessMessage	Выполняет предварительную обработку входящих сообщений в цикле обработки сообщений переди их отправкой.
RectangleToClient	Вычисляет размер и расположение указанного прямоугольника экрана в координатах клиента.
RectangleToScreen	Вычисляет размер и расположение указанной клиентской области в координатах экрана.
Refresh	Принудительно вызывает элемент управления, который в результате делает недоступной свою клиентскую область и немедленно перерисовывает себя и все дочерние элементы.
ResetBackColor	Восстанавливает значение по умолчанию свойства BackColor.
ResetBindings	Восстанавливает значение по умолчанию свойства DataBindings.
ResetCursor	Восстанавливает значение по умолчанию свойства Cursor.
ResetFont	Восстанавливает значение по умолчанию свойства Font.
ResetForeColor	Восстанавливает значение по умолчанию свойства ForeColor.

ResetImeMode	Восстанавливает значение по умолчанию свойства ImeMode.
ResetRightToLeft	Восстанавливает значение по умолчанию свойства RightToLeft.
ResetText	Восстанавливает значение по умолчанию свойства Text.
ResumeLayout	Перегружен. Восстанавливает обычную логику макета.
Scale	Перегружен. Масштабирует элемент управления и любые его дочерние элементы.
Select	Перегружен. Активирует элемент управления.
SelectNextControl	Активирует следующий элемент управления.
SendToBack	Помещает элемент управления в конец z-последовательности.
SetBounds	Перегружен. Задает границы элемента управления.
Show	Отображает элемент управления.
SuspendLayout	Временно приостанавливает логику макета для элемента управления.
ToString (унаследовано от Object)	Возвращает String, который представляет текущий Object.
Update	Вызывает перерисовку элементом управления недопустимых областей клиентской области.

Открытые события

BackColorChanged	Возникает при изменении значения свойства BackColor.
BackgroundImageChanged	Возникает при изменении значения свойства BackgroundImage.
BindingContextChanged	Возникает при изменении значения свойства BindingContext.
CausesValidationChanged	Возникает при изменении значения свойства CausesValidation.
ChangeUICues	Возникает при изменении фокуса или клавиатурных подсказок пользовательского интерфейса.
Click	Возникает при щелчке элемента управления.
ContextMenuChanged	Возникает при изменении значения свойства ContextMenu.
ControlAdded	Происходит при добавлении нового элемента управления к Control.ControlCollection.
ControlRemoved	Происходит при удалении элемента

	управления из Control.ControlCollection.
CursorChanged	Возникает при изменении значения свойства Cursor.
Disposed (унаследовано от Component)	Добавляет обработчик событий для отслеживания события Disposed для компонента.
DockChanged	Возникает при изменении значения свойства Dock.
DoubleClick	Возникает при двойном щелчке элемента управления.
DragDrop	Возникает, когда операция перетаскивания завершена.
DragEnter	Происходит при перемещении объекта внутрь границ элемента управления.
DragLeave	Происходит при перемещении объекта за границы элемента управления.
DragOver	Происходит при перетаскивании объекта над границами элемента управления.
EnabledChanged	Возникает при изменении значения свойства Enabled.
Enter	Возникает при входе в элемент управления.
FontChanged	Возникает при изменении значения свойства Font.
ForeColorChanged	Возникает при изменении значения свойства ForeColor.
GiveFeedback	Возникает при операции перетаскивания.
GotFocus	Возникает при получении фокуса элементом управления.
HandleCreated	Происходит при создании дескриптора для элемента управления.
HandleDestroyed	Возникает в процессе уничтожения дескриптора элемента управления.
HelpRequested	Происходит при запросе справки для элемента управления.
ImeModeChanged	Возникает при изменении свойства ImeMode.
Invalidated	Возникает, когда отображение элемента управления следует обновить.
KeyDown	Возникает при нажатии клавиши, если элемент управления имеет фокус.
KeyPress	Возникает при нажатии клавиши, если элемент управления имеет фокус.
KeyUp	Возникает, когда клавишу отпускают, если элемент управления имеет фокус.
Layout	Возникает, когда элемент управления должен переместить свои дочерние элементы управления.
Leave	Возникает, когда элемент управления

	лишается фокуса ввода.
LocationChanged	Возникает при изменении значения свойства Location.
LostFocus	Возникает при потере фокуса элементом управления.
MouseDown	Возникает, когда указатель мыши находится на элементе управления и нажата кнопка мыши.
MouseEnter	Возникает, когда указатель мыши оказывается на элементе управления.
MouseHover	Возникает, когда указатель мыши наведен на элемент управления.
MouseLeave	Возникает, когда указатель мыши покидает элемент управления.
MouseMove	Возникает, когда указатель мыши перемещается на элемент управления.
MouseUp	Возникает, когда указатель мыши находится на элементе управления и кнопка мыши не нажата.
MouseWheel	Возникает при движении колеса мыши, если элемент управления имеет фокус.
Move	Возникает при перемещении элемента управления.
Paint	Возникает при обновлении элемента управления.
ParentChanged	Возникает при изменении значения свойства Parent.
QueryAccessibilityHelp	Возникает при предоставлении справки объектом AccessibleObject для приложений со специальными возможностями.
QueryContinueDrag	Возникает во время операции перетаскивания и позволяет источнику перетаскивания определить, должна ли она быть отменена.
Resize	Возникает при изменении размеров элемента управления.
RightToLeftChanged	Возникает при изменении значения свойства RightToLeft.
SizeChanged	Возникает при изменении значения свойства Size.
StyleChanged	Возникает при изменении стиля элемента управления.
SystemColorsChanged	Происходит при изменении системных цветов.
TabIndexChanged	Возникает при изменении значения свойства TabIndex.
TabStopChanged	Возникает при изменении значения свойства TabStop.
TextChanged	Возникает при изменении значения свойства Text.

Validated	Возникает при окончании проверки элемента управления.
Validating	Возникает при проверке элемента управления.
VisibleChanged	Возникает при изменении значения свойства Visible.

Взаимодействие с пользователем

Изменением свойств, вызовом методов, выполнением обработчиков событий.

Два способа обработки события:

1. Замещение защищенного метода, вызывающего соответствующее событие (OnClick --> Click).

2. Подключение собственного обработчика, тело которого генерирует Дизайнер VS.

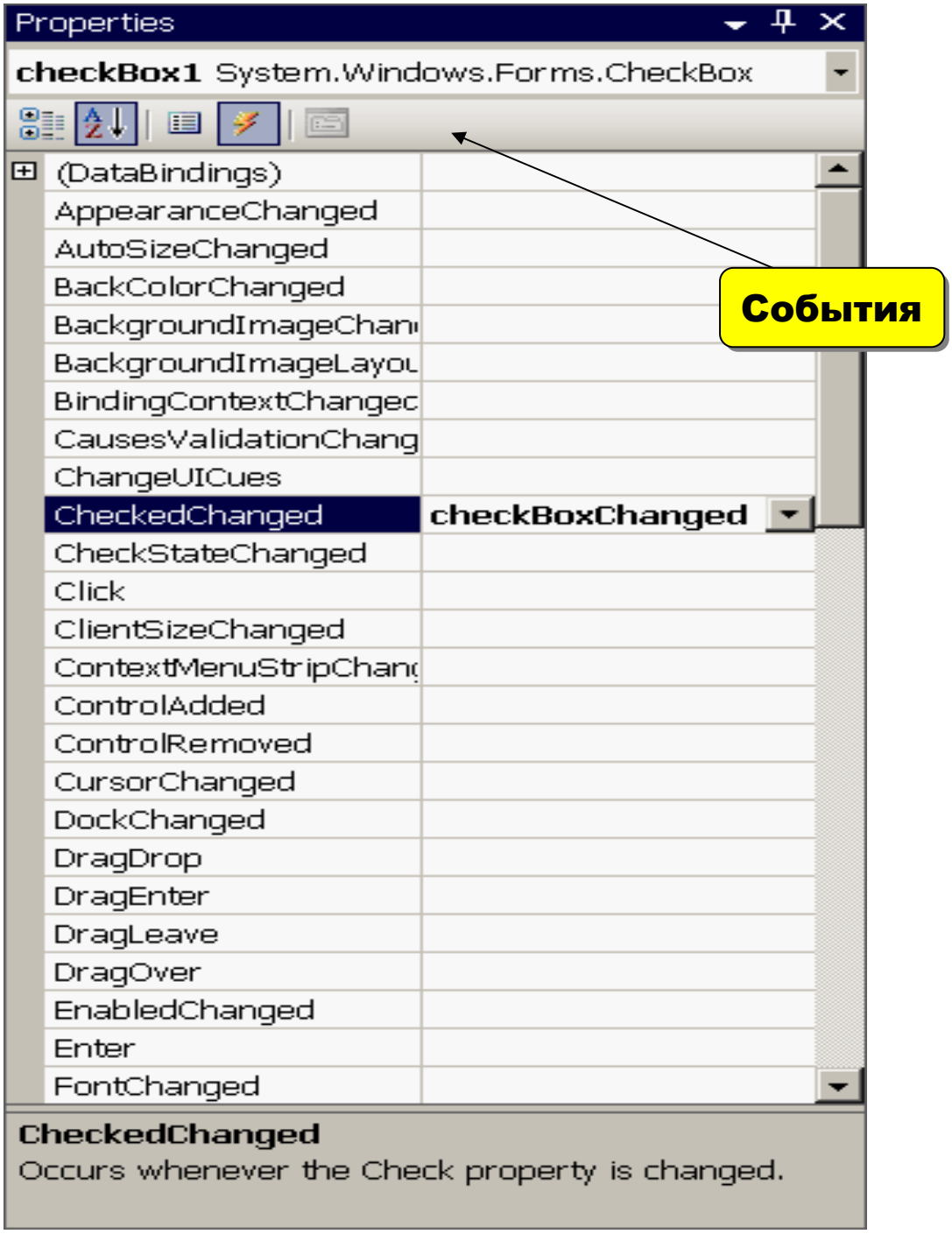


рис.4

2. События ЭУ

Наиболее часто используемые события ЭУ:

- Click
- DoubleClick
- Validating
- Enter
- Leave
- Paint

Enter, **Leave** – получить фокус элементом управления, потерять фокус (параметр EventArgs e).

Validating – проверить правильность данных (параметр CancelEventArgs e).

private void button1_Click (object sender, EventArgs e) { . . . }

sender – ссылка на ЭУ, сгенерировавший событие.

sender.Name – имя (идентификатор) ЭУ;

sender.Text – текст в ЭУ.

e – параметры сообщения.

Как установить фокус ввода: ЭУ.Focus()

Возвращаемое значение: значение true, если запрос фокуса ввода был успешным; в противном случае — false.

Примечания

Метод Focus возвращает значение true, если элемент управления успешно получил фокус ввода. Ели элемент управления имеет фокус ввода, не всегда имеются внешние признаки, указывающие на это. Такое поведение характерно в первую очередь для невыделяемых элементов управления, перечисленных ниже, а также для любых производных от них элементов управления.

Элемент управления может быть выделен и получить фокус ввода, если всем нижеследующим утверждениям соответствует значение true: бит стиля ControlStyles.Selectable задан как true; элемент управления содержится в другом элементе управления, и все его родительские элементы видимы и включены.

Список элементов управления форм Windows Forms, которые также являются невыделяемыми. Элементы управления, производные от этих элементов, также не выделяются.

- Panel
- GroupBox
- PictureBox
- ProgressBar
- Splitter
- Label
- LinkLabel (если ссылка в элементе управления отсутствует)

Пример

В следующем примере фокус передается указанному Control при условии, что он имеет возможность получать фокус.

```
public void ControlSetFocus(Control control)
{
  // Set focus to the control, if it can receive focus.
  if(control.CanFocus)
  {
    control.Focus();
  }
}
```

3. События клавиатуры

KeyDown – при нажатии
KeyUP – при отпускании

void Form1_KeyUp(object sender, KeyEventArgs e)

Свойства класса KeyEventArgs:

Alt, Control, Shift = true – нажата, false – не нажата.
KeyCode – код нажатой клавиши
KeyData – совокупность кодов нажатых клавиш

KeyPress – удержание, посылается серия событий
void Form1_KeyPress(object sender, KeyPressEventArgs e)

4. События мыши

MouseClick – щелчок
MouseDoubleClick (посылается и MouseClick)
MouseDown – нажатие
MouseUp – отпускание
MouseEnter – курсор находится на ЭУ
MouseHover – курсор мыши наведен на элемент управления
MouseLeave – курсор покидает ЭУ
MouseMove – перемещение курсора
MouseWheel – колесико

private void Form1_MouseDown(object sender, MouseEventArgs e)

Открытые свойства MouseEventArgs

Button	Возвращает значение перечисления MouseButtons - сведения о том, какая кнопка мыши была нажата.
Clicks	Возвращает число нажатий и отпусканий кнопки мыши.
Delta	Возвращает счетчик со знаком для количества щелчков вращающегося колесика мыши. Щелчок — это один зубчик колесика мыши.
X	Возвращает x-координату мыши.
Y	Возвращает y-координату мыши.

Начиная с Windows 2000, корпорация Майкрософт вводит поддержку пятикнопочной мыши Microsoft IntelliMouse Explorer.

Две новые кнопки мыши (XBUTTON1 и XBUTTON2) обеспечивают перемещение вперед-назад.

Перечисление MouseButtons

Имя члена	Описание	Значение
Left	Была нажата левая кнопка мыши.	1048576
Middle	Была нажата средняя кнопка мыши.	4194304
None	Никакая кнопка мыши не была	0

	нажата.	
Right	Была нажата правая кнопка мыши.	2097152
XButton1	Была нажата первая кнопка XButton.	8388608
XButton2	Была нажата вторая кнопка XButton.	16777216

Пример.

```
switch (e.Button)
{
  case MouseButtons.Left:
    eventString = "L";
    x1 = e.X;
    y1 = e.Y;
    break;
  case MouseButtons.Right:
    eventString = "R";
    break;
}
```

5. Событие Paint

```
private void Form1_Paint (object sender, Forms.PaintEventArgs e)
Graphics g = e.Graphics;
```

Элемент управления Button

Это простая командная кнопка.

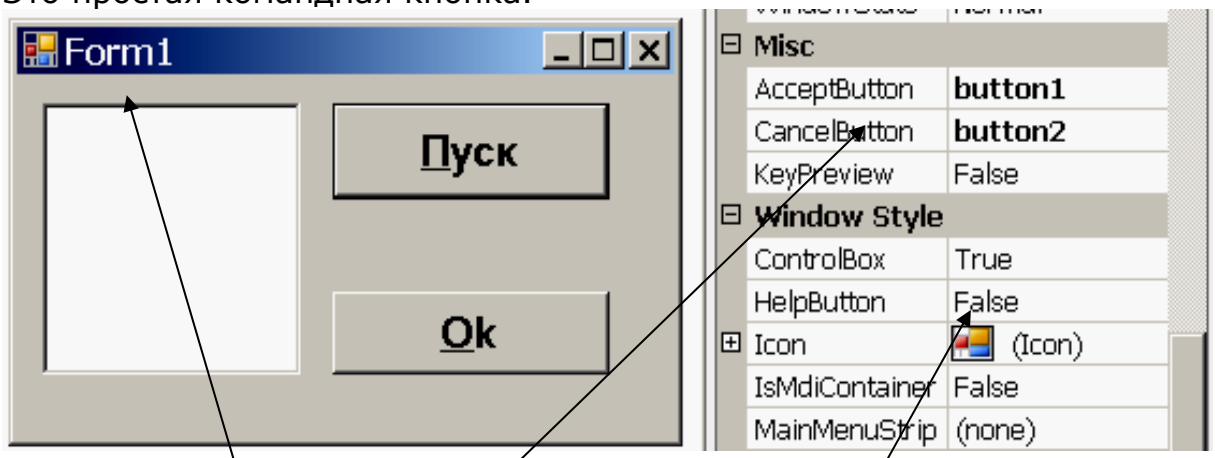


рис.5

Свойство формы AcceptButton, равное ссылке на кнопку, делает кнопку в качестве кнопки по умолчанию, т.е. активируемой при любом нажатии клавиши ВВОД, независимо от того, на каком элементе управления формы находится в этот момент фокус. На рис. вызывается button1_Click.

Примечание. Если для текстового поля установлено AcceptsReturn=false, эту возможность можно использовать для ввода по Enter текста в текстовые поля, например, такие как TextBox. Иначе, клавиша Enter в случае AcceptsReturn=false будет проигнорирована.

Свойство формы CancelButton, равное ссылке на кнопку делает кнопку кнопкой «Отмена». Кнопка «Отмена» активируется при любом нажатии клавиши ESC, независимо от того, на каком элементе управления формы находится в этот момент фокус. В нашем примере ызывается button2_Click().

&Пуск –> (Tab, Alt+П)

Создание главного меню

Рассмотрим разработку следующей программы.

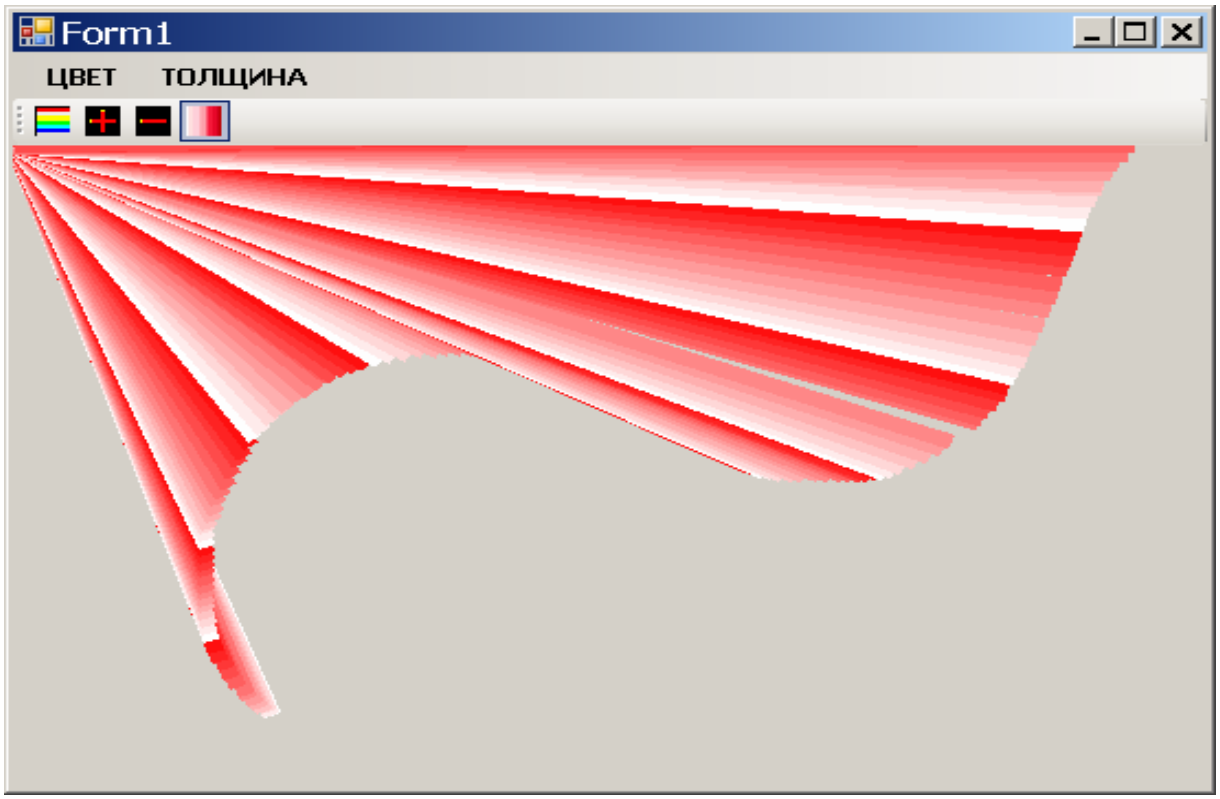


рис.6

Большинство Windows-приложений оснащено главным меню, которое представляет собой иерархическую структуру выполняемых функций и команд. Практически все функции, которые можно осуществить при помощи элементов управления, имеют свой аналог в виде пункта меню.

Для создания главного меню используется элемент управления MenuStrip (MainMenu – VS 2003). Перетаскиваем компонент MenuStrip на форму из ToolBox. MenuStrip отображается на панели невидимых элементов, а в форме отображаются элементы ToolStripMenuItem, которые в тексте программы включаются в контейнер компонента MenuStrip.

Примечание. Класс ToolStripMenuItem не наследует интерфейс Control, поэтому общим ЭУ не является. ОЭУ можно располагать на форме в любом месте и изменять размеры, а главное меню – только под заголовком.

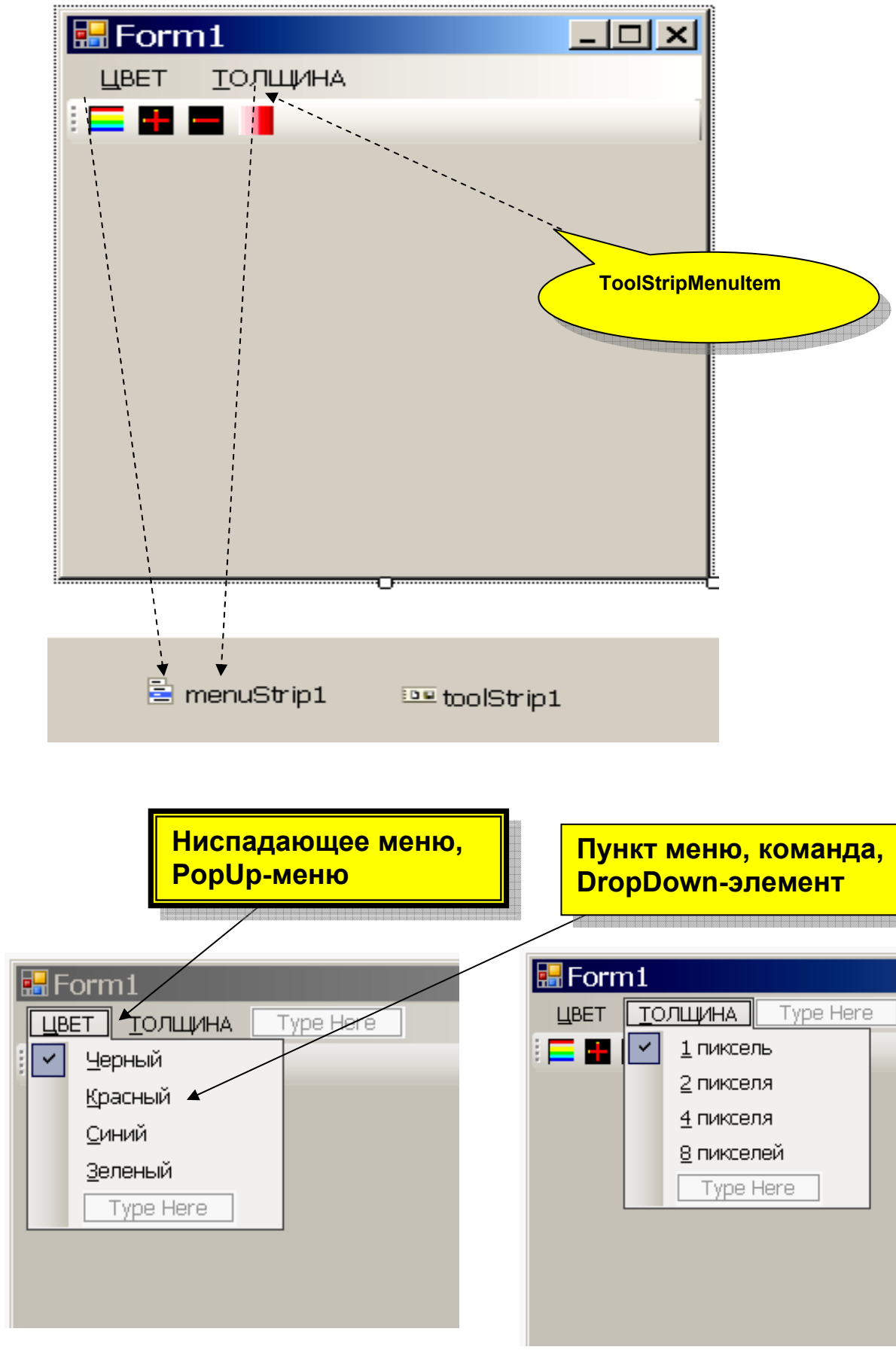


рис.7

Каждый пункт главного меню имеет свое окно свойств, в котором, подобно другим элементам управления, задаются значения свойств Name и Text (см. 1,2).

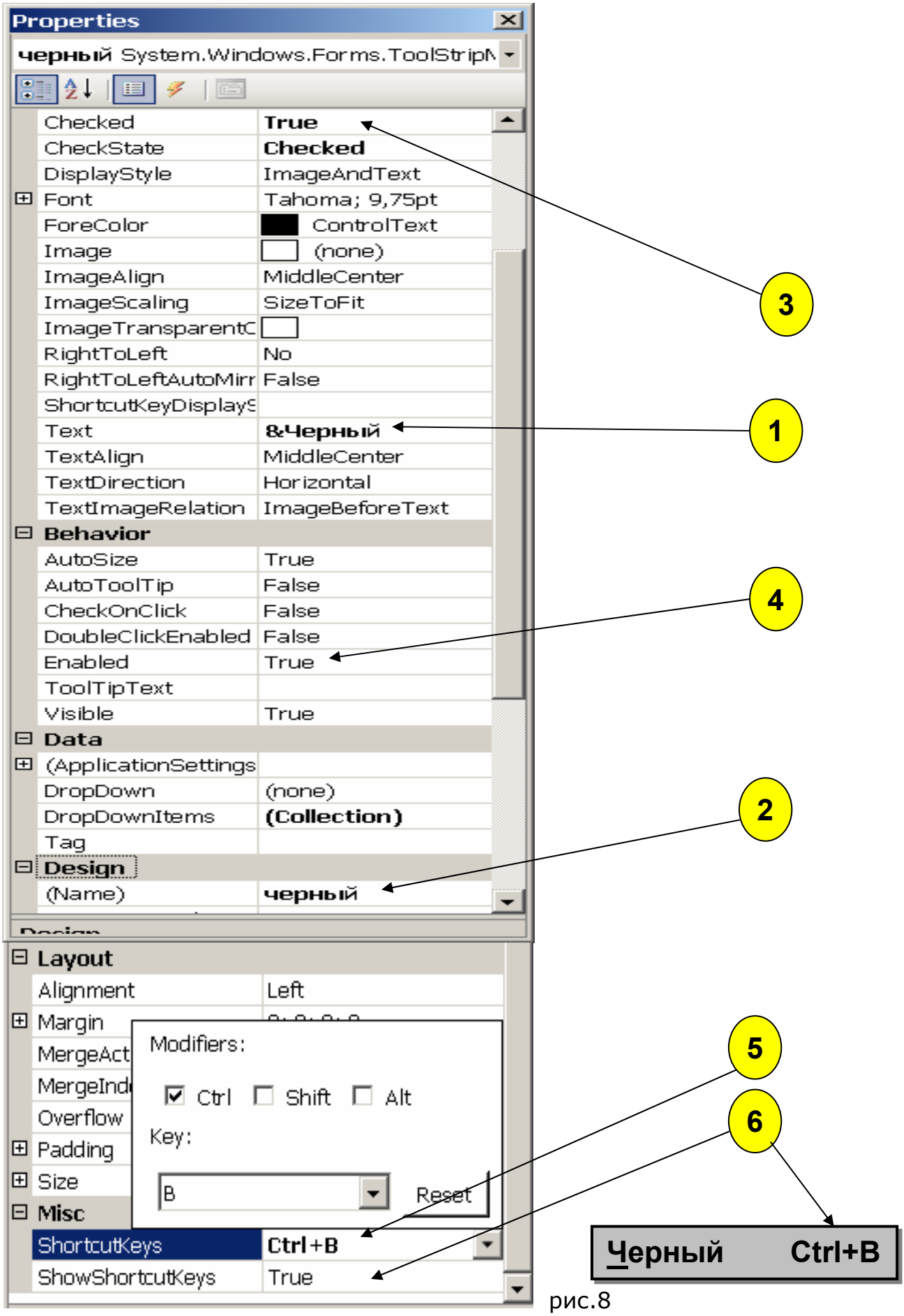


рис.8

В поле Text перед словом Черный стоит знак & — так называемый амперсанд, указывающий, что Ч должно быть подчеркнута и будет частью встроенного клавиатурного интерфейса Windows (см. 1). Когда

пользователь на клавиатуре нажимает клавишу Alt и затем Ч, выводится пункт Черный. Для разделения пунктов горизонтальной разделительной линией в свойстве Text пункта меню просто вводим знак тире. Существует и другой способ.

Пункты меню во время работы в дизайнере (конструкторе VS) с помощью свойства Checked=true можно пометить галочкой (см. 3) или с помощью свойства Enabled=false сделать пункт недоступным для работы (см. 4).

6. Быстрые клавиши

В Windows есть еще интерфейс для работы с так называемыми быстрыми клавишами, или акселераторами. Сочетание клавиш указывают в свойстве ShortcutKeys (см. 5). Следует назначать стандартным пунктам общепринятые сочетания клавиш.

С помощью свойства ShowShortcutKeys=false быстрые клавиши можно не отображать в меню. По умолчанию ShowShortcutKeys=true (см. 6).

7. Исходный текст программы

Классы и свойства:

public class MenuStrip : ToolStrip

public virtual ToolStripItemCollection Items { get; }

public class ToolStripMenuItem : ToolStripDropDownItem

public abstract class ToolStripDropDownItem : ToolStripItem

public ToolStripItemCollection DropDownItems { get; }

public class ToolStripItemCollection : ArrangedElementCollection, IList, ICollection, IEnumerable

На рис.9 показана связь объектов главного меню.

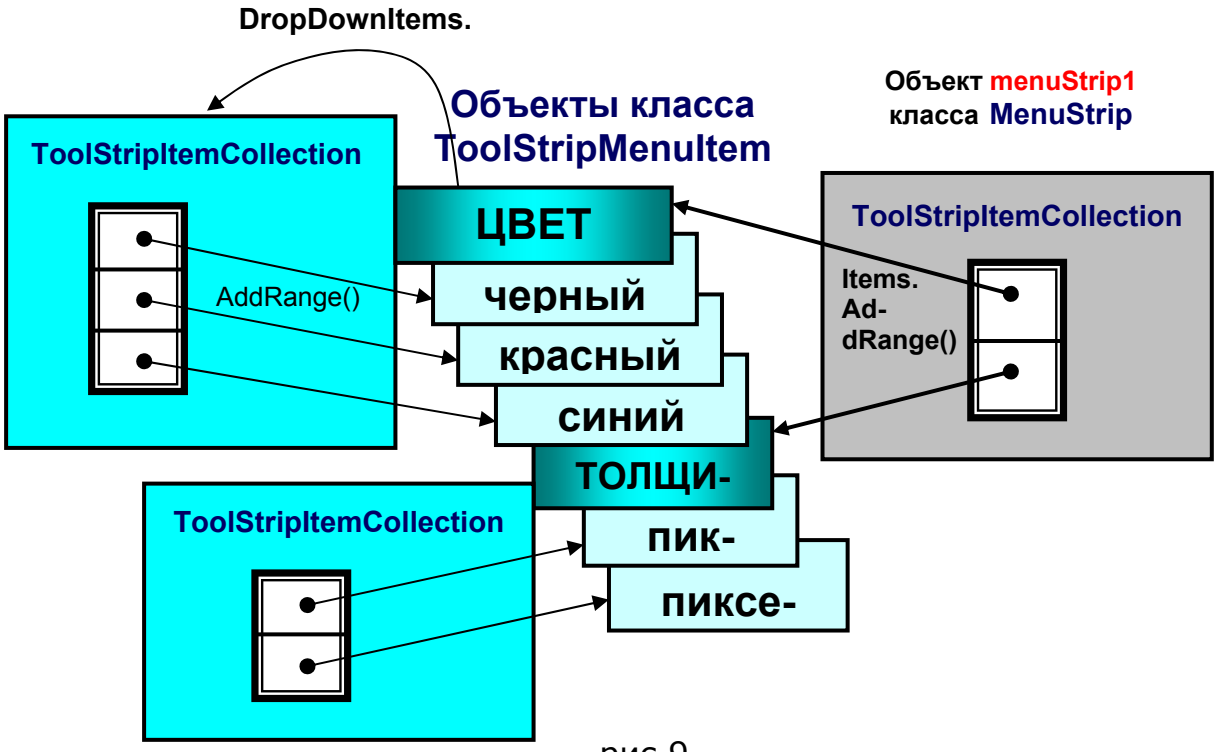


рис.9

Рассмотрим подробнее, как программируется меню на уровне исходного текста программы.

1.

Сначала создается контейнер MenuStrip и все объекты-пункты меню.

```
// создаем контейнер MenuStrip и пункты ToolStripMenuItem
MenuStrip menuStrip1 = new MenuStrip();

ToolStripMenuItem ЦВЕТ = new ToolStripMenuItem();
   ToolStripMenuItem черный   = new ToolStripMenuItem();
   ToolStripMenuItem красный = new ToolStripMenuItem();
   ToolStripMenuItem синий     = new ToolStripMenuItem();

ToolStripMenuItem ТОЛЩИНА = new ToolStripMenuItem();
   ToolStripMenuItem пикселей1 = new ToolStripMenuItem();
   ToolStripMenuItem пикселей2 = new ToolStripMenuItem();
```

2.

Компонент MenuStrip является контейнером пунктов меню самого верхнего уровня. Все пункты меню (и PopUp, и DropDown) являются экземплярами класса ToolStripMenuItem.

Используя свойство Items контейнера, получаем ссылку на коллекцию класса ToolStripItemCollection.

С помощью метода AddRange() этой коллекции в коллекцию включаются ссылки на объекты-меню верхнего уровня.

```
// Включаем в коллекцию контейнера menuStrip1  PopUp-пункты главного меню.

menuStrip1.Items.AddRange(new ToolStripItem[ ] {ЦВЕТ, ТОЛЩИНА});

menuStrip1.Location = new System.Drawing.Point(0, 0);
menuStrip1.Name = "menuStrip1";
menuStrip1.Text = "menuStrip1";
menuStrip1.Size = new System.Drawing.Size(707, 24);
menuStrip1.TabIndex = 0;
```

3.

Если пункт меню является PopUp-пунктом, то через его свойство DropDownItems становится доступной коллекция класса ToolStripItemCollection.

С помощью метода AddRange() этой коллекции в коллекцию включаются ссылки на DropDown-объекты (подчиненные подпункты).

```
// Формируем PopUp-меню ЦВЕТ, устанавливаем свойства его пунктов

ЦВЕТ.DropDownItems.AddRange (new ToolStripItem[ ] { черный, красный, синий } );
ЦВЕТ.Name = "ЦВЕТ";
ЦВЕТ.Text = "&ЦВЕТ";
ЦВЕТ.Size = new System.Drawing.Size (51, 20);

// черный

черный.Checked = true;                                    // установить флажок
черный.CheckState = Forms.CheckState.Checked;     // запомнить состояние
```

```
черный.Name = "черный";
черный.Size = new System.Drawing.Size (152, 22);
черный.Text = "&Черный";
черный.Click += new System.EventHandler (черный_Click);

// красный

красный.Name = "красный";
красный.Size = new System.Drawing.Size (152, 22);
красный.Text = "&Красный";
красный.Click += new System.EventHandler (красный_Click);
//
// синий
// . . .


// Создаем PopUp-меню ТОЛЩИНА, устанавливаем свойства его пунктов

ТОЛЩИНА.DropDownItems.AddRange(new ToolStripItem[ ] { пикселей1,
                                                                    пикселей2
});
ТОЛЩИНА.Name = "ТОЛЩИНА";
ТОЛЩИНА.Text = "&ТОЛЩИНА";
ТОЛЩИНА.Size = new System.Drawing.Size (82, 20);

// пикселей1

пикселей1.Checked = true;                                    // установить флажок
пикселей1.CheckState = Forms.CheckState.Checked;     // запомнить состояние
пикселей1.Name = "пикселей1";
пикселей1.Text = "&1 пиксель";
пикселей1.Size = new System.Drawing.Size (154, 22);
пикселей1.Click += new System.EventHandler (толщина_Click);

// пикселей2
// . . .
```

8. Установка и сброс отметки одного пункта меню

Без учета состояния остальных пунктов ниспадающего меню:

```
this.Пункт1.CheckOnClick = true;    // В дизайнере VS
```

9. Установка и сброс отметки пунктов PopUp-меню

Задача: установить контрольную отметку у выбранного пункта ниспадающего меню, а у взаимоисключающих пунктов аналогичную отметку убрать.

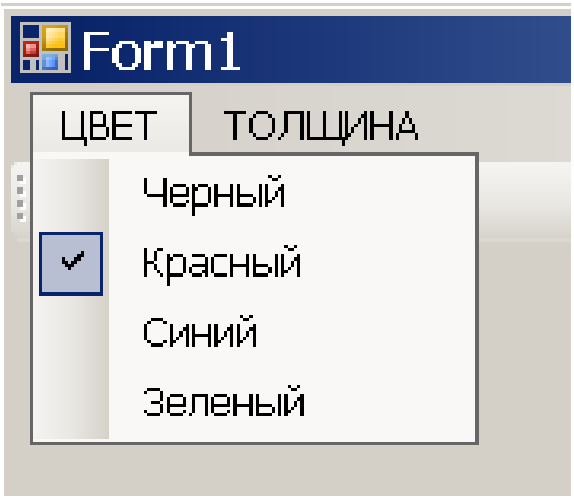


рис.10

```
private void черный_Click (object sender, EventArgs e)
{
   color = Color.Black;
   MenuItemCheck ("&Черный", ЦВЕТ);
}

private void красный_Click (object sender, EventArgs e)
{
   color = Color.Red;
   MenuItemCheck ("&Красный", ЦВЕТ);
}

private void синий_Click(object sender, EventArgs e)
{
   color = Color.Blue;
   MenuItemCheck ("&Синий", ЦВЕТ);
}

private void толщина_Click (object sender, EventArgs e)
{

        switch (((ToolStripMenuItem)sender).Text)
        {
           case "&1 пиксель":
              width = 1;
              break;
           case "&2 пикселя":
              width = 2;
              break;
           case "&4 пикселя":
              width = 4;
              break;
            . . . . . .
        }

        MenuItemCheck (( (ToolStripMenuItem)sender).Text, ТОЛЩИНА);
  }
```

```
//private void MenuItemCheck(ToolStripMenuItem name, ToolStripMenuItem popup)

private void MenuItemCheck ( string text,  ToolStripMenuItem  popup )
{
        foreach (ToolStripMenuItem item in popup.DropDownItems)
        {
           if ( item.Text != text )
              item.Checked = false;
           else
              item.Checked = true;
        }
}
```

11. Дополнительные возможности

Элементами меню могут быть:
пункты меню типа ToolStripMenuItem,
списки ComboBox типа ToolStripComboBox,
разделители типа ToolStripSeparator,
управляющие элементы TextBox типа ToolStripTextBox.

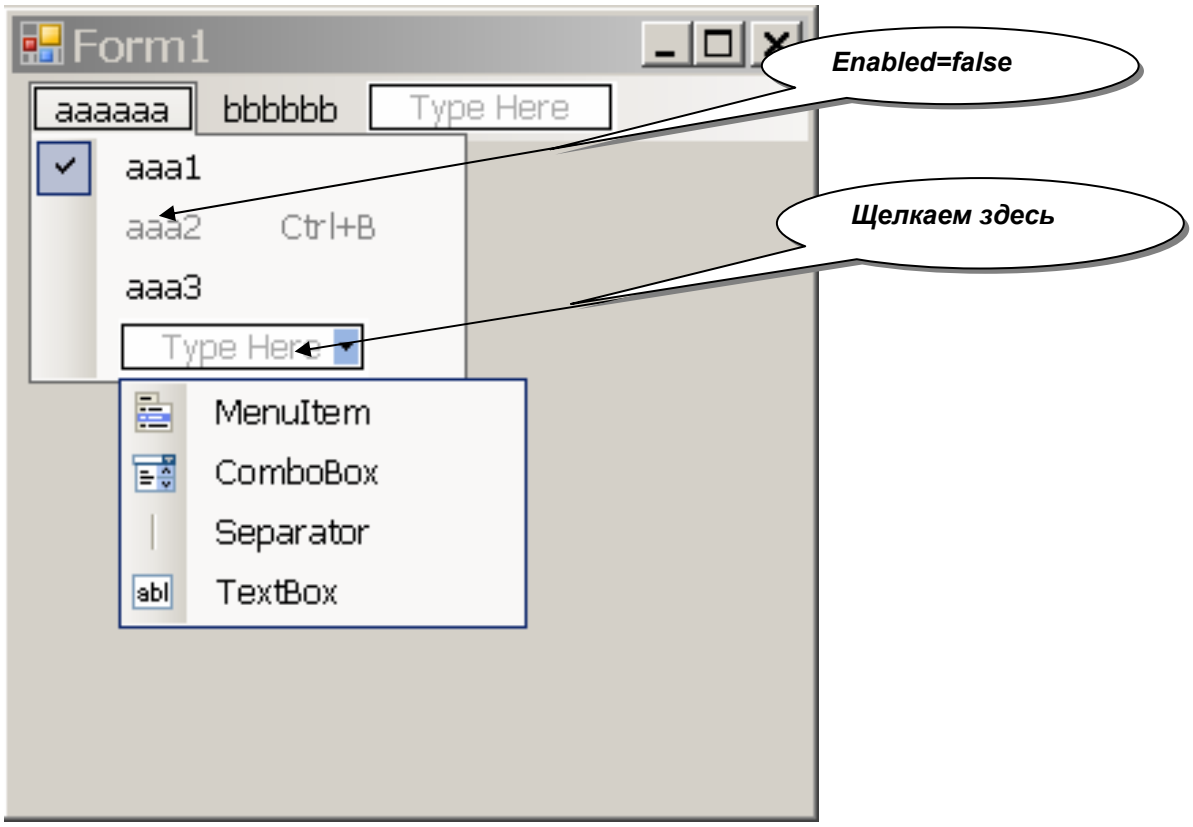


рис.11

Особенностью элементов ComboBox и TextBox является то, что они не исчезают с экрана после их выборки. Если выбран пункт меню, то меню исчезает.

```
private System.Windows.Forms.ToolStripMenuItem    toolStripMenuItem1;
private System.Windows.Forms.ToolStripSeparator   toolStripSeparator1;
private System.Windows.Forms.ToolStripTextBox     toolStripTextBox1;
private System.Windows.Forms.ToolStripComboBox    toolStripComboBox1;
```

Создание контекстного меню

Это меню, которое выводится по щелчку правой кнопки мыши, и вид которого зависит от управляющего элемента, на котором сделан щелчок. Контекстное меню, дублирующее некоторые действия основного меню, - самый привычный способ работы с программой для пользователя.

Ряд элементов управления (TextBox, ComboBox) имеют встроенное контекстное меню.

Пример: элемент управления TextBox содержит в себе простейшее контекстное меню, дублирующее действия подменю Edit.

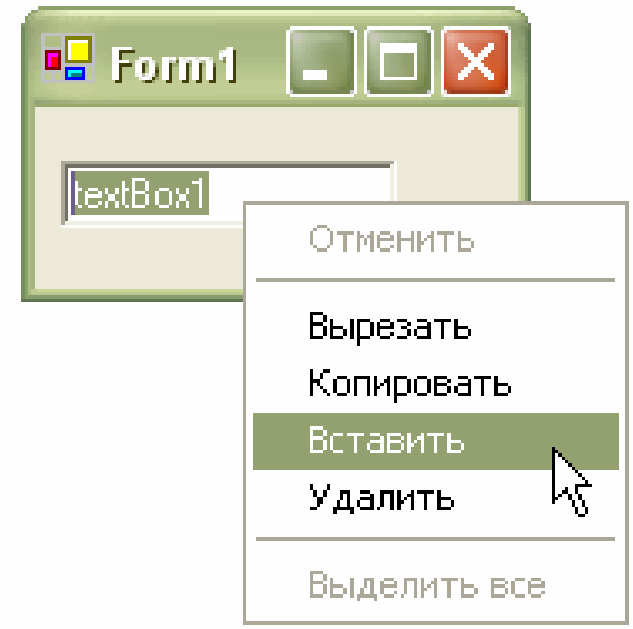


рис.12

Добавим элемент управления ContextMenuStrip (ContextMenu – VS 2003) из окна ToolBox на форму.

Контекстное меню включает только одну древовидную структуру элементов меню, поэтому оно наследует не класс ToolStrip, а класс ToolStripDropDownMenu.

```
public class ContextMenuStrip : ToolStripDropDownMenu
```

Пункты контекстного меню добавляются точно так же, как и для главного меню. Все, что нужно сделать, — это определить, где будет появляться контекстное меню.

Привязка контекстного меню к ОЭУ

Многие элементы управления имеют свойство ContextMenuStrip, с помощью которого можно прикрепить контекстное меню к элементу.

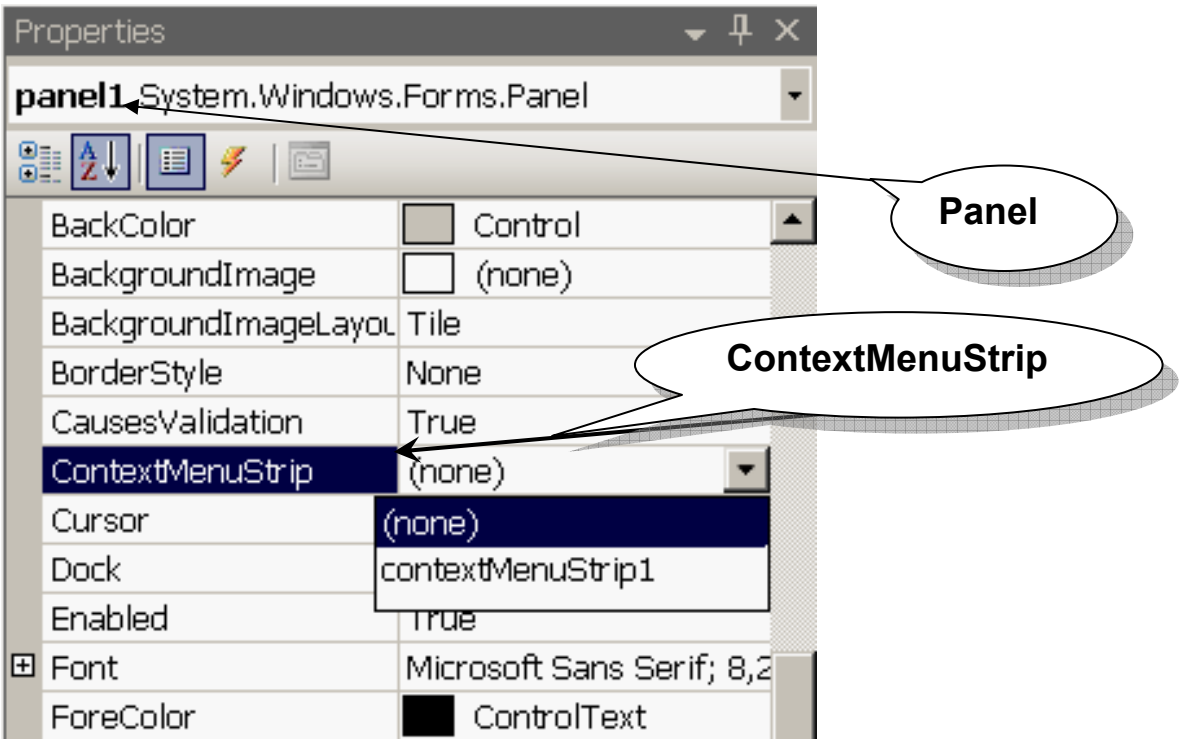


рис.13

Местоположение меню определяется свойством LayoutStyle относительно родительского окна.

12. Создание панели инструментов ToolStrip

public class ToolStrip : ScrollableControl, IComponent, IDisposable

В графических программах панели инструментов — основное средство работы. Панели инструментов ToolStrip содержат, как правило, наборы кнопок (Button), зачастую дублирующих пункты главного меню.

Перетащим на форму из окна ToolBox компонент ToolStrip.

На кнопках панели обычно располагаются иконки.

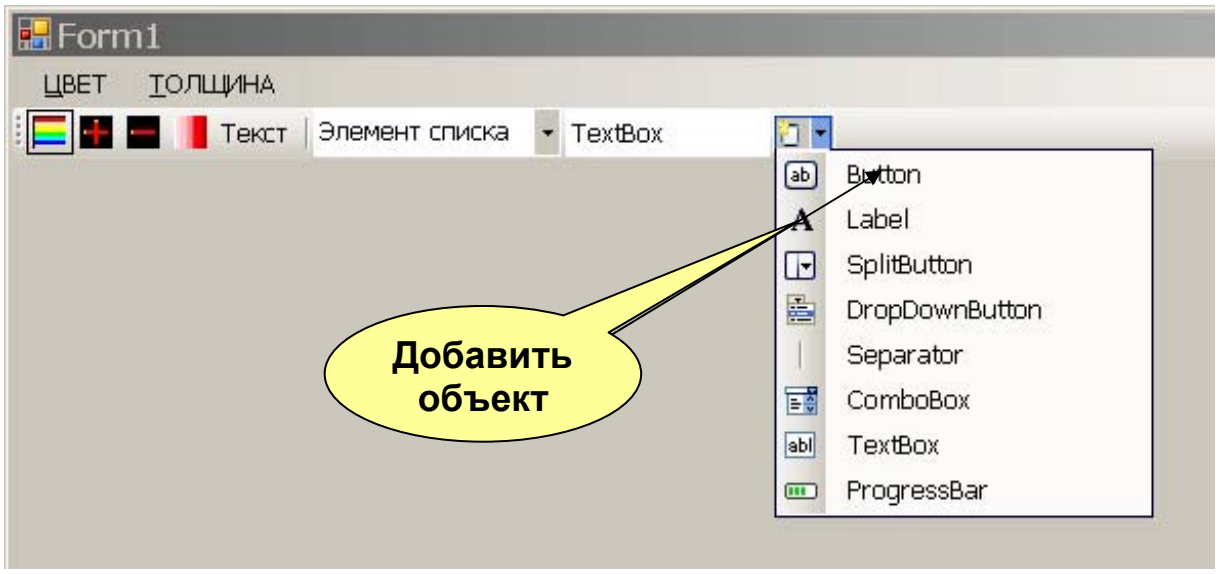


рис.14

Создание рисунков (иконок).

Щелкаем правой кнопкой на проекте, далее выбираем:
Add / New Item... / Icon File

Запустится графический редактор со своим главным меню. Редактор создает две иконки – 16x16 и 32x32 пикселей. Во избежание путаницы рекомендуется иконку 16x16 удалить.

Для этого в меню Image сначала удаляемую иконку делаем текущей:
Image / Current Icon Image Type / 16x16, 16 colors
а затем ее удаляем:
Image / Delete Image Type.

Создаем рисунок первого инструмента. Остальные инструменты создаем так же.

Добавление инструмента на панель

Добавить объект на панель инструментов можно двумя способами: с помощью щелчка на стрелке списка Add ToolStripButton в строке инструментов или через свойство Items, создающее коллекцию инструментов.

1-ый способ.

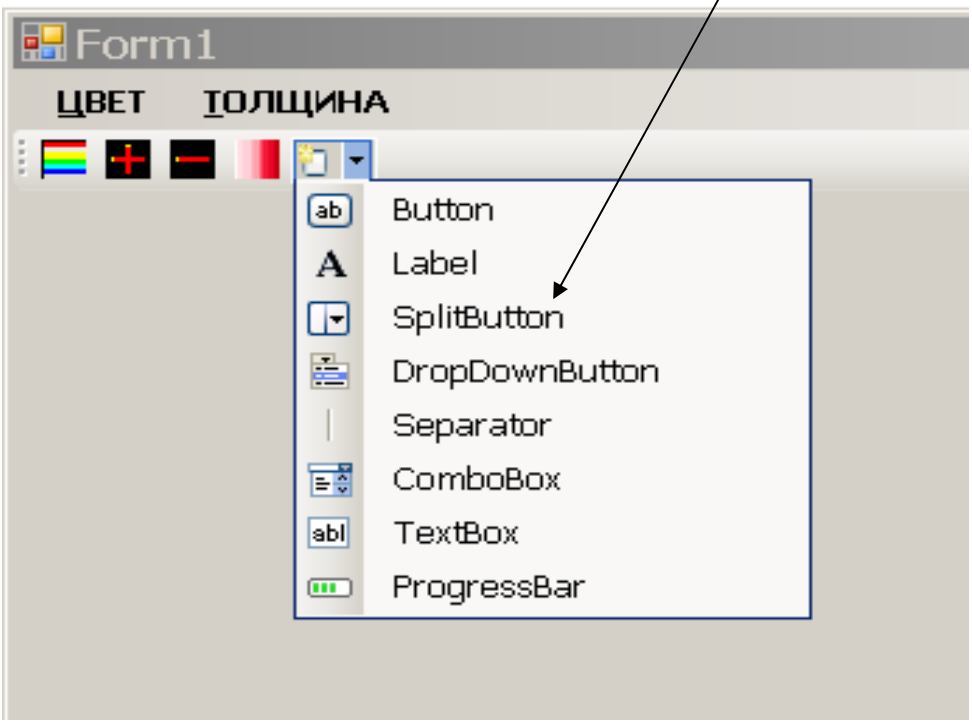


рис.15

Контекстное меню любой кнопки панели инструментов позволяет с помощью пункта Set Image... открыть окно Select resource и импортировать рисунок из файла.

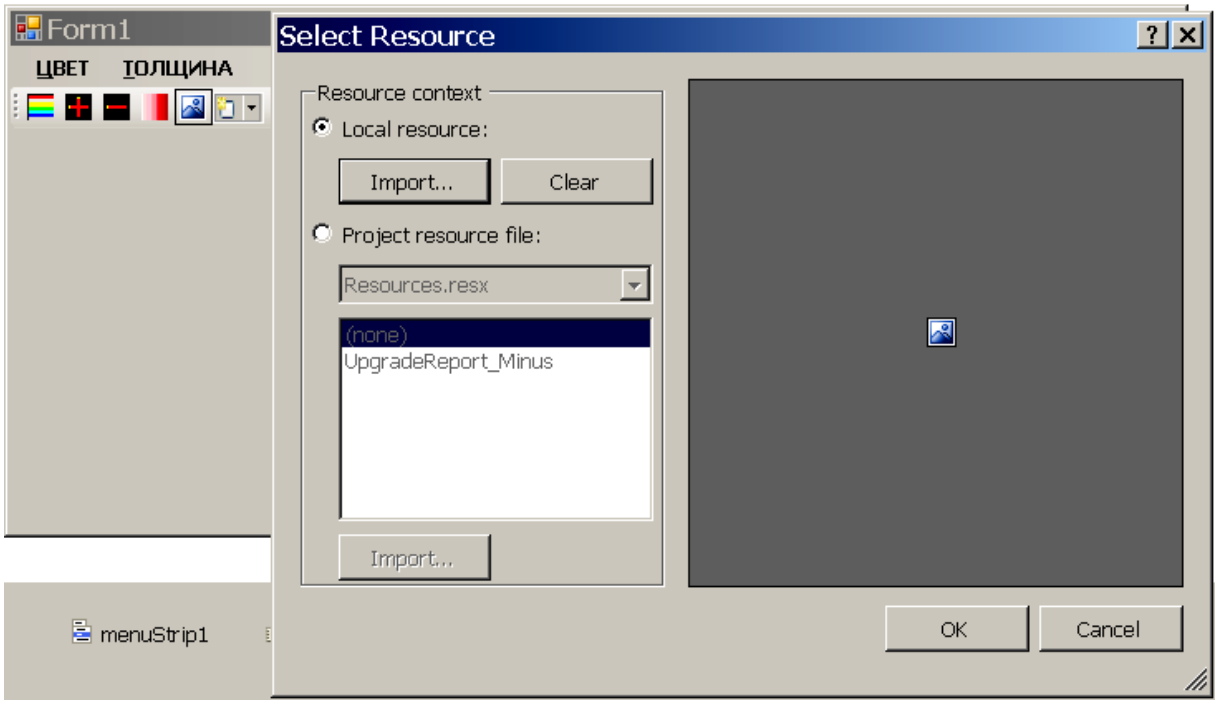


рис.16

2-ой способ. Щелкаем на кнопке ... свойства Items. Появляется редактор коллекции.

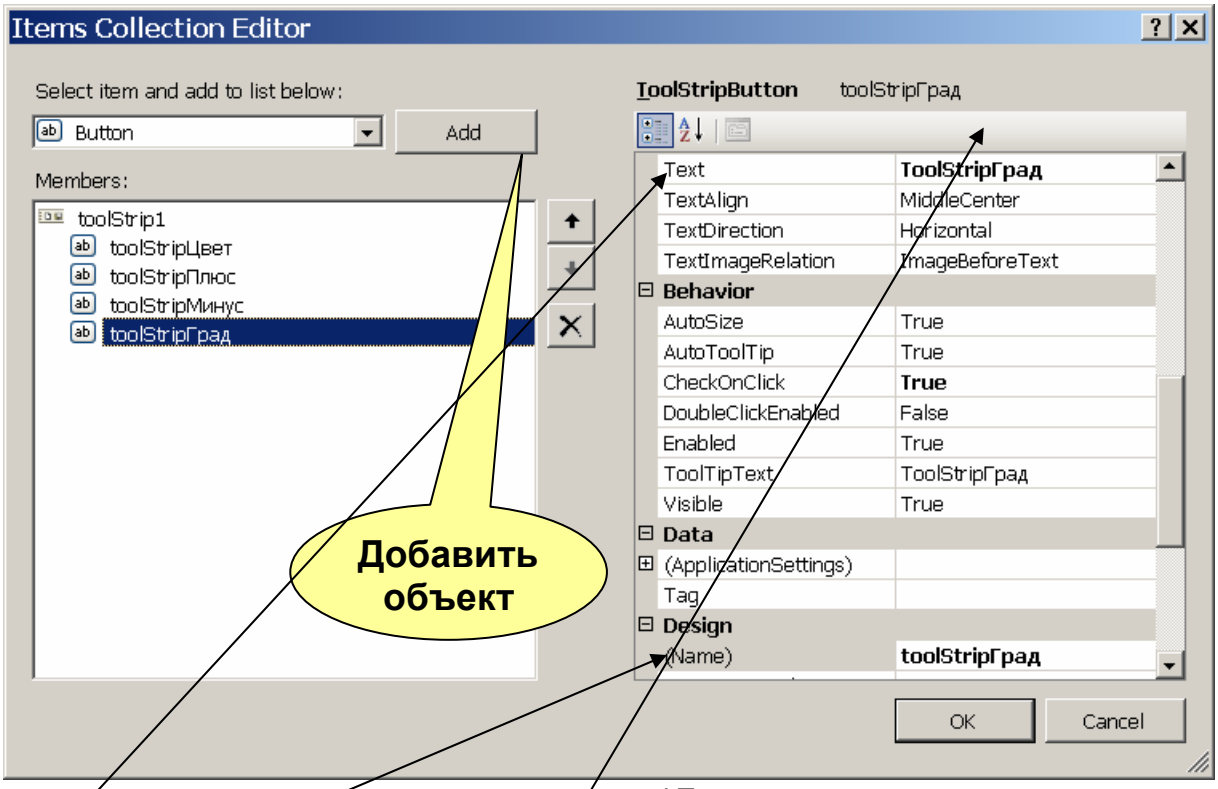


рис.17

1. Для каждого добавляемого инструмента изменяем его свойства Text и Name.

2. Используя свойство Image … инструмента, открываем окно Select resource (см. выше) и импортируем иконку из созданного файла.

Для каждого инструмента создаем либо свой обработчик события, либо указываем обработчик пункта меню, соответствующего инструменту.

```
private void toolStripЦвет_Click (object sender, EventArgs e)
{
    // ColorDialog colorDialog1 = new ColorDialog();    // Создать окно

     colorDialog1.Color = color;         // текущий цвет в рамке

     if (colorDialog1.ShowDialog() == DialogResult.OK)
         color = colorDialog1.Color;     // новый цвет
}

private void toolStripПлюс_Click (object sender, EventArgs e)
{
   width += 2;
}

private void toolStripМинус_Click (object sender, EventArgs e)
{
   if (width == 1) return;
   width -= 2;
}

private void   Form1_MouseMove (object sender, MouseEventArgs e)
{
 int  y = menuStrip1.Size.Height + toolStrip1.Size.Height;

 if  ( leftDown == true )
 {
  Graphics g = this.CreateGraphics();

  if ( toolStripГрад.Checked == true )
  {
    Pen pen2 = new Pen(Color.FromArgb(255, c, c), width);    //белый – красный
    // Pen pen2 = new Pen(Color.FromArgb(c, 0, 0), width);       //красный – черный

      g.DrawLine (pen2, 0, y, e.X, e.Y);

      if (c < 20)
          c = 255;
      else
          c = c – h;
  }
 else
      g.DrawLine(pen, 0, y, e.X, e.Y);
 }
}
```

Реализация на уровне исходного текста программы

1. Создается контейнер типа ToolStrip.
2. Создаются кнопки-инструменты.

3. С помощью свойства контейнера Items получаем коллекцию элементов.

4. Используя метод коллекции AddRange(), добавляем в коллекцию ссылки на кнопки-инструменты.

Следует заметить, что все инструменты панели, как и пункты меню, наследуют абстрактный класс ToolStripItem:

```
public class ToolStripButton : ToolStripItem

ToolStrip toolStrip1 = new ToolStrip();
ToolStripButton toolStripЦвет    = new ToolStripButton();
ToolStripButton toolStripПлюс   = new ToolStripButton();
ToolStripButton toolStripМинус = new ToolStripButton();
ToolStripButton toolStripГрад    = new ToolStripButton();


// toolStrip1
//
toolStrip1.Items.AddRange(new ToolStripItem[ ] { toolStripЦвет,
                                                 toolStripПлюс,
                                                 toolStripМинус,
                                                 toolStripГрад });
toolStrip1.Name = «toolStrip1»;
toolStrip1.Text    = «ToolStrip1»;
toolStrip1.Location = new System.Drawing.Point(0, 24);
toolStrip1.Size = new System.Drawing.Size(707, 25);
toolStrip1.TabIndex = 1;

// toolStripЦвет
//
toolStripЦвет.Name = «toolStripЦвет»;
toolStripЦвет.Text    = «ToolStripЦвет»;
toolStripЦвет.DisplayStyle = ToolStripItemDisplayStyle.Image;
toolStripЦвет.Image =
        ((System.Drawing.Image)(resources.GetObject(«toolStripЦвет.Image»)));
toolStripЦвет.ImageTransparentColor = System.Drawing.Color.Magenta;
toolStripЦвет.Size = new System.Drawing.Size(23, 22);
toolStripЦвет.Click += new System.EventHandler(toolStripButton1_Click);

// toolStripПлюс
//
. . .

// toolStripГрад
//
toolStripГрад.CheckOnClick = true;
. . .
```

13. Создание набора вкладок. Элемент управления TabControl.

В VS-2005 этот элемент управления изменен.

Контейнер TabControl содержит коллекцию страниц типа TabPage. Коллекция имеет тип Controls.

К коллекции страниц (вкладок) можно получить доступ и через свойство TabPages, которое доступно в окне свойств дизайнера. Каждая страница имеет закладку, текст которой определяется свойством Text.

Существует два способа добавления страниц в дизайнере.

1-ый способ. Используя управляющую кнопку элемента.

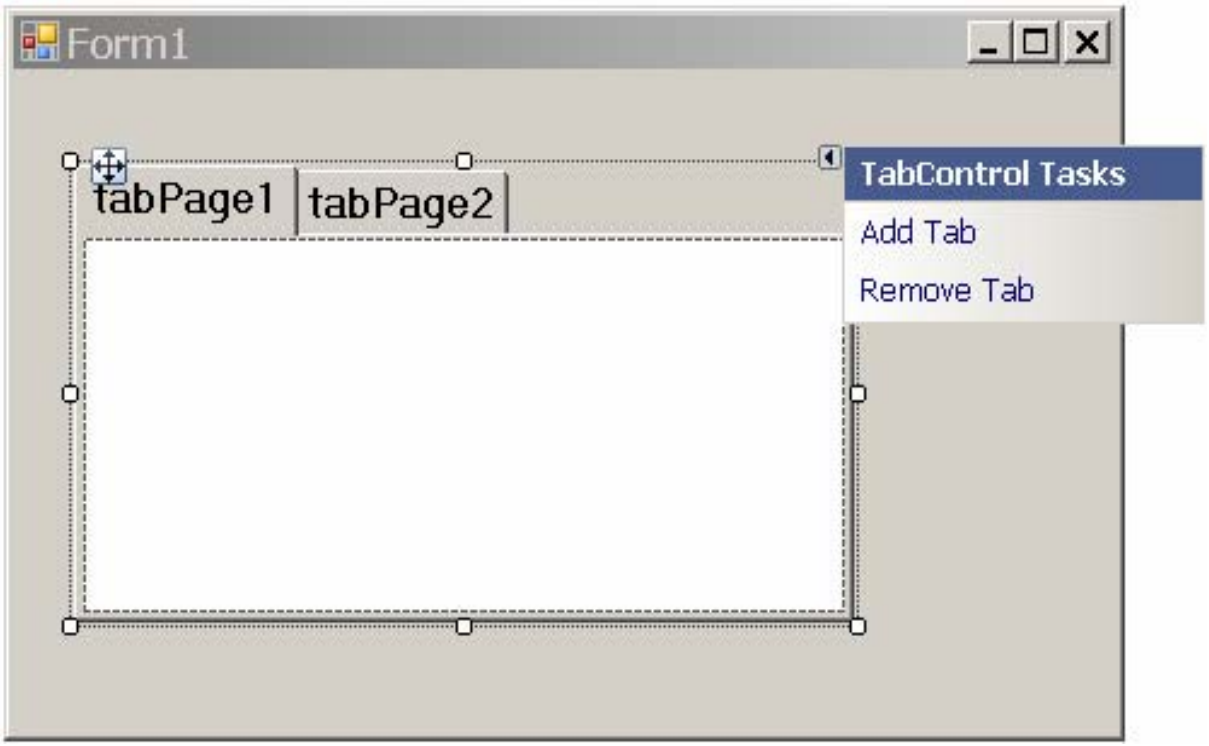


рис.18

Список закладок в одну строку:

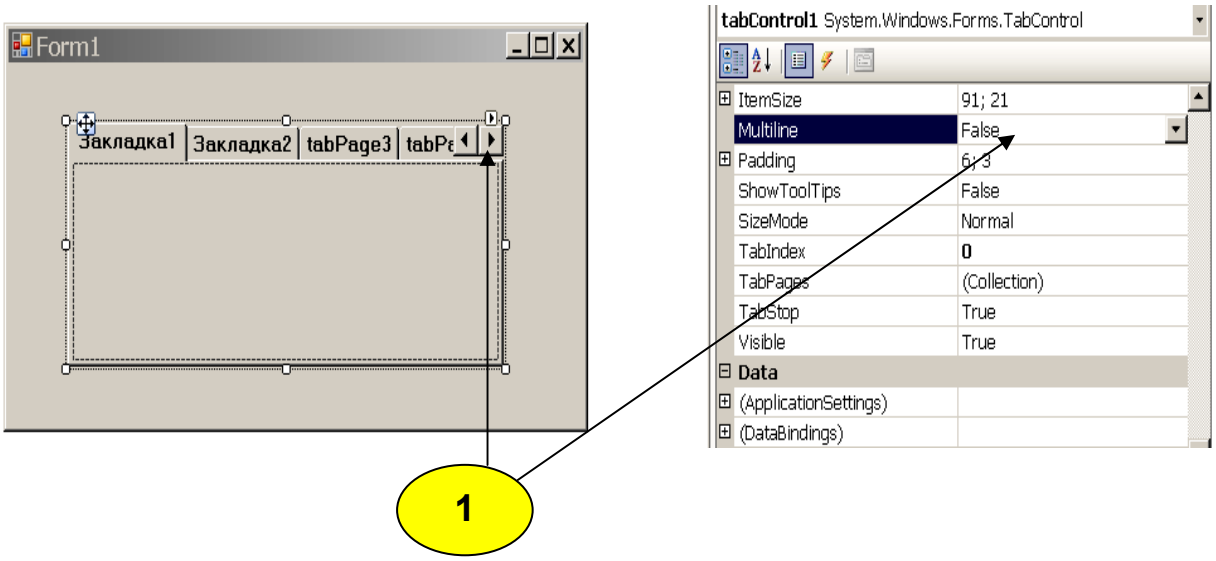


рис.19

Список закладок в несколько строк:

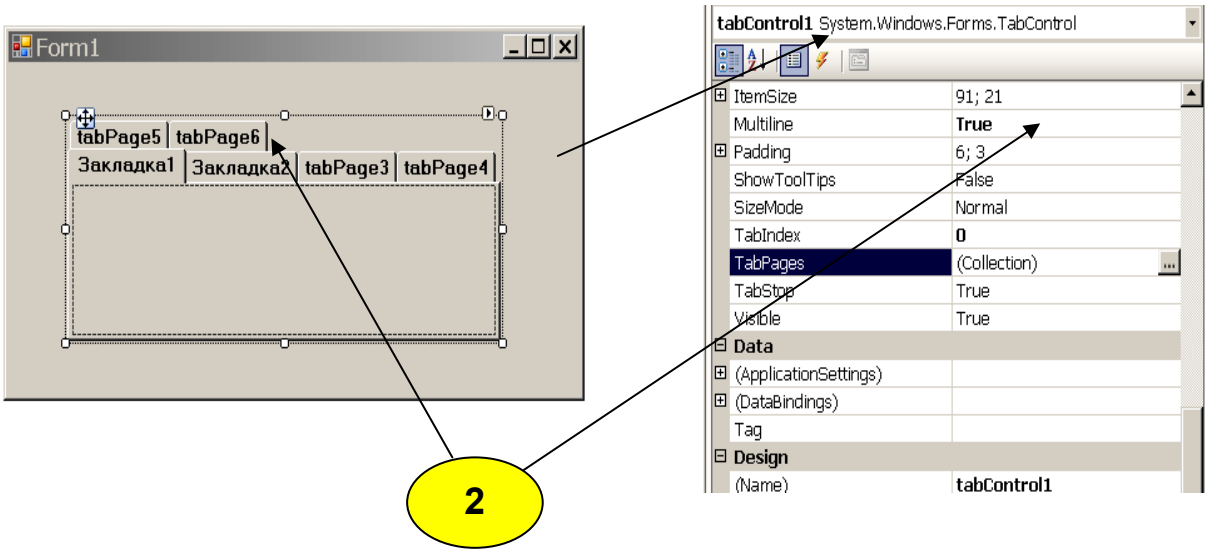


рис.20

Редактирование свойств TabControl (см. 2).
Как выбирать TabControl или конкретную страницу TabPage?
Ответ: надо выделять соответствующий объект.

Редактирование свойств страницы:

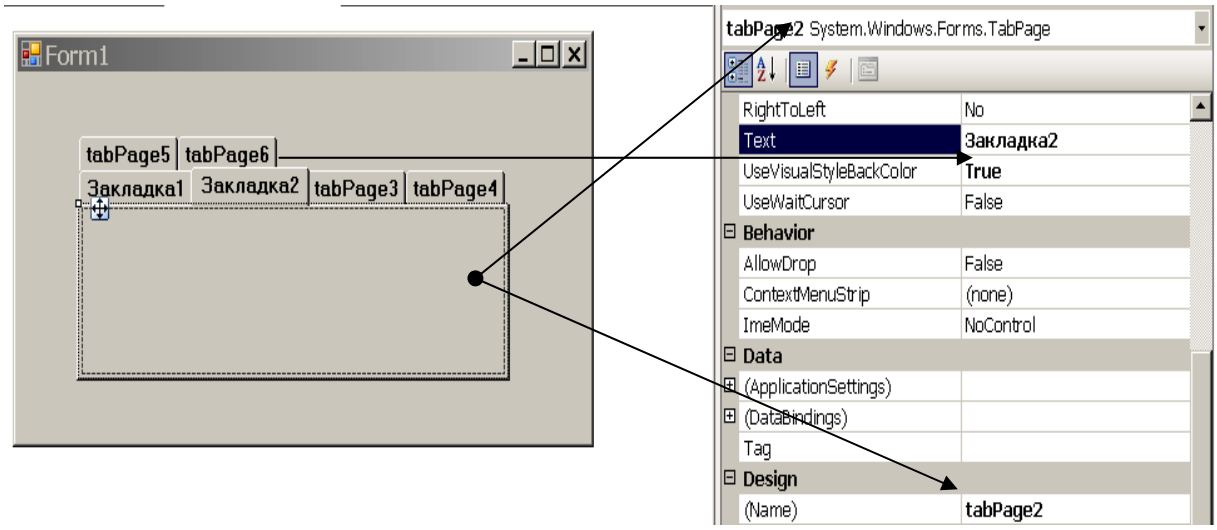


рис.21

2-ой способ. Используя свойство TabPages (3).

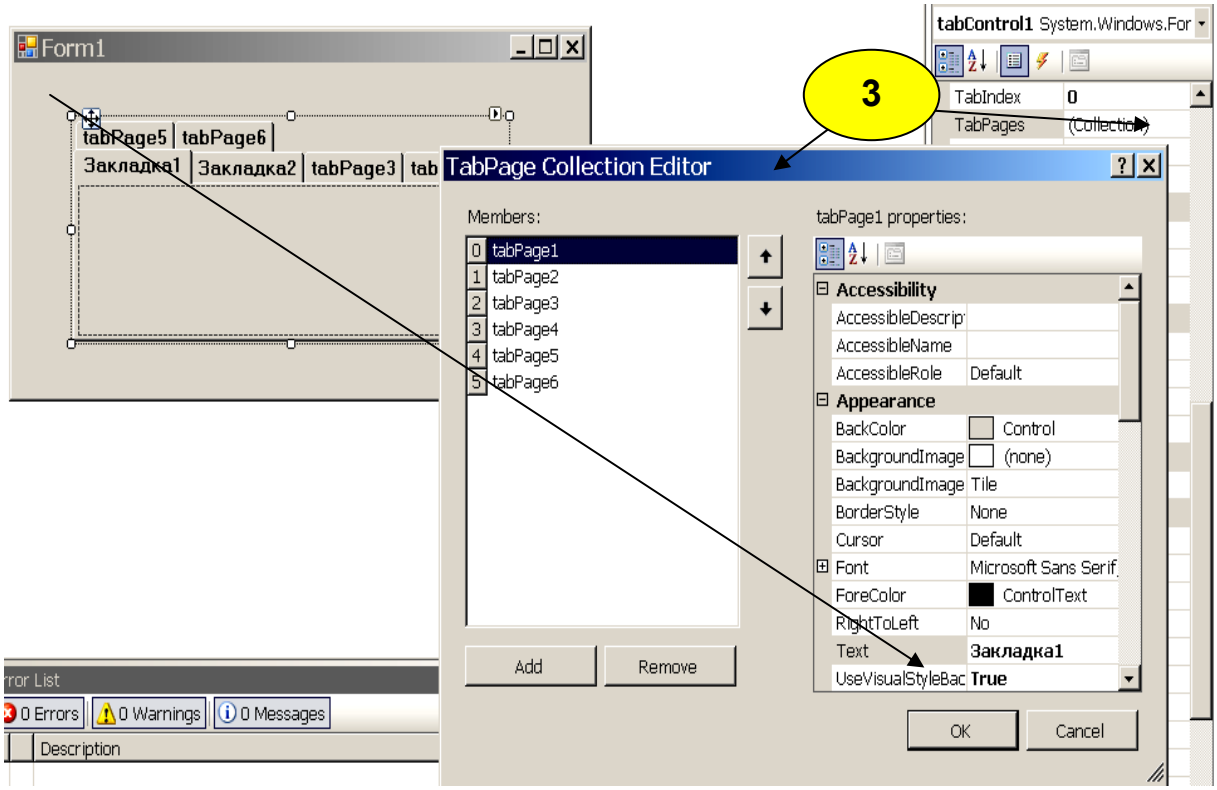


рис.22

Добавление ЭУ на вкладку и обработка событий осуществляется обычным образом. Все элементы управления, размещенные на странице, принадлежат коллекции ЭУ этой страницы.

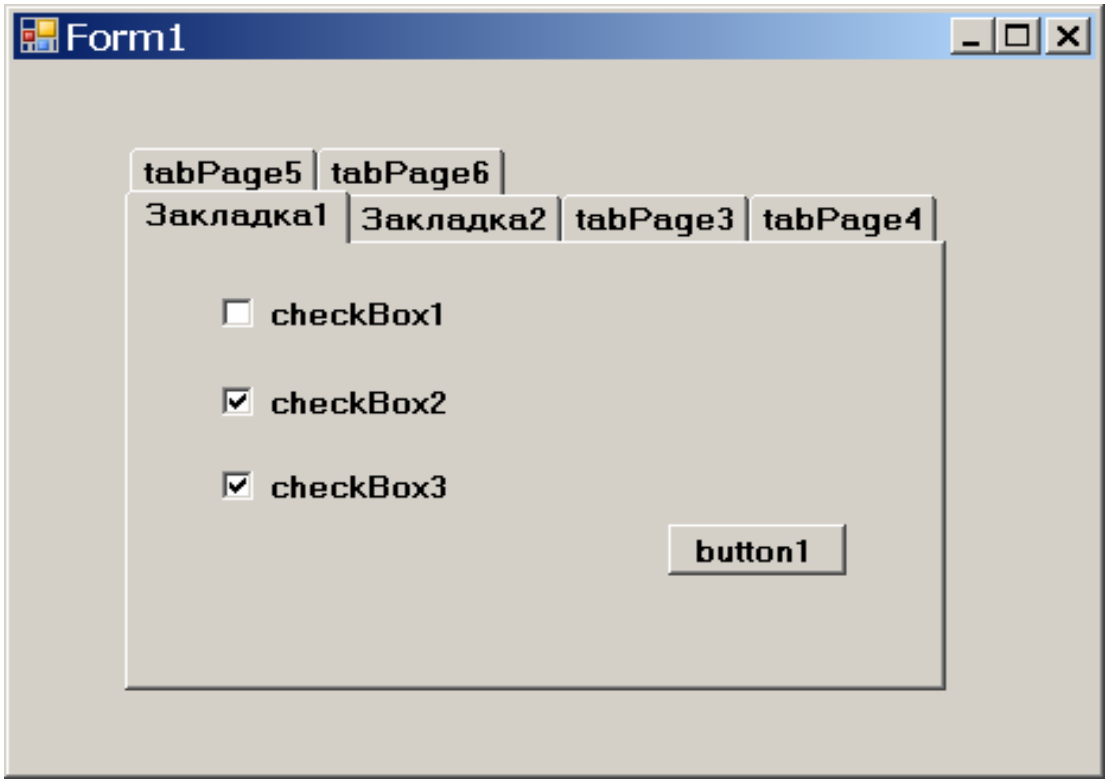


рис.22

Элемент управления CheckBox (флажок)

Флажок в виде галочки отмечает выбор пользователя или умалчиваемое значение.

Если свойство ThreeState установлено в true, то свойство CheckState элемента CheckBox может принимать одно из значений:

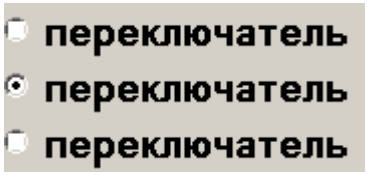
- Checked – элемент отмечен;
- Unchecked – элемент не отмечен;
- Indeterminate – элемент не доступен.

Для получения текущего состояния элемента используется свойство Checked.

```
if ( checkBox1.Checked)
{
   . . .
}
```

Элемент управления RadioButton (переключатель)

Элементы управления Windows Forms RadioButton представляют собой набор как минимум из двух взаимоисключающих вариантов выбора для пользователя.



Группирование элементов.

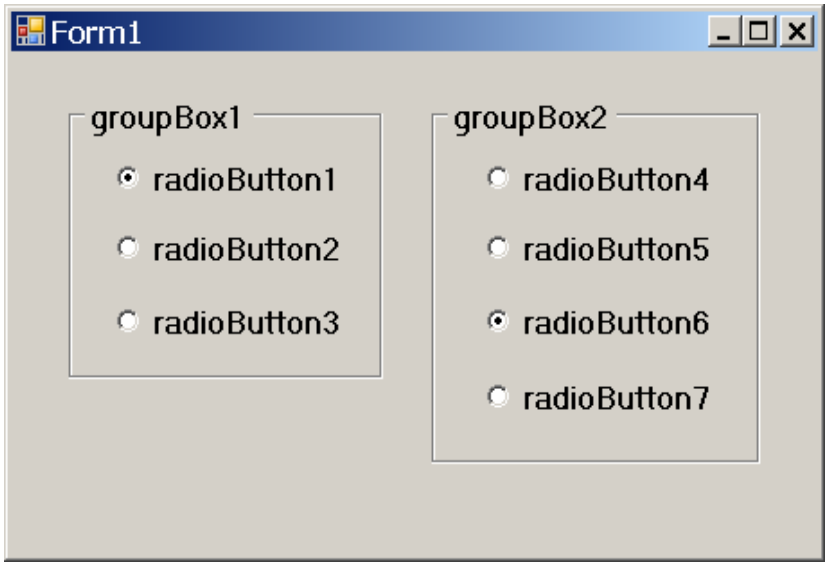


рис.23

При выборе в группе одного переключателя остальные автоматически очищаются. Все элементы управления RadioButton в данном контейнере, таком как Form, составляют группу.

Чтобы создать в одной форме несколько групп, поместите каждую группу в свой собственный контейнер, такой как элемент управления GroupBox или Panel.

Чтобы получить или установить состояние RadioButton, используйте свойство Checked. Вид переключателя можно изменять с помощью свойства Appearance - его можно задать в виде кнопки с фиксацией или в виде стандартного переключателя.

Пример.

Определение состояние флажков группы groupBox1. Группа, кроме флажков, включает и другие ЭУ. Свойство Controls элемента GroupBox возвращает коллекцию элементов управления, включенных в группу.

```
for ( int i=0;   i < groupBox1.Controls.Count;   i++)
{
if (groupBox1.Controls[i] is RadioButton)
{
        RadioButton rb = (RadioButton) groupBox1.Controls[i];

        if ( rb.Checked )
        {
                txt =  rb.Text;
                break;
        }
}
}
```

РАБОТА С ТЕКСТОМ

Диалоговые окна, TextBox, ErrorProvider, RichTextBox, ListBox, ComboBox, CheckListBox, DomainUpDown, NumericUpDown, DateTimePicker, MonthCalendar, DataGridView

1. Диалоговые окна

Диалоговое окно открытия файла - OpenFileDialog ЭУ отображается в панели компонентов.

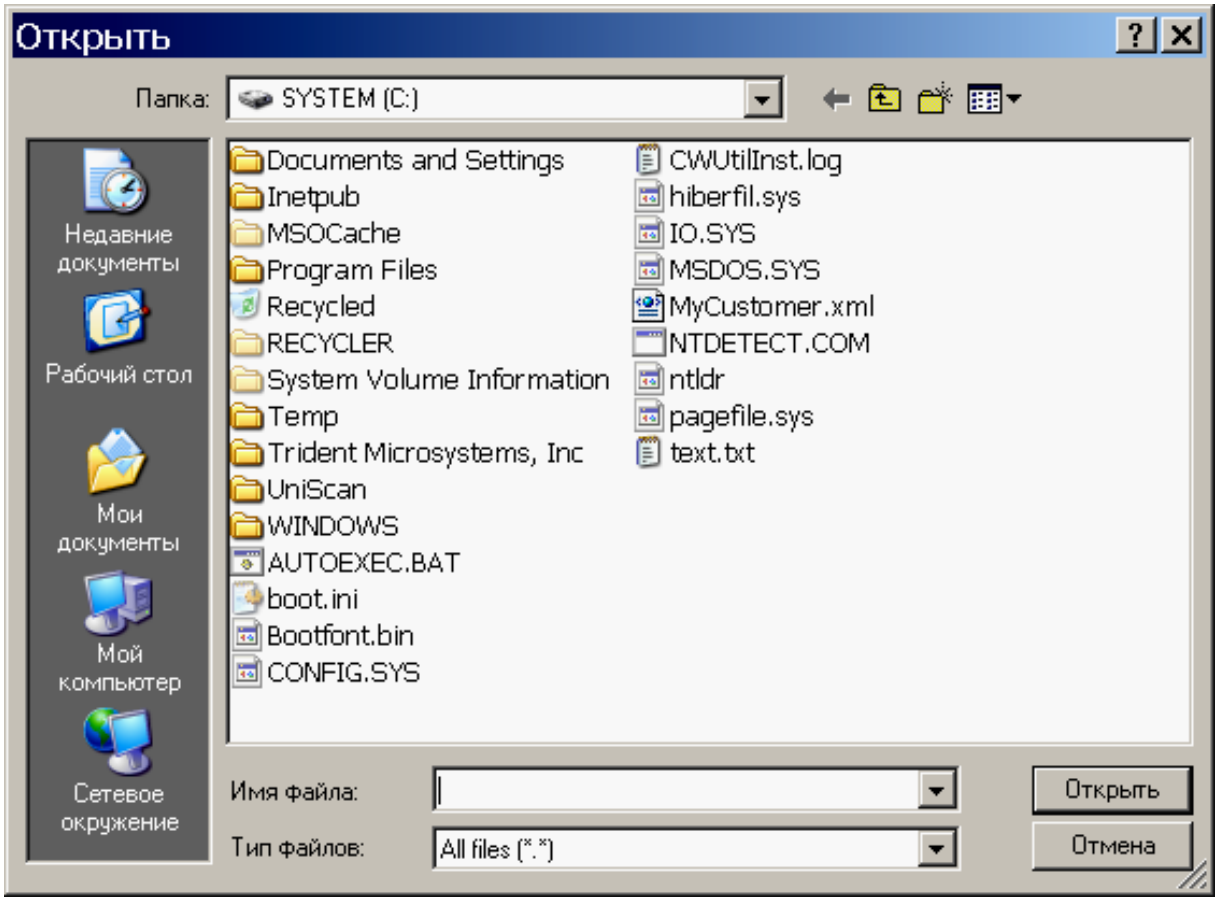


рис.1

В конструкторе Form1():

```
OpenFileDialog   openFileDialog1 = new OpenFileDialog();
```

Пример:

```
private void button1_Click(object sender, System.EventArgs e)
{
    Stream myStream;

   //  OpenFileDialog openFileDialog1 = new OpenFileDialog();

   openFileDialog1.InitialDirectory = "c:\\";
   openFileDialog1.Filter = "txt files (*.txt)|*.txt|All files (*.*)|*.*";
   openFileDialog1.FilterIndex = 2;
   openFileDialog1.RestoreDirectory = true;

   if (openFileDialog1.ShowDialog() == DialogResult.OK  &&
```

```
                                                    openFileDialog1.FileName.Length
> 0)
        {
            myStream = openFileDialog1.OpenFile();          // Только для чтения

            if (myStream != null)
            {
                // Сюда следует вставить код для чтения потока.
                myStream.Close();
            }
        }
    }
```

Свойства и методы класса **OpenFileDialog**:

- InitialDirectory - возвращает или устанавливает начальную папку, отображенную диалоговым окном файла.
- DefaultExt - возвращает или устанавливает расширение имени файла по умолчанию.
- Filter - возвращает или устанавливает текущую строку фильтра имен файлов, которая отображается в поле «Тип файлов» диалогового окна.
- FilterIndex - возвращает или устанавливает индекс фильтра, выбранного в настоящий момент в диалоговом окне файла.
- RestoreDirectory - возвращает или задает bool-значение, показывающее, восстанавливает ли диалоговое окно текущую папку перед закрытием диалога.
- OpenFile() - открывает выбранный пользователем файл в режиме «только чтение». Файл задается свойством FileName.
- FileName.Length – имя файла и длина имени файла.

Диалоговое окно сохранения файла - SaveFileDialog

В конструкторе VS:

```
SaveFileDialog  saveFileDialog1 = new  SaveFileDialog();
```

Далее - аналогично диалогу OpenFileDialog.

```
    // Выводим спецификацию файла и фильтр
    saveFileDialog1.DefaultExt = "*.rtf";

    saveFileDialog1.Filter = "RTF Files|*.rtf";

    if (saveFileDialog1.ShowDialog() == DialogResult.OK &&
                                                    saveFileDialog1.FileName.Length >
0)
    {
            // Сюда следует вставить код для открытия файла с именем
```

```
        //  saveFileDialog1.FileName и вывода информации в файл
    }
```

Диалоговое окно для выбора папки – FolderBrowserDialog

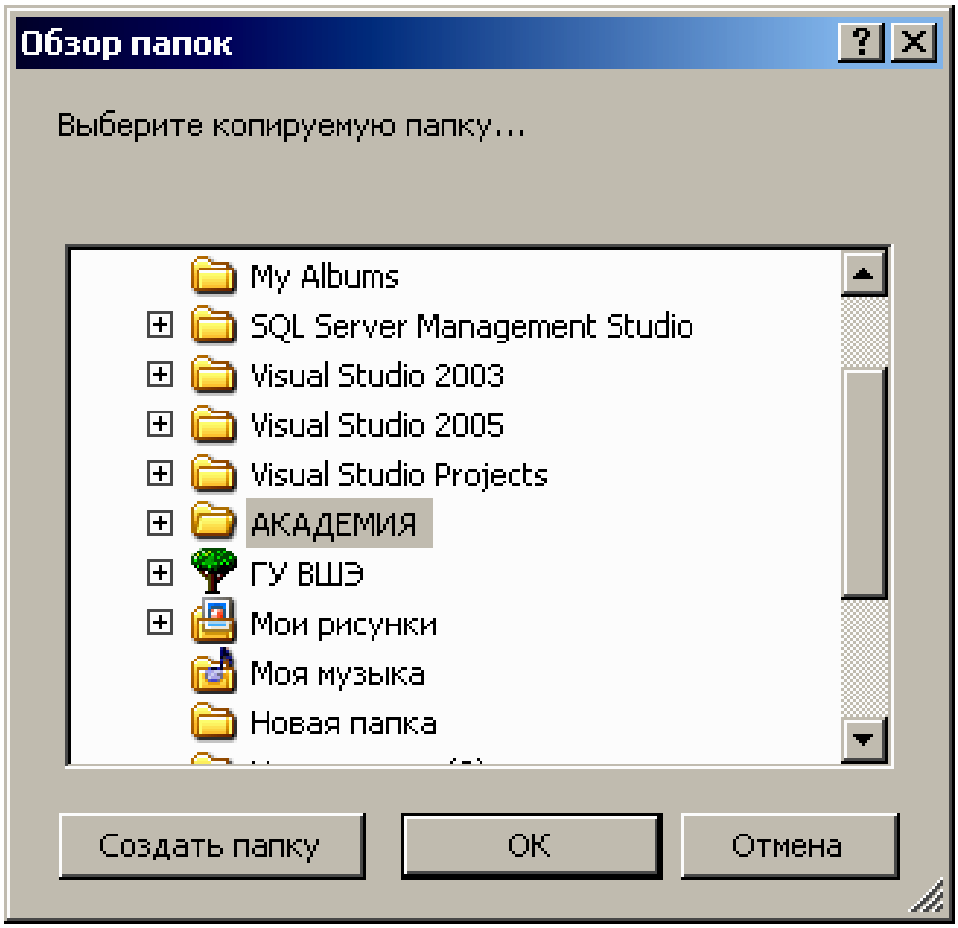


рис.2

Обычно после создания нового FolderBrowserDialog RootFolder устанавливается в расположение, с которого начинается обзор.

Дополнительно имеется возможность установить для SelectedPath абсолютный путь вложенной папки RootFolder, выбираемой изначально.

Также имеется возможность дополнительно установить свойство Description, чтобы предоставить пользователю дополнительные инструкции.

Наконец, вызывается метод ShowDialog для отображения диалогового окна для пользователя.

Когда диалоговое окно закрыто, а результатом диалога из ShowDialog является DialogResult.OK, SelectedPath будет являться строкой, содержащей путь к выбранной папке.

Имеется возможность использовать свойство ShowNewFolderButton для управления, если пользователь может создавать новые папки с помощью кнопки New Folder.

```
private FolderBrowserDialog folderBrowserDialog1;
folderBrowserDialog1 = new System.Windows.Forms.FolderBrowserDialog();

// Show the FolderBrowserDialog.
folderBrowserDialog1.Description = "Выберите копируемую папку...";
```

```
folderBrowserDialog1.SelectedPath = "C:\\Temp ";
DialogResult  result = folderBrowserDialog1.ShowDialog();
if( result == DialogResult.OK )
   folderName = folderBrowserDialog1.SelectedPath;
```

Диалоговое окно выбора шрифта - FontDialog

Отображает диалоговое окно для задания шрифта и его атрибутов.

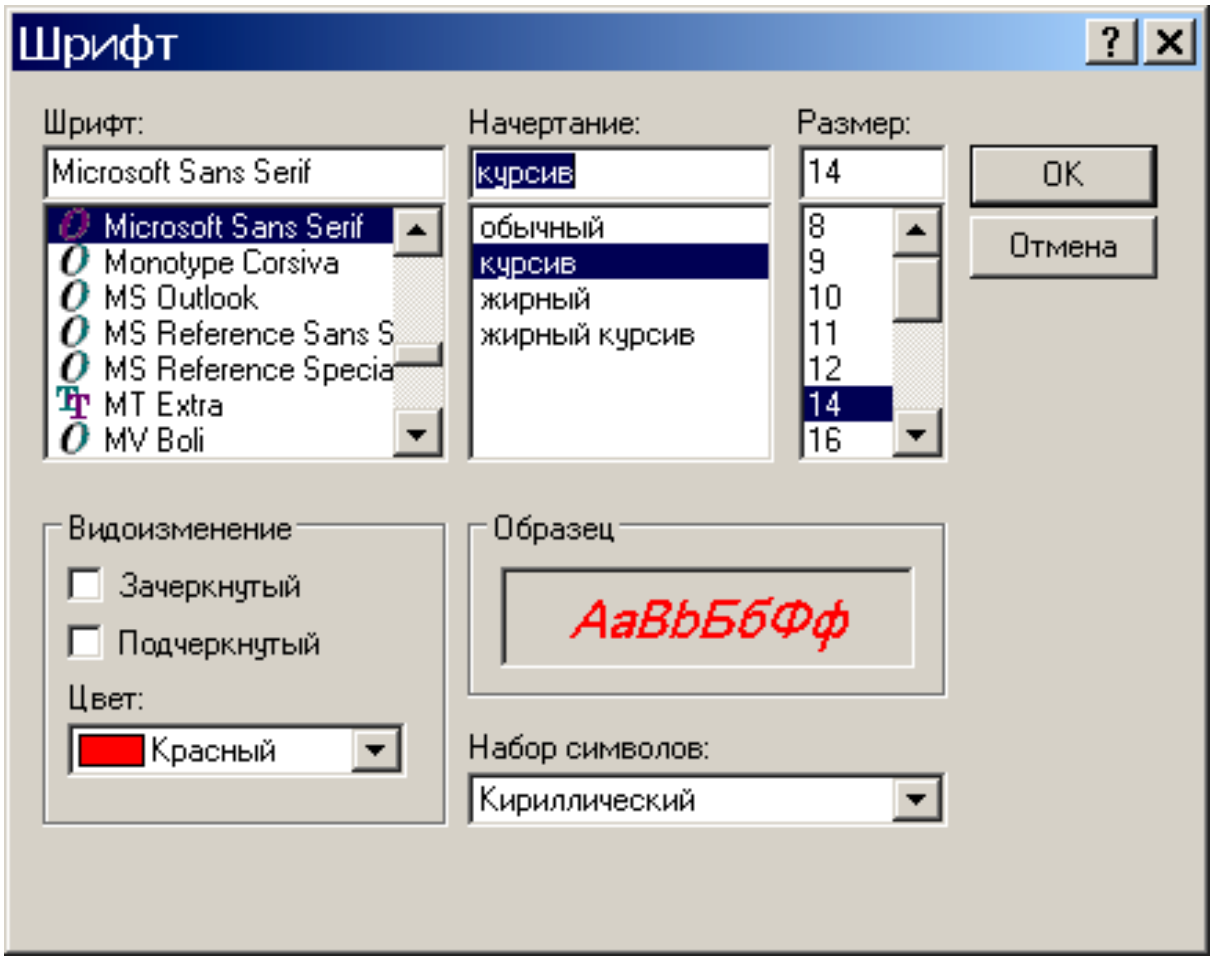


рис.3

Пример. Изменение шрифта и цвета текста ЭУ label1.

```
private void шрифт_Click(object sender, EventArgs e)
 {
    fontDialog1.ShowColor = true;   // Разрешить вывод списка цветов

    fontDialog1.Font   = label1.Font;                  // Текущие
    fontDialog1.Color = label1.ForeColor;              //                значения

    if  ( fontDialog1.ShowDialog() == DialogResult.OK )
    {
       label1.Font = fontDialog1.Font;            // Новый шрифт
       label1.ForeColor = fontDialog1.Color;    // Новый цвет шрифта
    }
 }
```

Диалоговое окно выбора цвета - ColorDialog

Отображает диалоговое окно выбора цвета, позволяющее задать цвет элемента интерфейса.

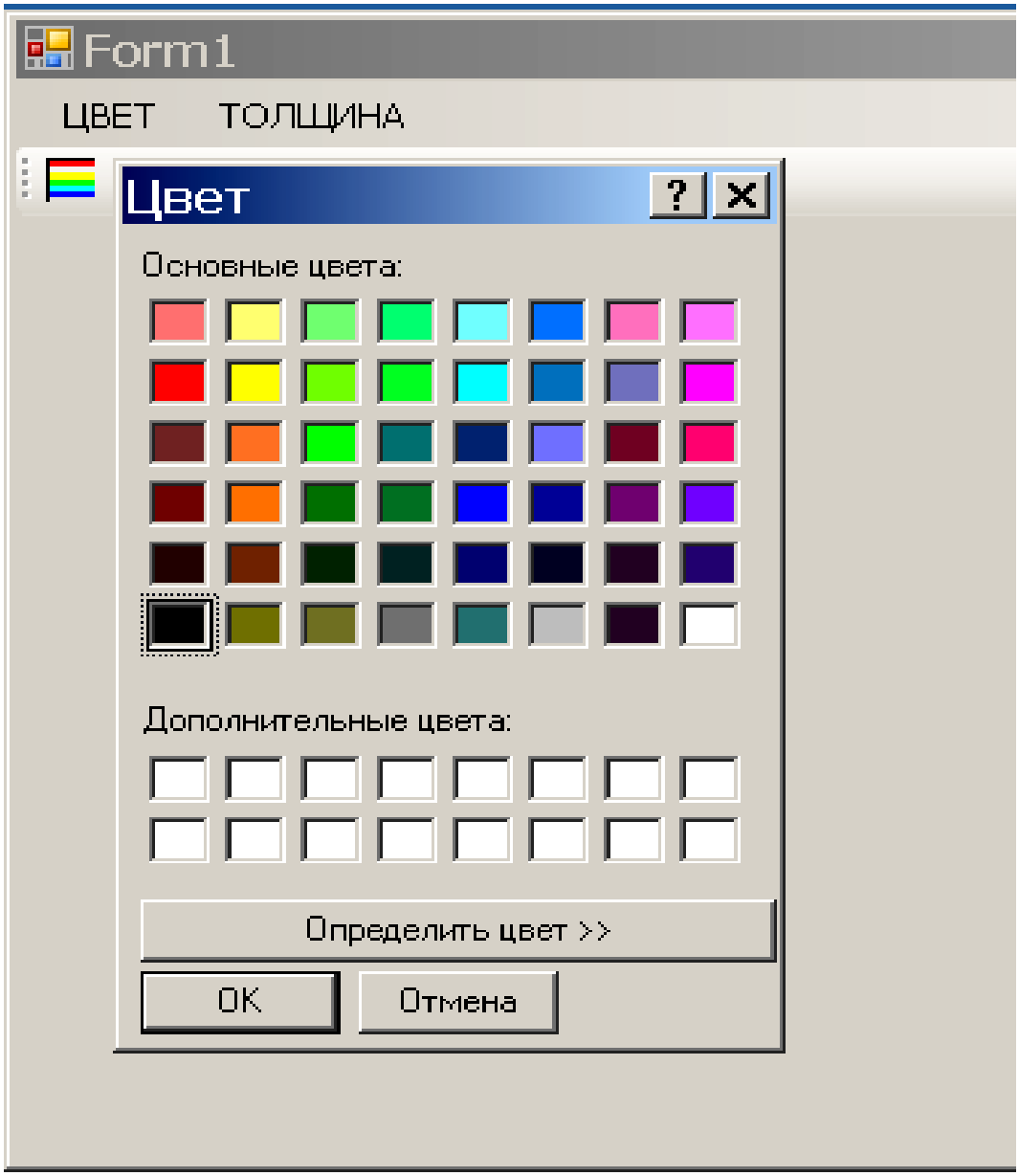


рис. 5

```
private void сектор1_Click(object sender, EventArgs e)
{
        colorDialog1.Color = color1;        // Текущий цвет в рамке

        if (colorDialog1.ShowDialog() == DialogResult.OK)
        {
           color1 = colorDialog1.Color;     // Новый цвет
        }
}
```

2. Элемент управления TextBox

Элемент управления TextBox обычно используется для ввода, отображения и редактирования текста.

В текстовых полях можно выводить несколько строк текста, размещать текст в соответствии с размером элемента управления и добавлять основные элементы форматирования.

В элементе управления TextBox можно вводить или отображать текст только в одном формате. Для отображения текста в различных форматах следует использовать элемент управления RichTextBox.

Текст, отображаемый в элементе управления, содержится в свойстве Text. По умолчанию в текстовом поле можно ввести до 2048 знаков.

Если свойству MultiLine присвоить значение true, это позволит вводить до 32 килобайт текста.

```
textBox1.Text = "Строка текста";
```

Выделение текста программными средствами

Свойство SelectionStart определяет положение курсора в текстовой строке, причем 0 указывает крайнюю левую позицию.

Свойство SelectionLength определяет количество выделяемых символов текста.

В следующем примере выделяется содержимое всего текстового поля textBox1 при обработке события Enter.

```
private void textBox1_Enter (object sender, EventArgs e)
{
  textBox1.SelectionStart = 0;
  textBox1.SelectionLength = textBox1.Text.Length;
  str =   textBox1.SelectedText;
}
```

Свойство TextLength можно использовать для определения числа знаков в строке при решении задач, для которых необходимо знание общего количества знаков, например, поиск определенных строк в тексте элемента управления.

Просмотр нескольких строк

По умолчанию в элементе управления Windows Forms TextBox отображается одна строка текста без полос прокрутки.

Если длина текста превышает размер доступного пространства, отображается лишь часть текста. Эту стандартную настройку можно изменить, присваивая соответствующие значения свойствам MultiLine, WordWrap и ScrollBars.

WordWrap - показывает, переносятся ли автоматически в начало следующей строки слова текста по достижении границы многострочного текстового поля.

AcceptsReturn - возвращает или задает значение, указывающее, что происходит при нажатии клавиши ВВОД в многострочном элементе управления TextBox:

- создается новая строка текста (= true) или
- активируется кнопка стандартного действия формы (=false).

AcceptsTab - возвращает или задает значение, указывающее, что происходит при нажатии клавиши TAB в многострочном элементе управления:

- вводится знак табуляции в текстовом поле (true)
- фокус ввода в форме перемещается к следующему элементу управления в порядке табуляции (false).

Пример

Создается многострочный элемент управления TextBox с вертикальными полосами прокрутки. Для создания многострочного элемента управления "текстовое поле", пригодного для полноценной работы с текстовыми документами, в примере используются свойства AcceptsTab, AcceptsReturn и WordWrap.

```
private void CreateMyMultilineTextBox()
{
   TextBox textBox1 = new TextBox();

   textBox1.Multiline = true;

   textBox1.ScrollBars = ScrollBars.Vertical;

   textBox1.AcceptsReturn = true;

   textBox1.AcceptsTab = true;

   textBox1.WordWrap = true;

   textBox1.Text = "Welcome!";
}
```

Создание текстового поля для ввода пароля

```
private void InitializeTextBox()
{
  // Очистить строку пароля.
  textBox1.Text = "";

  // Символ заполнитель строки пароля.
  textBox1.PasswordChar = '*';
```

```
  // Длина пароля не больше 14 символов.
  textBox1.MaxLength = 14;
}
```

Добавление текста в конец поля

public virtual int TextLength {get;} – длина строки.

1. AppendText() - добавляет текст в конец текущего текста поля.

```
textBox1.AppendText (RichTextBox1.SelectedText);
```

2. Добавляем текст в конец поля

```
textBox1.Text += " Добавляем текст";
textBox1.Text += " Еще раз добавляем текст";
```

3. Добавляем с переносом на новые строки (для Multiline = true).

```
textBox1.Text = "Строка 1\r\nстрока 2\r\nстрока 3\r\n";
textBox1.Text += "Строка 4\r\n";
```

Проверка вводимых значений.

Способ 1. Проверка символов в процессе их ввода.

Для проверки вводимых значений можно использовать обработчик события KeyPress, который получает управление при нажатии любой клавиши в поле TextBox. Событие KeyPress блокирует часть клавиатуры. Мышь работает.

Пример.

Проверяется элемент textBox1, который не должен содержать буквы. Если вводится буква, то в случае e.Handled=true она не отображается в поле.

```
private void textBox1_KeyPress (object sender, KeyPressEventArgs e)
{
   if ( ! char.IsDigit (e.KeyChar) )
   {                                        // Событие не обработано,
          e.Handled = true;                 // запретить отображение символа
          label1.Text ="Поле не может содержать буквы";
   }
}
```

Способ 2. Проверка результата ввода в конце.

При переключении фокуса ввода с элемента управления генерируется событие Validating, которое позволяет работать с клавиатурой, но блокирует другие действия пользователя, то есть не дает переключиться на другие ЭУ, пока не будет исправлена ошибка. Этим процессом управляет свойство Cancel параметра e обработчика события.

Свойство e.Cancel - получает или задает значение, показывающее, следует ли отменить событие. Если e.Cancel=true, то фокус ввода не покинет ЭУ, сгенерировавший это событие, до тех пор пока не будет установлено e.Cancel=false. По умолчанию e.Cancel=false.

Пример.

```
private void textBox1_Validating (object sender, CancelEventArgs e)
 {
    if (textBox1.Text == "")
            e.Cancel = false;
    else
    {
            try
            {
                    double.Parse(textBox1.Text);
                    e.Cancel = false;
            }
            catch
            {
                    e.Cancel = true;
                    label1.Text ="Поле не может содержать буквы";
            }
    }
}
```

Использование ЭУ ErrorProvider при проверке вводимых значений

Компонент ErrorProvider позволяет сигнализировать об ошибке с помощью небольшой иконку.

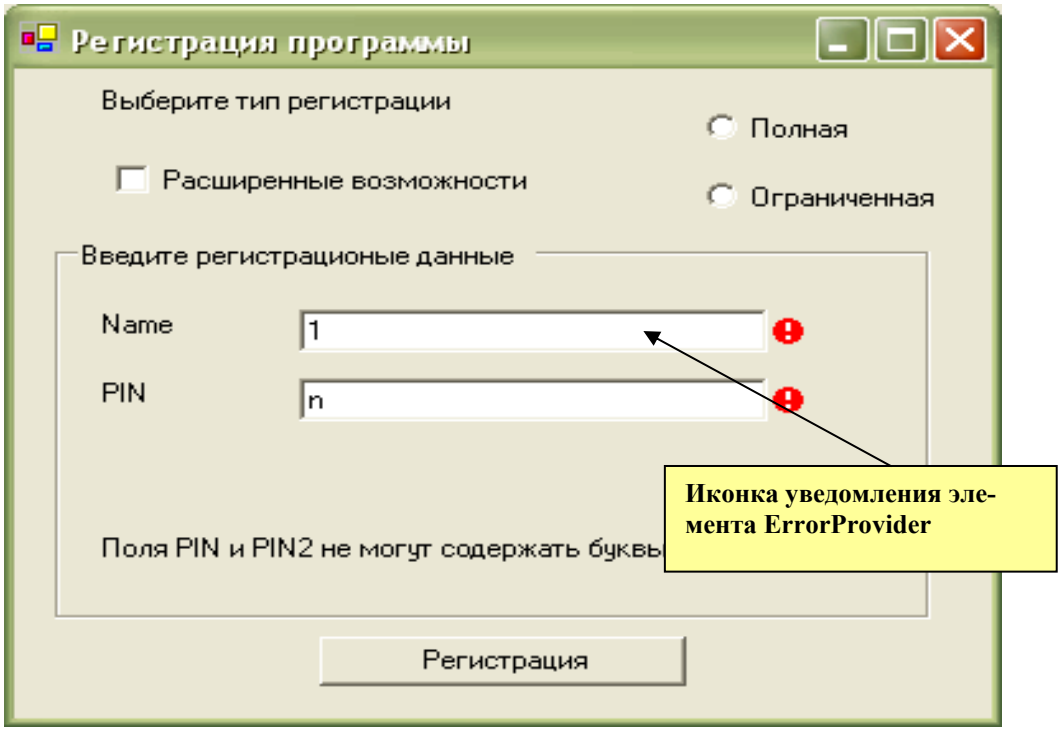


рис.6

Пример.

Перетащим на форму ЭУ ErrorProvider. Будет создан объект errorProvider1. Создадим обработчика textBox1_Validated.

```
private void textBox1_Validated (object sender,  EventArgs e)
{
   if (textBox1.Text != "")
   {
            try
            {   double.Parse(textBox1.Text);  }

            catch
            {
                errorProvider1.SetError (textBox1, "Must be number");
                    label1.Text = "Поле не может содержать буквы";
                 }
   }
}
```

Вторым параметром в методе SetError передается строка с описанием ошибки, которая может быть выведена на форму при наведении курсора на значок.

Событие Validated возникает в случае потери элементом фокуса ввода после события Validating, если при обработке последнего было установлено e.Cancel=false.

Некоторые свойства элемента ErrorProvider, отвечающие за внешний вид иконки:

Свойство	Описание	Значение по умолчанию
Blinkrate	Частота мерцания в миллисекундах	250
BlinkStyle	Стиль появления иконки. Возможны следующие варианты: BlinkIfDifferentError – иконка появляется при ошибке, мерцает несколько раз и останавливается; AlwaysBlink – при ошибке иконка мерцает постоянно; NeverBlink – иконка не появляется вообще	BlinkIfDifferentError
Icon	Изображение иконки. Можно использовать другие файлы иконок (.ico)	(Icon)

2. Элемент управления RichTextBox

Элемент управления RichTextBox обычно используется для предоставления возможностей изменения и отображения текста, схожих с возможностями текстовых редакторов, таких как Microsoft Word.

Элемент управления RichTextBox выполняет те же функции, что и элемент управления TextBox, но помимо этого он позволяет:

- отображать шрифты, цвета и ссылки,
- загружать текст и вложенные изображения из файлов,
- отменять и повторять операции редактирования, а также
- искать заданные символы.

Элемент управления RichTextBox, как и TextBox, позволяет отображать полосы прокрутки, однако в отличие от TextBox он по умолчанию отображает и горизонтальную, и вертикальную полосы прокрутки, а также поддерживает дополнительные параметры их настройки. Элемент управления RichTextBox содержит множество свойств, которые можно использовать при применении форматирования к любой части текста в элементе управления.

Перед тем как изменить форматирование текста, этот текст необходимо выделить. Только выделенному тексту можно назначить форматирование символов и абзацев.

После того как выделенному тексту был назначен какой-либо параметр, текст, введенный после выделенного, будет форматирован с тем же параметром, пока этот параметр не будет изменен, или не будет выделена другая часть документа элемента управления.

Свойство SelectionFont позволяет выделять текст полужирным шрифтом или курсивом. Кроме того, с помощью этого свойства можно изменять размер и гарнитуру текста. Свойство SelectionColor позволяет изменять цвет текста.

Свойство SelectionBullet следует использовать для создания маркированных списков.

Настройка форматирования абзацев осуществляется также с помощью свойств SelectionIndent, SelectionRightIndent и SelectionHangingIndent.

В элементе управления RichTextBox можно отображать содержимое обычного текстового файла, файла текста в формате Юникода или файла формата RTF. Для этого вызывается метод LoadFile().

Метод LoadFile() также можно использовать для загрузки данных из потока.

С помощью метода SaveFile() можно сохранить текст в файле в заданном формате. Метод SaveFile() позволяет сохранять данные в открытый поток примерно так же, как метод LoadFile().

Пример. При нажатии кнопки btnOpenFile отображается диалоговое окно Открытия файла. Имя выбранного файла используется в методе richTextBox1.LoadFile().

```
private void btnOpenFile_Click(object sender, System.EventArgs e)
{
   if(openFileDialog1.ShowDialog() == DialogResult.OK)

                richTextBox1.LoadFile (openFileDialog1.FileName);  // RTF
}
```

Пример

В следующем примере при обработке сообщения о загрузке формы элемент управления RichTextBox загрузит файл C:\MyDocument.RTF в элемент управления и осуществит поиск первого экземпляра слова «Текст».

После этого код изменит стиль, размер и цвет шрифта выделенного текста и сохранит изменения в новом файле C:\MyDocument2.RTF.

Ввод текста после слова «Текст» будет осуществляться с новыми параметрами.

```
private void Form1_Load (object sender, EventArgs e)
{
   richTextBox1.LoadFile ("C:\\MyDocument.RTF");
   richTextBox1.Find ( "Текст", RichTextBoxFinds.MatchCase);

   richTextBox1.SelectionFont = new Font ("Verdana", 12,
                                                                  FontStyle.Bold);
   richTextBox1.SelectionColor = Color.Red;

   richTextBox1.SaveFile ("C:\\MyDocument2.RTF",
                                          RichTextBoxStreamType.RichText);
}
```

Перечисление RichTextBoxStreamType:

Имя члена	Описание
PlainText	Поток открытого текста с пробелами вместо объектов OLE.
RichNoOleObjs	Поток в формате RTF с пробелами вместо объектов OLE. Это значение действительно только для использования с методом SaveFile элемента управления RichTextBox.
RichText	Поток в формате RTF.
TextTextOleObjs	Поток открытого текста с текстовым представлением объектов OLE. Это значение действительно только для использования с методом SaveFile элемента управления RichTextBox.
UnicodePlainText	Текст в формате Юникод. Поток текста с пробелами вместо объектов OLE.

Пусть в файле MyDocument.rtf хранится текст:

В следующем примере создается элемент управления RichTextBox, который загрузит файл RTF в элемент управления и осуществит поиск первого экземпляра слова «Текст».

После этого код изменит стиль, размер и цвет шрифта выделенного текста и сохранит изменения в исходном файле формата RTF корневой папки диска C.

Результат поиска слова «Текст»:

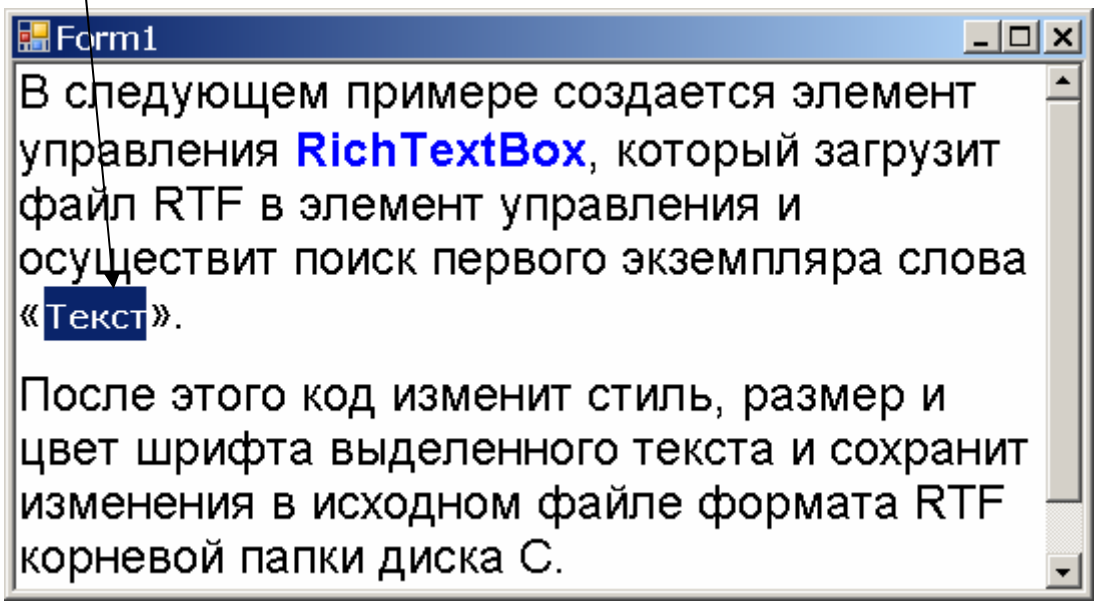


рис.7

Допишем текст до и после слова «Текст».

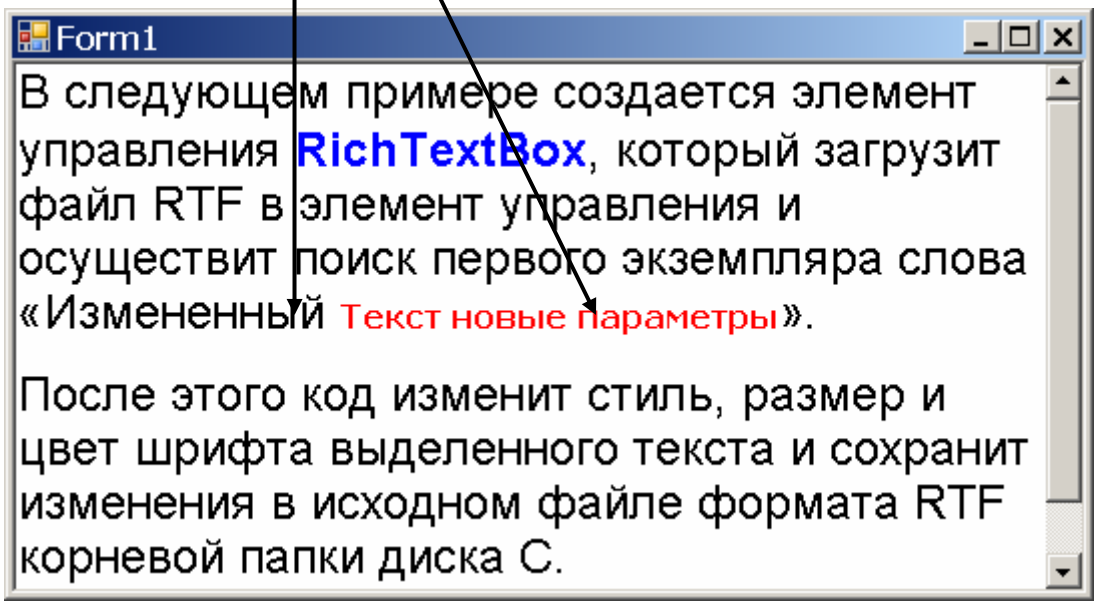


Рис.8

В следующем примере метод SaveFile() используется для указания того, что файл должен быть сохранен как текстовый файл в кодировке ASCII, а не в стандартном формате RTF.

```
SaveFileDialog saveFile1 = new SaveFileDialog();

  saveFile1.DefaultExt = "*.rtf";
  saveFile1.Filter = "RTF Files|*.rtf";

  if(saveFile1.ShowDialog() == System.Windows.Forms.DialogResult.OK
                                        &&   saveFile1.FileName.Length > 0)
  {
    // Сохранить как текстовый файл в кодировке ASCII
    richTextBox1.SaveFile (saveFile1.FileName,
                                                    RichTextBoxStreamType.PlainText);
  }
```

Добавление строк в конец

Осуществляется по аналогии с TextBox:

```
    richTextBox1.AppendText (textBox1.SelectedText);

   richTextBox1.Text += " Добавляемый текст";
```

Вместо \r\n достаточно \n:

```
   richTextBox1.Text = "Строка 1\nстрока 2\n";
   richTextBox1.Text += "строка 3\n";
```

3. Элементы управления ListBox, ComboBox и CheckListBox

Примеры ЭУ: ComboBox, ListBox и CheckListBox

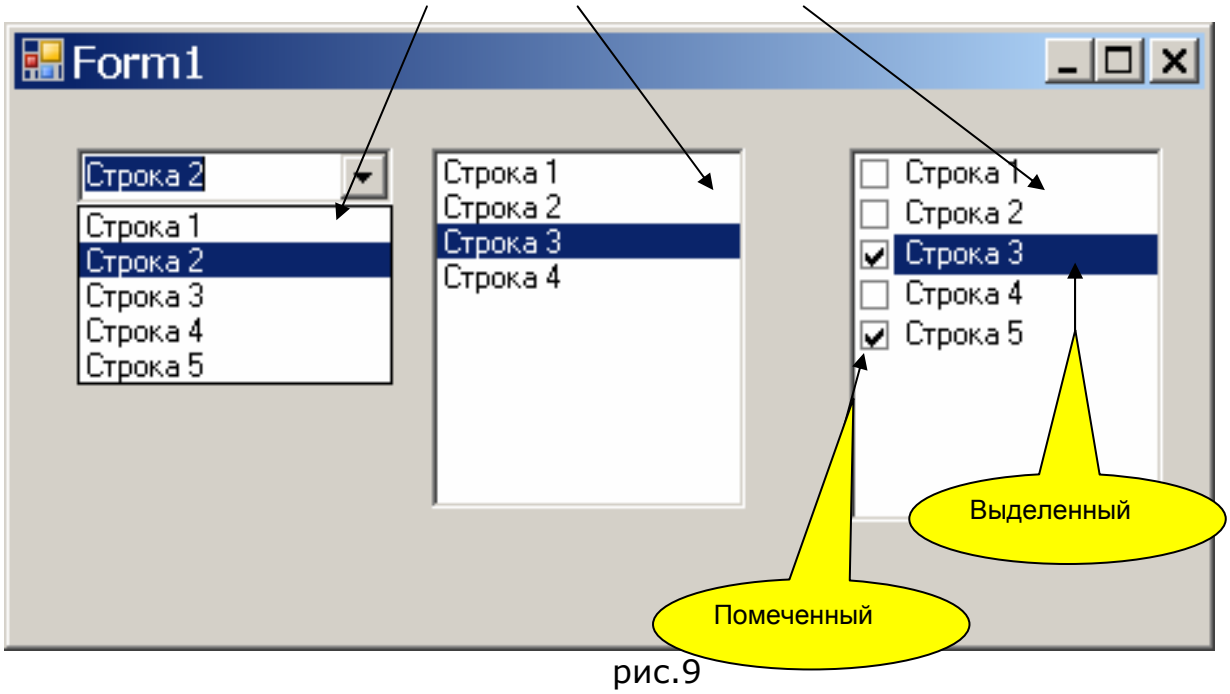


рис.9

Элемент управления ListBox позволяет отобразить список позиций, из которых пользователь может выбрать нужную щелчком мыши.

Элемент управления ComboBox используется для вывода данных в раскрывающемся поле со списком. По умолчанию элемент управления ComboBox появляется в виде двух частей:

● верхняя часть представляет собой текстовое поле, в которое пользователь может ввести элемент списка.

● вторая часть представляет собой список, в котором отображается список элементов, один из которых пользователь может выбрать.

Элемент управления CheckedListBox является расширением элемента управления ListBox. Он выполняет практически все функции списка, а кроме того, в нем может отображаться галочка рядом с элементами списка.

Другое различие между этими элементами управления заключается в том, что списки с помеченными элементами поддерживают только режим DrawMode.Normal; в таких списках можно выделить только один элемент или не выделить ни одного. Следует иметь в виду, что выделенный элемент отмечается в форме с помощью цвета и не обязательно является помеченным элементом.

Элемент управления ListBox

Полосы прокрутки

Если не все элементы могут одновременно отобразиться в поле списка, к элементу управления ListBox автоматически добавляется вертикальная полоса прокрутки.

Если для свойства MultiColumn задано значение true, элементы списка отображаются в нескольких столбцах и появляется горизонтальная полоса прокрутки. Это позволяет отобразить больше позиций списка и устраняет необходимость его вертикальной прокрутки для поиска нужной позиции.

Если для свойства MultiColumn задано значение false, элементы списка отображаются в одном столбце и появляется вертикальная полоса прокрутки.

Если для ScrollAlwaysVisible задано значение true, полоса прокрутки появляется независимо от числа элементов.

Выборка элементов

Свойство SelectionMode определяет, сколько элементов списка можно выбрать одновременно (ничего, одно или несколько: None, One, многострочный (MultiSimple) или многоколоночный (MultiExtended)).

Свойство SelectedIndex возвращает целочисленное значение, соответствующее первому выбранному элементу списка.

Выбранный элемент можно изменить программными средствами, изменив в коде значение SelectedIndex; соответствующий элемент списка будет выделен в форме Windows.

Если не выбран ни один из элементов, свойство SelectedIndex имеет значение -1.

Если выбран первый элемент в списке, значение свойства SelectedIndex равно 0.

Если выбрано несколько элементов, значение свойства SelectedIndex отражает выбранный элемент, появляющийся первым в списке.

Свойство SelectedItem аналогично свойству SelectedIndex, но возвращает сам элемент, обычно в виде строкового значения.

С помощью методов BeginUpdate() и EndUpdate() можно добавлять к ListBox большое число позиций, причем элемент управления не будет перерисовываться при добавлении к списку каждой новой позиции, пока не будет выполнен метод EndUpdate().

Методы FindString() и FindStringExact() позволяют найти в списке позицию, содержащую определенную строку поиска.

Свойства Items, SelectedItems и SelectedIndices предоставляют доступ к трем коллекциям, используемым ListBox.

Свойство Items получает коллекцию позиций элементов управления ListBox.

```
public ListBox.ObjectCollection Items {get;}
public class ListBox.ObjectCollection : IList, ICollection, IEnumerable
```

Свойство SelectedItems получает коллекцию выделенных позиций ЭУ ListBox.

```
public ListBox.SelectedObjectCollection SelectedItems {get;}
public class ListBox.SelectedObjectCollection : IList, ICollection, IEnumerable
```

Свойство SelectedIndices получает коллекцию индексов выделенных позиций ЭУ ListBox.

```
public ListBox.SelectedIndexCollection SelectedIndices {get;}
public class ListBox.SelectedIndexCollection : IList, ICollection, IEnumerable
```

Класс коллекции	Применение в списке
ListBox.ObjectCollection	Объединяет все позиции, содержащиеся в ЭУ ListBox.
ListBox.SelectedObjectCollection	Содержит коллекцию выделенных позиций, являющуюся подмножеством всех позиций, содержащихся в ListBox.
ListBox.SelectedIndexCollection	Содержит коллекцию индексов выделенных позиций, являющееся подмножеством множества индексов ListBox.ObjectCollection.

В списках хранятся ссылки на объекты!

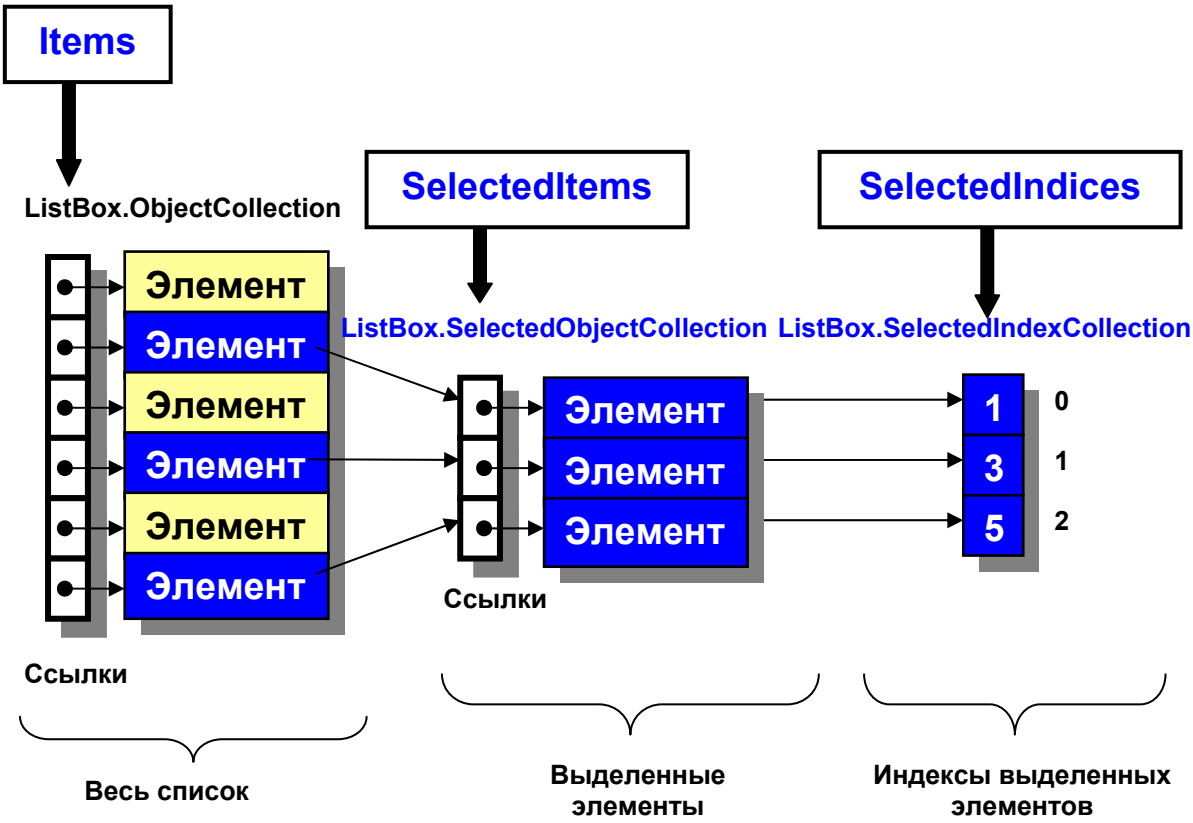


рис.10

Свойство Items.Count отражает число элементов в списке.

Это значение всегда на единицу больше максимально возможного значения свойства SelectedIndex, поскольку для свойства SelectedIndex индексация ведется от нуля.

Чтобы добавить или удалить элементы в список ListBox, используйте методы Items.Add(), Items.Insert(), Items.Clear() или Items.Remove(). Кроме того, можно добавить элементы в список с помощью свойства Items во время разработки.

Метод Add() может принимать любой объект при добавлении элементов в ListBox. Для отображения объекта в ListBox элемент управления использует текст, возврвщаемый методом ToString() объекта, если только имя элемента в объекте не было указано в свойстве DisplayMember.

Кроме добавления позиций с помощью метода Add() класса ListBox.ObjectCollection, добавлять их можно также с помощью свойства DataSource класса ListControl (см. ниже).

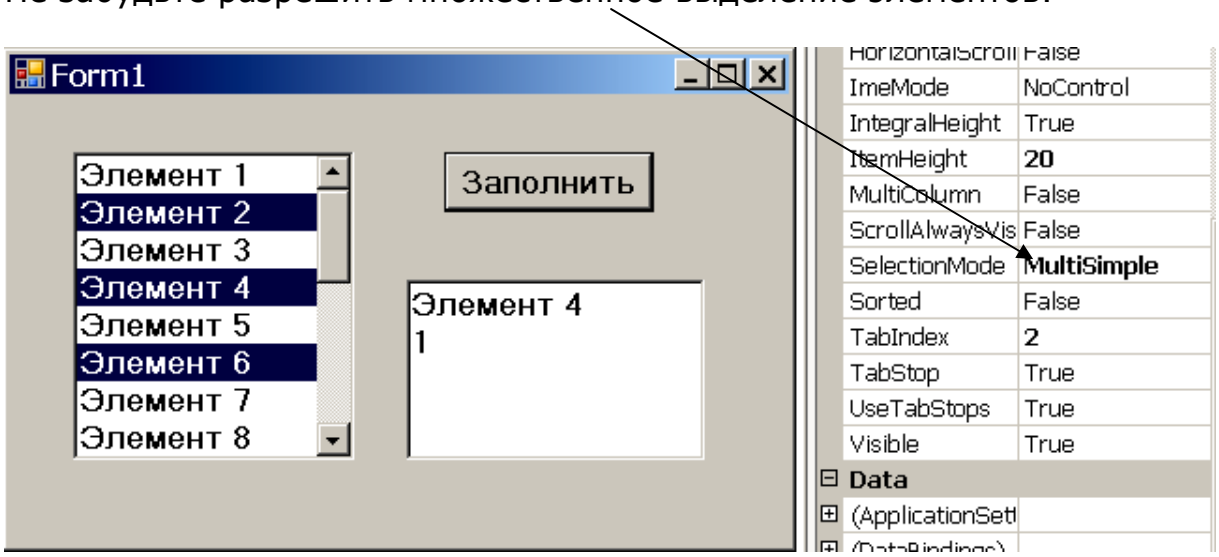
Пример 1

В следующем примере показан способ создания элемента управления ListBox, отображающего несколько позиций в столбцах. При этом в списке элемента управления может быть выбрано более одной позиции. Код, приведенный в примере, добавляет 50 позиций в ListBox с помощью метода Add класса ListBox.ObjectCollection, а затем выбирает три позиции из списка с помощью метода SetSelected. Затем отображаются значения из коллекций ListBox.SelectedObjectCollection (с помощью свойства SelectedItems) и ListBox.SelectedIndexCollection (с помощью свойства SelectedIndices). В примере предполагается, что код расположен в Form и вызывается оттуда же.

```
listBox1.SelectionMode = SelectionMode.MultiSimple; //в конструкторе VS

private void button1_Click (object sender, EventArgs e)
{
   listBox1.BeginUpdate();          // Начать добавление элементов.

      for ( int x = 1; x <= 20; x++)                  // Цикл добавления 50 элементов
        {
            listBox1.Items.Add ("Элемент " + x);
        }

   listBox1.EndUpdate();          // Высветить обновленный список

   // Выделить три элемента с индексами 1, 3 и 5
   listBox1.SetSelected (1, true);
   listBox1.SetSelected (3, true);
   listBox1.SetSelected (5, true);

   //Отобразить второй выделенный элемент [1] и индекс первого [0]
   richTextBox1.Text = listBox1.SelectedItems[1] + "\n";
   richTextBox1.Text += listBox1.SelectedIndices[0];
}
```

SetSelected() - выделяет указанную позицию в ListBox или снимает с нее выделение.

Не забудьте разрешить множественное выделение элементов.



Пример 2

Создание ListBox из объектов класса PhoneList, имеющих поля

string name, phone ;

Вопрос. Что отображать, а что возвращать в случае выделения строки, если в объекте несколько полей, а отображается только одно?

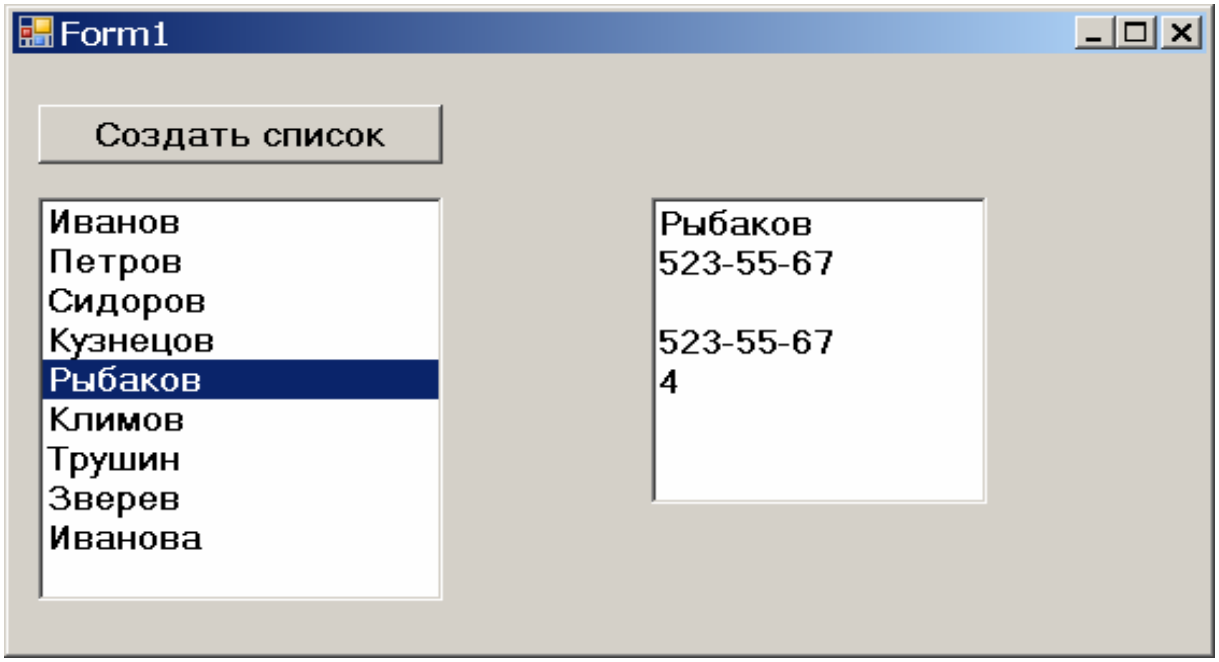


рис.11

Список отображает строки, которые определяет свойство DisplayMember. Если это свойство не использовать, то будут отображаться строки, возвращаемые методом ToString(). В примере он возвращает номер телефона.

Примечание. Свойство DisplayMember должно получать в качестве значения строку с именем открытого свойства, а не с именем открытого поля. Если вместо свойства использовать открытое поле, то данная возможность работать не будет.

Свойство SelectedItem возвращает ссылку на объект, соответствующий выбранному значению. Используя эту ссылку, можно получить доступ ко всем открытым членам объекта.

```
using System;
using System.Collections.Generic;
using System.Text;

namespace Ctrl_ListBox
{
   class PhoneList
   {
      string name, phone ;

      public PhoneList (string n, string p)
      {
         name = n;
         phone = p;
      }

      public string Name
      {
         get { return name; }
      }
```

```
        public string Phone
        {
            get { return phone; }
        }

        public override string ToString()
        {
            return Phone;
        }
    }
}

      private void button1_Click(object sender, EventArgs e)
        {
            listBox1.Items.Clear();
            listBox1.DisplayMember = "Name";

            listBox1.BeginUpdate();

              // Отображаются строки, указанные в свойстве DisplayMember
            listBox1.Items.Add (new PhoneList ("Иванов", "123-45-67"));
            listBox1.Items.Add (new PhoneList ("Петров", "223-45-68"));
            listBox1.Items.Add (new PhoneList ("Сидоров", "323-45-69"));
            listBox1.Items.Add (new PhoneList ("Кузнецов", "423-45-00"));
            listBox1.Items.Add (new PhoneList ("Рыбаков", "523-55-67"));
            listBox1.Items.Add (new PhoneList ("Климов", "623-66-67"));
            listBox1.Items.Add (new PhoneList ("Трушин", "723-77-67"));
            listBox1.Items.Add (new PhoneList ("Зверев", "823-88-67"));
            listBox1.Items.Add (new PhoneList ("Иванова", "923-99-67"));
            listBox1.EndUpdate();
        }

    private void listBox1_SelectedIndexChanged(object sender, EventArgs e)
    {
            PhoneList obj;
            obj = (PhoneList) listBox1.SelectedItem;

            richTextBox1.Text   = obj.Name + "\n" ;
            richTextBox1.Text += obj.Phone + "\n\n " ;

         // Выводится строка, возвращаемая методом ToString(), т.е. Phone.
            richTextBox1.Text += listBox1.SelectedItem.ToString()  + "\n";

            richTextBox1.Text += listBox1.SelectedIndex.ToString() + "\n";

              // Следующий фрагмент не работает, так как не используется
              // свойство DataSource
              //    listBox1.ValueMember = "Phone";
              //    richTextBox1.Text = listBox1.SelectedValue + "\n";

        }
    }
}
```

Пояснения:

Поскольку списки хранят ссылки на объекты, то необходимо приводить эту ссылку к типу объекта.

```
obj = (PhoneList) listBox1.SelectedItem;
```

ToString() можно не указывать, если значение участвует в операции со строками: +=, +.

Пример 3

В следующем примере вместо

```
listBox1.DisplayMember = "Name";
```

использовано

```
listBox1.DisplayMember = "Phone";
```

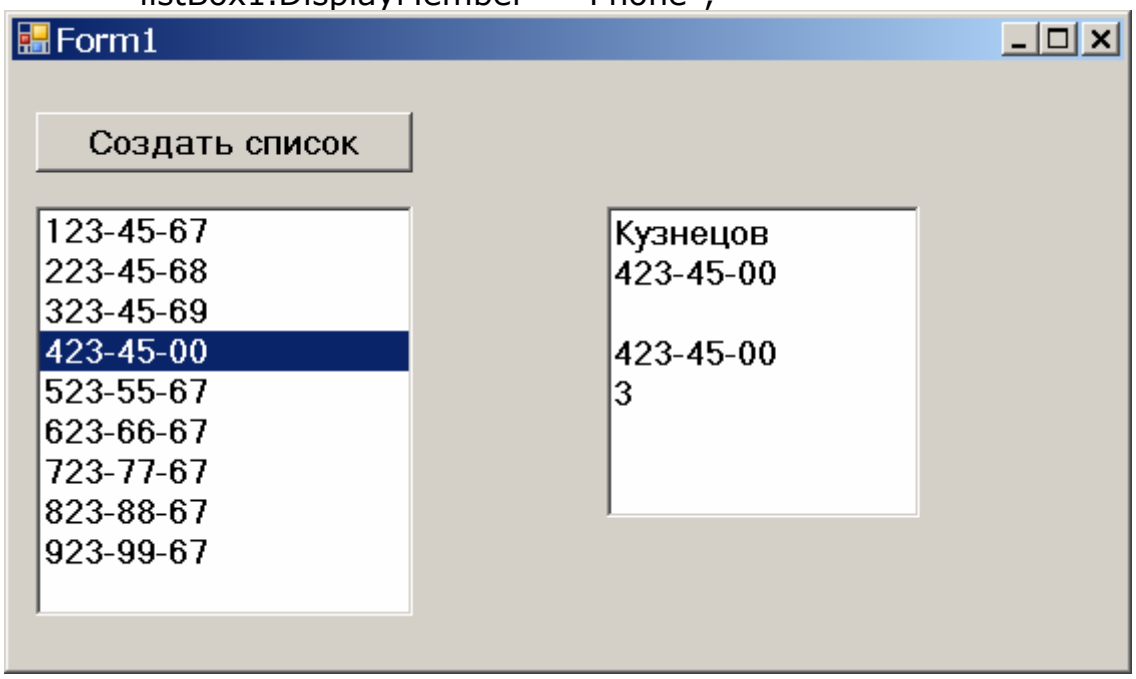


рис.12

Определение источника данных для ЭУ ListBox или ComboBox

Как быть, если данные для списка уже существуют в виде массива или таблицы?

Для элементов управления ListBox и ComboBox можно выполнить привязку к данным.

Чаще всего эти элементы управления используются для поиска сведений в базе данных, ввода новых данных или редактирования существующих.

DataSource - возвращает или задает источник данных для этого ListControl.

ValueMember - возвращает или задает строку, указывающую свойство источника данных, значение которого извлекается.

SelectedValue - возвращает или задает значение свойства, определенного свойством ValueMember.

В следующем примере используется связывание ЭУ ListBox с динамическим массивом ArrayList с помощью свойства DataSource:

```
listBox1.DataSource = arrayList;
```

Для привязки списка к объекту необходимо, чтобы класс объекта реализовывал интерфейс IList. Если этот класс реализует и интерфейс IBindingList, то при обновлении объекта будет автоматически обновляться и список ListBox.

1. В отличие от предыдущего примера, метод ToString() возвращает фамилию.
2. Кроме того, используется свойство
listBox1.ValueMember = "Phone";
которое показывает, значение какого свойства класса PhoneList должно возвращать свойство SelectedValue:
richTextBox1.Text += listBox1.SelectedValue + "\n";
3. Так как listBox1.DataSource = arrayList порождает событие SelectedIndexChanged с выборкой первого элемента, то вместо обработчика событий listBox1_SelectedIndexChanged используется listBox1_Click.

```
private void button1_Click(object sender, EventArgs e)
{
   listBox1.DataSource = null;     //для повторения

   ArrayList arrayList = new ArrayList();

   arrayList.Add (new PhoneList ("Иванов", "123-45-67"));
   arrayList.Add (new PhoneList ("Петров", "223-45-68"));
   arrayList.Add (new PhoneList ("Сидоров", "323-45-69"));
   arrayList.Add (new PhoneList ("Кузнецов", "423-45-00"));
   arrayList.Add (new PhoneList ("Рыбаков", "523-55-67"));
   arrayList.Add (new PhoneList ("Климов", "623-66-67"));
   arrayList.Add (new PhoneList ("Трушин", "723-77-67"));
   arrayList.Add (new PhoneList ("Зверев", "823-88-67"));
   arrayList.Add (new PhoneList ("Иванова", "923-99-67"));

   listBox1.Items.Clear();
   listBox1.DataSource = arrayList;

   listBox1.DisplayMember = "Name";
}

private void listBox1_Click (object sender, EventArgs e)
{
   PhoneList obj;
   obj = (PhoneList)listBox1.SelectedItem;
   richTextBox1.Text = obj.Name + "  ";
   richTextBox1.Text += obj.Phone + "\n\n";

   richTextBox1.Text += listBox1.SelectedItem + "\n";  //исп.ToString()
   richTextBox1.Text += listBox1.SelectedIndex + "\n"; //исп.ToString()

   listBox1.ValueMember = "Phone";
   richTextBox1.Text += listBox1.SelectedValue + "\n";
}
```

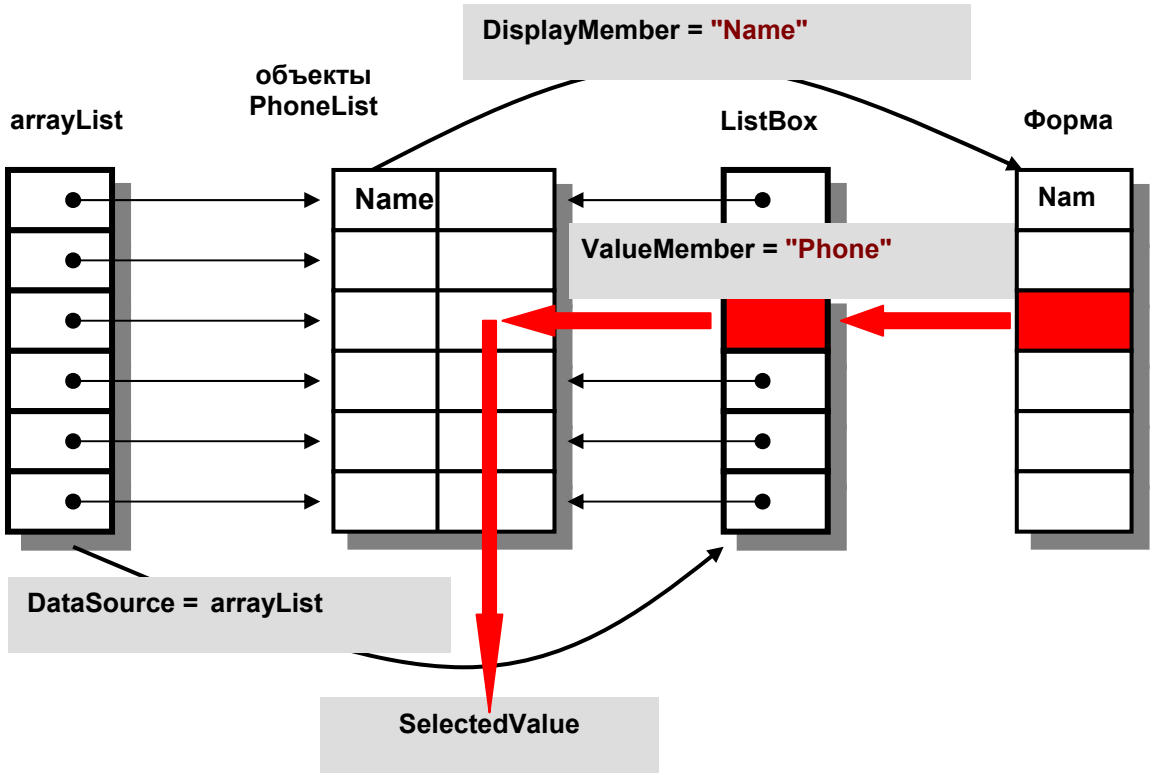


рис.13

Элементы управления Windows Forms не выполняют сортировку, если они привязаны к данным. Чтобы вывести отсортированные данные, используйте источник данных, поддерживающий сортировку данных, и выполните ее с его помощью.

Чтобы получить доступ к определенному элементу

```
comboBox1.Items[i].ToString();
```

Чтобы добавить элементы

```
// Добавить в конец
comboBox1.Items.Add ("Tokyo");

// Вставить с индексом  0:
checkedListBox1.Items.Insert (0, "Copenhagen");

// Присвоить коллекции Items полный массив:
object[] ItemObject = new object[10];

for (int i = 0; i <= 9; i++)
    ItemObject[i] = "Item" + i;

listBox1.Items.AddRange(ItemObject);
```

Чтобы удалить элемент

```
// Удалить с индексом  0:
comboBox1.Items.RemoveAt(0);

// Удалить выделенный элемент:
comboBox1.Items.Remove(comboBox1.SelectedItem);

// Удалить элемент "Tokyo":
comboBox1.Items.Remove("Tokyo");

// Удалить все элементы
listBox1.Items.Clear();
```

Особенности элемента управления ComboBox

Как правило, поле со списком используется при наличии списка возможных вариантов, а список — при необходимости ограничить данные, вводимые в список.

Поле со списком содержит текстовое поле, поэтому варианты, отсутствующие в списке, можно вводить, если это не запрещено свойством DropDownStyle.

Свойство DropDownStyle определяет стиль отображения поля со списком:

- DropDown – в поле ввода можно вводить новые значения. Полный список не отображается, пока пользователь не щелкнет кнопку со стрелкой вниз.
- DropDownList - в поле ввода нельзя вводить новые значения. Используется навигация по первой букве.
- Simple – в поле ввода можно вводить новые значения, окно списка постоянно открыто.

Пример формирования списка на базе его текстового поля:

```
if (comboBox1.FindStringExact ( comboBox1.Text) == -1 )

    comboBox1.Items.Add ( comboBox1.Text );
```

FindStringExact() – полное совпадение со строкой;
FindString() – совпадение подстрок (поиск вхождений);

Особенности ЭУ CheckedListBox

CheckedListBox позволяет:

- либо просмотреть коллекцию помеченных элементов, сохраненную в свойстве CheckedItems,
- либо пройти по списку с помощью метода GetItemChecked, чтобы определить, какие элементы помечены.

Метод GetItemChecked принимает номер элемента в качестве аргумента и возвращает значение true или false.

Свойства SelectedItems и SelectedIndices не определяют помеченные элементы: они определяют, какие элементы выделены.

Вывод помеченных элементов

Пример 1

Пройдите по коллекции CheckedItems помеченных элементов, начав с 0, поскольку нумерация коллекции начинается с нуля. Обратите внимание, что этот метод выдаст номер элемента в списке помеченных элементов, а не в полном списке.

Если первый элемент списка не помечен, но помечен второй, в приведенном ниже коде отобразится текст наподобие следующего: "Помеченный элемент 1 = ЭлементСписка2".

```
if ( checkedListBox1.CheckedItems.Count != 0 )
{
  string s = "";

  for ( int x = 0;   x < checkedListBox1.CheckedItems.Count ;   x++)
  {
       s = s + "\nПомеченный элемент " + (x+1) + " = "
                                          + checkedListBox1.CheckedItems[x];
  }
  MessageBox.Show (s);
}
```

Пример 2

Пройдите по коллекции Items всех элементов списка, начав с 0, поскольку нумерация коллекции начинается с нуля, и вызовите метод GetItemChecked() для каждого элемента. Обратите внимание, что этот метод выдаст номер элемента в полном списке; поэтому если первый элемент списка не помечен, но помечен второй, будет выведен текст наподобие следующего: "Помеченный элемент 2 = ЭлементСписка2".

```
string s =  "Отмеченые элементы:\n" ;

for ( int  i = 0;   i < checkedListBox1.Items.Count ;   i++ )
{
  if ( checkedListBox1.GetItemChecked(i) == true)
       s = s + "\nПомеченный элемент " + (i+1) + " = " + checkedListBox1.Items[i] ;
}
MessageBox.Show (s);
```

4. DomainUpDown

Элемент управления DomainUpDown отображает отдельное строковое значение, выделенное в коллекции Object путем нажатия на кнопки передвижения вверх/вниз по элементам коллекции.

рис.14

Пользователь также может вводить в элемент управления текст, если свойство ReadOnly имеет значение false (вводимая строка должна соответствовать элементу в коллекции, чтобы быть допустимой).

Доступ к элементам коллекции:

Вся коллекция объектов доступна через свойство Items.

Свойство SelectedIndex - получает или задает значение индекса для выделенного элемента.

Свойство SelectedItem - получает или задает текстовое представление выделенного элемента, основываясь на значении индекса элемента, выделенного в коллекции.

Пример:

```
MessageBox.Show ( "SelectedIndex: " + domainUpDown1.SelectedIndex + "\n" +
                  "SelectedItem: "   + domainUpDown1.SelectedItem);
```

Создать коллекцию объектов, отображаемых в элементе управления DomainUpDown, можно:

- в коде программы (можно добавлять и удалять отдельные элементы с помощью методов Items.Add() и Items.Remove())

-- или --

- на этапе конструирования (с помощью Редактора строк коллекции, вызываемого через свойство *Items* ...).

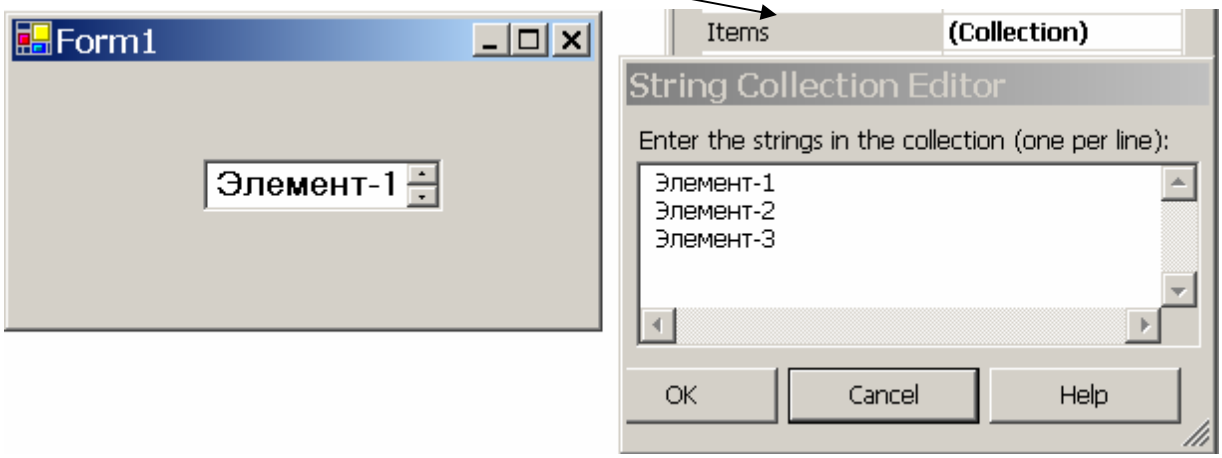


рис.15

// domainUpDown1. В конструкторе. Создаем коллекцию объектов-строк

```
this.domainUpDown1.Items.Add (" Элемент-1");
this.domainUpDown1.Items.Add (" Элемент-2");
this.domainUpDown1.Items.Add (" Элемент-3");
this.domainUpDown1.Name = "domainUpDown1";
this.domainUpDown1.Text = " Элемент-1";

this.domainUpDown1.TabIndex = 0;
this.domainUpDown1.Location = new System.Drawing.Point (96, 48);
```

Другие возможности:

Сортировка коллекции объектов может быть выполнена по алфавиту, если свойству Sorted задать значение true.

Когда Wrap имеет значение true, то список будет закольцованным (то есть при прокрутке последнего или первого объекта в коллекции просмотр списка начнется повторно с первого или последнего объекта соответственно).

Для отображения объекта в элементе управления «вверх/вниз» вызывается метод ToString() этого объекта.

Пример:

```
public class UDC
{
    int a, b;

    public UDC (int A, int B)
    { a=A; b=B;}

    public override string ToString()
    { return " " + a + "+" + b; }
}
```

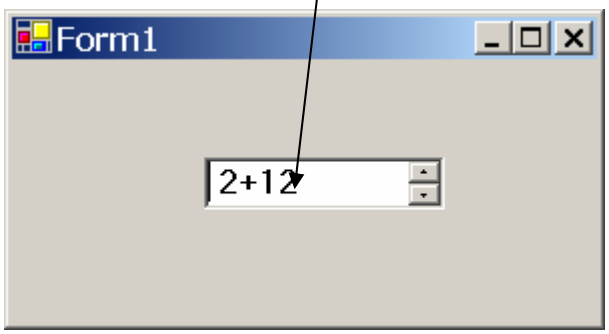


рис.16

```
UDC udc;
DomainUpDown  domainUpDown1 ;
// . . .
private void InitializeComponent()
{
    domainUpDown1 = new DomainUpDown();
    SuspendLayout();
    //
    // domainUpDown1
    //
```

```
udc = new UDC(0, 10);
domainUpDown1.Items.Add (udc);

udc = new UDC(1, 11);
domainUpDown1.Items.Add (udc);

udc = new UDC(2, 12);
domainUpDown1.Items.Add (udc);

udc = new UDC(3, 13);
domainUpDown1.Items.Add (udc);

domainUpDown1.Location = new System.Drawing.Point(96, 48);
domainUpDown1.Name = "domainUpDown1";
domainUpDown1.TabIndex = 0;
domainUpDown1.Text = " Элемент-1";
//
// Form1
// . . .
```

При вызове методов UpButton (отображает предыдущий элемент в коллекции) или DownButton (отображает следующий элемент в коллекции объектов) в коде либо при нажатии на кнопки передвижения вверх или вниз также вызывается защищенный метод UpdateEditText, обновляющий элемент управления новой строкой.

Если защищенное свойство UserEdit имеет значение true, строка соотносится с одним из значений в коллекции до обновления текстового отображения элемента управления.

Пример

В приведенном ниже примере создается и инициализируется элемент управления DomainUpDown. Данный пример позволяет задать значения для некоторых его свойств и создать коллекцию строк, отображаемых в элементе управления «вверх/вниз».

Предполагается, что TextBox, CheckBox и Button уже инициализированы на этой форме. В этом примере также предполагается, что на уровне классов имеется переменная myCounter – 32-разрядное целое число. Можно ввести строку в текстовое поле и добавить ее в коллекцию Items нажатием кнопки.

При помощи флажка можно включать и отключать свойство Sorted, наблюдая различия в коллекции элементов в элементе управления «вверх/вниз».

```
// СОЗДАЕТСЯ  КОНСТРУКТОРОМ:
protected DomainUpDown domainUpDown1;

private void MySub()
 {
   // Создать и проинициализировать DomainUpDown
   domainUpDown1 = new  System.Windows.Forms.DomainUpDown();

   // Добавить DomainUpDown в форму
```

```
    Controls.Add(domainUpDown1);
  }

  private void button1_Click (Object sender, System.EventArgs e)
  {
    // Добавить содержимое TextBox  в коллекцию DomainUpDown
    domainUpDown1.Items.Add ((textBox1.Text.Trim()) + " - " + myCounter);

    myCounter = myCounter + 1;          // Перейти к следующему номеру

    textBox1.Text = "";                 // Очистить TextBox.
  }

  private void checkBox1_Click(Object sender, System.EventArgs e)
  {
    // Если сортировка разрешена, то запретить сортировку, и наоборот
    if (domainUpDown1.Sorted)
          domainUpDown1.Sorted = false;
    else
          domainUpDown1.Sorted = true;
  }

  void domainUpDown1_SelectedItemChanged(Object s, EventArgs e)
  {
    // Вывести значение свойств SelectedIndex and SelectedItem
    MessageBox.Show(
                "SelectedIndex: " + domainUpDown1.SelectedIndex+ "\n" +
                "SelectedItem: "   + domainUpDown1.SelectedItem);
  }
```

5. NumericUpDown

Элемент управления NumericUpDown содержит одно численное значение, которое можно увеличить или уменьшить, нажав кнопку ВВЕРХ или ВНИЗ элемента управления. Пользователь сам может ввести значение, если только свойство ReadOnly имеет значение false.

Чтобы задать допустимый интервал значений элемента управления, присвойте значения свойствам Minimum и Maximum.

Задайте свойству Value начальное значение. В коде это свойство будет возвращать вам выбранное значение.

Задайте свойству Increment - шаг изменения значения, когда пользователь нажмет кнопку ВВЕРХ или кнопку ВНИЗ.

Чтобы значения отображались в шестнадцатеричном формате, присвойте свойству Hexadecimal значение true.

Чтобы в десятичных числах отображался разделитель групп разрядов, присвойте свойству ThousandsSeparator значение true.

Чтобы задать число десятичных разрядов, которые будут отображаться после разделителя целой и дробной частей, присвойте свойству DecimalPlaces значение, равное этому числу десятичных разрядов.

При вызове метода UpButton или DownButton, как в коде, так и вручную, нажатием кнопки ВВЕРХ или кнопки ВНИЗ, новое значение проверяется, и элемент управления обновляется, принимая это значение в соответствующем формате. В частности, если свойство UserEdit имеет зна-

чение true, метод ParseEditText вызывается до проверки и обновления значения. После этого проверяется, находится ли значение в интервале значений свойств Minimum и Maximum, а затем вызывается метод UpdateEditText.

Пример

В приведенном ниже примере создается и инициализируется элемент управления NumericUpDown, задаются некоторые из его общих свойств, а также дается разрешение пользователям изменять некоторые из них во время выполнения. В коде предполагается, что в форме были размещены три элемента управления CheckBox, а также были созданы обработчики для их событий Click. Свойства DecimalPlaces, ThousandsSeparator и Hexadecimal задаются по событию Click каждого флажка.

```
public void InstantiateMyNumericUpDown()
{
  // Create and initialize a NumericUpDown control.
  numericUpDown1 = new NumericUpDown();

  // Dock the control to the top of the form.
  numericUpDown1.Dock = System.Windows.Forms.DockStyle.Top;

  // Set the Minimum, Maximum, and initial Value.
  numericUpDown1.Value = 5;
  numericUpDown1.Maximum = 2500;
  numericUpDown1.Minimum = -100;

  // Add the NumericUpDown to the Form.
  Controls.Add(numericUpDown1);
}


// Check box to toggle decimal places to be displayed.
private void checkBox1_Click(Object sender, EventArgs e)
{
  /* If DecimalPlaces is greater than 0, set them to 0 and round the
     current Value; otherwise, set DecimalPlaces to 2 and change the
     Increment to 0.25. */
  if (numericUpDown1.DecimalPlaces > 0)
  {
    numericUpDown1.DecimalPlaces = 0;
    numericUpDown1.Value = Decimal.Round(numericUpDown1.Value, 0);
  }
  else
  {
    numericUpDown1.DecimalPlaces = 2;
    numericUpDown1.Increment = 0.25M;
  }
}

// Check box to toggle thousands separators to be displayed.
private void checkBox2_Click(Object sender, EventArgs e)
{
```

```
    /* If ThousandsSeparator is true, set it to false;
       otherwise, set it to true. */
    if (numericUpDown1.ThousandsSeparator)
    {
      numericUpDown1.ThousandsSeparator = false;
    }
    else
    {
      numericUpDown1.ThousandsSeparator = true;
    }
}


// Check box to toggle hexadecimal to be displayed.
private void checkBox3_Click(Object sender, EventArgs e)
{
    /* If Hexadecimal is true, set it to false;
       otherwise, set it to true. */
    if (numericUpDown1.Hexadecimal)
    {
      numericUpDown1.Hexadecimal = false;
    }
    else
    {
      numericUpDown1.Hexadecimal = true;
    }
}
```

6. DateTimePicker и MonthCalendar

Элемент управления DateTimePicker позволяет пользователю выбирать дату и время и отображать их в указанном формате.

Календарь MonthCalendar – частный случай ЭУ DateTimePicker. Позволяет выбрать дату и возвратить ее значение.

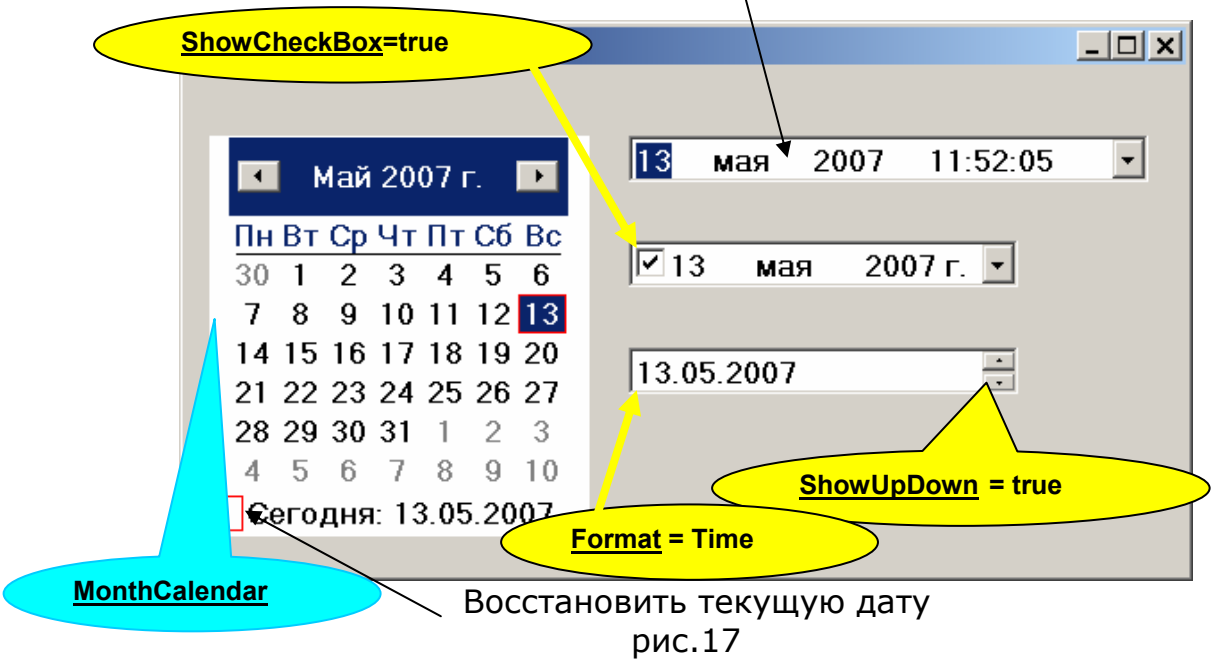


рис.17

Свойство Value получает или задает значение даты/времени, назначаемое элементу управления. Если свойство Value не было изменено ко-

дом или пользователем в окне свойств, ему присваивается значение текущей даты и времени (DateTime.Now).

Описание свойства:
public DateTime Value {get; set;}

Примеры установки текущих значений:

```
dateTimePicker1.Value = DateTime.Now.AddDays(1);
dateTimePicker2.Value = new System.DateTime(2007, 5, 13, 14, 34, 8, 0);
```

AddDays(1) - добавляет указанное число дней к значению этого экземпляра.

Ограничить количество выбираемых значений даты и времени можно с помощью настройки свойств MinDate и MaxDate.

Чтобы изменить вид части элементов управления календарем, следует настроить свойства CalendarForeColor, CalendarFont, CalendarTitleBackColor, CalendarTitleForeColor, CalendarTrailingForeColor и CalendarMonthBackground.

Свойство Format определяется значением из перечисления DateTimePickerFormat элемента управления. По умолчанию свойству Format для даты присваивается значение DateTimePickerFormat.Long.

Другие значения перечисления: Short, Time, Custom.

Если свойство Format имеет значение DateTimePickerFormat.Custom, можно создать собственный стиль формата путем настройки свойства CustomFormat и построения настраиваемой строки формата.

Пример, при указании для свойства CustomFormat значения "ddMMMMyyyy HH:mm:ss", дата будет отображена следующим образом: «13 мая 2007 г. 11:52:05».

Чтобы использовать элемент управления в стиле «вверх и вниз» для настройки значения даты и времени, следует задать для свойства ShowUpDown значение true. Если элемент управления календарем выделен, он не раскроется.

Значения даты/времени можно настроить путем выделения каждого элемента по отдельности и использования кнопок перемещения вверх и вниз для изменения значения.

При необходимости пользовательского форматирования даты и ограничения выделения только одной датой можно использовать элемент управления DateTimePicker вместо MonthCalendar. Если использовать DateTimePicker, необходимость в частых проверках значений даты/времени отпадает.

Примечание. Элемент управления DateTimePicker поддерживает только григорианский календарь.

Пример

В приводимом ниже примере производится создание нового экземпляра элемента управления DateTimePicker и его инициализация. Свойству CustomFormat элемента управления присвоено значение. Кроме того, свойство ShowCheckBox настроено таким образом, чтобы элемент управ-

ления отображал CheckBox, а свойство ShowUpDown настроено таким образом, чтобы элемент управления отображался как элемент управления «вверх и вниз».

```
public void CreateMyDateTimePicker()
{
 // DateTimePicker dateTimePicker1 = new DateTimePicker();

  // Установить минимальную и максимальную даты.
  dateTimePicker1.MinDate = new DateTime(1985, 6, 20);
  dateTimePicker1.MaxDate = DateTime.Today;

  // Установить строку форматирования
  dateTimePicker1.CustomFormat = "ddMMMMyyyy     HH:mm:ss";
  dateTimePicker1.Format =
            System.Windows.Forms.DateTimePickerFormat.Custom;

  // Установить CheckBox и высветить ЭУ up-down control.
  dateTimePicker1.ShowCheckBox = true;
  dateTimePicker1.ShowUpDown = true;
}
```

Пример. Создание календаря на 6 месяцев с номерами недель:
CalendarDimensions.Width = 3;   // см. 1
CalendarDimensions.Height = 2;

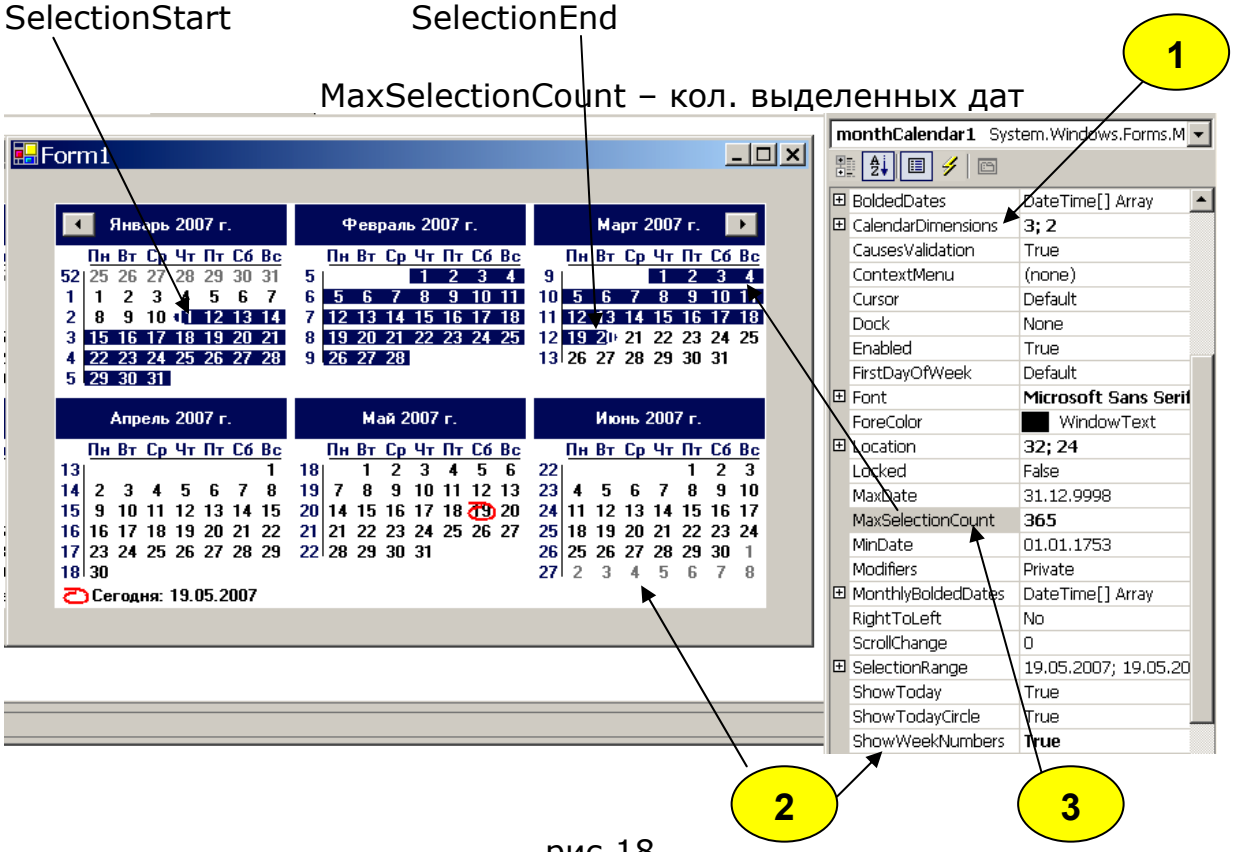


рис.18

Пример. Чтения даты из календаря monthCalendar1:

```
DateTime  dt  = monthCalendar1.SelectionStart;
string   дата =  dt.Day  +  "."  +  dt.Month  +  "."  +  dt.Year;
```

Свойства SelectionStart и SelectionEnd хранят дату в виде объектов структуры DateTime.

Свойство	Содержание
Day	Число
Mouhth	Номер месяца
Year	Год
Millisecond	Миллисекунды
Second	Секунды
Minute	Минуты
Hour	Часы
DayOfWeek	Номер дня в неделе
DayOfYear	Номер дня в году
Ticks	Количество периодов системного таймера
TimeOfDay	Время дня
Today	Текущая дата
UtcNow	Текущая локальная дата в терминах универсального координированного времени (coordinated universal time,

DataGridView - СЕТКА

1. Общие сведения

ЭУ DataGridView позволяет отобразить данные в виде набора строк и столбцов, то есть в виде таблицы.

Чтобы не путать таблицу DataGridView с реляционной таблицей базы данных, первую называют сеткой.

Некоторые классы и их свойства:

[ComVisibleAttribute(true)] [ClassInterfaceAttribute(ClassInterfaceType.AutoDispatch)] public class DataGridView : Control, ISupportInitialize
public DataGridViewColumnCollection Columns { get; }
public DataGridViewRowCollection Rows { get; }
public class DataGridViewColumnCollection : BaseCollection, IList, ICollection, IEnumerable
public class DataGridViewRowCollection : IList, ICollection, IEnumerable
DataGridViewRowCollection.Add () – добавить новую строку в коллекцию.
DataGridViewRow – класс, представляет одну строку в DataGridView.
DataGridViewColumnCollection.Add (DataGridViewColumn) – добавляет указанный столбец в коллекцию
DataGridViewColumn – класс, представляет один столбец в DataGridView.
DataGridViewColumnCollection.Add (String, String) - добавляет столбец с указанным именем и заголовком в коллекцию.
public class DataGridViewTextBoxColumn : DataGridViewColumn
DataGridViewTextBoxColumn – столбец, представляет коллекцию ячеек типа DataGridViewTextBoxCell.
public class DataGridViewTextBoxCell : DataGridViewCell
DataGridViewTextBoxCell – отображает редактируемый текст ячейки в ЭУ DataGridView.

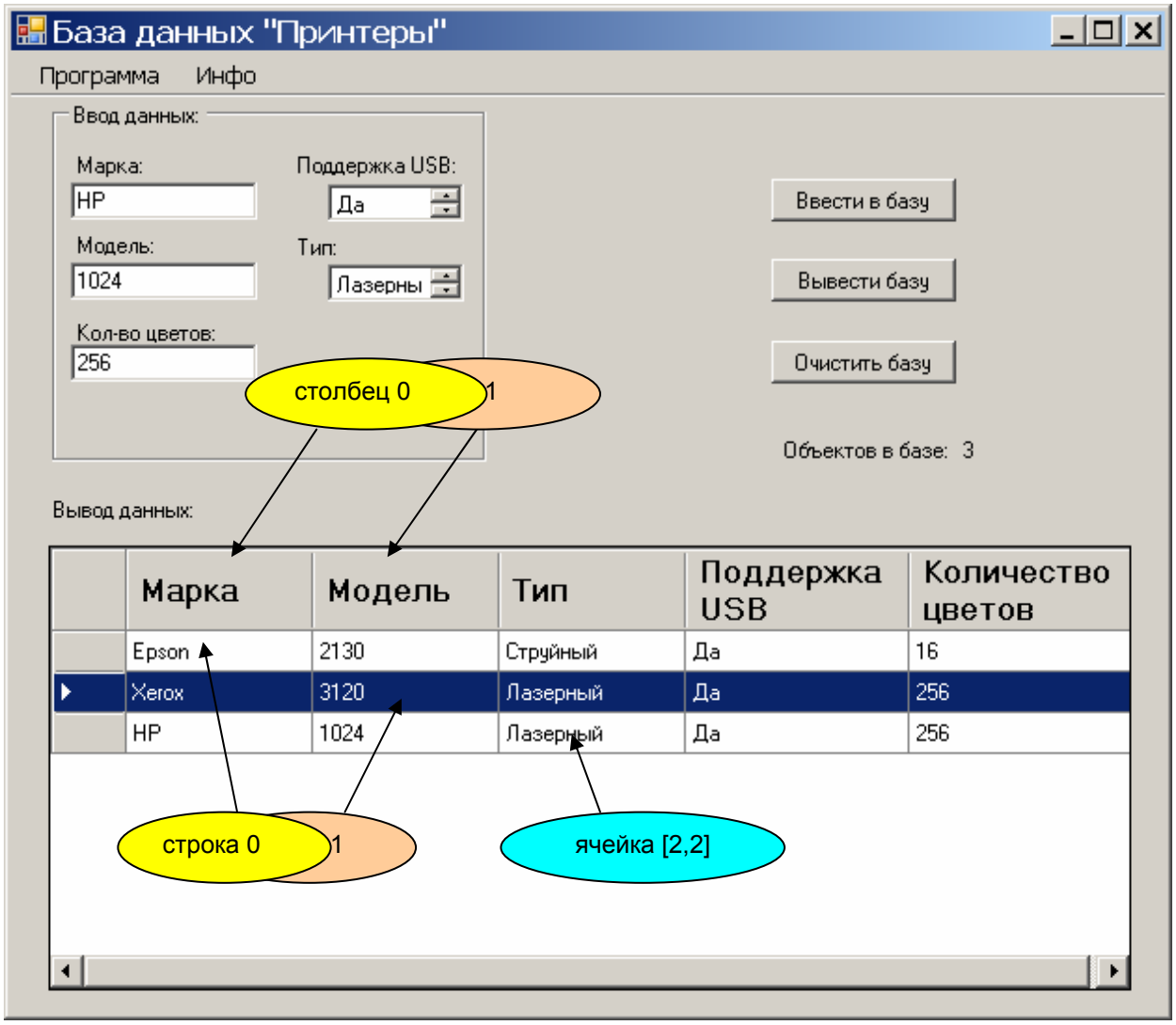


рис.1

Некоторые свойства класса DataGridView:

Columns	Возвращает коллекцию всех столбцов, содержащихся в ЭУ.
Rows	Возвращает коллекцию всех строк, содержащихся в ЭУ.
CurrentCellAddress	Возвращает индекс строки и столбца текущей ячейки.
CurrentRow	Возвращает строку, содержащую текущую активную ячейку.
CurrentCell	Возвращает или устанавливает текущую активную ячейку.
Item	Overloaded. Gets or sets the cell located at the intersection of the specified row and column. В языке C# это свойство является индексатором класса DataGrid.

Разрешить ДОБАВЛЕНИЕ, УДАЛЕНИЕ, и изменение РАЗМЕРА строк.
Разрешить изменение ПОРЯДКА и РАЗМЕРА столбцов.

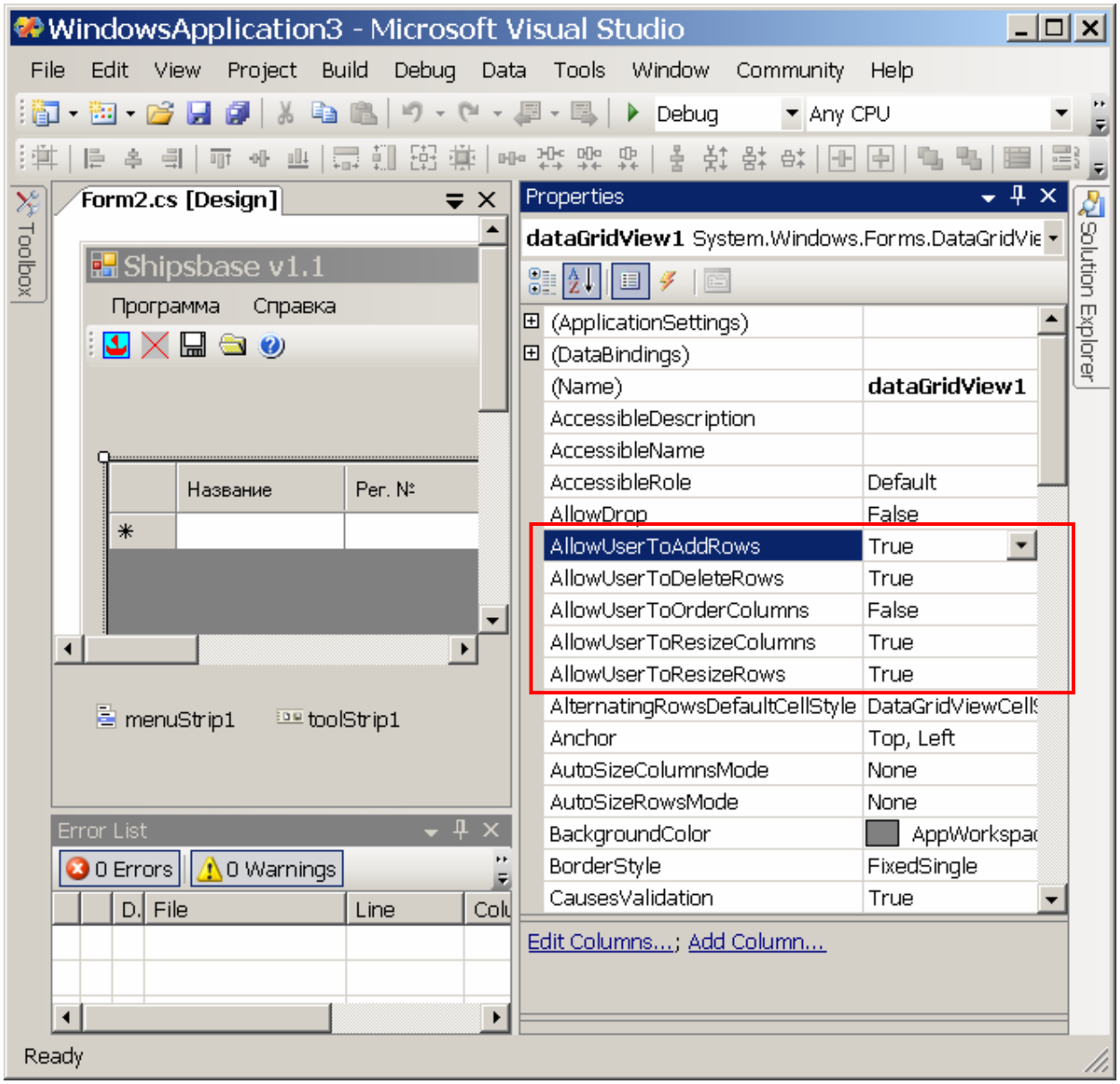


рис.2

2. Создание таблицы с помощью дизайнера VS 2005

Добавление столбцов

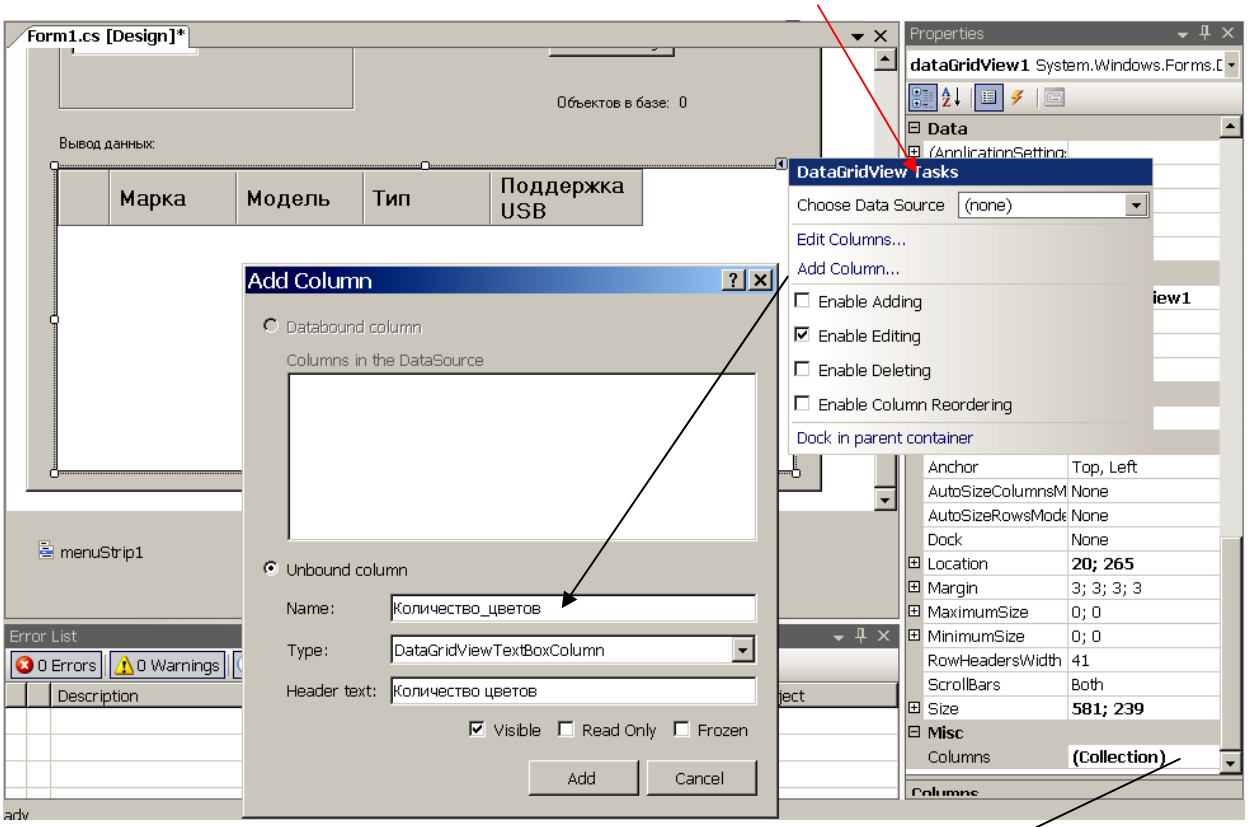


рис.3

Добавление через свойство Columns. Изменение свойств столбцов.

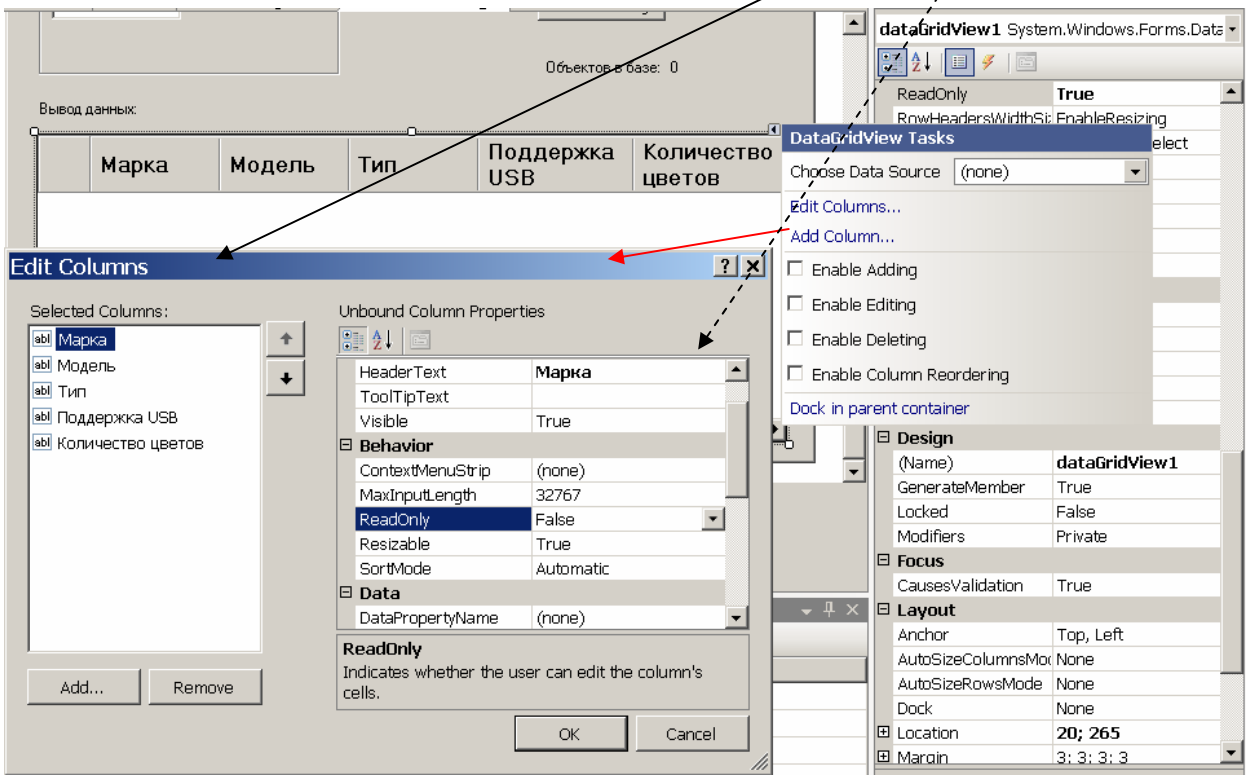


рис.4

Типы столбцов:

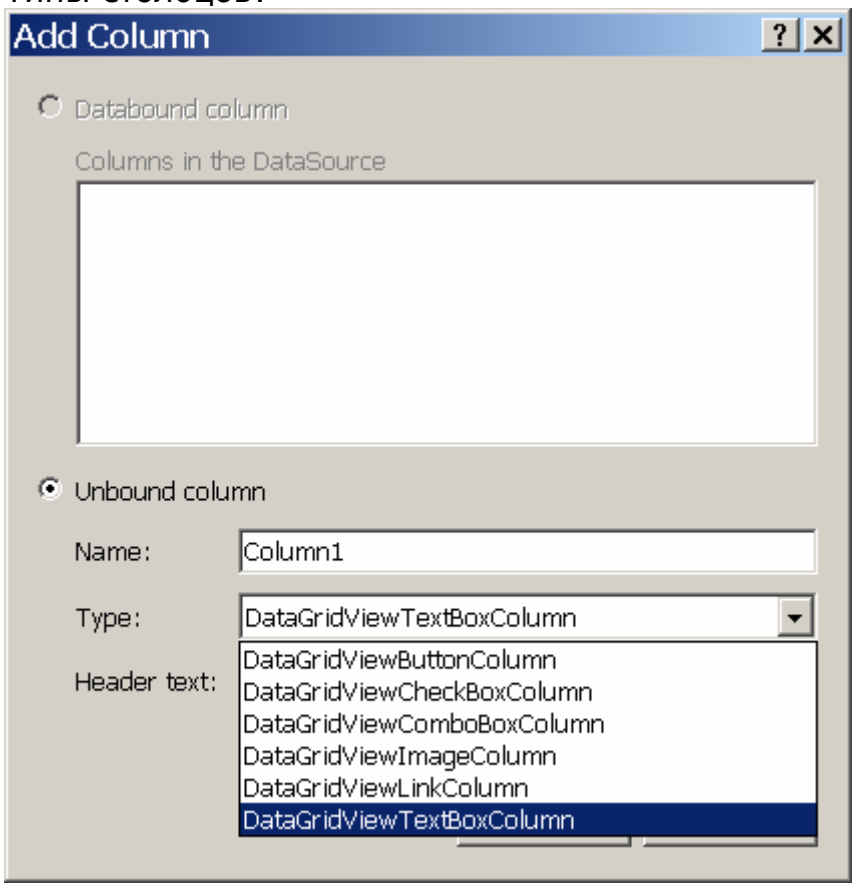


рис.5

Таблица создается следующей строкой:

```
DataGridView  dataGridView1 =  new  DataGridView();
```

Добавление строк

Через массивы row:

```
 string[] row0 = { "Xerox", "3120",  "Лазерный",   "Да",   "256" };
 string[] row1 = { "HP",      "2110",  "Лазерный",   "Да",   "1024" };
 string[] row2 = { "Epson", "1423",  "Струйный",   "Нет",  "1" };

 dataGridView1.Rows.Add(row0);

 DataGridViewRowCollection  rows  =  dataGridView1.Rows;
 rows.Add(row1);
 rows.Add(row2);

 dataGridView1.RowCount = 2;     // добавить 2 пустые строки

 dataGridView1.Rows.Add();    // добавить пустую строку
 dataGridView1.Rows.Add(3);  // добавить 3 пустые строки
```

Через перечень строк:

```
dataGridView1.Rows.Add ("HP", "2110", "Лазерный", "Да", "16");
```

Удаление строк

Строка 0:

```
DataGridViewRow dstr = dataGridView1.Rows[0];
dataGridView1.Rows.Remove(dstr);
```

Текущая строка:

```
DataGridViewRow dstr = dataGridView1.CurrentRow;
dataGridView1.Rows.Remove(dstr);
```

Все строки:

```
dataGridView1.Rows.Clear();
```

Обработчики событий

- Обработчик события «щелчок на ячейке»:

```
dataGridView1_CellContentClick (object sender,
                                DataGridViewCellEventArgs e)
```

Свойства: e.ColumnIndex - индекс столбца с акт. ячейкой, Y.
e.RowIndex - индекс строки с акт. ячейкой, X.

- Обработчик события «изменение текущей активной ячейки»:

```
dataGridView1_CurrentCellChanged (object sender, EventArgs e)
```

Работа с ячейками

```
DataGridViewCell  actCell = dataGridView1.CurrentCell;
```

Свойство Value – получить или изменить значение текущей активной ячейки (get и set).

actCell.Value – содержимое акт. ячейки.

```
int   y =   actCell.ColumnIndex;                    // индекс столбца
int   x =   actCell.RowIndex;                       // индекс строки
```

Point

```
int   y =   dataGridView1.CurrentCellAddress.Y;   // индекс столбца
int   x =   dataGridView1.CurrentCellAddress.X;   // индекс строки

 str = dataGridView1[x, y].Value;                   // на чтение

 dataGridView1[x, y].Value = "Лазерный";            // на запись


 str =dataGridView1.Rows[3].Cells[1].Value;
```

В обработчике CellContentClick:

```
str = dataGridView1 [e.ColumnIndex,  e.RowIndex].Value;
```

3. Создание таблицы вручную

Структура таблицы DataGridView

Создание таблицы (сетки):

```
DataGridView    dataGridView1          =  new  DataGridView();
```

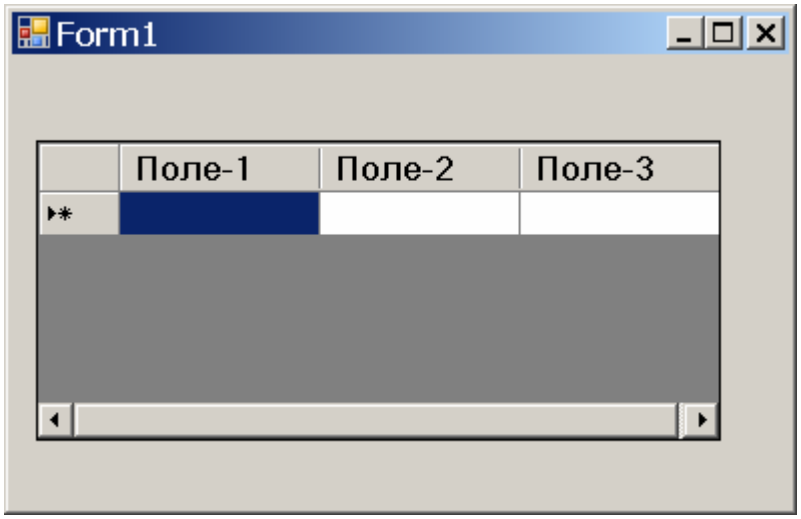


рис.6

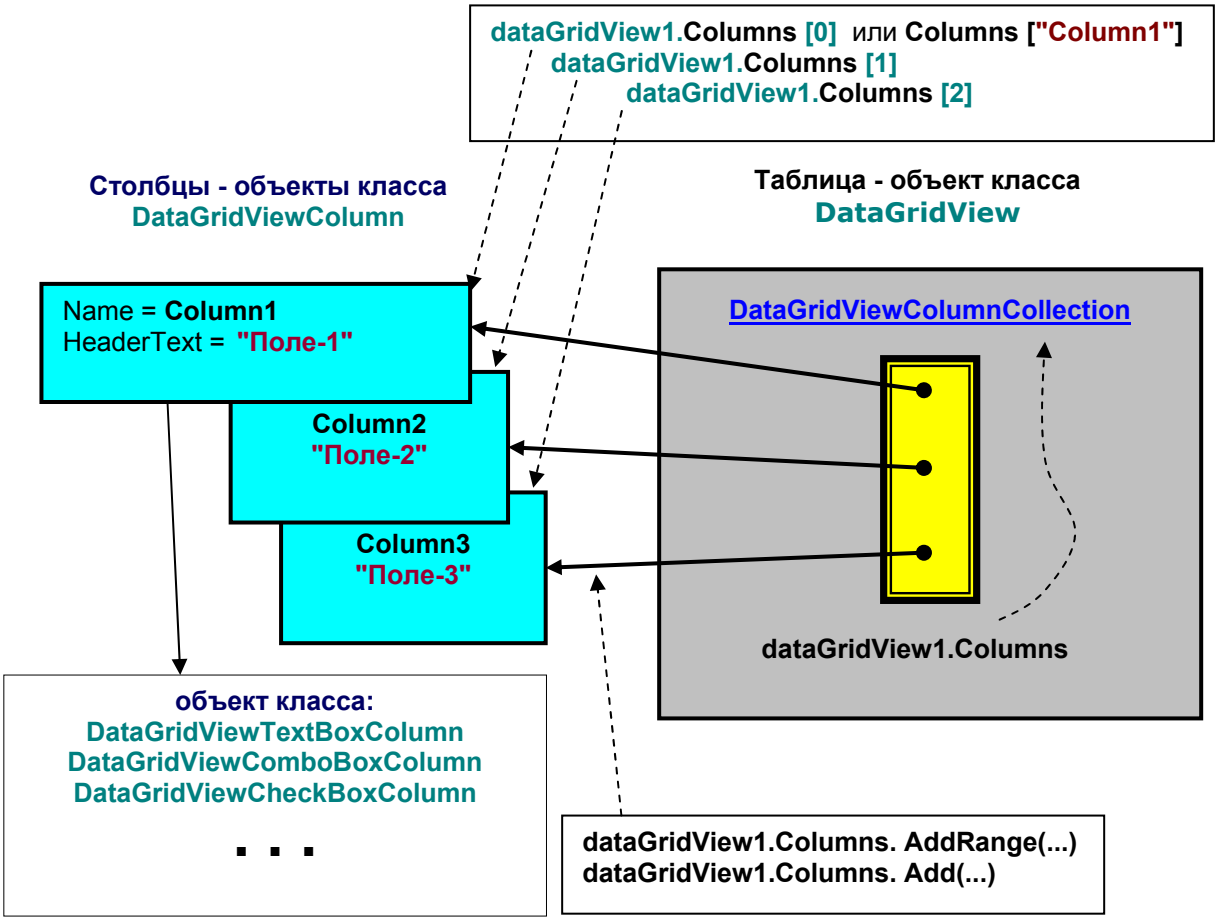


рис.7

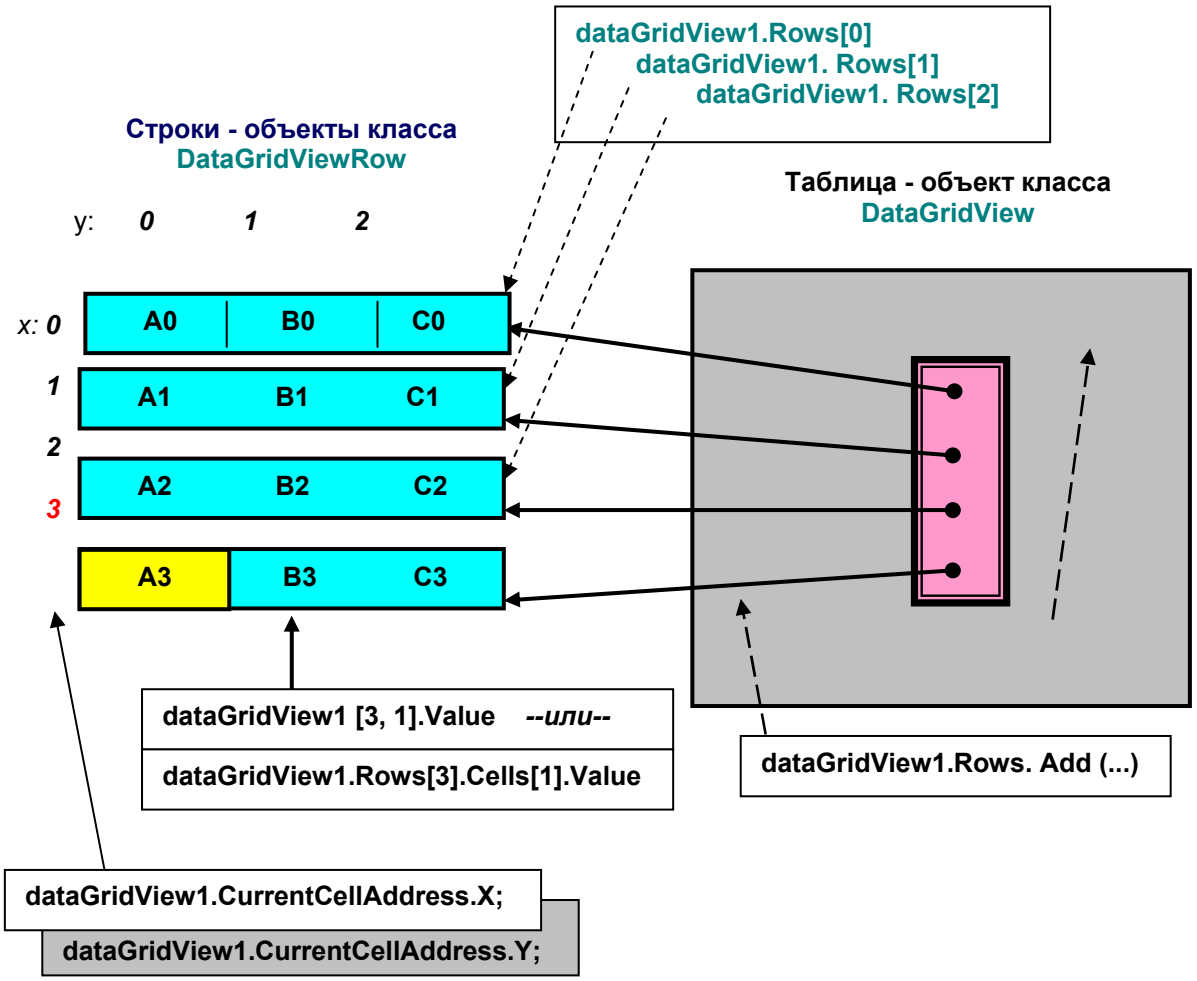


рис.8

Добавление столбцов

Предварительное создание столбцов как самостоятельных объектов типа DataGridViewColumn.

Столбцы предварительно не создаются.

Способ 1.

```
DataGridViewTextBoxColumn марка = new DataGridViewTextBoxColumn();

DataGridViewTextBoxColumn модель  = new DataGridViewTextBoxColumn();
                    . . . . . .
```

Добавление столбцов в dataGridView1:

```
dataGridView1.Columns.Add(марка);
марка.HeaderText = "Марка";

dataGridView1.Columns.Add(модель);
модель.HeaderText = " Модель";
```

dataGridView1.Columns.AddRange((new DataGridViewColumn[] { марка, модель, тип, поддержкаUSB, количество_цветов });

Вставка столбца column2 после второго столбца, то есть с индексом 2.

```
dataGridView1.Columns.Insert (2, column2)
```

Способ 2.

Создание и добавление DataGridViewTextBoxColumn-столбцов (имя столбца, заголовок столбца):

```
dataGridView1.Columns.Add("марка", "Марка");

dataGridView1.Columns.Add("модель", "Модель");
```

Примечание. Имя ссылки на объект-столбец неизвестно.

Создание и добавление DataGridViewTextBoxColumn-столбцов без имени:

```
dataGridView1.Columns.Add ("", "Марка");   // заголовок

dataGridView1.Columns.Add ("", "Модель");
```

Примечание. Ссылка на объект-столбец неизвестна.

Создание и добавление DataGridViewTextBoxColumn-столбцов без имени и заголовка:

```
dataGridView1.ColumnCount = 4;                // вставка 4-х столбцов
```

Установление свойств добавленных столбцов.

Доступ к столбцу по его индексу:

```
dataGridView1.Columns[0].Name = "New0";            // имя столбца
dataGridView1.Columns[1].Name = "New1";            // имя столбца

dataGridView1.Columns[0].HeaderText = "HeadNew0";  // заголовок
dataGridView1.Columns[1].HeaderText = "HeadNew1";  // заголовок
```

Примечание. Если заголовок не задан, то в качестве заголовка столбца используется его имя.

Другие свойства столбца

Доступ к столбцу по его имени:

```
dataGridView1.Columns ["имя"].DisplayIndex = 3;   //Место столбца
dataGridView1. Columns.Clear();     // Удалить все столбцы и строки
```

Наклонный стандартный шрифт:

```
dataGridView1.Columns[1].DefaultCellStyle.Font = new
                    Font(DataGridView.DefaultFont, FontStyle.Italic);
```

Значения в ячейках в центре по середине:

```
dataGridView1.Columns[1].DefaultCellStyle.Aligment =
                        DataGridViewContent Aligment.MiddleCenter;

марка.ReadOnly = true;  // Разрешить изменение ячейки
```

Добавление строк

4-е перегруженных метода:

Name	Description
DataGridViewRowCollection.Add ()	Adds a new row to the collection.
DataGridViewRowCollection.Add (DataGridViewRow)	Adds the specified DataGridViewRow to the collection.
DataGridViewRowCollection.Add (Int32)	Adds the specified number of new rows to the collection.
DataGridViewRowCollection.Add (Object[])	Adds a new row to the collection, and populates the cells with the specified objects.

Способы добавления строк были рассмотрены выше!

Пример.

База принтеров. Программа только отображает строки. Ячейки доступны только на чтение.

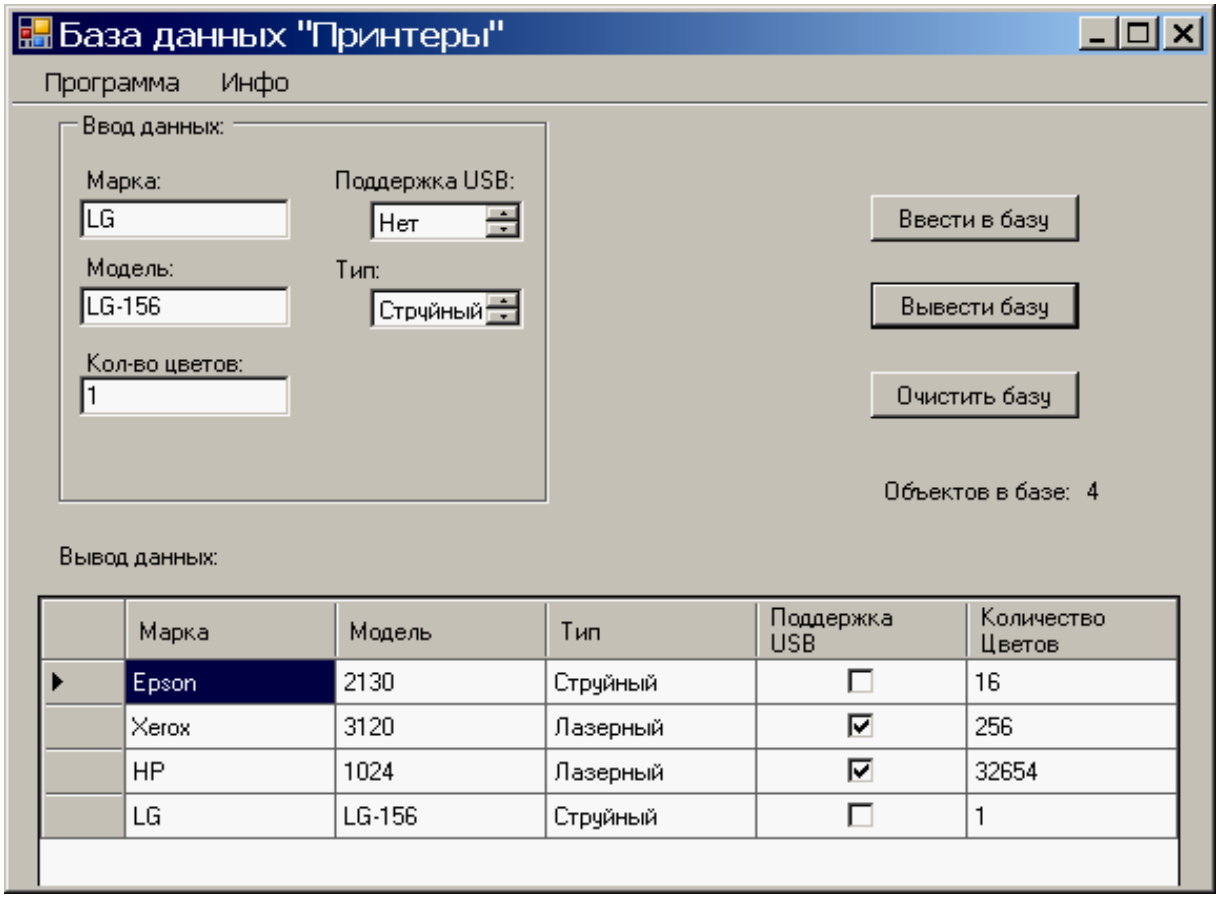


рис.9

-- Файл Printer.cs –

```
using System;
using System.Collections.Generic;
using System.Text;
using System.Collections;

namespace Printer_Virtual
{
```

```
class Printer
{
   protected string[] str = new string[5];   // информация о принтере
   private     string    mark, model;

   public Printer(string mark, string model)
   {
      this.mark = mark;
      this.model = model;
   }

   public virtual string[] Show()
   {
      str[0] = mark;
      str[1] = model;
      return str;
   }
}


class LaserPrinter : Printer
{
   string type = "Лазерный", USBsupport;
   int      SupportedColors;

   public LaserPrinter(string mark, string model,
                              string USBsupport, int SupportedColors)
      : base(mark, model)
   {
      this.USBsupport = USBsupport;
      this.SupportedColors = SupportedColors;
   }

   public override string[] Show()
   {
      base.Show();
      str[2] = type;
      str[3] = USBsupport;
      str[4] = SupportedColors.ToString();
      return str;
   }
}


class InkJetPrinter : Printer
{
   string type = "Струйный", USBsupport;
   int SupportedColors;

   public InkJetPrinter(string mark, string model,
                               string USBsupport, int SupportedColors)
      : base(mark, model)
   {
      this.USBsupport = USBsupport;
```

```
        this.SupportedColors = SupportedColors;
    }

    public override string[] Show()
    {
        base.Show();
        str[2] = type;
        str[3] = USBsupport;
        str[4] = SupportedColors.ToString();
        return str;
    }
  }
}
```

--Файл Form1.cs—

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Text;
using System.Windows.Forms;
using System.Collections;

namespace Printer_Virtual
{
  public partial class Form1 : Form
  {
    ArrayList obj = new ArrayList();              // Массив ссылок на объекты

    public Form1()
    {  InitializeComponent();  }

    private void button1_Click(object sender, EventArgs e)    // записать в  obj
    {
      Printer printer;
      try
      {
        string ucb;

        if (domainUpDownUSB.Text == "Да")

          ucb = "true";
        else
          ucb = "false";

        if (domainUpDownType.Text == "Струйный")

          printer = new InkJetPrinter(textBoxMark.Text, textBoxModel.Text,
                                      ucb, int.Parse(textBoxColors.Text));
        else
          printer = new LaserPrinter(textBoxMark.Text, textBoxModel.Text,
                                     ucb, int.Parse(textBoxColors.Text));
      }
```

```
        catch
        {
            MessageBox.Show("Заполнены не все поля " +
                                "или введен неверный тип данных!");
            return;
        }

        obj.Add(printer);
        label7.Text = obj.Count.ToString();
    }


    private void button2_Click(object sender, EventArgs e)    // в таблицу
    {
        dataGridView1.Rows.Clear();

        foreach (object prn in obj)
        {
            dataGridView1.Rows.Add (  ((Printer)prn).Show()  );
        }
    }


    private void выходToolStripMenuItem1_Click(object sender, EventArgs e)
    {
        Close();
    }


    private void button3_Click(object sender, EventArgs e)      // очистка базы
    {
        obj.Clear();
        dataGridView1.Rows.Clear();
        label7.Text = obj.Count.ToString();
    }


    private void очиститьПоляToolStripMenuItem_Click(object sender,
                                                     EventArgs e)
    {
        textBoxMark.Clear();
        textBoxModel.Clear();
        textBoxColors.Clear();
    }


private void оПрограммеToolStripMenuItem_Click(object sender, EventArgs e)
{
    MessageBox.Show("Программа 'База данных принтеров'.\n" +
    "Авторы: Горелов, Мошаров.");
}
```

```
    private void помощьToolStripMenuItem1_Click(object sender, EventArgs e)
    {
      MessageBox.Show("Программа 'База данных принтеров'.\n" +
        "Ввод данных производится в специально отведенные поля.\n" +
        "Типы данных ввода:\n   Марка - строка\n   Модель - строка\n" +
        "   Количество цветов - число.\n" +
        "После ввода информации о принтере в БД, выводите ее в таблицу.");
      }
    }
  }
```

-- Файл Form1.Designer.cs --

```
  namespace Printer_Virtual
  {
    partial class Form1
    {
      private System.ComponentModel.IContainer components = null;

      protected override void Dispose(bool disposing)
      {
        if (disposing && (components != null))
        {
          components.Dispose();
        }
        base.Dispose(disposing);
      }

      private void InitializeComponent()
      {
        this.textBoxMark = new System.Windows.Forms.TextBox();
        this.textBoxModel = new System.Windows.Forms.TextBox();
        this.textBoxColors = new System.Windows.Forms.TextBox();
        this.button1 = new System.Windows.Forms.Button();
        this.domainUpDownType = new System.Windows.Forms.DomainUpDown();
        this.button2 = new System.Windows.Forms.Button();
        this.domainUpDownUSB = new System.Windows.Forms.DomainUpDown();
        this.button3 = new System.Windows.Forms.Button();
        this.label1 = new System.Windows.Forms.Label();
        this.label2 = new System.Windows.Forms.Label();
        this.label3 = new System.Windows.Forms.Label();
        this.label4 = new System.Windows.Forms.Label();
        this.label5 = new System.Windows.Forms.Label();
        this.groupBox1 = new System.Windows.Forms.GroupBox();
        this.label6 = new System.Windows.Forms.Label();
        this.label7 = new System.Windows.Forms.Label();
        this.label8 = new System.Windows.Forms.Label();
        this.menuStrip1 = new System.Windows.Forms.MenuStrip();
        this.выходToolStripMenuItem = new Sys-
tem.Windows.Forms.ToolStripMenuItem();
        this.очиститьПоляToolStripMenuItem = new Sys-
tem.Windows.Forms.ToolStripMenuItem();
        this.выходToolStripMenuItem1 = new Sys-
tem.Windows.Forms.ToolStripMenuItem();
```

```
            this.помощьToolStripMenuItem = new System.Windows.Forms.ToolStripMenuItem();
            this.помощьToolStripMenuItem1 = new System.Windows.Forms.ToolStripMenuItem();
            this.оПрограммеToolStripMenuItem = new System.Windows.Forms.ToolStripMenuItem();
            this.dataGridView1 = new System.Windows.Forms.DataGridView();
            this.Марка = new System.Windows.Forms.DataGridViewTextBoxColumn();
            this.Модель = new System.Windows.Forms.DataGridViewTextBoxColumn();
            this.Тип = new System.Windows.Forms.DataGridViewTextBoxColumn();
            this.ПоддержкаUSB = new System.Windows.Forms.DataGridViewCheckBoxColumn();
            this.Количество_цветов = new System.Windows.Forms.DataGridViewTextBoxColumn();
            this.groupBox1.SuspendLayout();
            this.menuStrip1.SuspendLayout();
            ((System.ComponentModel.ISupportInitialize)(this.dataGridView1)).BeginInit();
            this.SuspendLayout();
            //
            // textBoxMark
            //
            this.textBoxMark.Location = new System.Drawing.Point(10, 43);
            this.textBoxMark.Name = "textBoxMark";
            this.textBoxMark.Size = new System.Drawing.Size(100, 20);
            this.textBoxMark.TabIndex = 1;
            //
            // textBoxModel
            //
            this.textBoxModel.Location = new System.Drawing.Point(10, 86);
            this.textBoxModel.Name = "textBoxModel";
            this.textBoxModel.Size = new System.Drawing.Size(100, 20);
            this.textBoxModel.TabIndex = 2;
            //
            // textBoxColors
            //
            this.textBoxColors.Location = new System.Drawing.Point(10, 130);
            this.textBoxColors.Name = "textBoxColors";
            this.textBoxColors.Size = new System.Drawing.Size(100, 20);
            this.textBoxColors.TabIndex = 5;
            //
            // button1
            //
            this.button1.Location = new System.Drawing.Point(408, 68);
            this.button1.Name = "button1";
            this.button1.Size = new System.Drawing.Size(99, 23);
            this.button1.TabIndex = 6;
            this.button1.Text = "Ввести в базу";
            this.button1.UseVisualStyleBackColor = true;
            this.button1.Click += new System.EventHandler(this.button1_Click);
            //
            // domainUpDownType
            //
            this.domainUpDownType.BackColor = System.Drawing.SystemColors.ActiveCaptionText;
```

```
            this.domainUpDownType.Items.Add("Струйный");
            this.domainUpDownType.Items.Add("Лазерный");
            this.domainUpDownType.Location = new System.Drawing.Point(148, 87);
            this.domainUpDownType.Name = "domainUpDownType";
            this.domainUpDownType.ReadOnly = true;
            this.domainUpDownType.Size = new System.Drawing.Size(73, 20);
            this.domainUpDownType.TabIndex = 8;
            this.domainUpDownType.Text = "Струйный";
            //
            // button2
            //
            this.button2.Location = new System.Drawing.Point(408, 111);
            this.button2.Name = "button2";
            this.button2.Size = new System.Drawing.Size(99, 23);
            this.button2.TabIndex = 9;
            this.button2.Text = "Вывести базу";
            this.button2.UseVisualStyleBackColor = true;
            this.button2.Click += new System.EventHandler(this.button2_Click);
            //
            // domainUpDownUSB
            //
            this.domainUpDownUSB.BackColor = Sys-
tem.Drawing.SystemColors.ActiveCaptionText;
            this.domainUpDownUSB.Items.Add("Да");
            this.domainUpDownUSB.Items.Add("Нет");
            this.domainUpDownUSB.Location = new System.Drawing.Point(148, 44);
            this.domainUpDownUSB.Name = "domainUpDownUSB";
            this.domainUpDownUSB.ReadOnly = true;
            this.domainUpDownUSB.Size = new System.Drawing.Size(73, 20);
            this.domainUpDownUSB.TabIndex = 10;
            this.domainUpDownUSB.Text = "Да";
            //
            // button3
            //
            this.button3.Location = new System.Drawing.Point(408, 155);
            this.button3.Name = "button3";
            this.button3.Size = new System.Drawing.Size(99, 23);
            this.button3.TabIndex = 11;
            this.button3.Text = "Очистить базу";
            this.button3.UseVisualStyleBackColor = true;
            this.button3.Click += new System.EventHandler(this.button3_Click);
            //
            // label1
            //
            this.label1.AutoSize = true;
            this.label1.Location = new System.Drawing.Point(10, 27);
            this.label1.Name = "label1";
            this.label1.Size = new System.Drawing.Size(43, 13);
            this.label1.TabIndex = 12;
            this.label1.Text = "Марка:";
            //
            // label2
            //
            this.label2.AutoSize = true;
```

```
this.label2.Location = new System.Drawing.Point(10, 70);
this.label2.Name = "label2";
this.label2.Size = new System.Drawing.Size(49, 13);
this.label2.TabIndex = 13;
this.label2.Text = "Модель:";
//
// label3
//
this.label3.AutoSize = true;
this.label3.Location = new System.Drawing.Point(10, 117);
this.label3.Name = "label3";
this.label3.Size = new System.Drawing.Size(82, 13);
this.label3.TabIndex = 14;
this.label3.Text = "Кол-во цветов:";
//
// label4
//
this.label4.AutoSize = true;
this.label4.Location = new System.Drawing.Point(128, 27);
this.label4.Name = "label4";
this.label4.Size = new System.Drawing.Size(93, 13);
this.label4.TabIndex = 15;
this.label4.Text = "Поддержка USB:";
//
// label5
//
this.label5.AutoSize = true;
this.label5.Location = new System.Drawing.Point(128, 71);
this.label5.Name = "label5";
this.label5.Size = new System.Drawing.Size(29, 13);
this.label5.TabIndex = 16;
this.label5.Text = "Тип:";
//
// groupBox1
//
this.groupBox1.Controls.Add(this.label5);
this.groupBox1.Controls.Add(this.label4);
this.groupBox1.Controls.Add(this.label3);
this.groupBox1.Controls.Add(this.label2);
this.groupBox1.Controls.Add(this.label1);
this.groupBox1.Controls.Add(this.domainUpDownUSB);
this.groupBox1.Controls.Add(this.domainUpDownType);
this.groupBox1.Controls.Add(this.textBoxColors);
this.groupBox1.Controls.Add(this.textBoxModel);
this.groupBox1.Controls.Add(this.textBoxMark);
this.groupBox1.Location = new System.Drawing.Point(22, 27);
this.groupBox1.Name = "groupBox1";
this.groupBox1.Size = new System.Drawing.Size(233, 193);
this.groupBox1.TabIndex = 17;
this.groupBox1.TabStop = false;
this.groupBox1.Text = "Ввод данных:";
//
// label6
//
```

```
            this.label6.AutoSize = true;
            this.label6.Location = new System.Drawing.Point(411, 207);
            this.label6.Name = "label6";
            this.label6.Size = new System.Drawing.Size(96, 13);
            this.label6.TabIndex = 18;
            this.label6.Text = "Объектов в базе:";
            //
            // label7
            //
            this.label7.AutoSize = true;
            this.label7.Location = new System.Drawing.Point(507, 207);
            this.label7.Name = "label7";
            this.label7.Size = new System.Drawing.Size(13, 13);
            this.label7.TabIndex = 19;
            this.label7.Text = "0";
            //
            // label8
            //
            this.label8.AutoSize = true;
            this.label8.Location = new System.Drawing.Point(19, 239);
            this.label8.Name = "label8";
            this.label8.Size = new System.Drawing.Size(83, 13);
            this.label8.TabIndex = 20;
            this.label8.Text = "Вывод данных:";
            //
            // menuStrip1
            //
            this.menuStrip1.Items.AddRange(new System.Windows.Forms.ToolStripItem[] {
                this.выходToolStripMenuItem,
                this.помощьToolStripMenuItem});
            this.menuStrip1.Location = new System.Drawing.Point(0, 0);
            this.menuStrip1.Name = "menuStrip1";
            this.menuStrip1.RenderMode                    =                    Sys-
tem.Windows.Forms.ToolStripRenderMode.System;
            this.menuStrip1.Size = new System.Drawing.Size(568, 24);
            this.menuStrip1.TabIndex = 21;
            this.menuStrip1.Text = "menuStrip1";
            //
            // выходToolStripMenuItem
            //
            this.выходToolStripMenuItem.DropDownItems.AddRange(new
                                                System.Windows.Forms.ToolStripItem[] {
                        this.очиститьПоляToolStripMenuItem,
                        this.выходToolStripMenuItem1});
            this.выходToolStripMenuItem.Name = "выходToolStripMenuItem";
            this.выходToolStripMenuItem.Size = new System.Drawing.Size(84, 20);
            this.выходToolStripMenuItem.Text = "Программа";
            //
            // очиститьПоляToolStripMenuItem
            //
            this.очиститьПоляToolStripMenuItem.Name      =      "очиститьПоляToolStrip-
MenuItem";
            this.очиститьПоляToolStripMenuItem.Size = new System.Drawing.Size(176, 22);
            this.очиститьПоляToolStripMenuItem.Text = "Очистить поля";
```

```
            this.очиститьПоляToolStripMenuItem.ToolTipText = "Очистить поля ввода";
            this.очиститьПоляToolStripMenuItem.Click += new
                                                Sys-
tem.EventHandler(this.очиститьПоляToolStripMenuItem_Click);
            //
            // выходToolStripMenuItem1
            //
            this.выходToolStripMenuItem1.Name = "выходToolStripMenuItem1";
            this.выходToolStripMenuItem1.Size = new System.Drawing.Size(176, 22);
            this.выходToolStripMenuItem1.Text = "Выход";
            this.выходToolStripMenuItem1.Click += new
                                                Sys-
tem.EventHandler(this.выходToolStripMenuItem1_Click);
            //
            // помощьToolStripMenuItem
            //
            this.помощьToolStripMenuItem.DropDownItems.AddRange(new
                                                System.Windows.Forms.ToolStripItem[] {
                    this.помощьToolStripMenuItem1,
                    this.оПрограммеToolStripMenuItem});
            this.помощьToolStripMenuItem.Name = "помощьToolStripMenuItem";
            this.помощьToolStripMenuItem.Size = new System.Drawing.Size(52, 20);
            this.помощьToolStripMenuItem.Text = "Инфо";
            //
            // помощьToolStripMenuItem1
            //
            this.помощьToolStripMenuItem1.Name = "помощьToolStripMenuItem1";
            this.помощьToolStripMenuItem1.Size = new System.Drawing.Size(165, 22);
            this.помощьToolStripMenuItem1.Text = "Помощь";
            this.помощьToolStripMenuItem1.Click += new
                                                Sys-
tem.EventHandler(this.помощьToolStripMenuItem1_Click);
            //
            // оПрограммеToolStripMenuItem
            //
            this.оПрограммеToolStripMenuItem.Name = "оПрограммеToolStripMenuItem";
            this.оПрограммеToolStripMenuItem.Size = new System.Drawing.Size(165, 22);
            this.оПрограммеToolStripMenuItem.Text = "О программе";
            this.оПрограммеToolStripMenuItem.Click += new
                                                Sys-
tem.EventHandler(this.оПрограммеToolStripMenuItem_Click);
            //
            // dataGridView1
            //
            this.dataGridView1.AllowUserToAddRows = false;
            this.dataGridView1.AllowUserToDeleteRows = false;
            this.dataGridView1.AllowUserToResizeColumns = false;
            this.dataGridView1.AllowUserToResizeRows = false;
            this.dataGridView1.BackgroundColor = System.Drawing.Color.White;
            this.dataGridView1.ColumnHeadersHeightSizeMode =
                                                DataGridViewColumnHeadersHeight-
SizeMode.AutoSize;
            this.dataGridView1.Columns.AddRange(new DataGridViewColumn[] {
                    this.Марка,
```

```
                this.Модель,
                this.Тип,
                this.ПоддержкаUSB,
                this.Количество_цветов});
            this.dataGridView1.Location = new System.Drawing.Point(12, 265);
            this.dataGridView1.Name = "dataGridView1";
            this.dataGridView1.Size = new System.Drawing.Size(543, 239);
            this.dataGridView1.TabIndex = 22;
            //
            // Марка
            //
            this.Марка.HeaderText = "Марка";
            this.Марка.Name = "Марка";
            this.Марка.ReadOnly = true;
            //
            // Модель
            //
            this.Модель.HeaderText = "Модель";
            this.Модель.Name = "Модель";
            this.Модель.ReadOnly = true;
            //
            // Тип
            //
            this.Тип.HeaderText = "Тип";
            this.Тип.Name = "Тип";
            this.Тип.ReadOnly = true;
            //
            // ПоддержкаUSB
            //
            this.ПоддержкаUSB.HeaderText = "Поддержка USB";
            this.ПоддержкаUSB.Name = "ПоддержкаUSB";
            this.ПоддержкаUSB.ReadOnly = true;
            this.ПоддержкаUSB.Resizable = Sys-
tem.Windows.Forms.DataGridViewTriState.True;
            this.ПоддержкаUSB.SortMode = DataGridViewColumnSortMode.Automatic;
            //
            // Количество_цветов
            //
            this.Количество_цветов.HeaderText = "Количество Цветов";
            this.Количество_цветов.Name = "Количество_цветов";
            this.Количество_цветов.ReadOnly = true;
            //
            // Form1
            //
            this.AutoScaleDimensions = new System.Drawing.SizeF(6F, 13F);
            this.AutoScaleMode = System.Windows.Forms.AutoScaleMode.Font;
            this.ClientSize = new System.Drawing.Size(568, 516);
            this.Controls.Add(this.dataGridView1);
            this.Controls.Add(this.label8);
            this.Controls.Add(this.label7);
            this.Controls.Add(this.label6);
            this.Controls.Add(this.groupBox1);
            this.Controls.Add(this.button3);
            this.Controls.Add(this.button2);
```

```
            this.Controls.Add(this.button1);
            this.Controls.Add(this.menuStrip1);
            this.MainMenuStrip = this.menuStrip1;
            this.Name = "Form1";
            this.Text = "База данных \"Принтеры\"";
            this.groupBox1.ResumeLayout(false);
            this.groupBox1.PerformLayout();
            this.menuStrip1.ResumeLayout(false);
            this.menuStrip1.PerformLayout();
            ((System.ComponentModel.ISupportInitialize)(this.dataGridView1)).EndInit();
            this.ResumeLayout(false);
            this.PerformLayout();

        }


        private System.Windows.Forms.TextBox textBoxMark;
        private System.Windows.Forms.TextBox textBoxModel;
        private System.Windows.Forms.TextBox textBoxColors;
        private System.Windows.Forms.Button button1;
        private System.Windows.Forms.DomainUpDown domainUpDownType;
        private System.Windows.Forms.Button button2;
        private System.Windows.Forms.DomainUpDown domainUpDownUSB;
        private System.Windows.Forms.Button button3;
        private System.Windows.Forms.Label label1;
        private System.Windows.Forms.Label label2;
        private System.Windows.Forms.Label label3;
        private System.Windows.Forms.Label label4;
        private System.Windows.Forms.Label label5;
        private System.Windows.Forms.GroupBox groupBox1;
        private System.Windows.Forms.Label label6;
        private System.Windows.Forms.Label label7;
        private System.Windows.Forms.Label label8;
        private System.Windows.Forms.MenuStrip menuStrip1;
        private System.Windows.Forms.ToolStripMenuItem выходToolStripMenuItem;
        private System.Windows.Forms.ToolStripMenuItem выходToolStripMenuItem1;
        private System.Windows.Forms.ToolStripMenuItem очиститьПоляToolStrip-
MenuItem;
        private System.Windows.Forms.ToolStripMenuItem помощьToolStripMenuItem;
        private System.Windows.Forms.ToolStripMenuItem помощьToolStripMenuItem1;
        private System.Windows.Forms.ToolStripMenuItem оПрограммеToolStripMenuItem;
        private System.Windows.Forms.DataGridView dataGridView1;
        private System.Windows.Forms.DataGridViewTextBoxColumn Марка;
        private System.Windows.Forms.DataGridViewTextBoxColumn Модель;
        private System.Windows.Forms.DataGridViewTextBoxColumn Тип;
        private System.Windows.Forms.DataGridViewCheckBoxColumn ПоддержкаUSB;
        private System.Windows.Forms.DataGridViewTextBoxColumn Количество_цветов;
    }
}
```

Пример 2

Предметная область - сеть аэропортов, связанных авиатрассами
Определить длину кратчайшего по количеству трасс маршрута

от заданного аэропорта до всех остальных аэропортов.

Представление сети аэропортов - граф, заданный матрицей смежности.

Вершина графа - аэропорт, ребро - авиатрасса.

Редактирование матрицы смежности - визуально щелчком мыши на соответствующей ячейке. Назначение аэропорта вылета - щелчком мыши на имени аэропорта в левой колонке таблицы.

Решение задачи основано на обходе графа в ширину.

Управление обходом через очередь.

Роль окраски вершин при обходе выполняет массив расстояний.

Отрицательное значение расстояния эквивалентно не пройденной вершине.

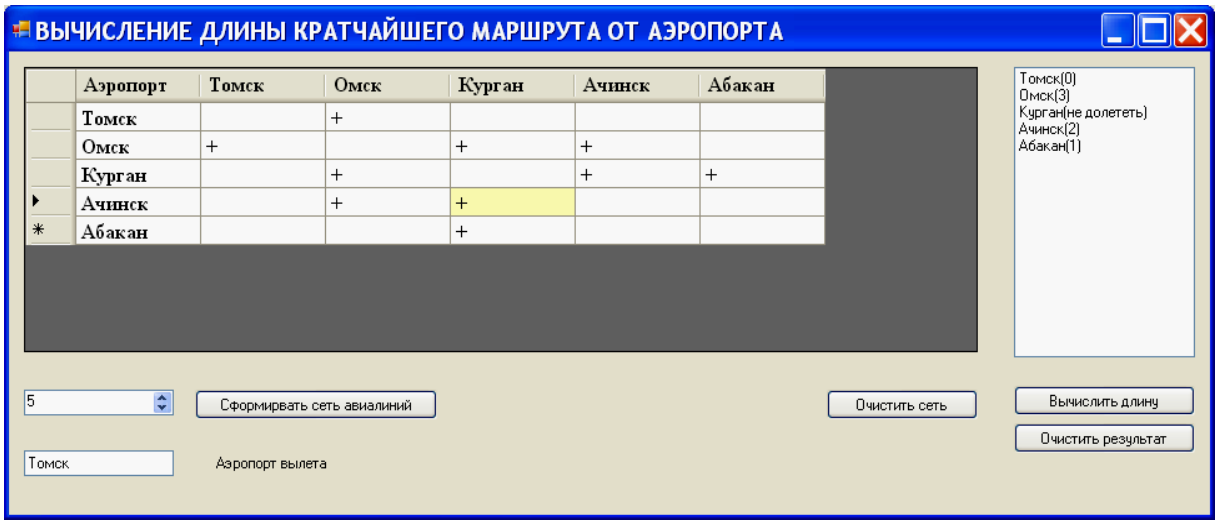


рис.10

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Text;
using System.Windows.Forms;
using System.Collections;


namespace EX1
{
   public partial class Form1 : Form
   {
      public Form1()
      {
         InitializeComponent();
      }

      string[] name;            //имена аэропортов
      int      n,       //Количество аэропортов
               старт;    //порядковый номер аэропорта вылета
      int[]    d;       //Расстояние от аэропорта вылета до всех аэропортов
```

```
private void Расстояние(bool[,] матр)
{
   int i;
   int номаэропорт,расстояние;
   Queue очередь = new Queue();
   очередь.Clear();
   d = new int[n];

   for(i=0; i<n; i++) d[i] = -1;

   d[старт] = 0;
   очередь.Enqueue(старт);

   while (очередь.Count != 0)
   {
      номаэропорт = (int)очередь.Dequeue();
      расстояние = d[номаэропорт] + 1;

      for (i = 0; i < n; i++)
         if (матр[номаэропорт, i] && d[i] == -1)
         {
            d[i] = расстояние;
            очередь.Enqueue(i);
         }
   }
}

private void Form1_Load(object sender, EventArgs e)
{
   name = new string[]
        { "Томск","Омск","Курган","Ачинск","Абакан",
          "Тюмень","Тайга","Барнаул","Бийск","Бердск"};
}

private void Сеть_Click(object sender, EventArgs e)
{
   n = (int)Количество.Value;
   Tpacca.Columns.Add("", "Аэропорт");

   for(int i = 0; i < n; i++) Tpacca.Columns.Add("", name[i]);
   for (int i = 0; i < n-1; i++) Tpacca.Rows.Add();
   for(int y = 0; y < n; y++) Tpacca[0, y].Value = name[y];

   for (int x = 1; x <=n; x++)
      for (int y = 0; y < n; y++)
         Tpacca[x, y].Value = " ";
}


private void Tpacca_CurrentCellChanged(object sender, EventArgs e)
{
   int x, y;
```

```
    x = Tpacca.CurrentCellAddress.X;
    y = Tpacca.CurrentCellAddress.Y;

    if (x < 0) return;

    if (x == 0)
    {
        старт = Tpacca.CurrentCellAddress.Y;
        аэропортВылета.Text = name[старт];
        return;
    }
    if (x == y + 1) return;
    if ((string)(Tpacca[x, y].Value) == " ")
    {
        Tpacca[x, y].Value = "+";
        Tpacca[y + 1, x - 1].Value = "+";
    }
    else
    {
        Tpacca[x, y].Value = " ";
        Tpacca[y + 1, x - 1].Value = " ";
    }
}
private void Вычислить_Click(object sender, EventArgs e)
{
    bool[,] матр = new bool[n, n];
    int i,j;
    for (i = 0; i < n; i++)
        for (j = 0; j < n; j++)
            if ((string)(Tpacca[j+1, i].Value) == "+")
                матр[i, j] = true;
            else
                матр[i, j] = false;
    Расстояние(матр);
    for (i = 0; i < n; i++)
        if(d[i]>=0)
          Маршрут.Items.Add(name[i]+"("+d[i]+")");
        else
          Маршрут.Items.Add(name[i]+"(не долететь)");
}


private void Очистить_Click(object sender, EventArgs e)
{
    Tpacca.Columns.Clear();
    Tpacca.Rows.Clear();
    Маршрут.Items.Clear();
}


private void ОчисткаMap_Click(object sender, EventArgs e)
{
    Маршрут.Items.Clear();
}
```

```
    }
}


using System;
using System.Collections.Generic;
using System.Windows.Forms;

namespace EX1
{
    static class Program
    {
        /// <summary>
        /// The main entry point for the application.
        /// </summary>
        [STAThread]
        static void Main()
        {
            Application.EnableVisualStyles();
            Application.SetCompatibleTextRenderingDefault(false);
            Application.Run(new Form1());
        }
    }
}




namespace EX1
{
    partial class Form1
    {
        private System.ComponentModel.IContainer components = null;

        protected override void Dispose(bool disposing)
        {
            if (disposing && (components != null))
            {
                components.Dispose();
            }
            base.Dispose(disposing);
        }

        #region Windows Form Designer generated code

        private void InitializeComponent()
        {
            DataGridViewCellStyle  dataGridViewCellStyle3 = new
                                                    Sys-
tem.Windows.Forms.DataGridViewCellStyle();
            this.Tpacca = new System.Windows.Forms.DataGridView();

            this.Сеть = new System.Windows.Forms.Button();
            this.Количество = new System.Windows.Forms.NumericUpDown();
```

```
            this.Маршрут = new System.Windows.Forms.ListBox();
            this.Вычислить = new System.Windows.Forms.Button();
            this.Очистить = new System.Windows.Forms.Button();
            this.ОчисткаМар = new System.Windows.Forms.Button();
            this.аэропортВылета = new System.Windows.Forms.TextBox();
            this.Вылет = new System.Windows.Forms.Label();

            ((System.ComponentModel.ISupportInitialize)(this.Tpacca)).BeginInit();
            ((System.ComponentModel.ISupportInitialize)(this.Количество)).BeginInit();
            this.SuspendLayout();
            //
            // Tpacca
            //
            dataGridViewCellStyle3.Alignment =
                System.Windows.Forms.DataGridViewContentAlignment.MiddleLeft;
            dataGridViewCellStyle3.BackColor =
                                    System.Drawing.Color.White;
            dataGridViewCellStyle3.Font = new System.Drawing.Font("Times New Roman",
                                        12F, System.Drawing.FontStyle.Bold,
                                        System.Drawing.GraphicsUnit.Point,
                                        ((byte)(204)));
            dataGridViewCellStyle3.ForeColor = System.Drawing.SystemColors.WindowText;
            dataGridViewCellStyle3.SelectionBackColor =
                                System.Drawing.Color.FromArgb(((int)(((byte)(255)))),
                                            ((int)(((byte)(255)))),
                                                    ((int)(((byte)(192)))));
            dataGridViewCellStyle3.SelectionForeColor = System.Drawing.Color.Black;
            dataGridViewCellStyle3.WrapMode =
                                        Sys-
tem.Windows.Forms.DataGridViewTriState.True;
            this.Tpacca.ColumnHeadersDefaultCellStyle = dataGridViewCellStyle3;
            this.Tpacca.ColumnHeadersHeightSizeMode =
                    Sys-
tem.Windows.Forms.DataGridViewColumnHeadersHeightSizeMode.AutoSize;
            this.Tpacca.Location = new System.Drawing.Point(12, 12);
            this.Tpacca.Name = "Tpacca";
            this.Tpacca.RowTemplate.DefaultCellStyle.Font = new
                                        System.Drawing.Font("Times New Roman", 12F,
                                        System.Drawing.FontStyle.Bold,
                                        System.Drawing.GraphicsUnit.Point,
                                        ((byte)(204)));
            this.Tpacca.RowTemplate.DefaultCellStyle.ForeColor =
                                                                Sys-
tem.Drawing.Color.Black;
            this.Tpacca.RowTemplate.DefaultCellStyle.SelectionBackColor =
                                    System.Drawing.Color.FromArgb(((int)(((byte)(255)))),
                                    ((int)(((byte)(255)))), ((int)(((byte)(192)))));
            this.Tpacca.RowTemplate.DefaultCellStyle.SelectionForeColor =
                                                                Sys-
tem.Drawing.Color.Black;
            this.Tpacca.Size = new System.Drawing.Size(764, 221);
            this.Tpacca.TabIndex = 0;
            this.Tpacca.CurrentCellChanged += new
                                    System.EventHandler(this.Tpacca_CurrentCellChanged);
```

```
//
// Сеть
//
this.Сеть.Location = new System.Drawing.Point(149, 262);
this.Сеть.Name = "Сеть";
this.Сеть.Size = new System.Drawing.Size(194, 23);
this.Сеть.TabIndex = 1;
this.Сеть.Text = "Сформирвать сеть авиалиний";
this.Сеть.UseVisualStyleBackColor = true;
this.Сеть.Click += new System.EventHandler(this.Сеть_Click);
//
// Количество
//
this.Количество.Location = new System.Drawing.Point(12, 262);
this.Количество.Maximum = new decimal(new int[] {
10,
0,
0,
0});
this.Количество.Minimum = new decimal(new int[] {
2,
0,
0,
0});
this.Количество.Name = "Количество";
this.Количество.Size = new System.Drawing.Size(120, 20);
this.Количество.TabIndex = 2;
this.Количество.Value = new decimal(new int[] {
5,
0,
0,
0});
//
// Маршрут
//
this.Маршрут.FormattingEnabled = true;
this.Маршрут.Location = new System.Drawing.Point(805, 12);
this.Маршрут.Name = "Маршрут";
this.Маршрут.Size = new System.Drawing.Size(145, 225);
this.Маршрут.TabIndex = 3;
//
// Вычислить
//
this.Вычислить.Location = new System.Drawing.Point(805, 259);
this.Вычислить.Name = "Вычислить";
this.Вычислить.Size = new System.Drawing.Size(145, 23);
this.Вычислить.TabIndex = 4;
this.Вычислить.Text = "Вычислить длину";
this.Вычислить.UseVisualStyleBackColor = true;
this.Вычислить.Click += new System.EventHandler(this.Вычислить_Click);
//
// Очистить
//
this.Очистить.Location = new System.Drawing.Point(655, 262);
```

```
            this.Очистить.Name = "Очистить";
            this.Очистить.Size = new System.Drawing.Size(121, 23);
            this.Очистить.TabIndex = 5;
            this.Очистить.Text = "Очистить сеть";
            this.Очистить.UseVisualStyleBackColor = true;
            this.Очистить.Click += new System.EventHandler(this.Очистить_Click);
            //
            // ОчисткаМар
            //
            this.ОчисткаМар.Location = new System.Drawing.Point(805, 288);
            this.ОчисткаМар.Name = "ОчисткаМар";
            this.ОчисткаМар.Size = new System.Drawing.Size(145, 23);
            this.ОчисткаМар.TabIndex = 6;
            this.ОчисткаМар.Text = "Очистить результат";
            this.ОчисткаМар.UseVisualStyleBackColor = true;
            this.ОчисткаМар.Click += new System.EventHandler(this.ОчисткаМар_Click);
            //
            // аэропортВылета
            //
            this.аэропортВылета.Location = new System.Drawing.Point(12, 309);
            this.аэропортВылета.Name = "аэропортВылета";
            this.аэропортВылета.Size = new System.Drawing.Size(120, 20);
            this.аэропортВылета.TabIndex = 7;
            //
            // Вылет
            //
            this.Вылет.AutoSize = true;
            this.Вылет.Location = new System.Drawing.Point(163, 312);
            this.Вылет.Name = "Вылет";
            this.Вылет.Size = new System.Drawing.Size(96, 13);
            this.Вылет.TabIndex = 8;
            this.Вылет.Text = "Аэропорт вылета";
            //
            // Form1
            //
            this.AutoScaleDimensions = new System.Drawing.SizeF(6F, 13F);
            this.AutoScaleMode = System.Windows.Forms.AutoScaleMode.Font;
            this.ClientSize = new System.Drawing.Size(962, 359);
            this.Controls.Add(this.Вылет);
            this.Controls.Add(this.аэропортВылета);
            this.Controls.Add(this.ОчисткаМар);
            this.Controls.Add(this.Очистить);
            this.Controls.Add(this.Вычислить);
            this.Controls.Add(this.Маршрут);
            this.Controls.Add(this.Количество);
            this.Controls.Add(this.Сеть);
            this.Controls.Add(this.Трасса);
            this.Name = "Form1";
            this.Text = "ВЫЧИСЛЕНИЕ ДЛИНЫ КРАТЧАЙШЕГО МАРШРУТА ОТ АЭ-
РОПОРТА";
            this.Load += new System.EventHandler(this.Form1_Load);
            ((System.ComponentModel.ISupportInitialize)(this.Трасса)).EndInit();
            ((System.ComponentModel.ISupportInitialize)(this.Количество)).EndInit();
            this.ResumeLayout(false);
```

```
            this.PerformLayout();

        }

        #endregion

        private System.Windows.Forms.DataGridView Трасса;
        private System.Windows.Forms.Button Сеть;
        private System.Windows.Forms.NumericUpDown Количество;
        private System.Windows.Forms.ListBox Маршрут;
        private System.Windows.Forms.Button Вычислить;
        private System.Windows.Forms.Button Очистить;
        private System.Windows.Forms.Button ОчисткаМар;
        private System.Windows.Forms.TextBox аэропортВылета;
        private System.Windows.Forms.Label Вылет;
    }
}
```

Обработчики событий:

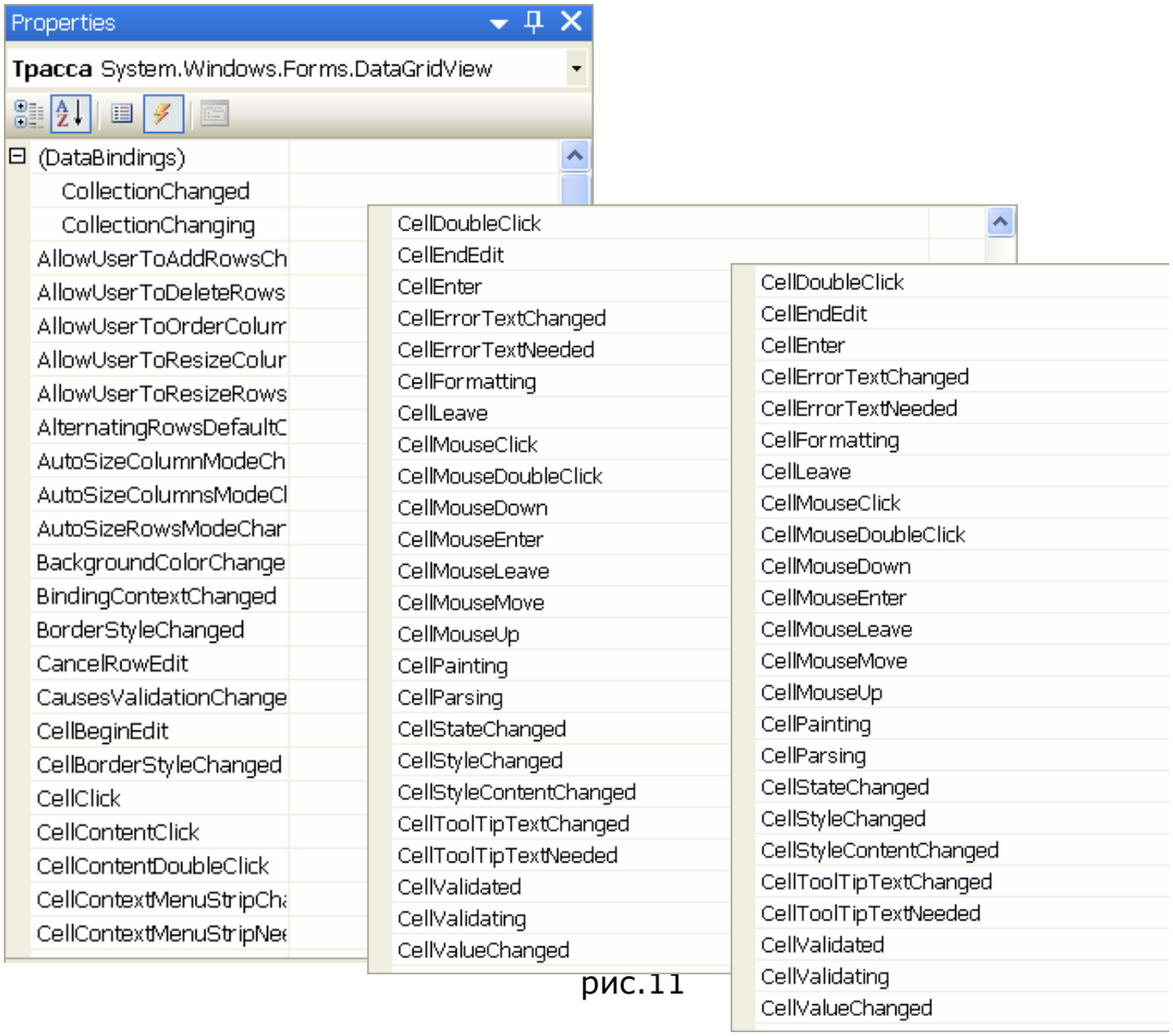


рис.11

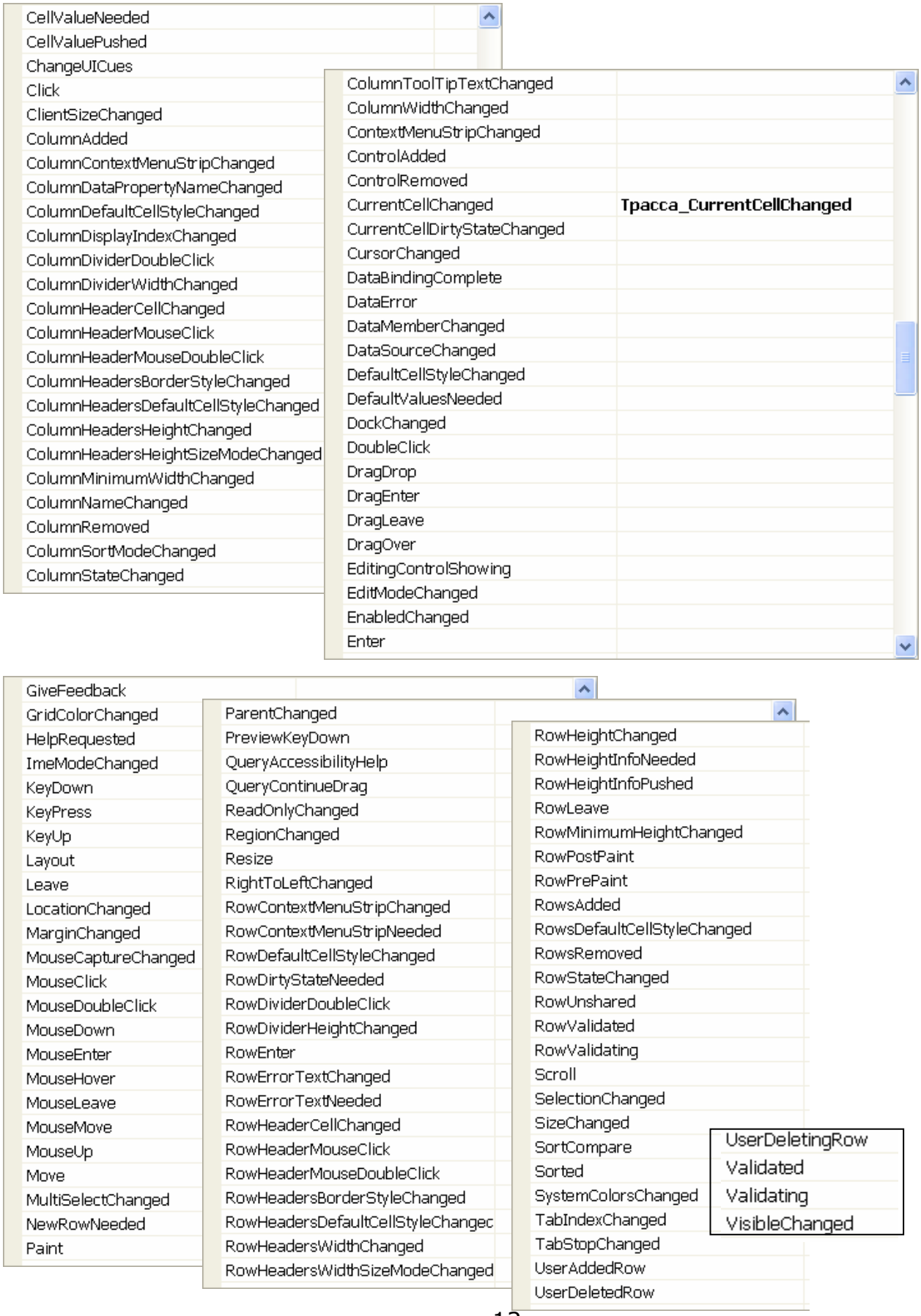

CellValueNeeded
CellValuePushed
ChangeUICues
Click
ClientSizeChanged
ColumnAdded
ColumnContextMenuStripChanged
ColumnDataPropertyNameChanged
ColumnDefaultCellStyleChanged
ColumnDisplayIndexChanged
ColumnDividerDoubleClick
ColumnDividerWidthChanged
ColumnHeaderCellChanged
ColumnHeaderMouseClick
ColumnHeaderMouseDoubleClick
ColumnHeadersBorderStyleChanged
ColumnHeadersDefaultCellStyleChanged
ColumnHeadersHeightChanged
ColumnHeadersHeightSizeModeChanged
ColumnMinimumWidthChanged
ColumnNameChanged
ColumnRemoved
ColumnSortModeChanged
ColumnStateChanged
ColumnToolTipTextChanged
ColumnWidthChanged
ContextMenuStripChanged
ControlAdded
ControlRemoved
CurrentCellChanged
Tpacca_CurrentCellChanged
CurrentCellDirtyStateChanged
CursorChanged
DataBindingComplete
DataError
DataMemberChanged
DataSourceChanged
DefaultCellStyleChanged
DefaultValuesNeeded
DockChanged
DoubleClick
DragDrop
DragEnter
DragLeave
DragOver
EditingControlShowing
EditModeChanged
EnabledChanged
Enter
GiveFeedback
GridColorChanged
HelpRequested
ImeModeChanged
KeyDown
KeyPress
KeyUp
Layout
Leave
LocationChanged
MarginChanged
MouseCaptureChanged
MouseClick
MouseDoubleClick
MouseDown
MouseEnter
MouseHover
MouseLeave
MouseMove
MouseUp
Move
MultiSelectChanged
NewRowNeeded
Paint
ParentChanged
PreviewKeyDown
QueryAccessibilityHelp
QueryContinueDrag
ReadOnlyChanged
RegionChanged
Resize
RightToLeftChanged
RowContextMenuStripChanged
RowContextMenuStripNeeded
RowDefaultCellStyleChanged
RowDirtyStateNeeded
RowDividerDoubleClick
RowDividerHeightChanged
RowEnter
RowErrorTextChanged
RowErrorTextNeeded
RowHeaderCellChanged
RowHeaderMouseClick
RowHeaderMouseDoubleClick
RowHeadersBorderStyleChanged
RowHeadersDefaultCellStyleChangec
RowHeadersWidthChanged
RowHeadersWidthSizeModeChanged
RowHeightChanged
RowHeightInfoNeeded
RowHeightInfoPushed
RowLeave
RowMinimumHeightChanged
RowPostPaint
RowPrePaint
RowsAdded
RowsDefaultCellStyleChanged
RowsRemoved
RowStateChanged
RowUnshared
RowValidated
RowValidating
Scroll
SelectionChanged
SizeChanged
SortCompare
Sorted
SystemColorsChanged
TabIndexChanged
TabStopChanged
UserAddedRow
UserDeletedRow
UserDeletingRow
Validated
Validating
VisibleChanged

рис.12

4. Определение источника данных для таблицы DataGridView

Таблица DataGridView может быть привязана к источнику данных.

Источник данных устанавливается свойством DataSource. Источниками могут быть:

- одномерный массив (Array, ArrayList, StringCollection и др.);
- компоненты, реализующие интерфейс IList или IListSource;
- обобщенные классы коллекций;
- DataTable;
- DataView;
- DataSet или DataViewManager.

Если элемент управления DataGridView привязан к данным с помощью нескольких связанных таблиц СУБД и в сетке разрешены переходы, в каждом ряду сетки отображаются расширители.

Расширитель позволяет переходить из главной таблицы в подчиненную. При щелчке узла отображается подчиненная таблица, а при нажатии кнопки возврата — исходная главная таблица. Таким образом, в сетке отображаются иерархические отношения между таблицами.

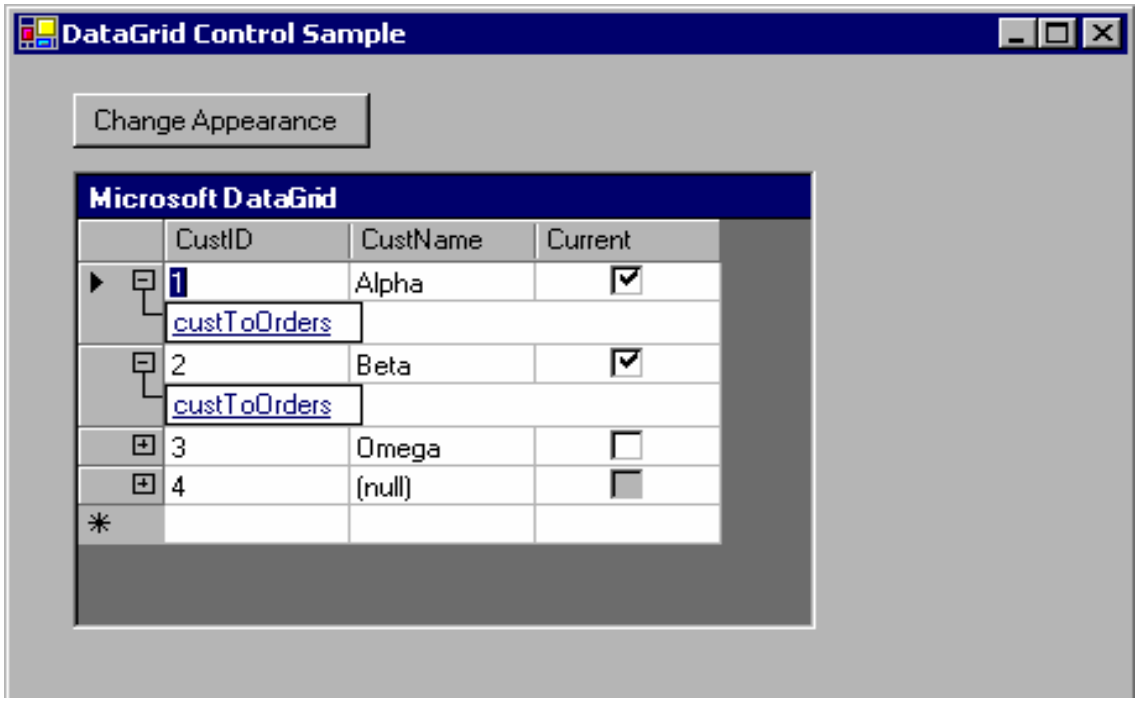


рис.13

Если установлено dataGridView1.AutoGenerateColumns = true; то после привязки таблицы к объекту столбцы и строки автоматически создаются, форматируются и заполняются.

После создания элемента управления DataGridView столбцы можно добавлять, удалять, упорядочивать и форматировать. Можно изменять заголовки столбцов и другие их свойства.

Когда в качестве источника используется массив, то ЭУ DataGridView ищет подходящие открытые свойства и создает из них столбцы и строки: имена свойств становятся именами столбцов, а значения свойств – значениями ячеек.

Свойство, возвращающее данное типа bool, порождает столбец типа DataGridViewCheckBoxColumn. Другие свойства, возвращающие значения типа string, int, float и т.д., отображаются в столбце типа DataGridViewTextBoxColumn.

Если свойство имеет аксессор set{}, то изменение значения в ячейке, соответствующей свойству, приводит к изменению источника (вызывается метод свойства get).

Чтобы открытое свойство не отображалось в таблице, необходимо перед ним определить атрибут [Browsable(false)].

Пример.

Источником данных таблицы выбран динамический массив, являющийся базовым классом для класса PersonList. Элементами массива являются экземпляры класса Person.

Так как определены get и set аксессоры свойств, можно изменять массив из таблицы.

Именами столбцов являются имена свойств.

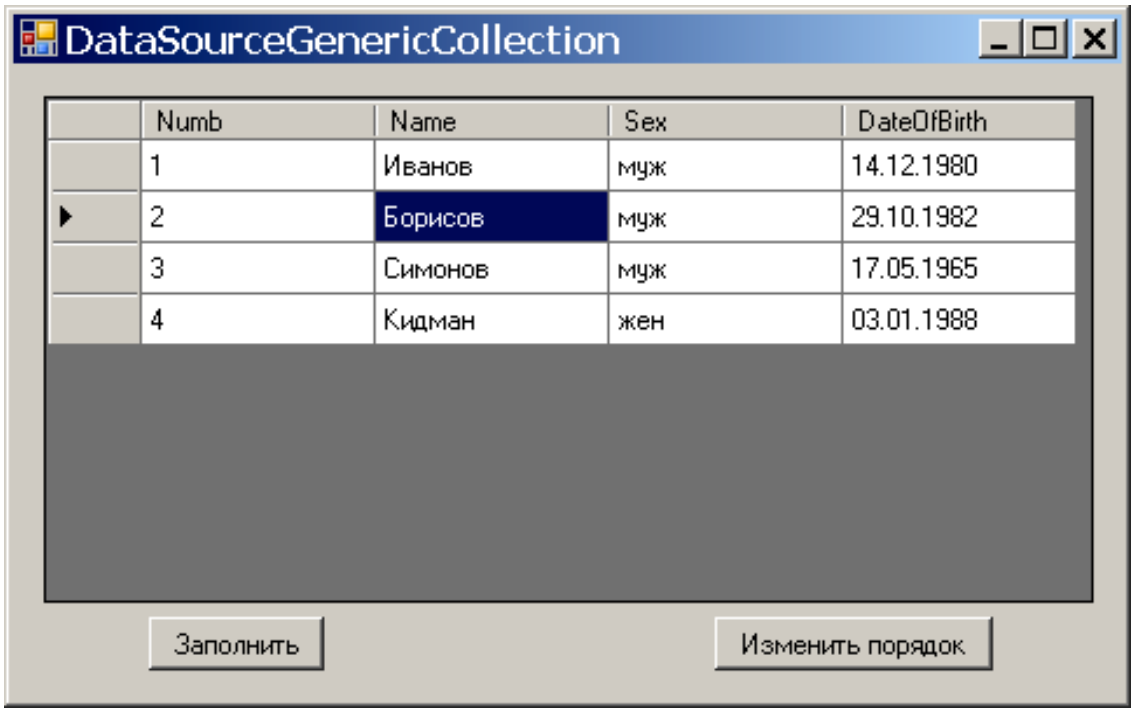


рис.14

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Drawing;
using System.Text;

public class Person
{
   private uint numb;
   private string name;
   private Sex sex;
   private DateTime dateOfBirth;
   static uint numbs = 0;
```

```
    public Person( string name, Sex sex, DateTime dob )
    {
       this.name = name;
       this.sex = sex;
       this.dateOfBirth = dob;
       numb = ++numbs;
    }

    // [Browsable(false)]
    public uint Numb
    {
        get { return numb; }
    }

    public string Name
    {
        get { return name;   }
        set { name = value; }
    }

    public Sex Sex
    {
        get { return sex;   }
        set { sex = value; }
    }

    public DateTime DateOfBirth
    {
       get { return dateOfBirth; }
       set { dateOfBirth = value; }
    }
}


   public enum Sex
    {
       муж,
       жен
    }

using System;
using System.Collections;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Text;
using System.Windows.Forms;

public partial class Form1 : Form
{
    public Form1()
    {
       InitializeComponent();
```

```
  }

 ArrayList   people;

 private void Form1_Load(object sender, EventArgs e)
 {
   // создадим массив объектов типа Person
   people = new ArrayList ();

   people.Add (new Person("Иванов", Sex.муж, new DateTime(1980, 12, 14)));
   people.Add(new Person("Борисов", Sex.муж, new DateTime(1982, 10, 29)));
   people.Add (new Person("Симонов", Sex.муж, new DateTime(1965, 5, 17)));
   people.Add (new Person("Кидман", Sex.жен, new DateTime(1988, 1, 3)));
 }

  private void getData_Click(object sender, EventArgs e)
  {
     dataGridView1.AutoGenerateColumns = true;

     dataGridView1.DataSource = people;

     // Изменить имя столбца
     dataGridView1.Columns["Sex"].HeaderText = "Пол";

     // установим последовательность столбцов в таблице
     DataGridViewColumnCollection columns = dataGridView1.Columns;
     columns[0].DisplayIndex = 2;
     columns[1].DisplayIndex = 3;
     columns[2].DisplayIndex = 0;
    //другой способ выборки столбца (не путать Name со свойством таблицы)
     dataGridView1.Columns["Name"].DisplayIndex = 1;
  }


     private void Изменить_порядок_Click(object sender, EventArgs e)
     {
        MessageBox.Show(((Person)people[1]).Name);  // старое значение
        ((Person)people[1]).Name = "aaaaaaa";           // новое значение имени
        // изменение значения в массиве находит отражение в таблице

        // перестановка столбцов
        dataGridView1.Columns["Numb"].DisplayIndex = 1;
        dataGridView1.Columns["Name"].DisplayIndex = 0;
        dataGridView1.Columns["Sex"].DisplayIndex = 3;
        dataGridView1.Columns["DateOfBirth"].DisplayIndex = 2;
     }
}
```

Изменение значения ячейки в таблице приводит к изменению соответствующего элемента в массиве.

Изменяем “Борисов” на “Антонов”.

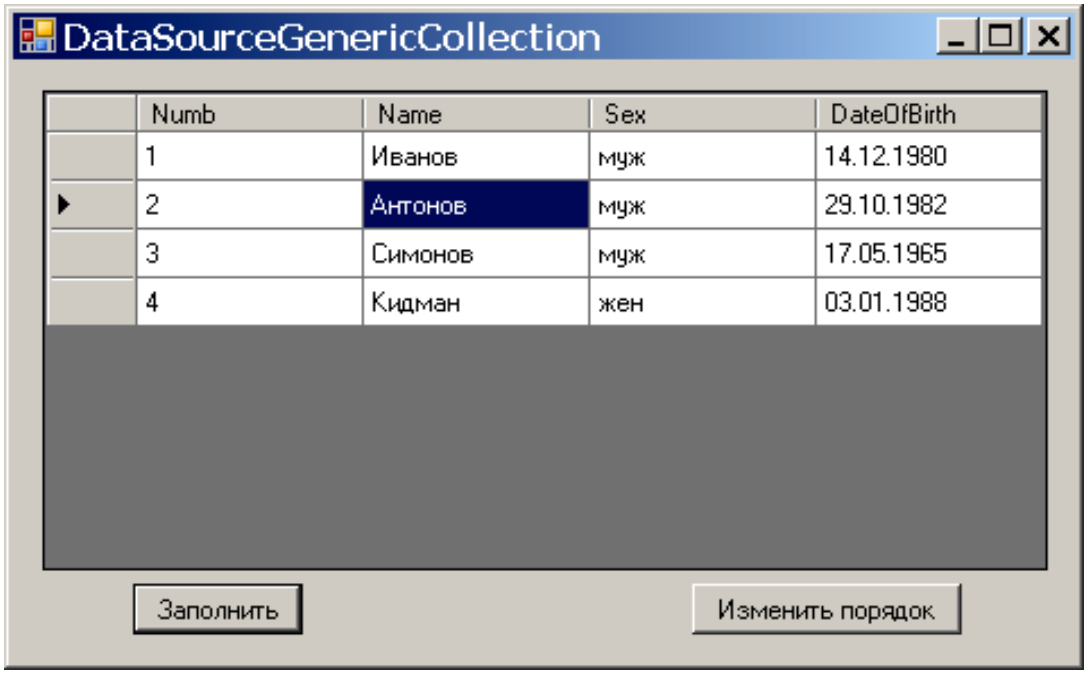


рис.15

Изменение элемента массива в программе

```
((Person)people[1]).Name = "aaaaaaa";
```

отображается в таблице.

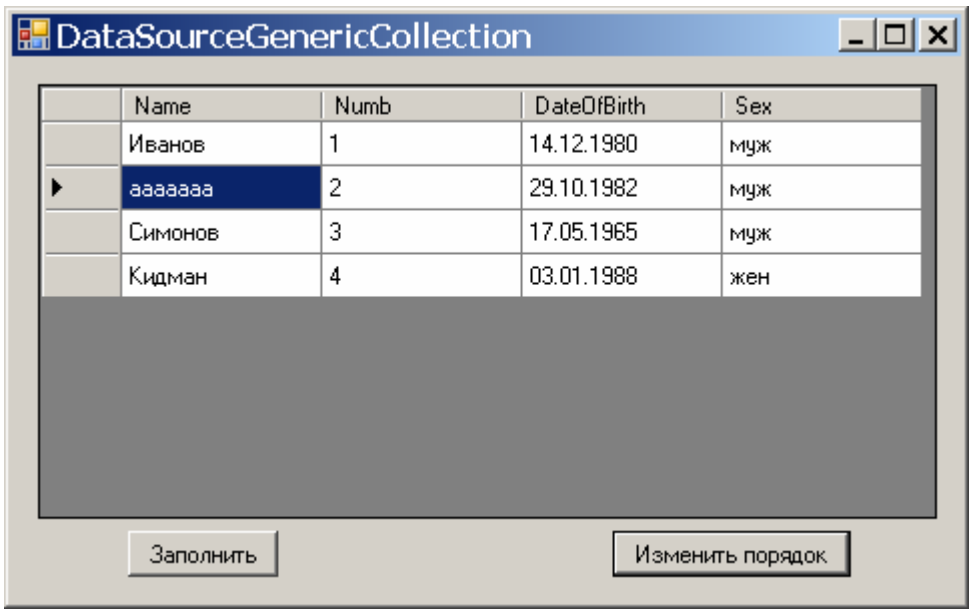


рис.16

Порядок следования столбцов не всегда соответствует порядку следования свойств массива.

Примеры изменения порядка следования столбцов:

```
dataGridView1.AutoGenerateColumns = true;
dataGridView1.DataSource = people;
```

Способ 1. Использование числового индекса

```
dataGridView1.Columns[3].DisplayIndex = 3;

DataGridViewColumnCollection columns = dataGridView1.Columns;
columns[0].DisplayIndex = 2;
columns[1].DisplayIndex = 1;
columns[2].DisplayIndex = 0;
```

Способ 2. Использование имени столбца в качестве индекса

```
dataGridView1.Columns["Numb"].DisplayIndex = 1;
dataGridView1.Columns["Name"].DisplayIndex = 0;
dataGridView1.Columns["Sex"].DisplayIndex = 3;
dataGridView1.Columns["DateOfBirth"].DisplayIndex = 2;
```

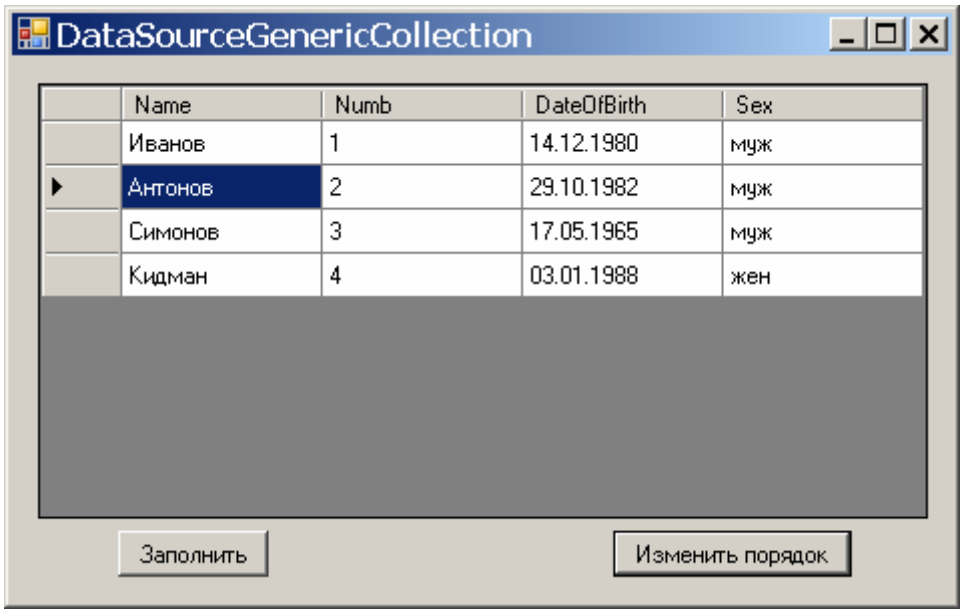


рис.17

Определение источника данных для столбца типа DataGridViewComboBoxColumn

Комбинированный список является общим для всех ячеек столбца DataGridViewComboBoxColumn. Поэтому в качестве источника данных для такого столбца может использоваться массив объектов. Имя массива необходимо указать в свойстве DataSource столбца DataGridViewComboBoxColumn.

Если объект массива содержит несколько открытых свойств, то необходимо в DisplayMember указать, какое свойство является отображаемым, а в ValueMember - какое свойство возвращает значение в программу. Свойство DataPropertyName указывает имя свойства в источнике данных таблицы, которое является индексом в списке.

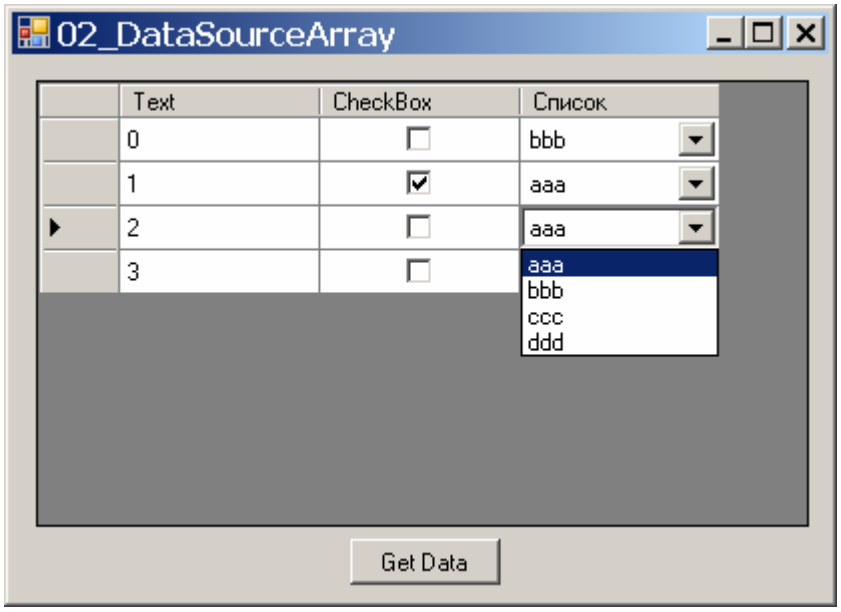


рис.18

```
/* Пример определения источника данных для таблицы.
 * Столбцы строятся автоматически.
*/
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Text;
using System.Windows.Forms;
using System.Collections;

public partial class Form1 : Form
{
   public Form1()
   {
      InitializeComponent();
   }

   Item[] items;

   private void getData_Click(object sender, EventArgs e)
   {
      // создать источник данных для таблицы
      items = new Item[4];

      items[0] = new Item(0, false);
      items[1] = new Item(1, true);
      items[2] = new Item(2, false);
      items[3] = new Item(3, false);

      // создать массив для списка
      ArrayList arl = new ArrayList();

      arl.Add (new ItemComboBox("aaa",0));
```

```
    arl.Add (new ItemComboBox("bbb",1));
    arl.Add (new ItemComboBox("ccc", 2));
    arl.Add (new ItemComboBox("ddd",3));

    dataGridView1.AutoGenerateColumns = true;
    dataGridView1.DataSource = items;
    dataGridView1.Columns["Boolean"].HeaderText = "CheckBox";

     {   // создание столбца – комбинированного списка
        DataGridViewComboBoxColumn cbc = new
                                        DataGridViewComboBoxColumn();
        cbc.HeaderText = "Список";

        cbc.DataSource = arl;         // массив в качестве источника
        cbc.DisplayMember = "TextCBox";        //из ItemComboBox !
        cbc.ValueMember = "Value";              //из ItemComboBox !
        //cbc.DataPropertyName = "TextBox";         //из Item !

       // добавить комб. список к коллекции столбцов
        dataGridView1.Columns.Add(cbc);
}


protected class Item
{
   private int textBox;
   private bool bolean;

   public Item(int textBox, bool bolean)
   {
      this.textBox = textBox;
      this.bolean = bolean;
   }

   public int TextBox
   { get { return textBox; } }

   public bool Boolean
   { get { return bolean; } }
}


protected class ItemComboBox
{
   private string textCBox;
   private int value;

   public ItemComboBox(string textCBox, int value)
   {
      this.textCBox = textCBox;
      this.value = value;
   }

   public string TextCBox
```

```
        { get { return textCBox; } }

        public int Value
        { get { return value; } }
    }
}
```

The DataGridViewComboBoxColumn class is a specialized type of DataGridViewColumn used to logically host cells that enable selection and editing. A DataGridViewComboBoxColumn has an associated DataGridView-ComboBoxCell in every DataGridViewRow that intersects it.

The following steps describe a common way to use this class:

Create a DataGridView.

Set the DataSource property of the DataGridView.

Create a DataGridViewComboBoxColumn.

If the data source contains multiple properties or columns, set the DataPropertyName property to the name of the property or column for populating the cell values.

DataPropertyName - gets or sets the name of the data source property or database column to which the DataGridViewColumn is bound.

Use the DataSource or Items properties to establish the selections.

Use the ValueMember and DisplayMember properties to establish a mapping between cell values and what will be displayed on the screen.

The DataGridViewComboBoxColumn will only work properly if there is a mapping between all its cell values that are populated by the DataGrid-View.DataSource property and the range of choices populated either by the DataSource property or the Items property. If this mapping doesn't exist, the message "An Error happened Formatting, Display" will appear when the column is in view.

The default sort mode for this column type is NotSortable.

Notes to Inheritors When you derive from DataGridViewComboBoxColumn and add new properties to the derived class, be sure to override the Clone method to copy the new properties during cloning operations. You should also call the base class's Clone method so that the properties of the base class are copied to the new cell.

Дополнительные примеры:

-Example

The following code example demonstrates how to create an unbound DataGridView; set the ColumnHeadersVisible, ColumnHeadersDefaultCellStyle, and ColumnCount properties; and use the Rows and Columns properties. It also demonstrates how to use a version of the AutoResizeColumnHeadersHeight and AutoResizeRows methods to properly size the column headers and the rows. To run this example, paste the following code into a form that contains a Data-GridView named dataGridView1 and a button named Button1, and then call the InitializeDataGridView method from the form's constructor or Load event handler. Ensure all events are connected with their event handlers.

```
private void InitializeDataGridView()
{
   // Create an unbound DataGridView by declaring a column count.
```

```
    dataGridView1.ColumnCount = 4;
    dataGridView1.ColumnHeadersVisible = true;

    // Set the column header style.
    DataGridViewCellStyle columnHeaderStyle = new DataGridViewCellStyle();

    columnHeaderStyle.BackColor = Color.Beige;
    columnHeaderStyle.Font = new Font("Verdana", 10, FontStyle.Bold);
    dataGridView1.ColumnHeadersDefaultCellStyle = columnHeaderStyle;

    // Set the column header names.
    dataGridView1.Columns[0].Name = "Recipe";
    dataGridView1.Columns[1].Name = "Category";
    dataGridView1.Columns[2].Name = "Main Ingredients";
    dataGridView1.Columns[3].Name = "Rating";

    // Populate the rows.
    string[] row1 = new string[] { "Meatloaf", "Main Dish", "ground beef",
        "**" };
    string[] row2 = new string[] { "Key Lime Pie", "Dessert",
        "lime juice, evaporated milk", "****" };
    string[] row3 = new string[] { "Orange-Salsa Pork Chops", "Main Dish",
        "pork chops, salsa, orange juice", "****" };
    string[] row4 = new string[] { "Black Bean and Rice Salad", "Salad",
        "black beans, brown rice", "****" };
    string[] row5 = new string[] { "Chocolate Cheesecake", "Dessert",
        "cream cheese", "***" };
    string[] row6 = new string[] { "Black Bean Dip", "Appetizer",
        "black beans, sour cream", "***" };

    object[] rows = new object[] { row1, row2, row3, row4, row5, row6 };

    foreach (string[] rowArray in rows)
    {
        dataGridView1.Rows.Add(rowArray);
    }
}

private void button1_Click(object sender, System.EventArgs e)
{
    // Resize the height of the column headers.
    dataGridView1.AutoResizeColumnHeadersHeight();

    // Resize all the row heights to fit the contents of all non-header cells.
    dataGridView1.AutoResizeRows(
        DataGridViewAutoSizeRowsMode.AllCellsExceptHeaders);
}

private void InitializeContextMenu()
{
    // Create the menu item.
    ToolStripMenuItem getRecipe = new ToolStripMenuItem("Search for recipe", null,
        new System.EventHandler(ShortcutMenuClick));
```

```
    // Add the menu item to the shortcut menu.
    ContextMenuStrip recipeMenu = new ContextMenuStrip();
    recipeMenu.Items.Add(getRecipe);

    // Set the shortcut menu for the first column.
    dataGridView1.Columns[0].ContextMenuStrip = recipeMenu;
    dataGridView1.MouseDown += new MouseEventHandler(dataGridView1_MouseDown);
}

private DataGridViewCell clickedCell;

private void dataGridView1_MouseDown(object sender, MouseEventArgs e)
{
// If the user right-clicks a cell, store it for use by the shortcut menu.
    if (e.Button == MouseButtons.Right)
    {
        DataGridView.HitTestInfo hit = dataGridView1.HitTest(e.X, e.Y);
        if (hit.Type == DataGridViewHitTestType.Cell)
        {
            clickedCell =
                dataGridView1.Rows[hit.RowIndex].Cells[hit.ColumnIndex];
        }
    }
}

private void ShortcutMenuClick(object sender, System.EventArgs e)
{
    if (clickedCell != null)
    {
        //Retrieve the recipe name.
        string recipeName = (string)clickedCell.Value;

        //Search for the recipe.
        System.Diagnostics.Process.Start(
            "http://search.msn.com/results.aspx?q=" + recipeName);
            //null);
    }
}
```

СОЗДАНИЕ МНОГООКОННЫХ ПРИЛОЖЕНИЙ. MDI ПРИЛОЖЕНИЯ. ЭЛЕМЕНТ УПРАВЛЕНИЯ NOTIFYICON

1. Диалоговые модальные SDI-окна

Новая форма добавляется в проект, как и любой другой элемент. В случае создания многооконного приложения, дочерних форм может быть несколько. Каждая форма может содержать свои ЭУ.

При создании формы события происходят в следующей последовательности:

- конструктор – форма создается;
- Load – форма существует, но не отображена;
- Activated – форма становится видимой и текущей. (Идет серия сообщений! Использовать не удалось.)

Исключение. Для Visible=true (Show с Visible=true):

- конструктор до Visible=true – форма создается, отображается и становится текущей;
- Load
- Activated
- конструктор после Visible=true – форма создается;

При закрытии формы события происходят в следующей последовательности:

- Closing – попытка закрыть форму
- (CancelEventArgs e.Cancel=true – оставить форму открытой, false – закрыть);
- Closed – после закрытия формы;
- Deactivate – после закрытия формы

Модальные диалоговые окна создаются методом ShowDialog().

Родительская форма блокируется до завершения работы с дочерней.

Завершить работу дочерняя форма может либо закрыв себя методом Close(), либо сделав себя невидимой (Visible=false). В последнем случае дочерняя форма остается открытой, но управление возвратится в родительскую форму, которая может выполнить Visible=true и опять сделать дочернюю форму видимой. Открытую дочернюю форму frm2 может закрыть родительская форма - frm2.Cancel().

```
private void Form1_Load(object sender, EventArgs e)
{
   Visible = true;  //принудительно высветить родительскую форму
   Enabled = false; //погасить все поля
   this.BackColor = System.Drawing.SystemColors.ControlDark;

   Form2 dialog = new Form2();
   dialog.ShowDialog();

   Enabled = true;
   this.BackColor = System.Drawing.SystemColors.Control;
   . . .
```

Для получения значения из дочернего окна можно установить свойство DialogResult кнопок в соответствующее значение: OK, Cancel и др. Выбор этих кнопок завершает работу с окном, но оставляет ее открытой. Метод ShowDialog() возвращает код нажатой клавиши в виде перечисления DialogResult.

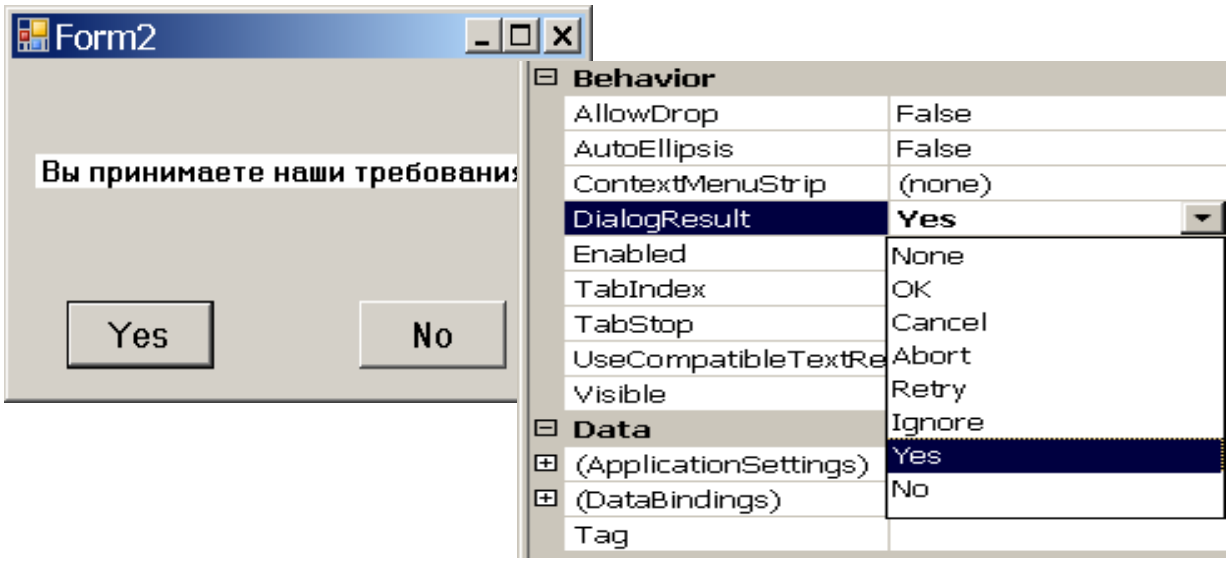


рис.1

Пример получения информации о завершении работы дочерней формы:

```
private void button3_Click(object sender, EventArgs e)
{
   Form2 dialog = new Form2();

   DialogResult result = dialog.ShowDialog();

   // здесь доступны свойства формы dialog
   // например, string str = dialog.val1;

   dialog. Close();
   switch (result)
   {
      case DialogResult.Yes:

         MessageBox.Show("Выбрано ДА");
         break;

      case DialogResult.No:

         MessageBox.Show("Выбрано НЕТ");
         break;
   }
}
```

Не зависимо от того закрыли ли мы дочернюю форму или нет, при выходе из метода, ее создавшего, ссылка на дочернюю форму выйдет из области видимости и будет уничтожена сборщиком мусора. Пока ссылка

не уничтожена, свойства дочерней формы (как и другие открытые члены) остаются доступными в родительской форме.

Так как ShowDialog() не создает новых экземпляров, то его можно выдавать повторно: все старые значения формы будут видны.

Пример получения информации от дочерней формы.

Необходимо в форму добавить свойства для возврата передаваемых значений. Пока управление не покинет обработчик, который создал дочернее окно, все открытые члены дочерней формы доступны.

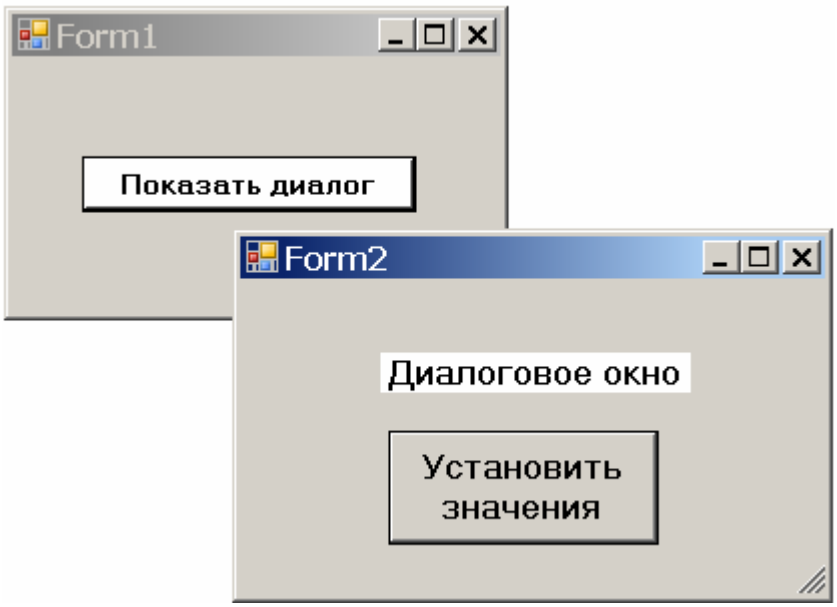


рис.2

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Text;
using System.Windows.Forms;

public partial class Form1 : Form
{
   public Form1()
   { InitializeComponent();  }

   private void button3_Click(object sender, EventArgs e)
   {
      Form2 dialog = new Form2();

      // здесь можно передать значения форме 2 через ее свойства

      dialog.ShowDialog();

      int v = dialog.Val1;
      MessageBox.Show ("Значение 1 = " + v);
      MessageBox.Show ("Значение 2 = " + dialog.Val2);
```

```
    }
}

public partial class Form2 : Form
{
   int val1, val2;

   public Form2()
   {
      InitializeComponent();
   }

   public int Val1
   {   get { return val1; }
       set { val1 = value; }
   }

   public int Val2
   {   get { return val2; }
       set { val2 = value; }
   }

   private void button1_Click(object sender, EventArgs e)
   {
      Val1 = 12;
      Val2 = 145;
      Close();
   }
}
```

Передача параметров в дочернюю форму с ее отображением с помощью ее же метода.

В вызываемой форме:

```
// формирование динамического массива
      . . .
// создание формы2 и вызов ее метода для ее визуализации
// с передачей параметра.
Form2 f2 = new Form2();
  f2.AddLab(array1);
        . . .
```

Форма 2:

```
public partial class Form2 : Form
   {
      private ArrayList   arrayList1;

      public Form2()
      {
         InitializeComponent();
      }

      public void AddLab (ArrayList  arList)
```

```
        {
          arrayList1= arList;          // получить доступ к массиву Формы 1

          this.ShowDialog();           // отображение Формы 2
        }

        private void button3_Click(object sender, EventArgs e)
        {
         . . . . . .
          this.Close();            // закрыть форму 2
        }
    }
}
```

2. Диалоговые НЕмодальные окна

Создание немодальных диалогов с передачей и возвратом параметров.

Окно немодального диалога создается методом Show().

Особенности:

1. Управление после создания формы немедленно передается на следующий оператор после Show().

2. Родительская форма продолжает получать сообщения.

3. Родительская форма не получает никакого уведомления о том, что дочерняя форма закрыта.

Статическое свойство OpenForms класса Application возвращает коллекцию открытых форм, из которой можно выбрать ссылку на требуемую форму:

```
Form1 f1 = (Form1)Application.OpenForms["Form1"];
```

Первый вариант.

Создаем немодальные окна. Управляем доступностью форм.

Все формы остаются видимыми на экране, но доступ к ним либо разрешается, либо запрещается.

Для запрещения создания других форм-дубликатов, кроме первой Form2, в Form1 используем свойство this.Enabled = false. При выходе из формы Form2 (события Form2_FormClosing или Form2_FormClosed) разрешаем (f1.Enabled = true) использование формы Form1.

По кнопке Закрыть или по меню Закрыть закрываем форму и делает ее невидимой.

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Text;
using System.Windows.Forms;
```

```
namespace WindowsApplication1
{
   public partial class Form1 : Form
   {
      public Form1()
      {
         InitializeComponent();
      }

      private void button1_Click (object sender, EventArgs e)
      {
         Form2 f2 = new Form2();
         this.Enabled = false;
         f2.Show();
      }
   }
}


using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Text;
using System.Windows.Forms;

namespace WindowsApplication1
{
   public partial class Form2 : Form
   {
      public Form2()
      {
         InitializeComponent();
      }


      private void Form2_FormClosing (object sender,
                                                    FormClosingEventArgs e)
      {
         // так как форма Form1 не закрыта, она доступна
         Form1 f1 = (Form1)Application.OpenForms["Form1"];
         f1.Enabled = true;
      }
   }
}
```

Второй вариант.

Создаем немодальные окна. Управляем видимостью форм.

Перед вызовом Form2, в Form1 устанавливаем ее невидимость: this.Visible = false. При выходе из формы Form2 (события Form2_FormClosing или Form2_FormClosed) устанавливаем видимость (f1.Visible = true) формы Form1.

Форма Form2 становится невидимой при выходе из формы по кнопке Закрыть или по меню Закрыть.

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Text;
using System.Windows.Forms;

namespace WindowsApplication1
{
   public partial class Form1 : Form
   {
      public Form1()
      {
         InitializeComponent();
      }

      private void button1_Click (object sender, EventArgs e)
      {
         Form2 f2 = new Form2();
         this.Visible = false;
         f2.Show();
      }
   }
}


using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Text;
using System.Windows.Forms;

namespace WindowsApplication1
{
   public partial class Form2 : Form
   {
      public Form2()
      { InitializeComponent(); }


      private void Form2_FormClosing (object s,
                                                    FormClosingEventArgs e)
      {
         Form1 f1 = (Form1)Application.OpenForms["Form1"];
         f1.Visible = true;
      }
   }
}
```

Способ использования данных из других форм

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Text;
using System.Windows.Forms;

namespace WindowsApplication1
{
   public partial class Form1 : Form
   {

      public Form1()
      { InitializeComponent(); }

      public string strF1 = "форма1 ";
      Form2   f2 = null;

      private void button1_Click(object sender, EventArgs e)
      {
         this.Enabled = false;
         f2 = new Form2();

         f2.Show();
        // show=true;
      }


      // Нельзя щелкать на второй кнопке раньше первой,
      // так как f2 будет = null
      private void button2_Click(object sender, EventArgs e)
      {
         // Здесь д.б. проверка на то, что форма 2 отработала: if (show) ...
         // Используем строку из формы 2,
         // так как объект f2 не уничтожен.
         MessageBox.Show(f2.strF2);
      }
   }
}


using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Text;
using System.Windows.Forms;

namespace WindowsApplication1
{
```

```
    public partial class Form2 : Form
    {

        public Form2()
        { InitializeComponent(); }

        public string strF2 = "форма2 ";


        private void Form2_FormClosed(object sender,
                                              FormClosedEventArgs e)
        {
            strF2 = "Изменили строку в Форме 2";

            // Используем строку из формы 1,
            // так как объект f1 не уничтожен.
            Form1 f1 = (Form1)Application.OpenForms["Form1"];
            MessageBox.Show (f1.strF1);
            f1.Enabled = true;
        }
    }
}
```

3. MDI приложения

MDI приложения позволяют приложению отображать сразу несколько документов одновременно, каждый в своем окне. Такие окна являются окнами немодальных диалогов.

В родительском окне необходимо установить свойство IsMdiContainer в true.

Как правило, у родительского окна создают пункт меню «Окно» со свойством MdiList=true. В этом случае в пункт «Окно» будут автоматически добавляться подпункты, образующие список всех открытых окон.

Новая форма добавляется в проект, как и любой другой элемент.

Для MDI приложения конструируется, как правило, только одна дочерняя форма, на основе которой создается несколько дочерних окон.

Для того чтобы создаваемое окно отображалось как дочерняя форма приложения, необходимо установить его свойство MdiParent равным this.

Окно создается в обработчике события меню методом Show().

Пример:

```
private void menuNew_Click(object sender, EventArgs e)
{
    Form2   wnd = new Form2();
    wnd.MdiParent = this;
    wnd.Show();
}
```

Дочерние окна не закрывают родительские элементы управления, поэтому эти ЭУ всегда доступны.

4. Выполнение фоновых приложений. Элемент управления NotifyIcon

Элемент управления NotifyIcon отображает значок в области уведомлений (SystemTray) панели задач, соответствующий приложению, выполняемому в фоновом режиме.

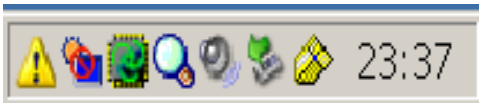


рис.3

Создайте проект SystemTray. Перетащите на форму из окна ToolBox элементы управления ContextMenu и NotifyIcon. Добавьте в контекстное меню пункты "Показать" (имя пункта – menuShow) и "Скрыть" (имя пункта – menuHide).

Установите следующие свойства элемента notifyIcon1:

свойство	Значение
ContextMenu	contextMenu1
Icon	Icon\ eventlogWarn.ico
Text	Задача SystemTray

В результате будет сгенерирован следующий код:

```
private System.Windows.Forms.NotifyIcon notifyIcon1;

notifyIcon1.ContextMenu = contextMenu1;
notifyIcon1.Icon =
 ((System.Drawing.Icon)(resources.GetObject("notifyIcon1.Icon")));
notifyIcon1.Text = "Задача SystemTray ";
notifyIcon1.Visible = true;
notifyIcon1.DoubleClick += new EventHandler(menuShow_Click);
```

Изображение, используемое в качестве иконки (свойство Icon) элемента notifyIcon1, будет выводиться в область уведомлений.

Значение свойства Text представляет собой текст всплывающей подсказки, появляющейся при наведении курсора на иконку приложения.

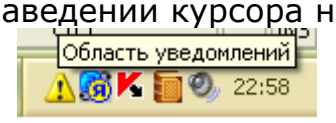


рис.4

В конструкторе формы свойству ShowInTaskbar присваиваем значение false для удаления с панели задач иконки нашего приложения:

```
public class Form1 : Form
{
    InitializeComponent();

        // скрываем видимость приложения на панели задач
    this.ShowInTaskbar = false;
 }
```

Добавляем обработчик пункта меню menuShow:

```
private void  menuShow_Click (object sender, System.EventArgs e)
 {
    // Отобраем приложение на панели задач при запуске
    this.ShowInTaskbar = true;

    //Показываем форму
    this.Show();

    //Отключаем доступность пункта меню menuShow
    menuShow.Enabled = false;

    //Включаем доступность пункта меню menuHide
    menuHide.Enabled = true;
}
```

Обработчик пункта меню menuHide изменяет эти значения на обратные:

```
private void  menuHide_Click (object sender, System.EventArgs e)
 {
    this.ShowInTaskbar = false;
    this.Hide();

    menuShow.Enabled = true;
    menuHide.Enabled  = false;
}
```

В режиме дизайна в окне Properties элемента управления notifyIcon1 переключаемся на события и в поле DoubleClick выбираем из списка обработчик menuShow_Click.

Примечание.

Можно было создать свой обработчик для события DoubleClick, а из него вызвать menuShow_Click():

```
private void notifyIcon1_DoubleClick(object sender, EventArgs e)
{
  menuShow_Click (this, new EventArgs());
}
```

В результате обработчиком события notifyIcon1_DoubleClick будет обработчик menuShow_Click.

Запускаем приложение.

В области уведомлений появляется иконка, связанная с notifyIcon1.

Пункты контекстного меню отображают и удаляют иконку приложения с панели задач.

Так как контекстное меню было установлено для ЭУ notifyIcon1, то двойной щелчок мышки на иконке в области уведомлений выводит пункты "Показать" и "Скрыть":

рис.5

Эти пункты отображают и удаляют иконку приложения с панели задач. При этом окно приложения по пункту "Показать" отображается на экране только в том случае, если перед этим оно не было свернуто на панель задач.

ВВЕДЕНИЕ В ГРАФИКУ

1. Интерфейсы GDI и GDI+

GDI – интерфейс графических устройств. Входит в состав Windows. Обеспечивает независимость Windows от графических устройств.

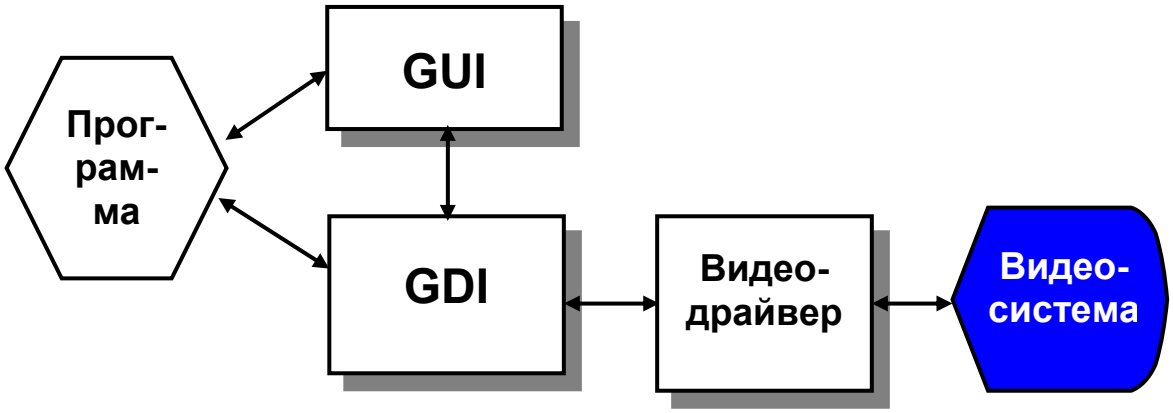


рис.1

GDI+ - объектно-ориентированная подсистема. Состоит из набора базовых классов .NET и опирается на GDI.

Реализована в пространстве имен System.Drawing.

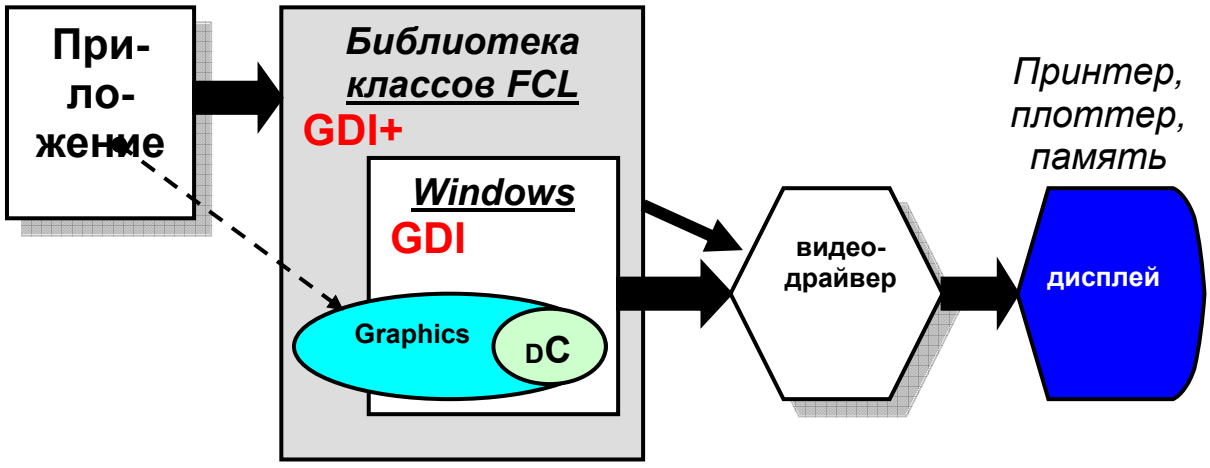


рис.2

2. Контекст устройства (DC) и контекст отображения (Graphics).

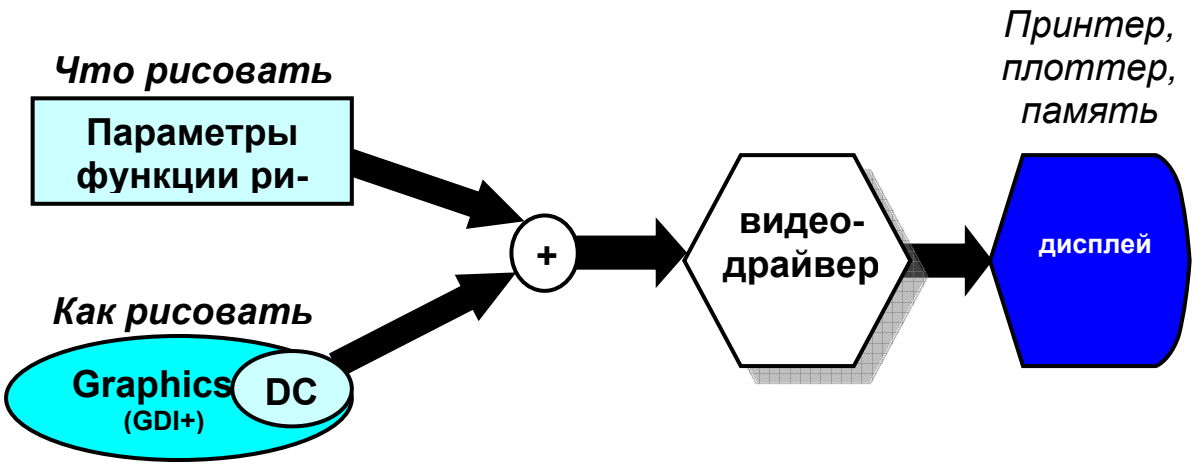


рис.3

Система координат по умолчанию (в пикселях)

Рис. 10.4. Система координат по умолчанию

рис.4

Класс Graphics шире DC. Класс реализует и свойства, и инструменты для рисования в контексте DC.

Пример метода рисования прямоугольника:
g.FillRectangle (Brush, X, У, width, height);

здесь g - объект класса Graphics - контекст отображения.

3. Перья

Перья используются для рисования линий и простейших геометрических фигур и создаются как объекты класса Pen. Вот соответствующие конструкторы:

```
public Pen (Color);
public Pen (Color, float);                  // Цвет + толщина
public Pen (Brush);                         // На основе кисти
public Pen (Brush, float);
public Pen (Color.FromArgb(255,155,100) );  // Оранжевое
```

После того как перо создано, программа может использовать его атрибуты при помощи свойств класса Pen. Некоторые из этих свойств перечислены в табл.

Свойства пера

Свойство	Описание
Alignment	Выравнивание пера
Width	Ширина линии
Brush	Кисть, используемая пером
Color	Цвет пера
DashStyle	Стиль пунктирных и штрихпунктирных линий
DashCup	Вид точек и штрихов пунктирных и штрихпунктирных линий
DashOffset	Расстояние от начала линии до начала штриха
DashPattern	Массив шаблонов для создания произвольных штрихов и пробелов штриховых и штрихпунктирных линий
StartCup EndCup	Стиль концов линий
LineCap	Формы концов линий
LineJoin	Стиль соединения концов двух различных линий
MiterLimit	Предельная толщина в области соединения остроконечных линий

Пример:

```
Color   color;

        color = Color.Black
        pen = new Pen(color, 2);

        color = Color.White
        pen = new Pen(color, 4);
```

4. Кисти

Внутренняя область окна и замкнутых геометрических фигур может быть закрашена при помощи кисти. Кисти создаются на базе классов, производных от абстрактного класса Brush. Это следующие классы:

- Brushes;
- SolidBrush;
- HatchBrush;
- TextureBrush;
- LinearGradientBrush;

- PathGradientBrush.

Кисть стандартного цвета.

```
Brush brB = Brushes.Blue;
Brush brR = Brushes.Red;
```

Кисть для сплошной закраски:

```
Brush solidBaige = new SolidBrush(Color.Baige);
Brush solidOrangy = new SolidBrush(Color.FromArgb(255,155,100) );  // Оранжевая
```

Кисти типа HatchBrush - прямоугольная кисть
заданного стиля с заданным цветом изображения и фона.

Кисти типа TextureBrush – текстурная кисть,
может иметь любой внешний вид и любой цвет.

Градиентные кисти - обеспечивают плавное изменение интенсивности цвета.

Члены класса Brushes.
Все перечисленные в таблице открытые свойства возвращают объект Brush, определенный системой

AliceBlue	GhostWhite	NavajoWhite
AntiqueWhite	Gold	Navy
Aqua	Goldenrod	OldLace
Aquamarine	Gray	Olive
Azure	Green	OliveDrab
Beige	GreenYellow	Orange
Bisque	Honeydew	OrangeRed
Black	HotPink	Orchid
BlanchedAlmond	IndianRed	PaleGoldenrod
Blue	Indigo	PaleGreen
BlueViolet	Ivory	PaleTurquoise
Brown	Khaki	PaleVioletRed
BurlyWood	Lavender	PapayaWhip
CadetBlue	LavenderBlush	PeachPuff
Chartreuse	LawnGreen	Peru
Chocolate	LemonChiffon	Pink
Coral	LightBlue	Plum
CornflowerBlue	LightCoral	PowderBlue
Cornsilk	LightCyan	Purple
Crimson	LightGoldenrodYellow	Red
Cyan	LightGray	RosyBrown
DarkBlue	LightGreen	RoyalBlue
DarkCyan	LightPink	SaddleBrown
DarkGoldenrod	LightSalmon	Salmon

DarkGray	LightSeaGreen	SandyBrown
DarkGreen	LightSkyBlue	SeaGreen
DarkKhaki	LightSlateGray	SeaShell
DarkMagenta	LightSteelBlue	Sienna
DarkOliveGreen	LightYellow	Silver
DarkOrange	Lime	SkyBlue
DarkOrchid	LimeGreen	SlateBlue
DarkRed	Linen	SlateGray
DarkSalmon	Magenta	Snow
DarkSeaGreen	Maroon	SpringGreen
DarkSlateBlue	MediumAquamarine	SteelBlue
DarkSlateGray	MediumBlue	Tan
DarkTurquoise	MediumOrchid	Teal
DarkViolet	MediumPurple	Thistle
DeepPink	MediumSeaGreen	Tomato
DeepSkyBlue	MediumSlateBlue	Transparent
DimGray	MediumSpringGreen	Turquoise
DodgerBlue	MediumTurquoise	Violet
Firebrick	MediumVioletRed	Wheat
FloralWhite	MidnightBlue	White
ForestGreen	MintCream	WhiteSmoke
Fuchsia	MistyRose	Yellow
Gainsboro	Moccasin	YellowGreen

5. Создание контекста отображения

Контекст отображения – это объект класса Graphics.

1. Для рабочей области:

```
Graphics g = Graphics.FromHwnd (this.Handle);
```

или

```
Graphics g =this.CreateGraphics();
```

this.CreateGraphics() – возвращает объект класса Graphics (контекст отображения).

2. Для элемента управления:

```
Graphics g = Graphics.FromHwnd (<Эл.управления>.Handle);
```

или

```
Graphics g =<Эл.управления>.CreateGraphics();
```

Пример. `Graphics g = panel1.CreateGraphics();`

В обработчике событий

```
Form1_Paint  (object sender, Forms.PaintEventArgs e)

Graphics g  =  e.Graphics;
```

Примечание. Такой контекст устройства делает доступным для перерисовки только реально «испорченную» часть окна.

Если запрограммирована перерисовка всего окна, то контекст устройства надо создавать первым или вторым способом.

Пример 1. Рисование траектории перемещения курсора при нажатой кнопке мыши.

```
//  Move_Veer – первая программа с графикой.
// Программа рисует траекторию перемещения курсора мыши при
// нажатой клавише и перестает рисовать траекторию при
// отпускании клавиши.
// Способ рисования: при перемещении курсора поступает
// сообщение Move с координатами курсора,
// в которые выводится квадрат со стороной 4 пикселя.

using System;
using System.Drawing;
using System.Windows.Forms;

public class Form1 : Forms.Form
{
   private System.ComponentModel.Container components = null;
   public Form1() { InitializeComponent(); }
   protected override void Dispose( bool disposing )
   {        if( disposing )
                    if (components != null) { components.Dispose(); }
            base.Dispose( disposing );
   }

+  Windows Form Designer generation code

   static void Main()
   { Application.Run(new Form1()); }

      bool  doDraw = false;                  // клавиша мыши отпущена

   private void Form1_MouseDown (object sender, MouseEventArgs e)
   {  doDraw = true;  }              // клавиша нажата


   private void Form1_MouseUp (object sender, MouseEventArgs e)
   {  doDraw = false; }              // клавиша отпущена


   private void Form1_MouseMove (object sender, MouseEventArgs e)
      //  e – объект, определяющий координаты курсора  (e.X, e.Y)
   {
      if (doDraw)
      {
            Graphics g = this.CreateGraphics();

                  /* Создать объект  redBrush – сплошная кисть красного
                        цвета. Color – структура System.Drawing.Color */
```

```
                SolidBrush  redBrush = new SolidBrush (Color.Red);

               /* Вывод закрашенного прямоугольника в координатах e.X, e.Y
                  шириной 4 и высотой 4 пикселя с использ. кисти redBrush  */
                  g.FillRectangle (redBrush, e.X, e.Y, 4, 4);
                  g.Dispose();
            }
      }
}
```

6. Добавление обработчиков событий

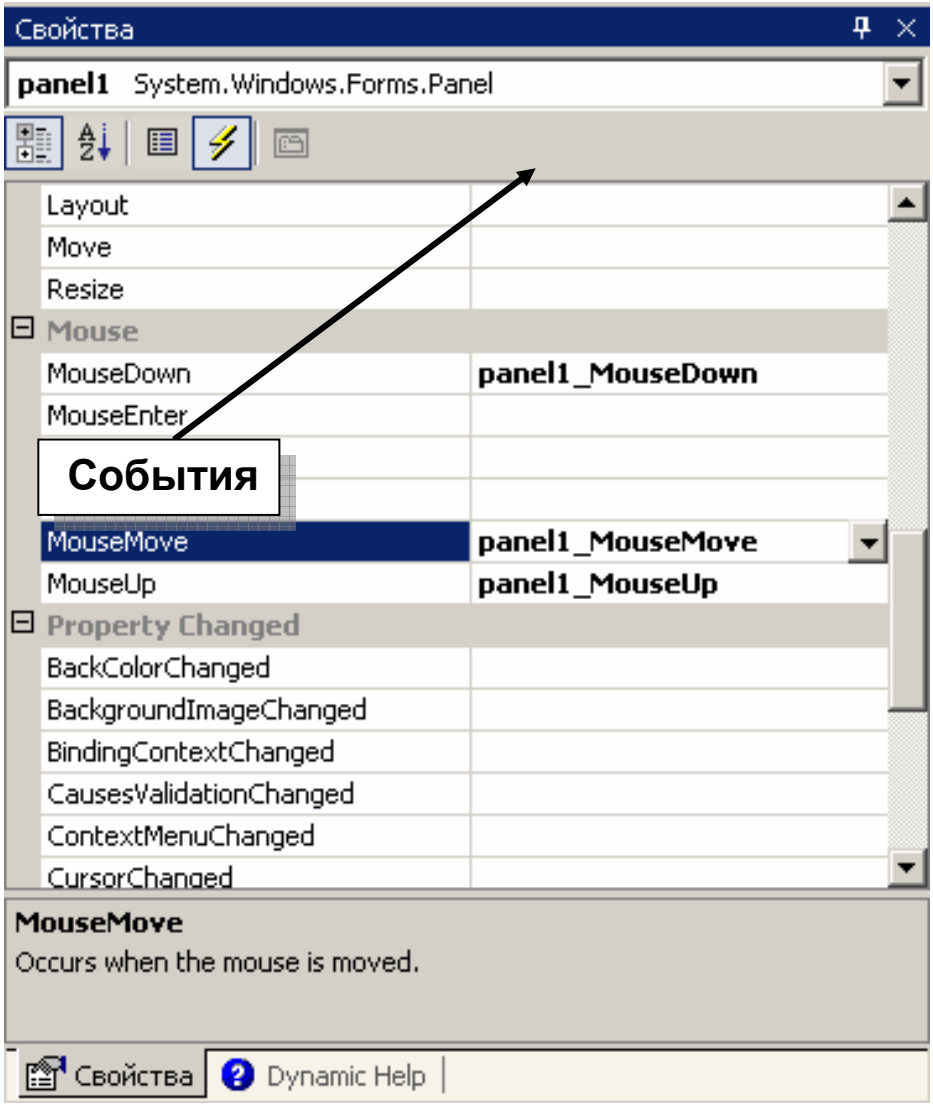


рис.5

7. Рисование в окне элемента управления

Необходимо получить контекст отображения для окна элемента управления (например, панели)

/* Graphics_Panel – вторая программа с графикой .

Программа рисует в панели траекторию перемещения курсора мыши при нажатой клавише и перестает рисовать траекторию при отпускании клавиши.

Способ рисования: при перемещении курсора в окне панели поступает сообщение Move с координатами курсора, в которые выводится круг диаметром 10 пикселей.

Отличия от программы Graphics_Move показаны жирным шрифтом.

```
*/
using System;
using System.Drawing;
using System.Windows.Forms;

public class Form1 : Forms.Form
{
   private System.ComponentModel.Container components = null;

   public Form1() { InitializeComponent(); }

   protected override void Dispose( bool disposing )
   {        if( disposing )
                    if (components != null) { components.Dispose(); }
            base.Dispose( disposing );
   }

+  Windows Form Designer generation code

   static void Main()
   { Application.Run(new Form1()); }

    bool  doDraw = false;          // клавиша мыши отпущена

   private void  panel1_MouseDown(object sender, MouseEventArgs e)
   {  doDraw = true;  }            // клавиша нажата

   private void  panel1_MouseUp (object sender, MouseEventArgs e)
   {  doDraw = false; }            // клавиша отпущена


   private void panel1_MouseMove(object sender, MouseEventArgs e)
   {
       if (doDraw)
     {
          Graphics g = panel1.CreateGraphics();
             SolidBrush  redBrush = new SolidBrush (Color.Red);

               /* Вывод закрашенного эллипса в координатах e.X, e.Y
                     шириной 10 и высотой 10 пикселей с использ. кисти redBrush */
               g.FillEllipse (redBrush, e.X, e.Y, 10, 10);
               g.Dispose();
      }
     }
}
```

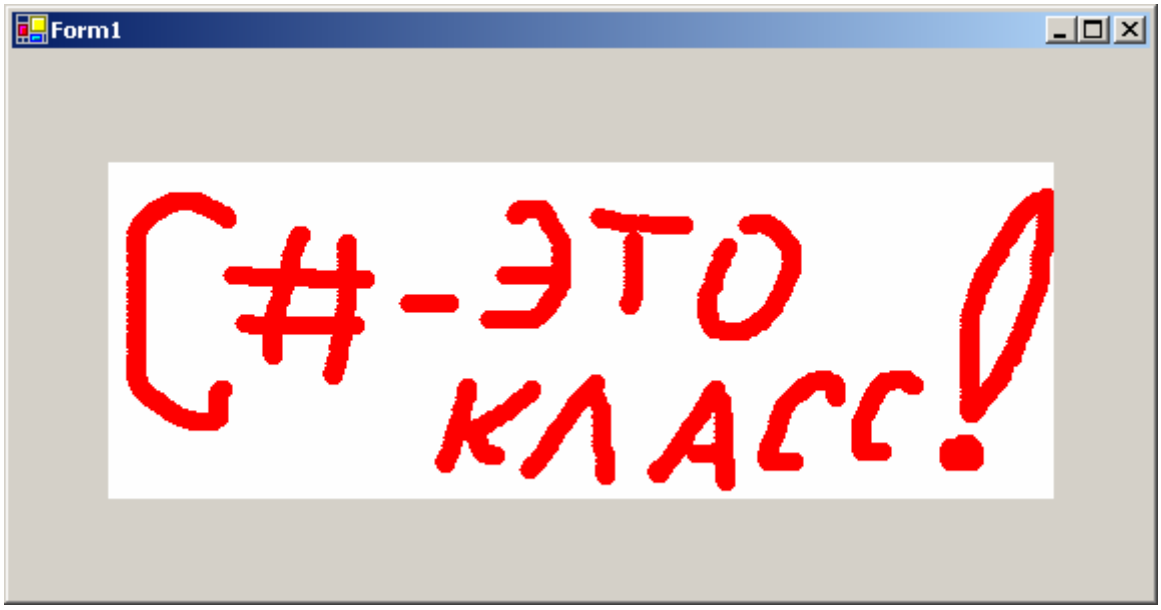


рис.6

Событие Paint

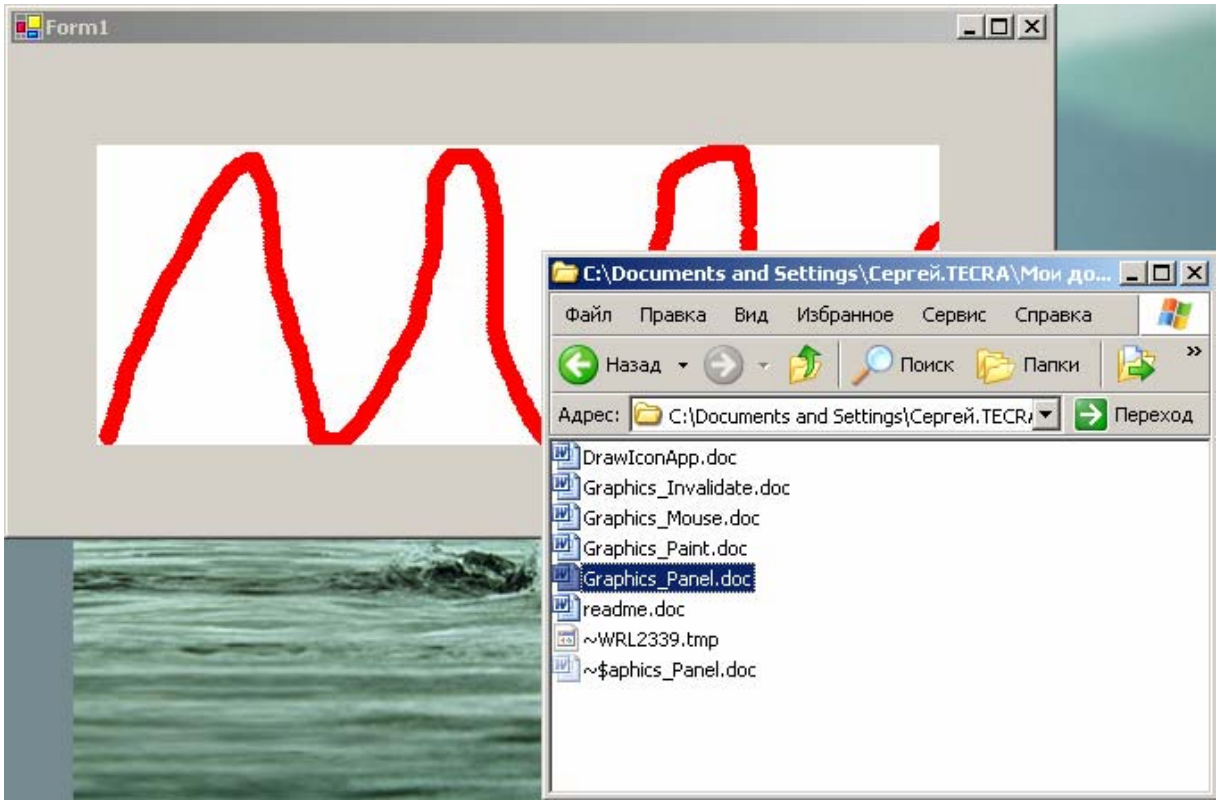


рис.7

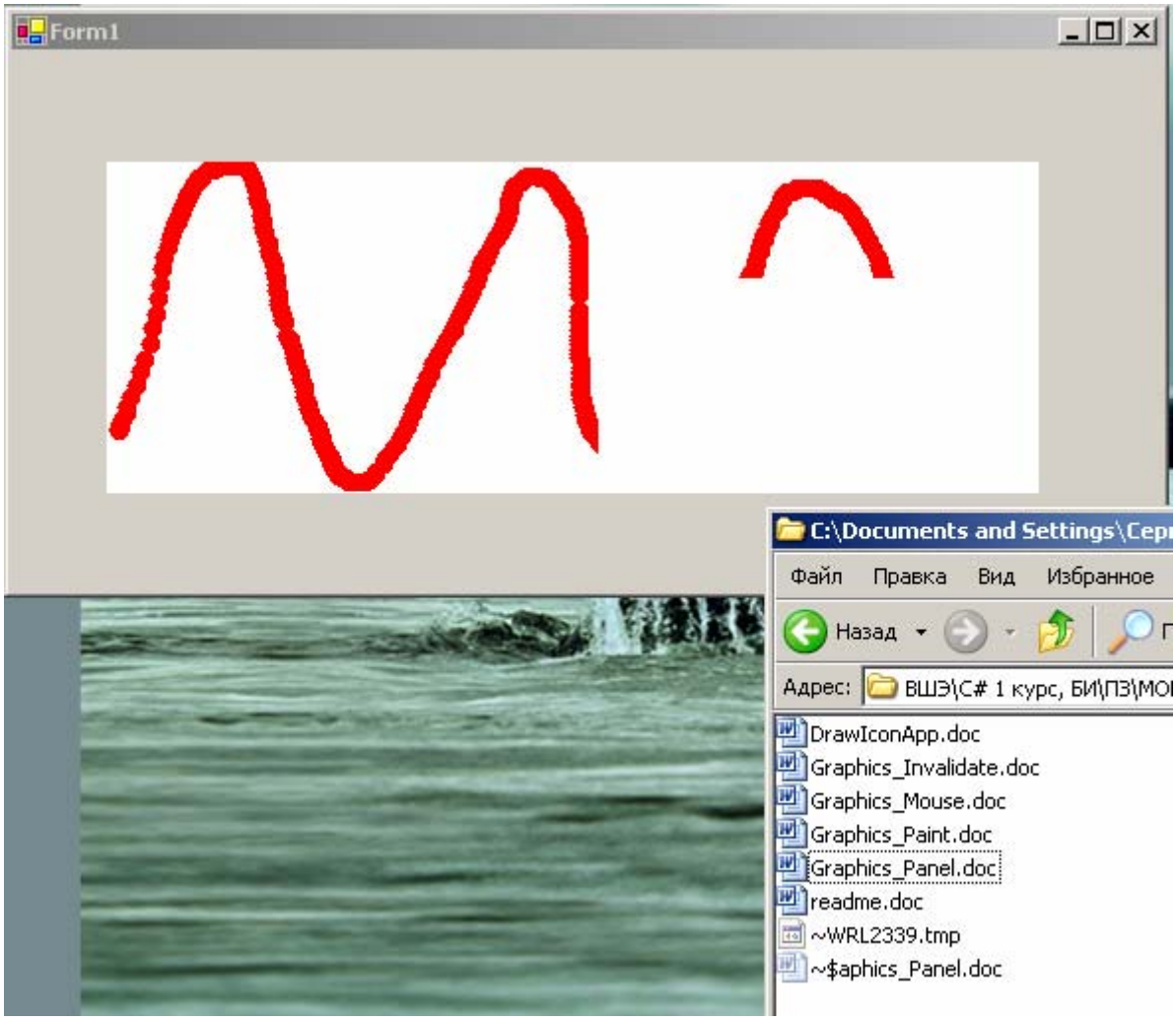


рис.8

Когда вся клиентская область окна формы или часть этой области требует перерисовки, форме передается событие Paint.

Элементы управления выполняют обработку события Paint сами, перерисовывая при необходимости свои окна.

```
// Демонстрация перерисовки содержимого экрана
// Программа выводит строку текста, прямоугольник и эллипс при
// обработке сообщения Paint.

using System;
using System.Drawing;
using System.Windows.Forms;

public class Form1 : Forms.Form
{
    private System.ComponentModel.Container components = null;

    public Form1() { InitializeComponent(); }

    protected override void Dispose( bool disposing )
    {
            if( disposing )
```

```
                    if (components != null)
                        components.Dispose();
                base.Dispose( disposing );
    }

    +  Windows Form Designer generation code

       static void Main()
       {  Application.Run(new Form1());  }



       public string text = "Обработка события Paint";      // моя строка

       private void  Form1_Paint  (object sender, Forms.PaintEventArgs e)
       {
         Graphics g  =  e.Graphics;   /*  e – ссылка на объект, имеющий
                     свойства:  - Graphics – контекст устройства
                                - ClipRectangle – границы области перерисовки
                                                                          */
         g.Clear (Color.White);        // Закрасить окно белым цветом

                 /* Вывод строки text  с использ. объекта Font  и свойства Black
                       статич. класса  System.Drawing.Brushes, в координатах 0,0 */
         g.DrawString (text, new Font(”Arial”, 15), Brushes.Black, 0, 0);

               /* Вывод незакрашенного прямоугольника в координатах
                 10,30  шириной 200 и высотой 100 пикселей с использ.
                 пера - объекта Pen, инициализированного цветом
                 Brushes.Green с шириной линий 2 пикселя */
         g.DrawRectangle (new Pen(Brushes.Green,2), 10, 30, 200, 100);

                          // Вывод незакрашенного эллипса
         g.DrawEllipse (new Pen(Brushes.Red,8), 150, 120, 100, 130);
       }
    }
```

1.
Graphics g = e.Graphics; e определяет только часть окна для перерисовки.

2.
Graphics g = Graphics.FromHwnd (this.Handle); e определяет для перерисовывания все окно.

8. Генерация сообщения Paint и перерисовка содержимого окна

Можно сгенерировать собщение Paint вручную с помощью метода
окно.Invalidate();
При этом произойдет очистка окна.

```
// Демонстрация  генерации сообщения Paint
// и перерисовки содержимого окна.
// Программа запоминает координаты курсора в момент  щелчка
// клавиши мыши и генерирует сообщение Paint.
// При обработке  сообщения Paint  выводится прямоугольник в
// запомненных координатах.

using System;
using System.Drawing;
using System.Windows.Forms;

public class Form1 : Forms.Form
{
    private System.ComponentModel.Container components = null;

    public Form1() { InitializeComponent(); }

    protected override void Dispose( bool disposing )
    {
            if( disposing )
                    if (components != null)
                     components.Dispose();
            base.Dispose( disposing );
    }

+   Windows Form Designer generation code

    static void Main()
    {  Application.Run(new Form1());  }

     int  X, Y;        // координаты курсора мыши

    void  Form1_MouseDown (object sender, MouseEventArgs e)
    {
            X = e.X;  Y = e.Y;
            Invalidate();              // генерация Paint
    }

    private void  Form1_Paint  (object sender, Forms.PaintEventArgs e)
    {
       if ( X > 0 & Y > 0 )
             g.DrawRectangle (new  Pen (Brushes.Green,2),  X, Y, 200, 100);
    }
}
```

9. Методы и свойства класса Graphics

Очистка окна. Пример: закрасить окно белым цветом

```
g.Clear (Color.White);
```

Рисование геометрических фигур

Имена большого количества методов, определенных в классе Graphics, начинаются с префиксов Draw и Fill.

Первые из них предназначены для рисования текста, линий и незакрашенных фигур (таких, например, как прямоугольные рамки),
а вторые - для рисования закрашенных геометрических фигур.

Линия

```
public void DrawLine (Pen, Point, Point);
public void DrawLine (Pen, PointF, PoinF);
public void DrawLine (Pen, int, int, int, int);
public void DrawLine (Pen, float, float, float, float);
```

Для получения такого окна добавьте в обработку события Paint код, приведенный ниже:

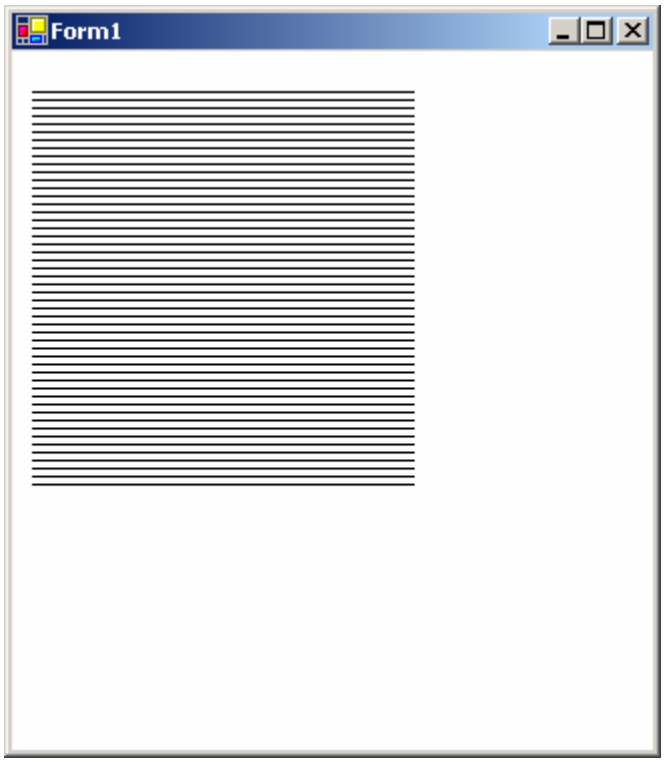


рис.9

```
private void Form1_Paint(object sender, PaintEventArgs e)
{
    Graphics g = e.Graphics ;
    g.Clear (Color.White) ;

    Pen  pen = new Pen (Brushes.Black, 2);

    for (int i=0; i<50; i++)
            g.DrawLine (pen, 10, 4*i+20, 200, 4*i+20) ;
}
```

Набор линий

public void DrawLines (Pen, Point[ ]);
public void DrawLines (Pen, PointF[ ]);

Пример вывода соединяющихся линий.

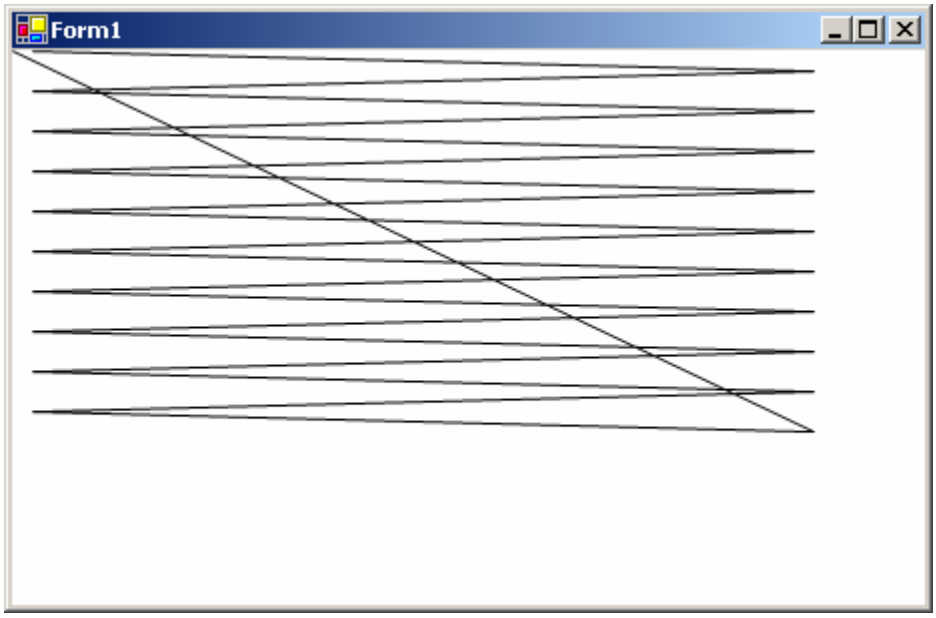


рис.10

```
using System.Drawing.Drawing2D;

//  Добавить в конструктор
            Point[] points = new Point[50];

            for (int i=0; i < 20; i++)
            {
                  int xPos;
                  if (i%2 == 0)
                        xPos=10;
                  else
                        xPos=400;

                  points[i] = new Point(xPos, 10 * i);
            }

            Pen pen = new Pen(Brushes.Black, 2);
. . . . . .

            private void Form1_Paint(object sender, PaintEventArgs e)
            {
                  Graphics g = e.Graphics ;
                  g.Clear (Color .White) ;

                  // Высококачественное сглаживание
                  g.SmoothingMode = SmoothingMode.HighQuality;
                g.DrawLines (pen, points);
```

}

Незакрашенный прямоугольник

Метод DrawRectangle позволяет рисовать прямоугольники, заданные координатами верхнего левого угла, а также шириной и высотой.

```
public void  DrawRectangle (Pen, Rectangle);
public void  DrawRectangle (Pen, int, int, int, int);
public void  DrawRectangle (Pen, float, float, float, float);
```

Класс Rectangle имеет свойства:
X и Y,
Width и Height.

Набор незакрашенных прямоугольников

```
public void DrawRectangles(Pen, Rectangle[]);
public void DrawRectangles(Pen, RectangleF[]);
```

Незакрашенный многоугольник

```
public void DrawPolygon (Pen, Point []) ;
public void DrawPolygon (Pen, PointF[]);
```

Незакрашенный эллипс

Эллипс вписывается в прямоугольник.

```
public void DrawEllipse (Pen, Rectangle);
public void DrawEllipse (Pen, RectangleF) ;
public void DrawEllipse (Pen, int, int, int, int);
public void DrawEllipse (Pen, float, float, float, float);
```

Сегмент эллипса

```
public void  DrawArc (Pen, Rectangle, float, float);
public void  DrawArc (Pen, RectangleF, float, float);
public void  DrawArc (Pen, int, int, int, int, int, int);
public   void  DrawArc (Pen, float, float, float, float, float, float);
```

Незакрашенный замкнутый сегмент эллипса

```
public void  DrawPie (Pen,        Rectangle, float, float);
public void  DrawPie (Pen,        RectangleF, float, float);
public void  DrawPie (Pen,        int, int, int, int, int, int);
public   void  DrawPie (Pen, float, float, float, float, float, float);
```

Кривые Безье

Кривая проходит через 4 точки: нач., кон. и 2 управл.

```
public void DrawBezier (Pen,     Point, Point, Point, Point);
```

Еще 4 метода.

Канонические сплайны

Кривая проходит через все точки.

```
public void DrawCurve (Pen, Point []) ;
```

Еще несколько методов.

Закрашенные фигуры

Префикс Fill:

Метод	Описание
FillRectangle	Рисование закрашенного прямоугольника
FillRectangles	Рисование множества закрашенных многоугольников
FillPolygon	Рисование закрашенного многоугольника
FillEllipse	Рисование закрашенного эллипса
FillPie	Рисование закрашенного сегмента эллипса
FillClosedCurve	Рисование закрашенного сплайна
FillRegion	Рисование закрашенной области типа Region

Вместо пера используется кисть:

```
Brush br = Brushes.Red;
g.FillRectangle (br, e.X, e.Y, 4, 4);
```

10. Ресурсы приложения

Приложение Microsoft Windows может хранить в виде ресурсов текстовые строки, значки, курсоры, графические изображения, меню, диалоговые окна, произвольные массивы данных и т. д.

Физически ресурсы находятся внутри exe-файла приложения.

Они могут загружаться в оперативную память автоматически при запуске приложения или по запросу приложения (явному или неявному).

В приложениях Microsoft .NET Framework тоже используется концепция ресурсов. В частности, ресурсы таких приложений могут содержать значки и графические изображения.

Данные из исходных файлов, содержащих значки или изображения, будут переписаны в файл сборки приложения. Благодаря этому, Вы сможете поставлять приложение как единый загрузочный файл.

Методы DrawIcon – рисование значка:

- С растяжением/сжатием
  ```
  public void  DrawIcon (Icon, Rectangle) ;
  ```
- Без растяжения/сжатия
  ```
  public void  DrawIcon (Icon, int, int) ;
  ```
- Без растяжения/сжатия. Значок обрезается.
  ```
  public void  DrawIconUnstretched(Icon, rect);
  ```

Пример.

1. Добавьте в проект файл со значком.

2. Установите у этого файла свойство Build Action, равное Embedded Resource (встраиваемый ресурс).

3. Создайте обработчик события Form1_MouseDown.

Поэкспериментируйте с размерами области отображения.

```
Rectangle rect;
private void Form1_MouseDown(object sender, MouseEventArgs e)
{
   Graphics g = Graphics.FromHwnd(Handle);

   /* Создать объект-иконку. Конструктору передается:
                        1) местоположение иконки в виде метода
                                    GetType() - ссылка на объект-сборку
                        2) имя иконки.  */
   Icon myIcon = new Icon(GetType(), "myIcon.ico");

   // нарисовать иконку
   rect.X = e.X; rect.Y = e.Y; rect.Width = 50; rect.Height = 50;

   //g.DrawIcon(myIcon, rect);

   // нарисовать иконку без растяжения фиксированного размера
   g.DrawIcon(myIcon, e.X+100, e.Y+100);

   g.DrawIconUnstretched(myIcon, rect);         // Как есть

}
```

Пример (для продвинутых).

В следующем примере показано, как, используя разные события мыши, изобразить путь указателя мыши на панеле.

Для каждого из происходящих событий MouseMove и MouseDown в GraphicsPath добавляется сегмент линии.

Чтобы обновлять рисунок, при каждом событии MouseDown или MouseUp для Panel вызывается метод Invalidate.

Кроме того, в случае события MouseWheel (вил) (колесо) графический путь прокручивается вверх или вниз.

На экране также отображаются дополнительные события мыши, такие как MouseHover (зависание).

Кроме того, отображаются и дополнительные сведения о мыши, содержащиеся в классе SystemInformation.

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Text;
using System.Windows.Forms;

public partial class Form1 : Form
{
   public Form1()
```

```
{
    mousePath = new System.Drawing.Drawing2D.GraphicsPath();
     InitializeComponent();
     label2.Text = "SystemInformation.DoubleClickSize: " +
        SystemInformation.DoubleClickSize.ToString();
     label3.Text = "SystemInformation.DoubleClickTime: " +
        SystemInformation.DoubleClickTime.ToString();
     label4.Text = "SystemInformation.MousePresent: " +
        SystemInformation.MousePresent.ToString();
     label5.Text = "SystemInformation.MouseButtons: " +
        SystemInformation.MouseButtons.ToString();
     label6.Text = "SystemInformation.MouseButtonsSwapped: " +
        SystemInformation.MouseButtonsSwapped.ToString();
     label7.Text = "SystemInformation.MouseWheelPresent: " +
        SystemInformation.MouseWheelPresent.ToString();
     label8.Text = "SystemInformation.MouseWheelScrollLines: " +
        SystemInformation.MouseWheelScrollLines.ToString();
     label9.Text = "SystemInformation.NativeMouseWheelSupport: " +
        SystemInformation.NativeMouseWheelSupport.ToString();
}

 // GraphicsPath представляет последовательность соединенных линий и кривых.
private System.Drawing.Drawing2D.GraphicsPath mousePath;
private int fontSize = 20;


private void panel1_MouseDown(object sender, MouseEventArgs e)
{
   // Update the mouse path with the mouse information
   Point mouseDownLocation = new Point(e.X, e.Y);

   string eventString = null;
   switch (e.Button)
   {
      case MouseButtons.Left:
         eventString = "L";
         break;
      case MouseButtons.Right:
         eventString = "R";
         break;
      case MouseButtons.Middle:
         eventString = "M";
         break;
      case MouseButtons.XButton1:
         eventString = "X1";
         break;
      case MouseButtons.XButton2:
         eventString = "X2";
         break;
      case MouseButtons.None:
      default:
         break;
   }
```

```
    if (eventString != null)
    {
        mousePath.AddString(eventString, FontFamily.GenericSerif,
                    (int)FontStyle.Bold, fontSize, mouseDownLocation,
                    StringFormat.GenericDefault);
    }
    else
    {
        mousePath.AddLine(mouseDownLocation, mouseDownLocation);
    }
    panel1.Focus();
    panel1.Invalidate();
}

private void panel1_MouseEnter(object sender, EventArgs e)
{
    // Update the mouse event label to indicate the MouseEnter event occurred.
    label1.Text = sender.GetType().ToString() + ": MouseEnter";
}

private void panel1_MouseHover(object sender, EventArgs e)
{
    // Update the mouse event label to indicate the MouseHover event occurred.
    label1.Text = sender.GetType().ToString() + ": MouseHover";
}

private void panel1_MouseLeave(object sender, EventArgs e)
{
    // Update the mouse event label to indicate the MouseLeave event occurred.
    label1.Text = sender.GetType().ToString() + ": MouseLeave";
}

private void panel1_MouseMove(object sender, MouseEventArgs e)
{
    // Update the mouse path that is drawn onto the Panel.
    int mouseX = e.X;
    int mouseY = e.Y;

    mousePath.AddLine(mouseX, mouseY, mouseX, mouseY);
}

private void panel1_MouseUp(object sender, MouseEventArgs e)
{
    Point mouseUpLocation = new System.Drawing.Point(e.X, e.Y);

    // Show the number of clicks in the path graphic.
    int numberOfClicks = e.Clicks;
    mousePath.AddString("    " + numberOfClicks.ToString(),
                FontFamily.GenericSerif, (int)FontStyle.Bold,
                fontSize, mouseUpLocation, StringFormat.GenericDefault);
    panel1.Invalidate();
}

private void panel1_Paint (object sender, PaintEventArgs e)
```

```
    {
      // Perform the painting of the Panel.
      e.Graphics.DrawPath(System.Drawing.Pens.DarkRed, mousePath);
    }

    private void clearButton_Click (object sender, EventArgs e)
    {
      // Clear the Panel display.
      mousePath.Dispose();
      mousePath = new System.Drawing.Drawing2D.GraphicsPath();
      panel1.Invalidate();
    }

    private void panel1_MouseWheel (object sender,
                                        System.Windows.Forms.MouseEventArgs e)
    {
      // Update the drawing based upon the mouse wheel scrolling.

      int numberOfTextLinesToMove = e.Delta *
                    SystemInformation.MouseWheelScrollLines / 120;
      int numberOfPixelsToMove = numberOfTextLinesToMove * fontSize;

      if (numberOfPixelsToMove != 0)
      {
        System.Drawing.Drawing2D.Matrix translateMatrix =
                          new System.Drawing.Drawing2D.Matrix();
        translateMatrix.Translate(0, numberOfPixelsToMove);
        mousePath.Transform(translateMatrix);
      }
      panel1.Invalidate();
    }
  }
```

11. События клавиатуры

События:

- KeyDown – при нажатии
- KeyUP – при отпускании
- KeyPress – удержание, посылается серия событий

Свойства класса KeyEventArgs:

Alt, Control, Shift - true – нажата, false – не нажата.

KeyCode – код нажатой клавиши

KeyData – совокупность кодов нажатых клавиш

KeyValue – десятичное значение свойства KeyData

Handled – флаг, указывающий было ли сообщение обработано. true – дальнейшая обработка нажатия не требуется. По умолчанию – false.