ИНФОРМАТИКА 8. клас
ОБЩООБРАЗОВАТЕЛНА ПОДГОТОВКА

• Красимир Манев •

• Нели Манева •

• Велислава Христова •

издателство
изкуства

София, 2017

Необходимите за практическите занятия материали, задачи за самостоятелна работа, допълнителни упражнения и разработки, помощни изходни файлове можете да намерите на сайта www.izkustva.net. За да получите достъп до електронните ресурси, описани в учебника, изтеглете архива Resources.zip от сайта и го разархивирайте в кореновата папка на устройство C:.

ISBN 978-619-7243-21-5

СЪДЪРЖАНИЕ

I Основи на информатиката

1 Числата и техните представяния

Числата са основен обект на математиката, но и безценен инструмент в ежедневието на човек. Без числата трудно можем да си представим важни човешки дейности като инженерното проектиране, финансовото дело, търговията и т.н. За да можем да извършваме пресмятания с числа, те трябва да бъдат представени по подходящ начин. Една съвкупност от правила за представяне на естествените числа наричаме ***бройна система***, тъй като с естествените числа означаваме броя на някакви обекти – 0, 1, 2, ..., 397, и т.н. Бройните системи делим на ***непозиционни*** и ***позиционни***. Обичайната бройна система днес е ***десетичната позиционна бройна система***. Затова в примерите по-долу използваме представянето на числата в тази система.

Непозиционни бройни системи

Първите бройни системи са непозиционни. В непозиционните бройни системи стойността на всеки използван за представянето ***знак*** (по-късно знаците, използвани в представянето са наречени ***цифри***) е постоянна и не зависи по никакъв начин от нейното място в числото. Така, при представянето на числа с пръстите и дланите на ръцете всеки пръст е имал стойността на десетичното число 1, дланта на едната ръка – стойността на десетичното число 5 и т.н.

Друг пример за такава бройна система е ***римската***. В нея цифрата I има стойност 1, цифрата V – стойност 5, цифрата X – стойност 10, и т.н. Числото 12 се представя в римската бройна система като XII, защото XII = X + I + I = 10 + 1 + 1 = 12. В римската система, когато цифра с по-малка стойност се постави преди цифра с по-голяма стойност, тогава по-малката се изважда от по-голямата, а не се прибавя. Затова, например, IV = –1 + 5 = 4.

Извършването на аритметични действия с числа, представени в непозиционна бройна система, трябва да е било изключително трудно. Затова те не са просъществували твърде дълго и са отхвърлени от времето.

Позиционни бройни системи

Бройните системи са ***позиционни***, когато стойността на цифрата зависи от това, на коя позиция се намира тя в даденото число. Всяка позиционна бройна система се определя от ***основата*** си – цяло число p, $p > 1$. За представяне на число в ***p-ична бройна система*** са ни необходими p цифри. В десетичната позиционна бройна система това са знаците 0, 1, 2, 3, 4, 5, 6, 7, 8 и 9. Числото се записва в няколко ***позиции***, като във всяка позиция стои една от цифрите на системата. Позициите на числото, наричани още ***разряди***, са номерирани от дясно наляво с 0, 1, 2, и т.н. В общия случай число с n цифри в p-ична бройна система записваме като $a_{n-1}\ a_{n-2} \ldots a_{0(p)}$.

Стойността на всяка цифра c се мени в зависимост от позицията ѝ и е равна на $c.p^i$. Например, в числото 542, записано в десетична бройна система, цифрата 2 в нулевия разряд има стойност $2 = 2.10^0$, докато в числото 1245 тя е във втория разряд и има стойност $200 = 2.10^2$. Стойността на числото, представено в позиционна бройна система е сума от стойностите на всичките му цифри. Така $1245 = 1.10^3 + 2.10^2 + 4.10^1 + 5.10^0$.

Когато дробно число е представено в позиционна бройна система, тогава всяка от цифрите му след десетичната запетая се умножава по съответната отрицателна степен на основата p. Например $0{,}1245_{(10)} = 1.10^{-1} + 2.10^{-2} + 4.10^{-3} + 5.10^{-4}$.

Числата с дробна част се представят в паметта на компютъра по особен начин – с *мантиса* и *порядък*. Мантисата е число с дробна част m, порядъкът – цяло число p, а стойността на представяното число с мантиса m и порядък p е произведението $m.10^p$. Така например, числото 3,14 може

да се запише като $3{,}14.10^0$, $0{,}314.10^1$ или 314.10^{-2}. Този начин на запис позволява да се запазват числа с много малко значещи цифри. Така числото 1000000000000 може да се запише като 1.10^{12}, а 0,000000000001 като 1.10^{-12}.

Двоична позиционна бройна система

Десетичната бройна система не е подходяща за използване в компютрите, тъй като технически е много трудно да се представят десетичните цифри. Естествено е в електронните устройства да се използва ***двоична позиционна бройна система***. В нея вместо 10 се използват само 2 цифри – 0 и 1, които лесно се представят електронно с отсъствие или наличие на електрически ток, или пък с ненамагнитеност или намагнитеност. Както знаем от часовете по Информационни технологии, разрядите на числата, представени в двоична система, наричаме ***битове***. Стойността на всяка цифра в двоична система се умножава по съответните степени на основата 2. Така двоичното число $a_{n-1}\, a_{n-2} \ldots a_{0(2)}$ е равно на $a_{n-1}.2^{n-1} + a_{n-2}2^{n-2} + \ldots + a_0 2^0$.

Аналогично на десетичната бройна система, при представянето на дроби в двоична система всяка цифра се умножава по съответната отрицателна степен на 2. Така двоичното число $0{,}b_1\, b_2 \ldots b_{k(2)}$ е равно на $b_1.2^{-1} + b_2 2^{-2} + \ldots + a_k 2^{-k}$.

Преминаване от двоична система в десетична и обратно

Преминаването от двоична в десетична система става по формулата, дадена по-горе. За целта е добре да имаме предварително пресметнати стойностите на степените на основата 2. Първите няколко от тях са дадени в следната таблица:

Степенен показател	7	6	5	4	3	2	1	0
Степен на двойката	128	64	32	16	8	4	2	1

Да преобразуваме представеното в двоична позиционна бройна система число $10101011_{(2)}$ в десетична система. За по-лесно да поставим цифрите на числото в таблица, срещу съответните им степени на двойката:

Двоично число	1	0	1	0	1	0	1	1
Степен на двойката	128	64	32	16	8	4	2	1

Умножаваме всеки бит на двоичното число със съответната степен на двойката:

$10101011_{(2)} = 1.2^7 + 0.2^6 + 1.2^5 + 0.2^4 + 1.2^3 + 0.2^2 + 1.2^1 + 1.2^0 =$

$= 1.128 + 0 + 1.32 + 0 + 1.8 + 0 + 1.2 + 1.1 =$

$= 128 + 32 + 8 + 2 + 1 = 171_{(10)}$.

А как да преминем от десетична в двоична бройна система? Забележете, че ако разделим числото $N = b_{n-1}.2^{n-1} + b_{n-2}.2^{n-2} + \ldots + b_1.2^1 + b_0.2^0$ на 2 ще получим в резултат цялото частно $b_{n-1}.2^{n-2} + b_{n-2}.2^{n-3} + \ldots + b_1.2^0$ и остатък b_0 – най-младшия разряд на представянето на N в двоична система. Получаваме следната:

Процедура. *Дадено:* Естествено число N в десетична бройна система. *Действия:* Делим числото N на 2 и записваме полученото частно и остатък. Делим частното на две и отново записваме полученото частно и остатък. Продължаваме по същия начин, докато получим частно 0. *Резултат:* Получените остатъци, записани в ред, обратен на реда на получаването им, са двоично представяне на N.

```
171:2 = 85 и остатък 1
85:2 = 42  и остатък 1
42:2 = 21  и остатък 0
21:2 = 10  и остатък 1
10:2 = 5   и остатък 0
5:2 = 2    и остатък 1
2:2 = 1    и остатък 0
1:2 = 0    и остатък 1
171(2) = 10101011(2)
```

Фиг. 1.

На *Фиг. 1* е показан пример, в който процедурата е приложена върху числото $171_{(10)}$. Резултатът, разбира се, е $10101011_{(2)}$.

Превръщането на дробно число от двоична в десетична система също извършваме по формулата. За целта отново си приготвяме таблица със стойностите на отрицателните степени на двойката, като имаме предвид че $2^{-k} = 1/2^k$:

Степенен показател	–1	–2	–3	–4	–5	–6	–7
Степен на двойката	1/2=0,5	1/4=0,25	1/8=0,125	1/16=0,0625	0,03125	0,015625	0,0078125

Да превърнем дробното число $0{,}1001101_{(2)}$ от двоична в десетична система. Подреждаме цифрите на числото до съответните степени:

Двоично число	1	0	0	1	1	0	1
Степен на двойката	0,5	0,25	0,125	0,0625	0,03125	0,015625	0,0078125

и получаваме $0{,}1001101_{(2)} = 1.2^{-1} + 1.2^{-4} + 1.2^{-5} + 1.2^{-7} = 1/2 + 1/16 + 1/32 + 1/128 =$
$= 0{,}5 + 0{,}0625 + 0{,}03125 + 0{,}0078125 = 0{,}6015625_{(10)}$.

Преминаването обратно, от десетична в двоична система, при дробни числа става с умножение по 2. Забележете, че когато умножим по 2 число по-малко от единица, цялата част на получения резултат става 0 или 1 – първата двоична цифра на дробта. Така получаваме следната:

Процедура. *Дадено:* Дробно число D в десетична бройна система. *Действия:* Умножаваме дробната част на числото по 2. Отделяме получената цяла част – 0 или 1 – като поредна цифра на двоичното представяне и отново умножаваме само дробната част на резултата по 2. Когато получим в резултат дробна част 0, процедурата се прекратява. *Резултат:* Получените цели части, в реда на получаването им, са двоично представяне на D.

```
0 | 6015625 x 2 =
1 | 203125  x 2 =
0 | 40625   x 2 =
0 | 8125    x 2 =
1 | 625     x 2 =
1 | 25      x 2 =
0 | 5       x 2 =
1 | 0
```

Фиг. 2.

На *Фиг. 2* е показано преобразуването на числото $0.6015625_{(10)} = 0{,}1001101_{(2)}$.

За някои числа тази процедура никога не завършва, защото в двоична система те са безкрайни дроби. В такъв случай прекратяваме процедурата, когато намерим достатъчно много цифри след двоичната точка.

2 Числата и техните представяния – продължение

Аритметични действия с двоични числа

Процедурите за извършването на аритметични действия с естествени числа, записани в двоична бройна система по нищо не се различават от познатите ни процедури за извършване на аритметични действия в десетична система. Достатъчно е само да знаем съответните таблици за събиране, умножение и изваждане. В двоична бройна система тези таблици са много по-прости (виж *Фиг. 1*).

Събирането на две числа извършваме поразрядно. Ако при събирането на две цифри получим число по-голямо от основата p, тогава в резултата записваме последната цифра на числото, т.е. оста-

Събиране	Изваждане	Умножение
0 + 0 = 0	0 – 0 = 0	0 x 0 = 0
1 + 0 = 1	1 – 0 = 1	1 x 0 = 0
0 + 1 = 1	1 – 1 = 0	0 x 1 = 0
1 + 1 = 0 и пренос 1	10 – 1 = 1	1 x 1 = 1

Фиг. 1. Таблици за събиране, изваждане и умножение в двоична система

тъка при деление на p, а цялата част при деление на p е число, което наричаме ***пренос***. За събиране на числа в произволна позиционна система с основа p имаме следната:

Процедура. *Дадено:* Двоични числа $A = a_n.2^n+a_{n-1}2^{n-1}+...+a_02^0$ и $B = b_n.2^n+b_{n-1}2^{n-1}+...+b_02^0$ (ако едно от числата има по малко цифри, попълваме го отляво с нули). *Действия:* В началото преносът q е нула. Обработваме разрядите от дясно наляво. Когато сме в i-тия разряд, пресмятаме $C = q + a_i + b_i$. Последната цифра на C, да я означим с c_i, записваме в резултата, а преноса запомняме в q. Когато обработим и последния разряд, ако преносът е ненулев – записваме го най-вляво в резултата като c_{n+1}. *Резултат:* $c_{n+1}.2^{n+1}+c_n.2^n+c_{n-1}2^{n-1}+...+c_02^0$ е сумата на A и B.

Изваждането на числа в позиционна бройна система с основа p има особеност. Когато поредната цифра на умалителя b_i е по-голяма от съответната цифра на умаляемото a_i, трябва да добавим p_m от най-близкия вляво ненулев разряд на умаляемото и да извадим b_i от $a_i + p$. За да контролираме това непросто действие, поставяме точки над засегнатите цифри. Така всяка цифра, която е била 0, става $p – 1$, а тази, от която сме взели единица, намалява с 1. Т.е. в двоичната бройна система освен правилото `10 - 1 = 1`, трябва да се имат предвид и правилата `100 - 1 = 011`, `1000 - 1 = 0111` и т.н. За **изваждане на числа** имаме следната:

Процедура. *Дадено:* Двоични числа $A = a_n.2^n+a_{n-1}2^{n-1}+...+a_02^0$ и $B = b_n.2^n+b_{n-1}2^{n-1}+...+b_02^0$, $A \geq B$ (ако едно от числата има по-малко цифри, попълваме го отляво с нули, а ако $A<B$, изваждаме A от B и вземаме резултата със знак минус). *Действия:* Обработваме разрядите от дясно наляво. Когато сме в i-тия разряд $c_i = a_i – b_i$, ако $a_i \geq b_i$. А ако $a_i < b_i$ редуцираме a_i и цифрите преди нея до първата ненулева цифра, както е описано по-горе и изваждаме b_i от редуцираната a_i. *Резултат:* $c_n.2^n+c_{n-1}2^{n-1}+...+c_02^0$ е разликата на A и B. Ако един или няколко от най-левите разряди на полученото число са нули – премахваме ги от резултата.

Умножението в бройна система с основа p е доста по-сложна операция. Да припомним как се извършва тя. Основна стъпка в случая е умножаването на цифра b от множителя по множимото. В началото преносът q е 0. Започваме с нулевия разряд на множимото ($i = 0$), като вървим наляво. Пресмятаме $t = b.a_i+q$. Остатъкът при делението на t и p записваме пред получения до момента резултат, а цялата част на частното става стойност на преноса. Последният пренос записваме пред получения до момента резултат. След като сме умножили най-лявата цифра на множителя с множимото, правим същото и със следващата отляво цифра, но записваме резултата изместен наляво на една позиция – освободилата се позиция има стойност 0, която може и да не записваме. Така правим с всички цифри на множителя и накрая събираме получените числа.

На *Фиг. 2* са показани примери за събиране, изваждане и умножение на две числа в двоична бройна система.

```
 111   1          ...
  11011011       11011011          11011011
+  1110010     - 1110010         x      1010
 101001101       1101001           110110110
                                  11011011000
                                  100010001110
```

Фиг. 2. Събиране, изваждане и умножение в двоична система

Сравняването на числа в позиционна бройна система – независимо от основата, извършваме със следната:

Процедура. *Дадено:* Двоични числа $A = a_n.2^n + a_{n-1}2^{n-1} +...+ a_0 2^0$ и $B = b_m.2^m + b_{m-1}2^{m-1} +...+ b_0 2^0$. *Действия:* Ако $n > m$, тогава $A > B$. Ако $n < m$, тогава $A < B$. Ако $n = m$, тогава сравняваме цифрите на двете числа в едни и същи разряди, започвайки от най-левия. Ако не намерим различаващи се цифри в нито една от позициите, значи $A = B$. *Резултат:* Нека i-тият разряд е първият, в който двете цифри са различни. Ако $a_i > b_i$, тогава $A > B$. В противен случай $A < B$.

Шестнадесетична бройна система

Освен двоична система, в информатиката понякога се използва и шестнадесетичната система. За да работим с бройни системи с основа по-голяма от 10, трябва да изберем допълнителни знаци за недостигащите цифри. В шестнадесетична система за цифри, съответни на 10, 11, 12, 13, 14 и 15, се използват латинските букви A, B, C, D, E и F. Например, стойността на числото $1AE_{(16)}$ в десетична бройна система е:

$1AE_{(16)} = 1.16^2 + 10.16^1 + 14.16^0 = 256 + 160 + 14 = 430_{(10)}$.

Шестнадесетичната система е характерна с това, че много лесно се преминава от нея в двоична система и обратно. Преминаването от двоична в шестнадесетична система става, като разбием цифрите на двоичното представяне на четворки от дясно наляво – ако цифрите в началото не са достатъчни за пълна четворка, тогава добавяме необходимия брой нули отляво. След това всяка четворка от двоични цифри заместваме със съответната шестнадесетична цифра, равна по стойност на двоичното число, съответно на четворката. Съответната шестнадесетична цифра за всяка двоична четворка е дадена в следната таблица:

0	1	2	3	4	5	6	7
0000	0001	0010	0011	0100	0101	0110	0111
8	9	A	B	C	D	E	F
1000	1001	1010	1011	1100	1101	1110	1111

За преминаване от шестнадесетична в двоична система пък е достатъчно да заменим всяка шестнадесетична цифра със съответната четворка двоични цифри. Например, $1111110010_{(2)}$=(0011)(1111)(0010) = $3F2_{(16)}$, а $3B7_{(16)}$ = (0011)(1011)(0111) = $1110110111_{(2)}$.

Шестнадесетичната система е полезна, когато трябва да се представи много голяма двоична стойност. Така например двоичното съдържание 10011110 на един байт се записва много по-компактно шестнадесетично като 9E, а числото $100101000101010111110001_{(2)}$ – като 9455F1.

Въпроси и задачи

1. *Каква е разликата между позиционна и непозиционна система? Коя от двете е по-удобна за извършване на аритметични пресмятания?*
2. *Какви числа може да са основа на позиционна бройна система?*
3. *Защо 1 не може да бъде основа на позиционна бройна система? Как биха изглеждали числата в позиционна бройна система с една цифра?*
4. *Бързото преминаване от двоична в шестнадесетична бройна система и обратно се дължи на факта, че $16=2^4$ е точна степен на 2. Затова всяка четворка двоични цифри представя точно една от цифрите на шестнадесетичната бройна система. Дефинирайте* ***осмична бройна система****. Кои ще бъдат цифрите ѝ? Как ще стане превръщането на двоично число в осмично и обратно?*
5. *Може ли да се разбере от цифрите на зададено число в коя позиционна бройна система е то? Обосновете отговора си.*

3 Числата и техните представяния – упражнение

Задача 1. Нека си припомним първо действията с цели числа, представени в десетична бройна система. Пресметнете:

а) 129876 + 3499809; б) 3499809 – 129876; в) 129876 x 184.

Задача 2. Намерете двоичното представяне на:

а) $0_{(10)}$; б) $1_{(10)}$; в) $100_{(10)}$; г) $2048_{(10)}$.

Решения и упътвания:

а) Нулата във всяка позиционна бройна система е 0. Затова $0_{(10)} = 0_{(2)}$.

в) Това, че числото $100_{(10)}$ съдържа само двоични цифри, не означава, че то е в двоична бройна система. Според процедурата за превръщане на числа от десетична в двоична система делим на новата основа, докато частното стане 0 и записваме остатъците от деленето в обратен на получаването ред:

100 : 2 = 50 и остатък **0**;
50 : 2 = 25 и остатък **0**;
25 : 2 = 12 и остатък **1**;
12 : 2 = 6 и остатък **0**;
6 : 2 = 3 и остатък **0**;
3 : 2 = **1** и остатък **1**;
1 : 2 = **0** и остатък **1**.

Следователно $100_{(10)} = 1100100_{(2)}$.

Задача 3*. Разгледайте решението на Задача 2.г). Можете ли да обясните на какво се дължи елементарното представяне на $2048_{(10)}$ в двоична бройна система? Посочете друго число в десетична бройна система, което се представя по подобен начин.

Задача 4. Кое десетично число е $101101_{(2)}$? А кое десетично е $1101_{(16)}$?

Задача 5. Превърнете в двоична бройна система числото:

а) $0{,}57_{(10)}$; а) $0{,}11_{(10)}$; в) $3{,}14_{(10)}$.

Забележка. Знакът, разделящ цялата от дробната част на число, представено в двоична бройна система, наричаме ***двоична запетая***.

Решения и упътвания:

а) При решаване на тази задача трябва да имаме предвид, че процедурата по превръщане на дробно число от десетична бройна система в двоична може да е безкрайна и затова трябва да спрем, след като сме получили достатъчно двоични цифри.

$0.57_{(10)}$ x 2 = **1**.14
$0{,}14_{(10)}$ x 2 = **0**.28
$0{,}28_{(10)}$ x 2 = **0**.56
$0.56_{(10)}$ x 2 = **1**.12
$0.12_{(10)}$ x 2 = **0**.24
. . .

Следователно $0.57_{(10)} \approx 0.10010$.

в) *Упътване.* Превърнете в двоична система отделно цялата и отделно дробната част.

Задача 6. Превърнете в десетична бройна система числото:

а) $0{,}00001_{(2)}$; б) $0{,}1111_{(2)}$; $1010{,}1010_{(2)}$.

Упътване. Използвайте таблицата от урока с отрицателните степени на двойката.

Задача 7. Нека $A = 110110_{(2)}$ и $B = 1001_{(2)}$. Намерете:

а) сумата на A и B; б) разликата на A и B; в) произведението на A и B.

Задача 8. Сравнете по големина числата:
а) $101010_{(2)}$ и $11111_{(2)}$;
б) $1101010010_{(2)}$ и $1101010100_{(2)}$.

Въпроси и задачи

1. *Превърнете в десетична бройна система*
   а) $1101010001001_{(2)}$; *б)* $1101010001001_{(8)}$; *в)* $1101010001001_{(16)}$.

Упътване. *Използвайте приложението Калкулатор.*

2. *Намерете стойността на:*
   а) $31_{(10)}$ *в двоична бройна система;* *б)* $32_{(10)}$ *в осмична бройна система;*
   в) $12_{(8)}$ *в двоична бройна система;* *г)* $C8_{(16)}$ *в двоична бройна система.*

Упътване*. В подзадачите в) и г) може първо да превърнете съответното число в десетична бройна система, но може да го направите и по-бързо.*

3. *Нека* $A = 100110_{(2)}$ *и* $B = 11011_{(2)}$. *Намерете:*
   а) сумата на A и B; *б) разликата на A и B;* *в) произведението на A и B.*
4. *Дадено е числото* $10111_{(2)}$. *Кое е следващото по големина цяло число? Опишете процедура за добавяне на единица към число в двоична бройна система, която да е по-проста от общата процедура за събиране на две числа. Как ще изглежда подобна процедура за записани в десетична бройна система числа? А за произволна p-ична?*
5*. *Опишете процедури за преминаване от осмична бройна система в шестнадесетична и обратно.*

4 Информационни дейности и процеси

Информация

Според Речника на българския език, информацията е „съобщение, сведения за някого или за нещо, осведомяване“. Всяко нещо на този свят – обект, явление, процес и т.н. се отличава с някакви особености – ***характеристики*** (или ***свойства***). Например, всяка точка на земята има местоположение (географски координати), а във всеки момент от времето – определена температура, влажност, височина над морското равнище и т.н. Някои характеристики са ***реални***, като температурата, влажността и височината на една точка от земната повърхност. Други са **абстрактни** – като географските ѝ координати (при различен избор на отправните земни линии, от които се определят координатите – екватора и Гринуичкия меридиан – всяка точка ще получава различни координати). Едни характеристики са постоянни, например координатите, при фиксирано начало на координатната система. Други, като температурата и влажността са променливи във времето.

Всяка характеристика е съществена, когато може да приема различни стойности и не всички обекти, притежаващи една и съща характеристика имат една и съща стойност за нея. Ако всички точки на пространството имаха една и съща температура, например, характеристиката „температура“ не би предизвикала интереса ни.

Характеристиките на обект (явление, процес и т.н.), оформени по начин, който може да бъде възприеман от човека, наричаме ***информация*** за този обект.

Познаването на характеристиките на обектите е съществено за човека. Живеем в среда с много важни за съществуването ни обекти, явления и процеси. Ако не притежаваше информация за обкръ-

жаващия свят, не я оценяваше и не реагираше в съответствие с направените оценки, човечеството щеше да изчезне както някои съществували преди хиляди години животински видове. Човешкият вид се отличава не само с възможност да **добива информация**, но и да я **предава** от поколение на поколение и да я **използва**, за да се приспособи към промените на средата, да я изменя в своя полза и да създава нови обекти с нужните му характеристики и с избрани техни стойности.

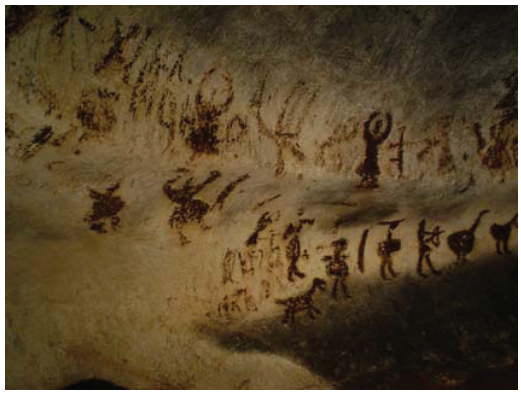
Човек възприема някои от реалните характеристики на обектите благодарение на своите ***сетива*** – зрение, слух, осезание, обоняние и вкус. За установяването на други характеристики са необходими специализирани ***измервателни инструменти*** – измервателна линия, кантар, термометър и т.н.

Абстрактните характеристики не могат нито да се възприемат сетивно, нито да се измерят с инструмент, освен ако не са създадени условия за това. Те се създават от човека, „закрепят“ се за съответния обект и стават информация за него. Така например, името на човека и неговият ЕГН се вписват в гражданския регистър. А координатите на точките от земната повърхност могат да бъдат пресметнати със съвременните технически средства (GPS) при избрано начало на координатната система.

За да могат стойностите на реални характеристики, наблюдавани от един човек, да се предадат на друг човек, е необходимо те да се представят по някакъв разбираем начин, т.е. да се превърнат в информация.

Съхраняване на информацията

Способността на човек да извлича информация е важно качество, но същевременно поражда и проблеми. Един от тези проблеми е **съхраняването на информацията**. Първобитните хора са съхранявали наученото от тях под формата на рисунки върху стените на пещерите, в които са живеели (*Фиг. 1*). С възникването на писмеността, човечеството създало удобен инструмент за съхраняване на информацията в писмен вид върху различни носители: папирусите в древен Египет, глинените плочки във Вавилон, пергаментите в древна Европа и т.н.



Фиг. 1. Пещерни рисунки

Фиг. 2. Печатарска машина

Многобройните и трудни за запомняне знаци – ***йероглифи***, обозначаващи различни понятия и действия, постепенно били заменени с малко на брой, нямащи собствено значение знаци – ***букви*** – така, че всяка информация може да

Фиг. 3. Библиотека

бъде изразена с последователности от букви. С изобретяването на хартията и книгопечатането (*Фиг. 2*) човечеството за дълго време решило проблема за съхраняване на добитата информация и предаването ѝ на бъдещите поколения.

И до днес книгите не са загубили своето значение. Многобройни публични и частни библиотеки съхраняват информацията, добита от човечеството за хилядолетия (*Фиг. 3*).

Използване на информация

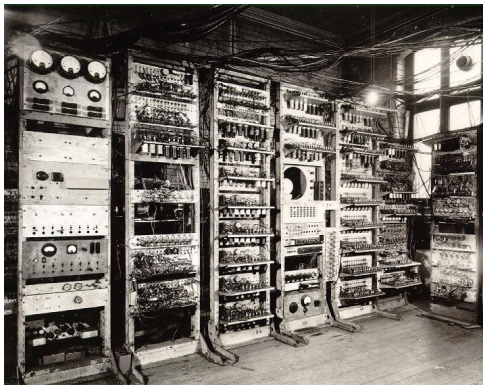
Добиването и съхраняването на информация, обаче, не е било достатъчно за развитието на човечеството. За да се реши сложен проблем, освен събирането на информацията, от която зависи решаването на проблема, е необходимо тази информация да бъде **намерена и използвана** – да се прегледа внимателно, да се определят измежду многобройните характеристики тези, които са най-важни, да се съпоставят стойностите на важните характеристики и да се търсят връзки между тях (както факта, че повишаването на температурата води до увеличаване на обема на телата, довел до създаване на инструмент за измерване на температурата), да се създават абстрактни понятия и характеристики на реалните обекти (както абстрактните координати, например) и да се използват тези характеристики за получаване на нова информация (какви са координатите на интересуващ ни обект, например).

Неудобството на книжните носители е в това, че те **не са подходящи за работа с наличната в тях информация** – в тях **трудно се търси**. Затова книгите се снабдяват със съдържание или различни индекси на понятията, а за търсене в океана от книги са изобретени библиотечните каталози. С непрекъснатото нарастване на обема на информацията, обаче, тези примитивни средства стават недостатъчни.

Друг проблем при използването на информация е, че способностите на човек да **съпоставя информация** с цел да извлече важни следствия и да вземе **важни решения**, са ограничени. При огромните обеми от натрупана информация един човек и даже големи групи от хора не са в състояние да прегледат всичко изписано по даден проблем и да извлекат нужната за решаването му информация **за разумно време**. В същото време, човечеството непрекъснато изгражда сложни технически системи – ядрените реактори, например, управлението на които изисква обработването на големи обеми информация и взимане на важни решения за много кратко време.

Затова, за да не спре в развитието и усъвършенстването си, човечеството е трябвало да се справи и с този проблем. Натрупаното знание още преди векове позволява той да бъде решен теоретично. От тогава до днес, едни от най-великите умове на човечеството правят стъпка след стъпка за практическото му решаване, но едва през 30-те и 40-те години на миналия век напредъкът на технологиите дава реална възможност това да се случи.

Автоматизирано обработване на информацията



Фиг. 4. Компютърът MARK1

Естествен подход за решаване на проблема е създаването на **машина**, която да усили интелектуалната мощ на човека, подобно на машините, които той вече е създал, за да увеличи физическата си мощ. За да може да се справи с тежките задачи, които трябва да решава такава машина, тя трябва да е изключително **бърза**. Тук даже повторяемостта на работния цикъл на съвременните автомобилни двигатели от няколко хиляди оборота в минута е абсолютно недостатъчна. В този смисъл е било изключено при работата на машината да се налага намеса от страна на човека, както често става с останалите машини.

Такъв начин на работа на една машина, при който намесата на човек е ненужна, наричаме ***автоматизиран***.

Към средата на 40-те години на миналия век започва построяването на първите машини за автоматизирана обработка на информация – ***компютрите*** (*Фиг. 4*). Тъй като е машина, компютърът не може да възприема информацията така, както човекът. Компютърът няма зрение, слух, обоняние и не може директно да възприема образи, звуци или аромат. За да може компютърът да съхранява и обработва информация, тя трябва да бъде представена по подходящ начин.

Въпроси и задачи

1. Посочете в природата източници на информация, които се възприемат директно от човека. С кое сетиво става това възприемане?
2. Посочете в природата източници на информация, която не се възприема директно от човека. Как сме достигнали до знанието за съществуване на всеки такъв източник?
3. Съществуват ли природни източници на информация, за които не подозираме?
4. Запознайте се по-подробно с работата на родителите си или на други близки. Посочете източници на информация на тяхното работно място.
5. Вгледайте се във вашето ежедневие. Посочете източници на информация, която представлява интерес за вас. Как добивате необходимата ви информация от тези източници?
6. Можете ли да откриете някаква информация във всяко от следните изображения:

а)  б)  в)  г) 

7. Може ли да откриете някаква информация в шума на работещ на високи обороти автомобилен двигател?
8. Можете ли (без да използвате специална техника), да откриете някаква информация в течност, налята в чаша, която няма никакъв цвят и миризма?
9. Има ли информация:
а) в романа „Под игото“; *б) в стихотворението „Две хубави очи“;*
в) в рапсодия „Вардар“; *г) в изображението на Мадарския конник;*
д) в предпочитания от вас видеоклип на любима група?

5 Информационни дейности и процеси – продължение

Представяне на информацията

От уроците по *Информационни технологии* знаем, че компютърът е двоична машина и може да съхранява и работи само с последователности от битове – нули и единици. Затова всяко нещо, което искаме да представим в компютъра, трябва да бъде ***кодирано*** двоично, за да може да се съхранява и обработва от компютър. Информацията, представена така, че да може да се обработва с компютър, наричаме ***данни***.

Вече знаем как се представят естествени числа в двоична бройна система. За да се представят отрицателни числа, първият бит на представянето се разглежда като ***знаков бит*** – 0 в знаковия бит означава плюс, а единица – минус. При положителните числа битовете след знаковия са двоичното

представяне на числото, а при отрицателните – някаква трансформация на абсолютната стойност така, че по-лесно да се извършват операциите. За дробните числа могат да се използват две представяния. При представянето с фиксирана ***запетая*** част от битовете е за цялата част, а останалите – за дробната, разглеждана като цяло число. Така дробното число 3,14 се представя с двете цели 3 и 14. Представянето с ***плаваща запетая*** е специфично експоненциално представяне – мантисата се избира така че да е максимално възможно, но по-малко от 1 число, т.е. да няма нула след десетичната запетая. Например, за числото 3,14 се избира експоненциалния вид $0{,}314.10^2$ и то се представя с целите числа 314 и 2, а числото 0,000001 – с целите 1 и –6 .

Нечисловите обекти също се представят с цели числа в двоична система. Всяка буква, цифра или знак, които използваме при създаването на текстове, се представят с цяло число в двоично представяне. А цветовете в системата RGB се кодират с три цели числа, представляващи интензивността на червената, зелената и синята съставящи на съответния цвят. Удобно е данните за по-сложни обекти да се събират в едно, за да се обработват по-лесно. Такова обединение на данните за един обект наричаме ***структура***. Някои обекти, представяни в компютъра, са доста по-сложни и имат много характеристики. Така информацията за един човек в системата за гражданска регистрация на населението съдържа три имена, дата на раждане, месторождение, единен граждански номер, единни граждански номера на родителите, братята и сестрите, постоянен и адрес на местоживеене и др. Без обединяване на данните за един човек в структура обработката им би била силно затруднена.

Науката информатика

С построяването на първите компютри възниква и науката (компютърна) ***информатика***.

Думата **ИНФОРМАТИКА** е съставена от части на думите **ИНФОР**мация и авто**МАТИКА**, т.е. това е науката, която изучава възможностите, начините и средствата за автоматична обработка на информацията.

От уроците по Информационни технологии познаваме огромните възможности на съвременната компютърна техника да решава различни задачи, които ежедневието поставя, да обогатява познанията ни за света и да ни помага да прекарваме по-интересно и по-приятно свободното си време. Всичко това става благодарение както на развитието на техниката, позволило да се създават компютри с все по-малки и по-малки размери, така и на науката, занимаваща се с компютърното програмиране, наречена информатика.

Отделни елементи на това, което днес наричаме информатика, се появяват още през 17 век. Истинските ѝ основи са поставени през 30-те и 40-те години на XX век, а обособяването ѝ като самостоятелна наука става преди не повече от 60 години. Тя, обаче, се развива изключително бързо и днес почти няма област в живота ни, която да не използва постиженията на информатиката.

Информационни дейности

Създаването на машина, която може **автоматизирано да обработва данни**, промени принципно възможностите на човека да изучава и променя обкръжаващата го действителност. Сложните задачи на математическата физика за пръв път получиха шанс да бъдат задоволително решени. Ядрената енергетика, космическите изследвания, моделирането на икономическите процеси, телекомуникациите и ред други области осъществиха огромен напредък, благодарение на използването на компютърна техника.

Умните машини, обаче, са приложими не само за решаване на сложни научни задачи. В настоящия момент в **почти всяка област на човешката дейност** се използват, по-малко или повече, компютри. Подготовка на печатни материали, управление на производство, финанси и персонал на раз-

лични по големина фирми, инженерно проектиране (включително на нови компютърни системи), публична администрация, банково дело, образование, развлекателни (но и сериозни) игри – това са само част от областите, в които компютрите са навлезли трайно.

Всичко това стана възможно, защото компютрите, освен да обработват данни, са в състояние да **съхраняват** огромни количества данни. Устройството за съхраняване на данни на нормален домашен персонален компютър днес е в състояние да съхрани съдържанието на около 1 000 000 средно големи книги, а малката, събираща се в джоб, флаш-памет – около 10 000. Ако приемем, че в целия свят в момента работят един милиард компютърни системи, можем да си представим, макар и приблизително, обема на наличната съхранена информация. В сравнение с класическото хранилище на информация от началото на миналия век – библиотеката, има една съществена разлика. Компютърът е в състояние да намира съхранените в него данни за части от секундата. Със създаването на световната **компютърна мрежа** интернет, цялата натрупана до момента информация може да се **предава** практически от всяка точка на земното кълбо до всяка друга точка, които са включени в мрежата.

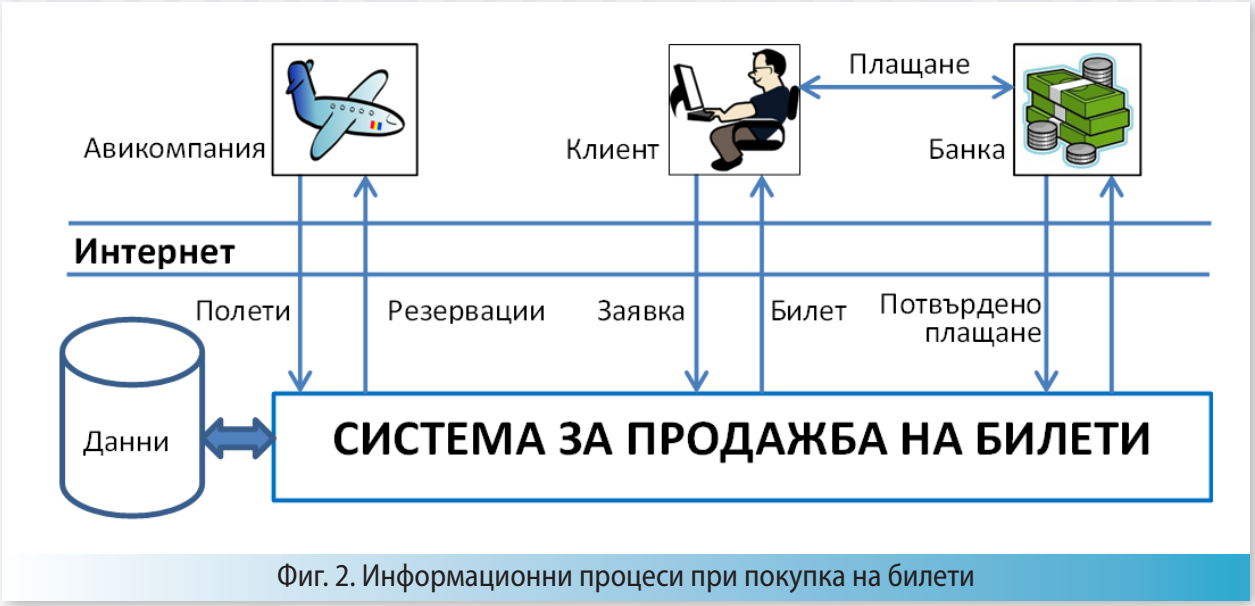
Събирането на данни, съхраняването им на компютърни носители, обработката на данни с цел получаване на важни за човека и обществото нови данни, както и предаването на данни между компютрите в локалните и световната мрежа за нуждите на общуването между хората, наричаме ***информационни дейности***. На *Фиг. 1* е показана схема на информационните дейности и потоците от данни, произтичащи от дейностите. В наши дни за всички информационни дейности се разработват методики, хардуерни платформи и различни софтуерни инструменти, така че информационните дейности да могат да бъдат извършвани ефективно и лесно. Съвкупността от методики за извършване на информационни дейности и предназначените за целта хардуер и софтуер наричаме ***информационни технологии***.

Фиг. 1. Информационни дейности

Информационни процеси

Информационният процес е съвкупност от взаимно обвързани информационни дейности (събиране на информация, съхраняване като данни, обработка на данни и разпространяване/използване на получените резултати) предназначени да осигурят извършването на някаква лична или обществено необходима дейност. В наши дни е невъзможно реализирането на ефективен информационен процес без използването на компютри, мрежата и съвременни информационни технологии.

За пример да разгледаме една популярна и обществено значима дейност като закупуването на самолетни билети (същите или много подобни процеси може да се наблюдават и при други дейности). На *Фиг. 2.* са показани участниците в дейността закупуване на билети през интернет и информационните процеси. Както се вижда на фигурата, могат да се разграничат 5 отделни съвкупности от свързани дейности, обединението на които решава задачата. Това са взаимодействията „клиент ↔ система“, „клиент ↔ банка“, „авиокомпания ↔ система“, „банка ↔ система“ и „система ↔ база от данни“. И във всеки от локалните процеси може да се наблюдават основните информационни дейности събиране и съхраняване на данни, обработване на данните и предоставяне на съответните резултати. Разделянето на глобалния информационен процес на локални информационни процеси е



Фиг. 2. Информационни процеси при покупка на билети

поради спецификата на всеки един от участниците. Например, информационните процеси, в които участва банката изискват много голяма сигурност, за да се предпазят банковите операции от злонамерена намеса.

Основна задача на Информатиката и Информационните технологии е да подпомагат обществото и гражданите, като:

- анализират осъществяваните в реалния живот дейности;
- идентифицират съпътстващите ги информационни процеси;
- създават формални (математически или компютърни) модели на тези процеси във формата на структури от данни и правила за тяхното обработване, с цел постигане на необходимите за дейността резултати;
- използват подходящи хардуер и софтуер за автоматизирането на процесите, а когато наличният софтуер не е достатъчен или не е достатъчно ефективен – създават необходимия софтуер за автоматизирането на процесите.

Създаването на софтуер е дейността, към която са ориентирани всички следващи уроци в този учебник.

Въпроси и задачи

1. *Посочете информация, която си струва да бъде съхранена. Защо смятате, че такава информация е полезна?*
2. *Посочете информация, която не си струва да бъде съхранена. Защо смятате, че такава информация е безполезна?*
3. *Посочете информация, която трудно може да бъде използвана, ако е съхранена по класически начин – на хартия.*
4. *Съставете схема на информационния процес, осъществяван при компютърна игра с двама играчи в мрежа.*

6 Алгоритми

Понятия и основни характеристики

Понятието ***алгоритъм*** е основно за информатиката. То произлиза от името на средноазиатския математик Мохамед ибн Муса ал Хорезми (от град Хорезм, днес Хива в Узбекистан), роден около 780 и починал в 847 г. Той е автор на съчинението „За индийското смятане“, посветено на представянето на числата в десетична бройна система и извършването на аритметични операции с тях. Затова в средните векове с algorismus или algorithmus са обозначавали правилата за извършване на операции в десетичната бройна система, които припомнихме в предишен урок.

Мохамед ибн Муса ал Хорезми

Постепенно смисълът на понятието се разширява и се достига до съвременното му разбиране, отразено в различни речници:

Речник на българския език, изд. БАН, том I, 1977: Система от правила, които определят последователност от изчислителни операции, прилагането на които води до решението на дадена задача. Алгоритъм на Евклид.

Larousse de la langue Francaise, Lexis, 1979: Съвкупност от правила или предписания за получаване с краен брой операции на определен резултат.

Webster's New Colegiate Dictionary, Webster, 1980: Процедура за решаване на математически проблеми (например намиране на най-голям общ делител) с краен брой стъпки, която често съдържа повтарящи се операции.

От цитираните описания на алгоритмите можем да извлечем някои основни техни характеристики:

1. За създаване на алгоритъм е необходимо множество от ***обекти*** и ***операции*** с тях. Например, естествените числа с аритметичните операции и сравняването.

2. Алгоритъмът решава някаква ***задача***, върху допустимите обекти, само с помощта на допустимите операции. Например, посочената в едно от описанията задача за намиране на най-голям общ делител (НОД) на две естествени числа. Задаваните обекти се наричат ***входни данни*** или ***вход***, а получаваните обекти – ***резултат*** или ***изход***. Такива задачи ще наричаме ***масови***, тъй като множеството от възможните входни данни може да е много голямо или както в задачата НОД – безкрайно. Ако фиксираме входните данни, получаваме ***екземпляр*** на задачата. Например: „Дадени са числата 12 и 30. Намерете техния НОД.“

3. Алгоритъмът е ***процедура***, определяща последователност от операции, която изпълнена над входните данни на произволен екземпляр на задачата, дава очаквания резултат. Споменатият в едно от определенията Алгоритъм на Евклид е такава процедура за намиране НОД на две положителни цели числа.

4. Процедурата трябва да е ***детерминирана*** – който и да я изпълни над едни и същи входни данни, трябва да получи един и същ резултат.

5. Процедурите, които наричаме алгоритми, трябва да водят до намиране на резултата с прилагане ***краен брой пъти*** на допустима операция. Този брой определя ***бързодействието*** на алгоритъма. Ако за една масова задача могат да бъдат построени няколко алгоритъма с различно бързодействие, добре е да знаем кой е най-бързият от тях.

6. Възможно е някои последователности от операции на алгоритъма да бъдат повтаряни многократно. Такива повторения се наричат ***цикли***. Организирането на цикли, както е посочено в един от споменатите по-горе речници, е характерна черта на много алгоритми.

В Урок 4 разгледахме алгоритмични процедури за решаване на задачите:

- да се представи в двоична бройна система число (цяло или дробно), зададено в десетична система;
- да се представи в десетична бройна система число (цяло или дробно), зададено в двоична система;
- да се намери сумата/разликата/произведението на две естествени числа, представени в позиционна бройна система;
- да се сравнят по големина две естествени числа, представени в позиционна бройна система;
- да се преобразува зададено число от двоична в шестнадесетична система и обратно.

Представяне на алгоритми

Естествените езици не могат да бъдат използвани за описване на алгоритми, заради възможност от неясно, двусмислено изразяване. Описаният на естествен език алгоритъм може да се окаже недетерминиран. Средства за недвусмислено представяне на алгоритмите са единствено ***машинните езици*** и ***езиците за програмиране***, но програмирането не е умение, което се постига лесно. В следващи уроци ще се учим да представяме алгоритмите на езика за програмиране C#.

```
1. Въведи A
2. Въведи B
3. Пресметни X = -B/A
4. Изведи X
5. Край
```

а.

```
1. Въведи A
2. Въведи B
3. Ако A==0 премини_към 6
4. X = -B/A
5. Изведи X и премини_към 7
6. Изведи "Няма решение"
7. Край
```

б.

```
1. Въведи A
2. Въведи B
3. Ако A==0 премини_към 6
4. X = -B/A
5. Изведи X и премини_към 8
6. Ако B==0 изведи "Всяко число
е решение" и премини_към 8
7. Изведи "Няма решение"
8. Край
```

в.

Фиг. 1.

В този раздел ще се спрем на по-лесни начини за описване на алгоритми – чрез ***ограничен естествен език*** и ***блок-схеми***. На *Фиг. 1.а* е представена, на ограничен естествен език, процедура за решаване на линейното уравнение $A.X + B = 0$. Да изпълним процедурата, като в стъпки 1 и 2 зададем за A стойност –7, а за B стойност 14. На стъпка 3, в X се пресмята $–B/A = –14/–7 = 2$ и процедурата ще изведе 2. Забележете различното значение на знака за равенство при описанието на алгоритмични процедури – пресмята се изразът, който е отдясно на знака и стойността му е новата стойност на променливата отляво. Казваме, че пресметнатата стойност на израза ***се присвоява*** на променливата.

Ако се опитаме да изпълним тази процедура с коя да е стойност за B и $A = 0$, тогава стъпка 3 няма да може да се изпълни заради невъзможността да се дели на 0. Процедурата не е добре дефинирана, защото не успява да завърши при някои стойности на входните данни. На *Фиг. 1.б* към процедурата от *Фиг. 1.а* е добавена проверката дали $A = 0$. Тъй като знакът за равенство вече е запазен за присвояването на стойност, знак за сравняването на две стойности ще ни бъде двойното равенство. Сега процедурата няма да спре аварийно при $A = 0$, но пък има друг дефект – ако $A = 0$ и $B = 0$, тогава всяка стойност на x е решение, а процедурата ще изведе невярното съобщение „Няма решение“. С още една проверка, показана на *Фиг. 1.в* като стъпка 6, поправяме и този дефект, за да получим алгоритъма за решаване на линейни уравнения.

Алгоритми като този от *Фиг. 1.а* наричаме ***линейни***, а тези от *Фиг. 1.б* и *1.в* – ***разклонени***.

Блок-схемният език е използван много в зората на програмирането. Днес той е позагубил своето практическо значение, но остава най-подходящото средство за обучение в техниката на разработване и описване на алгоритми. Основното му качество е, че е графичен и позволява да се представят по-лесно разклоненията, които при линейното представяне с ограничен естествен език или език за програмиране са трудни за проследяване. Съставните елементи на блок-схемите са блоковете и стрелките. Всеки блок определя действие, а когато действието е изпълнено, работата на алгоритъма продължава с блока, до който води излизащата стрелка.

Основните типове използвани блокове, представени на *Фиг. 2,* са:

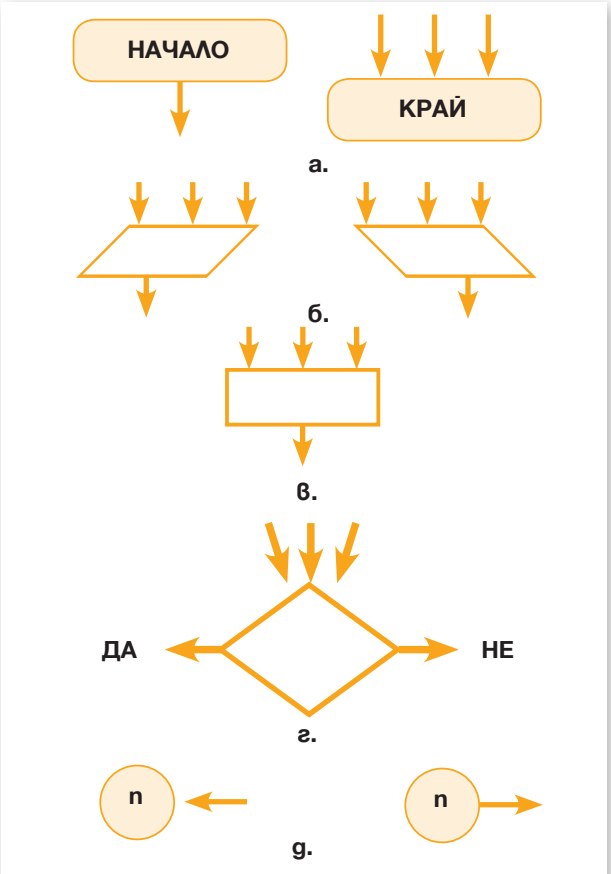
1. Блокове за начало и край (*Фиг. 2.а*). Тези два блока са с овална форма. Блокът НАЧАЛО определя мястото, от което започва изпълнението на алгоритъма и се среща еднократно в блок-схемата. В него не влизат стрелки и излиза една стрелка, показваща кой е следващият блок. Блокът КРАЙ определя място, където се прекратява изпълнението на алгоритъма. Такива блокове могат да бъдат няколко, като трябва да има поне един. От него не излиза стрелка, а може да влизат няколко.

2. Блокове за вход и изход (*Фиг. 2.б*). В първия блок се описват входните данни и се указва моментът, в който алгоритъмът получава входни данни. Вторият блок посочва момента на извеждането и извежданите междинни или крайни резултати. Произволен брой стрелки влизат и точно една стрелка излиза от тези два блока. Алгоритъмът изпълнява предписания вход или изход и продължава изпълнението с блока, който е посочен от излизащата стрелка.

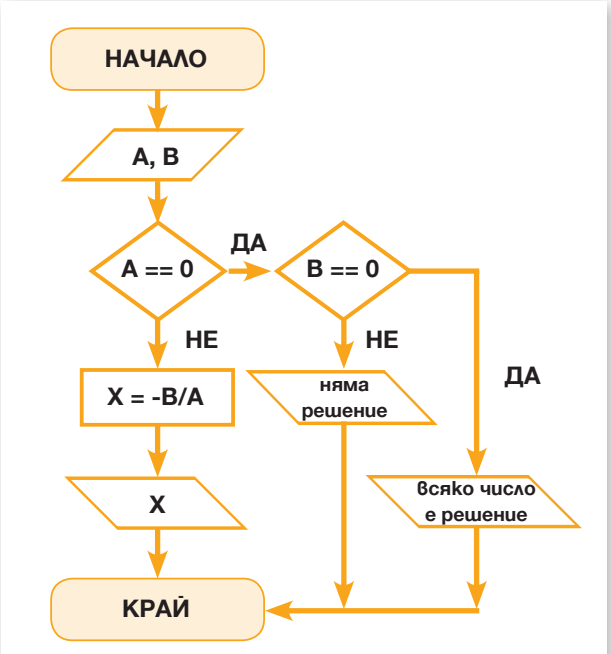
3. Обработващият блок (*Фиг. 2.в*) има форма на правоъгълник и в него се описват операции над обекти, които не са проверки, например изчисляване на изрази и запазване на резултата. Произволен брой стрелки могат да влизат в този блок и точно една стрелка излиза от него.

4. С блока от *Фиг. 2.г* се предписва проверка на някакво условие. В резултат процедурата се разклонява на две, в зависимост от това изпълнено ли е условието или не. Блокът има произволен брой входни стрелки и две изходни. Ако условието е в сила, изпълнението продължава с блока, към който сочи стрелката, надписана с ДА, а ако не – с блока, към който сочи стрелката, надписана с НЕ.

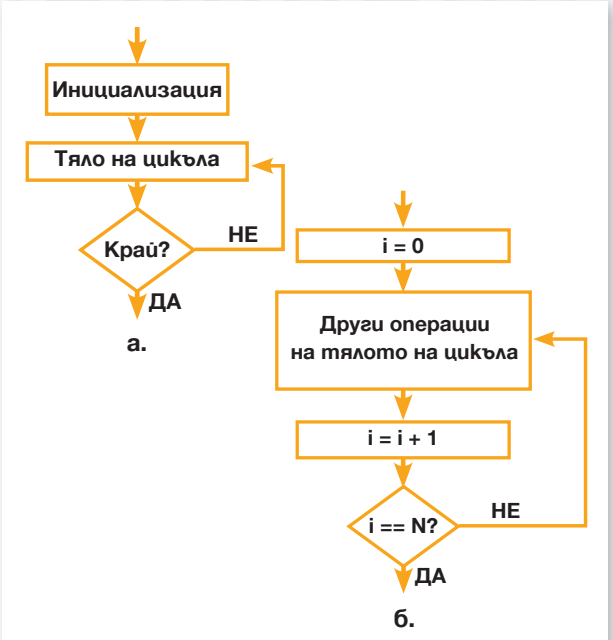
5. За прегледност при правене на блок-схемите се използват свързващи блокове с кръгла



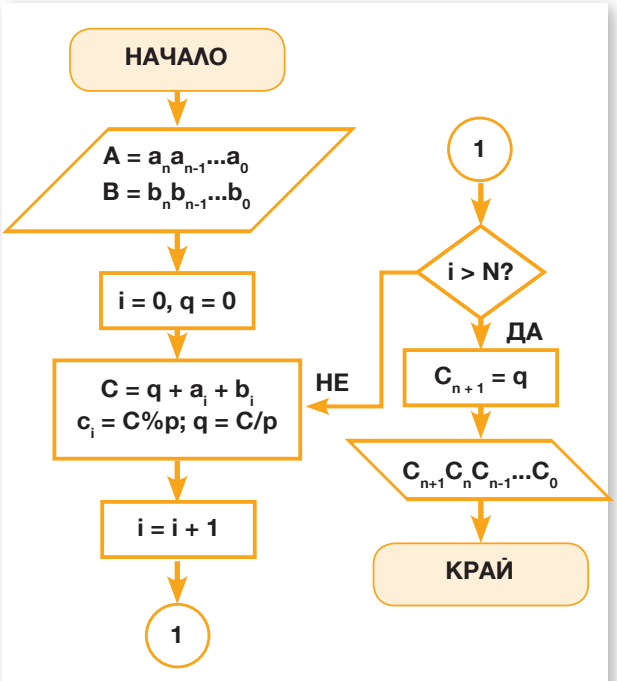
Фиг. 2. Блокове



Фиг. 3. Решаване на линейно уравнение



Фиг. 4.



Фиг. 5.

форма, в които се записват номера (*Фиг. 2.д*). За всяко цяло число n, в блок-схемата може да има точно един блок от всеки от двата типа, надписани с n. Използването на двойката такива блокове позволява да се разкъса прекалено дълга стрелка или стрелка, която сочи към блок от друга страница.

На *Фиг. 3* е показана блок-схемата на алгоритъма за решаване на линейно уравнение.

А сега да построим блок-схема за алгоритъма за събиране на числа в позиционна бройна система с основа p. Ще отбележим две важни особености. Първата е, че индексите в информатиката, както и другите данни, се съхраняват в променливи, чиито стойности могат да се изменят. Следователно, възможно е да записваме операции като $c_i = a_i + b_i + q$, които в зависимост от стойностите на индекса i имат различно действие. Ако $i = 0$, горната операция пресмята $c_0 = a_0 + b_0 + q$, но ако $i = 5$, тогава се пресмята $c_5 = a_5 + b_5 + q$. Това дава възможност при повтаряне на последователности от операции в цикъл, при всяко повторение да извършваме едни и същи действия, но с различни данни, в случая – с различни цифри на двете събираеми и сумата.

Втората особеност е, че в този алгоритъм има повтаряне на действия. Такъв алгоритъм наричаме ***цикличен***. Една от трудностите при описване на цикличен алгоритъм е описанието на циклите. На *Фиг. 4.а* е показана схематично последователност от блокове за организиране на цикъл. Тялото на цикъла се изпълнява, докато зададеното условие за край не бъде удовлетворено. Често срещан случай е, когато тялото на цикъла трябва да се изпълни определен брой пъти – да речем N. Фрагмент от блок-схема, осъществяващ такъв цикъл, е показан на *Фиг. 4.б*.

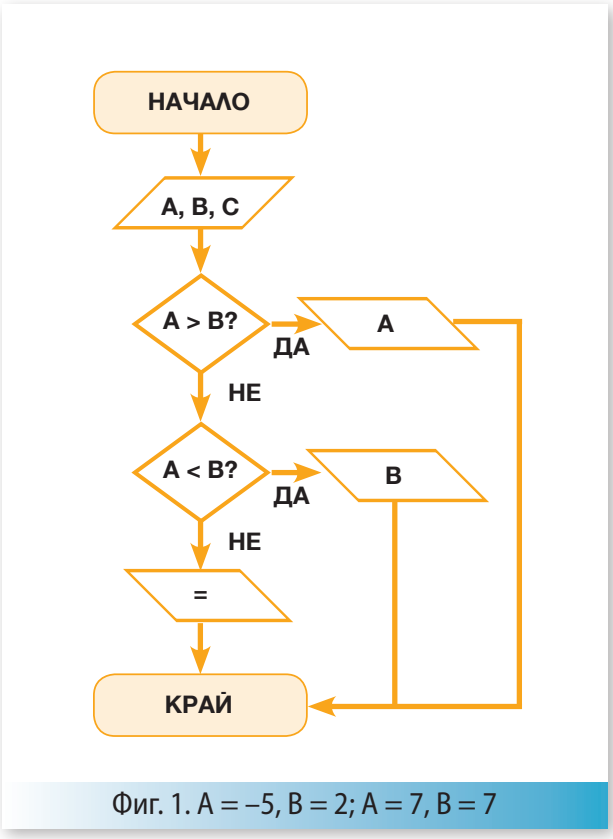
Съществено в тялото на цикъла е изменението на стойността на променливата i, която участва в условието за край. Само така условието може да бъде удовлетворено и да се излезе от цикъла. Още веднъж да обърнем внимание, че знакът за равенство има тук различен смисъл от този, с който го използваме в математиката. Математически, равенството $i = i + 1$ е еквивалентно на $0 = 1$, което е безсмислица. В блок-схемата $i = i + 1$ означава, че съдържанието на променливата се увеличава с единица. Алгоритъмът за събиране на две числа в p-ична система е показан на *Фиг. 5*.

Въпроси и задачи

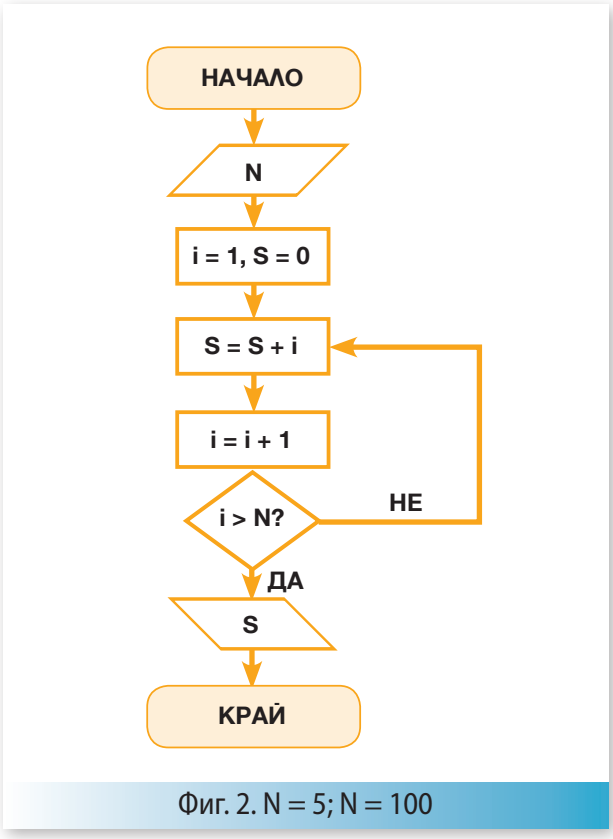
1. *Обяснете изискването алгоритъмът да е процедура за решаване на масова задача.*
2. *Защо кулинарните рецепти не са алгоритми?*
3. *Ако за дадена задача има няколко алгоритъма, какви могат да бъдат критериите, по които да се избере този от тях, който да се използва?*
4. *Прегледайте описаните в Урок 6 алгоритмични процедури. Защо многократното повтаряне на едни и същи на пръв поглед стъпки довежда до получаването на смислени резултати?*
5. *Дайте пример на процедура, която удовлетворява всички изисквания за алгоритъм, освен изискването за: а. детерминираност; б. крайност; в. масовост; г. наличие на вход.*

7 Алгоритми – упражнение

Задача 1. За всяка от следващите блок-схеми определете какъв ще бъде изведеният резултат при зададените стойности на входните данни:



Фиг. 1. A = –5, B = 2; A = 7, B = 7



Фиг. 2. N = 5; N = 100

Задача 2. Съставете блок-схема на алгоритъм за намиране лицето и периметъра на правоъгълник със страни *A* и *B*. *Упътване.* Този алгоритъм е линеен.

Задача 3. Съставете блок-схема на алгоритъм за проверка дали три дадени числа *A, B* и *C* са страни на триъгълник. *Упътване.* Три числа може да са дължини на страните на триъгълник, ако сумата на всеки 2 от тях е по-голяма от третото. Този алгоритъм е разклонен.

Задача 4. Съставете блок-схема на алгоритъм, който да разменя стойностите на две променливи X и Y. *Упътване.* Използвайте трета променлива Z, като първо преместите стойността на X в Z.

Задача 5. Съставете блок-схема на алгоритъм, който въвежда цяло число в променливата N и след това N цели числа в променливите $A_1, A_2, ..., A_N$. *Упътване.* Използвайте модела от *Фиг. 2*.

Задача 6. Съставете блок-схема на алгоритъма от Урок 1 на този раздел за сравняване по големина на две числа $A_{n-1}A_{n-2}...A_{0(p)}$ и $B_{n-1}B_{n-2}...B_{0(P)}$. *Упътване.* Използвайте модела от *Фиг. 2*. Отговорете си първо на въпросите: Каква ще бъде първата стойност на променливата i, за която трябва да се изпълни тялото на алгоритъма? С колко трябва да се промени i след изпълнение на тялото на цикъла? Коя ще бъде първата стойност на променливата i, за която трябва да прекрати цикъла?

Задача 7. Съставете блок-схема на Алгоритъма на Евклид за намиране на най-голям общ делител на целите числа (НОД[A,B]).

Упътване. Използвайте твърдението, че ако $A > B > 0$, тогава НОД[A,B] = НОД[$A – B,B$], модела от *Фиг. 4а* от предния урок и решението на Задача 4.

Задача 8. Съставете блок-схема на алгоритъм за намиране на най-малкото от числата $A_1, A_2, ..., A_N$. *Упътване.* В началото поставете в променливата *min* числото A_1, а след това продължете търсенето на най-малкото число по модела от *Фиг. 2*.

Задача 9*. Съставете блок-схема на тази стъпка в алгоритъма за изваждане на две числа $A_{n-1}A_{n-2}...A_{0(p)}$ и $B_{n-1}B_{n-2}...B_{0(P)}$, $A > B$, когато от по-малка цифра a_i трябва да се извади по-голяма. *Упътване.* Направете цикъл, който да върви наляво в търсене на първата различна от 0 цифра. Намалете тази цифра с единица. Направете нов цикъл, който върви надясно и заменя всяка от пропуснатите нули с $p – 1$, а цифрата, от която ще изваждате – с $p + a_i$.

Задача 10. Напишете на ограничения естествен език от урока алгоритмите от Задача 2, Задача 3 и Задача 4.

Задача 11*. Напишете на ограничения естествен език от урока алгоритъма от Задача 9.

8 Езици за програмиране

Съвременният компютър, чието принципно устройство познаваме от часовете по ИТ (наричан фон-Нойманов на името на учения, който за пръв път формулира всички изисквания, на които трябва да отговаря), е ***машина с програмно управление***. Устройството, което осъществява програмното управление на компютъра, е Централният процесор (ЦП). Същността на програмно управляваната машина е следната:

ЦП на компютъра изпълнява ***инструкции***, всяка от които предписва някаква ***операция*** с данни от оперативната памет (ОП). ***Кодът*** на инструкцията определя коя е операцията, а ***аргументите*** са или данни за операцията, или адреси в паметта, където са данните, с които да се извърши предписаната операция.

Системата от инструкции на компютъра и правилата за изпълнението им наричаме ***машинен език***. Поредица от инструкции наричаме ***програма***. Програмите се разполагат в ОП, както и данните. Така всяка инструкция получава ***адрес*** в паметта.

ЦП извлича инструкцията и зададените в нея аргументи от ОП, извършва операцията, записва резултата в ОП и определя следващата инструкция. Ако изпълнената инструкция е за пресмятане,

тогава ЦП продължава с инструкцията, която в програмата е **непосредствено след изпълнената**. Специални инструкции, за ***преход***, на машинния език могат да променят реда на изпълнение на инструкциите, като задават адрес на следващата инструкция, която трябва да се изпълни.

На *Фиг. 1* е показана програма от четири инструкции и част от ОП с необходимите за изпълнението данни. За удобство, двоичните кодове са заменени с разбираеми означения. Числото пред всяка инструкция е нейният адрес в ОП, кодовете на инструкциите са заменени с разбираеми думи, адресите на данните в паметта – с букви на променливи. След изпълнението на всяка от тези инструкции ЦП преминава към изпълнение на следващата инструкция.

```
1000 СЪБЕРИ X Y
1001 СЪБЕРИ С X 5
1002 УВЕЛИЧИ Y
1003 ПРЕМЕСТИ X Y
1004 КРАЙ
```

	...
X	1000
	...
Y	8
	...

Фиг. 1. Програма

Първата инструкция предписва да се съберат числата, намиращи се в паметта на адреси X и Y, а резултатът да се съхрани на адрес X. В резултат, съдържанието на X ще стане 1008 и ЦП ще премине към втората инструкция. Тя прибавя 5 към съдържанието на X и то става 1013. Третата инструкция увеличава съдържането на Y с единица, а четвъртата – заменя съдържанието на X с това на Y. Инструкцията КРАЙ е указание за процесора, да преустанови работата на програмата.

На *Фиг. 2* са показани примери на инструкции за преход. Инструкцията с код **ПРИ=0...НА...** предписва да се провери съдържанието на X и ако то е 0, ЦП трябва да премине към изпълнение на инструкцията с адрес A. В противен случай, т.е. ако X не е нула, ЦП трябва да продължи със следващата инструкция на програмата в ОП. Възможни са и други проверки, както в **ПРИ<0...НА...**, която извършва прехода, когато стойността на първия аргумент е по-малък от 0. Тези два прехода се наричат ***условни***. Инструкцията **СКОЧИ_НА...** предписва преходът да се извърши на адрес A без да се проверява условие – ***безусловен преход***. На *Фиг. 3* е показана програма, която намира абсолютната стойност на сумата на X и Y. Проследете работата на програмата, ако X = 3 и Y = –7.

```
ПРИ=0 X НА A
ПРИ<=0 X НА A
СКОЧИ_НА A
```

Фиг. 2. Преходи

```
1000 СЪБЕРИ X Y
1001 ПРИ>=0 X НА 1003
1002 УМНОЖИ С X -1
1003 ПОКАЖИ X
1004 КРАЙ
```

Фиг. 3.

Машинният език е труден за усвояване и всекидневна употреба. Затова за създаването на програми се използват ***езици за програмиране***, които са по-близки до човешкия, а машинният език се използва в изключително редки случаи – когато например се изисква написване на неголяма програма с максимално бързодействие.

Асемблери

Първите компютърни програми са написани, без съмнение, на машинните езици на съответните компютри, съдържащи стотици инструкции, в които кодовете на операциите и адресите на аргументите се изписват в цифров вид. Запомнянето на ***синтаксиса*** на инструкциите (кой е кодът на инструкцията, както и колко и какви аргументи има) и ***семантиката*** им (какво точно предписва всяка инструкция) е доста трудно. За избягване на грешки се налагат чести справки в описанието на езика. Освен това програмата, написана на машинния език на един компютър, не може да бъде пренесена на друг компютър.

За облекчаване работата на програмиста в зората на програмирането се създават ***асемблерните езици***. Разликата между тях и машинния език е, че в програмите, написани на асемблерен език, се допуска използване на ***нечислови*** последователности от знаци за означаване кодовете на операциите и адресите на операндите. Служебните думи `ADD, SUB, MULT и DIV,` например, извлечени от английските наименования на аритметичните операции – addition, subtraction, multiplication и division, могат да се използват за означаване на аритметичните операции. За поле от паметта, съдържащо обема на конкретен обект, може да се избере името `volume`, за поле, съдържащо възраст – `age` и т.н. Езикът на програмите от *Фиг. 1* и *Фиг. 3* по вида си е асемблерен.

„Преводът“ на програма от асемблерен на машинен език се извършва от специална програма – ***асемблер***, която замества имената на инструкциите и имената на аргументите със съответните кодове и адреси.

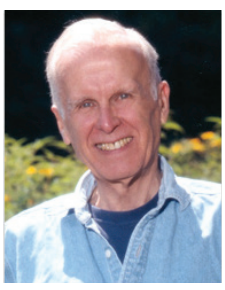
Алгоритмични езици

Програмирането на асемблерен език е доста по-лесно от програмирането на машинен език и до днес не е загубило своята роля. Използването на асемблерен език, обаче, носи твърде малко облекчения на програмиста. С разширяване на сферата на приложение на компютрите, в 60-те години на миналия век се появяват езици, първоначалната цел на които е програмистите да обменят алгоритми помежду си, затова отначало са наричани ***алгоритмични***.

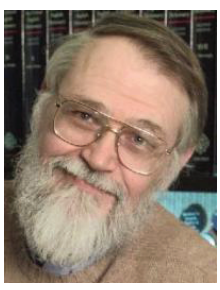
Алгоритмичните езици са от по-високо ниво от асемблерните. Една инструкция на езика от високо ниво (наричана ***оператор***) съответства на няколко машинни инструкции. Тези езици се доближават до човешкия, защото операторите им се съставят от разбираеми за човека фрази (обикновено от английския език). Например, `if … then` ... (ако … тогава ...), или `while … do …` (докато ... прави …) и др. Програмите, написани на алгоритмичен език се четат много по-лесно, което ги прави най-доброто средство за обмен на алгоритми и обучение в алгоритмизация на задачи за програмиране. Постепенно алгоритмичните езици изместват асемблерните от програмистката практика и започват да се наричат ***езици за програмиране***.

До идеята за език за програмиране пръв е достигнал Конрад Цузе, създателят на едни от първите реално действащи програмируеми, но не електронни, изчислителни устройства. През 1945 г. той проектира езика Plancalcul, който остава незавършен, но редица елементи от този проект се срещат по-късно в много езици за програмиране. През 1954-57 г., под ръководството на Джон Бекъс в IBM е разработен езикът FORTRAN (от англ. FORmula TRANslation, превод на формули). Предназначен в началото за бързо програмиране на числени пресмятания, по-късно той се използва като универсален език и оказва голямо влияние върху развитието на езиците за програмиране, независимо от някои свои несъвършенства. Начинът за пресмятане на стойността на математически изрази при зададени стойности на променливите във FORTRAN се среща и в най-съвременните езици за програмиране. Езикът FORTRAN претърпя голяма еволюция, като следващите му версии FORTRAN II, FORTRAN IV, FORTRAN 77, FORTRAN 85 отразяваха развитието на езиците за програмиране, а най-новата му версия FORTRAN 9x се използва и днес.


Конрад Цузе


Джон Бекъс


Грейс Хопър


Брайън Кернигън


Денис Ричи

По същото време, колектив под ръководството на Грейс Хопър работи над специализирани езици за обработка на икономическа информация. Най-сполучлив от тях е езикът COBOL (от англ. COmmercial and Business Oriented Language, език за бизнес приложения). Счита се, че 60% от софтуера за управление на бизнеса, който работи и в момента, е написан на COBOL.

В края на 50-те години специален научен комитет се занимава с разработването на принципите на универсален език за програмиране, който да не е ориентиран само към един тип задачи. Създаденият от този комитет език Algol (от англ. ALGOrithmic Language, алгоритмичен език) не успява да се наложи в практиката, но оказва голямо влияние върху развитие на езиците за програмиране.

До неотдавна много популярни бяха езиците BASIC (създаден от Джон Кемени и Томас Курц) и Pascal (създаден от Никлаус Вирт). Важен за развитието на езиците за програмиране е езикът C (създаден в началото но 70-те години от Брайън Кернигън и Денис Ричи), тъй като много от най-разпространените днес езици – C++, Java, Perl, C# и др. – са породени от C.

Транслатори

Езикът FORTRAN пожънал огромен успех, защото Джон Бекъс и сътрудниците му създали програма – ***транслатор***, която превежда програмите от FORTRAN на машинен език. По качествата си получените програми не се различавали много от написаните от опитен програмист. При това програмирането на FORTRAN е много по-бързо и по-надеждно от програмирането на асемблерен език. Разработките на Бекъс предизвикват революция в програмирането. Различаваме два вида транслатори. Първият вид са ***компилаторите***. Компилаторът избира за всяка конструкция на езика за програмиране подходящ, предварително подготвен фрагмент на машинен език и от всички фрагменти съставя (***компилира***) програма. В резултат на работата на компилатора се получава пълен превод на ***изходния текст*** на програмата (англ. source code) на машинен език – ***изпълнима програма*** (англ. executable). По този начин не се налага превеждане на програмата всеки път, когато потребителят пожелае да я изпълни върху конкретни данни.

Различен е подходът при ***интерпретатори***те. Те също подбират подходяща последователност от машинни команди за всяка от конструкциите на езика, но тя се изпълнява веднага (казваме, че конструкцията се ***интерпретира***), след което се преминава към интерпретиране на следващата конструкция в програмата. Ако по време на изпълнение на програмата съответната конструкция се достигне още веднъж, например в тялото на цикъл, инструкцията ще бъде интерпретирана отново. Интерпретатори се създават много по-лесно от компилаторите. Интерпретирането на програмата обаче е много по-бавно от изпълнението на компилирана програма. Като пример на интерпретатори ще посочим многобройните версии на транслатори за езика BASIC в първите години от появяването му.

Въпроси и задачи

1. *Защо наричат ЦП „мозък“ на компютъра?*
2. *Напишете програма с командите от урока за пресмятане на израза:*
   а. A + B + C; б. A + B.C; в. A.B + C.D, ако стойностите на A, B, C и D се намират в едноименни полета на паметта, а стойността на израза се запише в поле от паметта, означено с X.
3. *Напишете програма с командите от урока за пресмятане на верността на условието:*
   а. A + B + C < 0; б. A + B > C; в. A.B ≥ C.D, ако стойностите на A, B, C и D се намират в едноименни полета на паметта, а верността на условието – означена с 0 за неистина и 1 за истина – се записва в поле от паметта, означено с X. Константите 0 и 1 да бъдат в паметта на адреси `X+1` и `X+2`.
4. *Напишете програма с командите от урока за решаване на линейно уравнение с коефициенти A и B, зададени в едноименни адреси на паметта, а текстовете* `Няма решение` *и* `Безброй решения` *се намират на адреси `Y` и `Z` в паметта.*
5. *Намерете в Интернет информация за историята на езиците FORTRAN и C и направете презентация за създателите им и тяхното развитие.*

9 Езици за програмиране – продължение

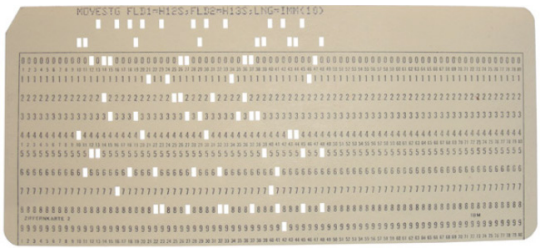
Ръчно програмиране

Първите програмисти – учени и университетски преподаватели – са създавали програми за собствени нужди на машинен или асемблерен език. Тъй като авторът на програмата е бил и единствен потребител, не се е налагало да планира предварително нейната функционалност, да създава документи, описващи логиката или функциите ѝ. Проверката за работоспособност на програмата се е извършвала в процеса на използване и когато са се установявали грешки, авторът-потребител е внасял необходимите корекции.

За разработването на програмите не се е използвал никакъв софтуерен инструментариум. Не са били изобретени още съвременните входни и изходни устройства – клавиатурата и мониторът. Програмите са се пишели на хартиени бланки, перфорирали се върху хартиени носители – перфоленти (*Фиг. 1*) и перфокарти (*Фиг. 2*) и се въвеждали в компютъра от устройства за четене на такива носители. На същите носители се перфорирали и входните данни. Резултатите от работата на програмата се отпечатвали от специализирани механични печатащи устройства върху по-тясна или по-широка хартиена лента (*Фиг. 3*).

С изобретяването на езиците за програмиране от високо ниво през 60-те години на миналия век, писането на програми става значително по-лесно. Програми създават не само математиците и информатиците, а и специалисти от близки специалности – физици, инженери и др. Възниква професията ***програмист***. Освен програми за свои нужди – транслатори от езиците за програмиране, операционни системи и т.н., които наричаме ***системен софтуер*** – програмистите започнали да пишат програми за нуждите и на други потребители – ***приложен софтуер***.

Фиг. 1. Перфолента



Фиг. 2. Перфокарта

Фиг. 3. Печатащо устройство

Технологично програмиране

В началото на 70-те години на миналия век компютрите получават все по-широко разпространение. Големите компании предпочитат да имат свои компютри за нуждите на управлението на

предприятията. Необходимостта от програми става все по-голяма, а програмите все по-сложни. От ***ръчната*** форма на създаване на програми се преминава към по-съвършената ***технологична*** форма.

През тези години се налага стилът, наричан ***структурно програмиране***. Сложните програми се изграждат от по-прости, логически свързани ***модули (подпрограми)***. Програмите се ***планират (специфицират)*** като съставени от подпрограми, специфицира се всяка от подпрограмите и тяхното взаимодействие и написването на различните подпрограми може да бъде възложено на различни програмисти. Характерен резултат от въвеждането на структурното програмиране е възможността някои важни подпрограми да бъдат написани веднъж завинаги от много опитни програмисти и да се използват наготово при създаване на нови програми. Такива подпрограми се наричат ***стандартни***, а архивите, в които са съхранени, ***библиотеки от стандартни подпрограми***. Използването на библиотеки от стандартни подпрограми ускорява програмирането и гарантира качественото решение на често срещани важни програмистки задачи.

За да могат програмистите да използват подпрограмите, написани от техни колеги, се налагало подпрограмите да бъдат ***документирани***. Наложило се и създаването на ***потребителска документация*** за по-сложните програми, за да може потребителите да работят с тях без помощта на създателите. Така, с програмата се свързват и различни придружаващи я компоненти и вместо за програма, вече се говори за ***програмен продукт***. Програмните продукти стават ***стока***, имат си цена и с тях се търгува, както с всяка друга стока.

Неизменни етапи от технологичното създаване на програмни продукти са ***тестването*** и ***съпровождането*** им. По време на тестването се откриват грешки, дават се предложения относно функционалността, удобствата за потребителя и др. Препоръките, забележките и предложенията се предоставят на програмистите, които извършват необходимите промени. По време на съпровождането на програмния продукт се отстраняват грешки в програмите, неоткрити при тестването. Внасят се и налагащи се изменения, например когато програмата е свързана с прилагането на нормативен документ, който се променя периодично и др. Затова, много от програмните продукти първоначално се разпространяват в ***пробни версии*** (алфа версия, бета версия и т.н.) и периодично се ***обновяват*** (англ. update).

Компютърният терминал

Истинска революция в технологията на програмирането предизвиква създаването на компютърния ***терминал*** (или ***конзола***) състоящ се от:

- едно входно устройство (клавиатура), от което да се въвежда текстът на програмите и командите към ОС;
- едно изходно у стройство (принтер), на което да се извежда това, което работещият на терминала въвежда („ехо“), както и резултатите от изпълнението на програмата.

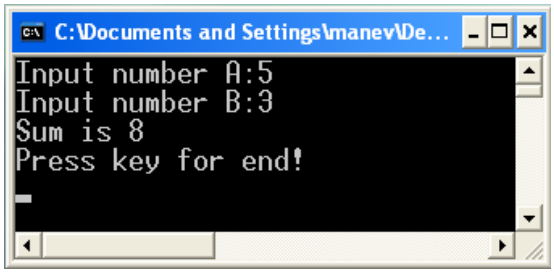
Фиг. 4. Терминал

Първите компютърни терминали, по същество, са управлявани от компютъра пишещи машини (*Фиг. 4*). Разликата между такъв терминал и използваната в живота пишеща машина е, че терминалът работи не с отделни листи, а с непрекъсната хартиена лента.

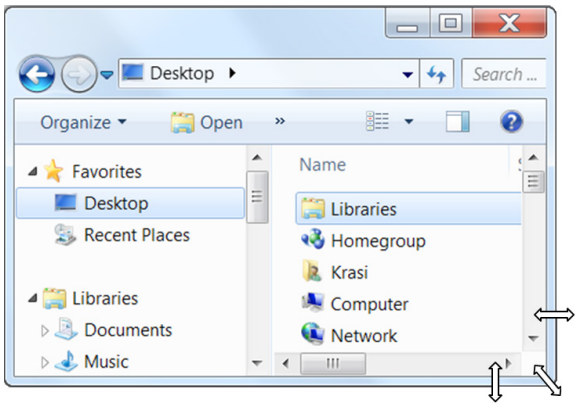
С изобретяването на терминала се създават принципно нови възможности за облекчаване работата на програмистите. Създават се първите ***текстови редактори***. С помощта на текстовия редактор програмистът може да въвежда текста на програмата от терминала във файл, без да се налага да го перфорира на лента или карти. Той може във всеки момент да разпечата избрана част от програмата

или цялата програма, за да види състоянието ѝ, след което да зададе нужните му изменения, да съхрани програмата във файла и да стартира транслатора. Тъй като добре познаваме тези инструментални програми от уроците по ИТ, ще отбележим само съществуването на специализирани текстови редактори за създаване на програми, които предлагат редица специални възможности – оцветяване на отделните части на програмата в различни цветове, подреждане на текста, подсказване за евентуални грешки и т.н.

Транслаторът също извежда своите съобщения на терминала и програмистът може веднага да се заеме с поправяне на откритите от компилатора грешки. След като програмата се компилира без грешки, програмистът може да я изпълни многократно, задавайки входни данни от терминала или от предварително подготвени с текстовия редактор файлове с данни. Ако до изобретяването на терминала програмистът е можел да разчита на 2-3 компилации и изпълнения на програмата на ден, то при работата от терминал той може да стигне до стотици компилации и изпълнения на програмата за един ден.



Фиг. 5. Програма с буквено-цифров интерфейс



Фиг. 6. Програма с графичен интерфейс

При работа с терминал става възможно създаване на програми, улесняващи тестването – ***дебъгери***. Дебъгерът позволява програмата да се изпълнява постъпково, като след всяка стъпка се наблюдава състоянието ѝ и така се откриват грешките във функционирането. Създават се и други програми за облекчаване на процеса на програмиране – програми, които подреждат по определени правила текста на програмата и го правят по-лесен за четене, програми, които поддържат последователни версии на една програма, за да може бързо да се върнем от неудачна нова версия към по-добрата стара, без да се пазят поотделно двете версии и т.н.

Постепенно, на мястото на електромеханичния терминал идва електронният – добре познатите днес ***компютърна клавиатура*** и ***монитор***. Отначало мониторите са с възможност да изобразяват само знаците на клавиатурната азбука и не носят някаква принципна новост, освен изчезването на неприятния шум на пишещата машина, произвеждан от удара на устройството, носещо съответната буква, по хартията. Затова и създаваните програми са само с ***буквено-цифров интерфейс*** (*Фиг. 5*). С въвеждането на графичните монитори, и най-вече на цветните графични монитори, се появява възможността за създаване на програми с ***графичен интерфейс*** (*Фиг. 6*).

Програми с графичен интерфейс

Създаването на програми с графичен интерфейс, което е приоритет за софтуерната индустрия днес, дължим на напредъка на технологиите в областта на хардуера: повсеместното използване на качествени цветни монитори, непрекъснато увеличаване на обемите на компютърната памет, подобряването на качествата на видеокартите, повишаването на скоростта на обмен на данните в компютъра, развитието на технологиите за визуализация на данни и т.н.

Програмите с графичен интерфейс се изграждат по еднотипен начин. При стартирането си, такава програма отваря на екрана един ***основен прозорец***, в който са изобразени елементи на графичния

интерфейс – ***бутони***, ***менюта***, ***текстови*** и ***комбинирани текстови кутии***, ***пояснителни надписи*** и др. С елементи като текстовите кутии и менютата с различни възможности за избор, например, потребителят задава данни на програмата, а с бутоните и менютата с команди посочва действията, които програмата трябва да изпълни. По време на работа, за изпълнение на една или друга функция, програмата може да отвори и други – ***работни*** или ***диалогови*** – прозорци. Програмите, които познаваме от уроците по ИТ, са програми с графичен интерфейс.

Визуално програмиране

Създаването на програма с графичен интерфейс може да се осъществи с класически форми на програмиране. Използват се стандартни подпрограми за създаване на графика, да се изчертават прозорците, като за всеки прозорец се помни мястото му на екрана, каква част от него се вижда във всеки момент и т.н. По същия начин се изчертават и отделните елементи на графичния интерфейс и се запомнят техните места в прозореца, съобразявайки се с мястото на прозореца върху екрана, програмират се цветът им на запълване, изписването на съдържащите се в тях текстове и т.н. При това, програмата следи за действията на потребителя и при всяко внесено от него изменение, цялото изображение се прерисува отново. Да се напише такава програма на ръка е изключително трудно. Освен това, какво точно се получава на екрана става ясно след като програмата се компилира и изпълни. Такъв начин на изграждане на програми с графичен интерфейс е изключително неудобен.

Затова, при създаване на приложни програми с графичен интерфейс, се използва по-подходящ стил за програмиране, наричан ***визуално програмиране***, и съответни инструментални програми. При този стил на програмиране програмистът разполага с голям набор от готови елементи, с които да изгради графичния интерфейс на създаваната от него програма, без да се налага да програмира тези елементи съвсем от начало, а трябва само да настрои техните параметри. В следващи уроци ще се запознаем с инструментариум за създаване на програми с графичен интерфейс.

Обекти и обектно-ориентирано програмиране

Както споменахме в предишен урок, при създаване на компютърен модел на реален обект от живота, може да се наложи да се съхраняват като данни стойностите на много негови характеристики. Това става с подходящо структуриране на данните и написване на подпрограми за необходимите операции с данните на обекта. Съществуват различни програмистки механизми за това, но господстващ в момента е механизмът, наричан ***клас от обекти***, а стилът на създаване на програми с използването на класове от обекти – ***обектно-ориентирано програмиране*** (ООП). Ще се запознаем с основните елементи на ООП.

Програмният ***обект*** е компютърен образ, съответстващ на реалния обект или явление. Например, за програма, която е предназначена да обслужва учебния процес, естествено е да дефинираме обекта Ученик. Основание за това е фактът, че за всеки ученик трябва да бъдат запазени в компютъра няколко характеристики, наричани ***атрибути*** (или ***свойства***) на обекта: име, дата на раждане, месторождение, идентификатор на класа, в който ученикът учи, номер в класа, оценки по учебните предмети, брой отсъствия и др.

С всеки обект в компютърната програма обикновено се свързват някакви обработки, извършвани в компютъра от подпрограми. Подпрограмите, свързани с обработката на определен обект се наричат ***методи***. Например, за обекта Ученик може да има методи, които определят името му, датата му на раждане или кой да е друг атрибут, средния му успех и т.н.

Всички еднотипни обекти – с еднакви атрибути и методи – образуват клас от обекти или просто ***клас***. Всеки конкретен обект със специфични стойности на атрибутите се нарича ***екземпляр*** на класа. Класът е средството на езика за обединяване на всички еднотипни обекти.

Подобно на стандартните подпрограми в процедурното програмиране, езиците за ООП предоставят на програмиста ***библиотеки от стандартни класове*** от обекти. Това са често използвани класове с общоприети методи или класове, за които трудно бихме написали сами съответни методи.

В следващи уроци ще се запознаем с езика за обектно-ориентирано програмиране C# (чете се „Си шарп“), с някои от стандартните класове от обекти на този език и техните методи.

Въпроси и задачи

1. *По какво се различава създаването на програма за собствени нужди от създаването на програма за друг потребител?*
2. *По какво се различава създаването на програма от един програмист от създаването на програмен продукт от колектив от програмисти?*
3. *Защо в програмните продукти от 80-те години не се използват бутони, менюта, текстови кутии и др.?*
4. *В кои от етапите на разработване на програмен продукт може да не участват програмисти?*
5. *Дайте примери за програмни продукти, използвани от вас, при които след името им има посочена последната версия.*
6. *Кога се налагат поправки в програмния продукт, които не са продиктувани от грешки на алгоритъма или на кода?*

II Среда за визуално програмиране

10 Интегрирана среда за визуално програмиране

Етапи и инструменти за създаване на програми

Както споменахме в предишен урок, в ерата на нетехнологичното програмиране, а и в началните години на технологичното програмиране, програмистът не разполага с много удобства в работата си. Той написва текста на програмата (***изходен код***) на хартия. Написаната на хартия програма се пренася на перфоленти или перфокарти. Ако **транслаторът** от езика за програмиране е **интерпретатор**, програмата се интерпретира и резултатът се разпечатва или записва на магнитен носител за следващо използване като входни данни.

Ако транслаторът е **асемблер** или **компилатор**, тогава програмата се превежда на машинен език, като полученият резултат се нарича ***обектен код***. При написването на програмата, програмистите често допускат грешно използване на езика за програмиране – неправилно изписан текст, некоректно поставени специални знаци и т.н. Такива грешки наричаме ***синтактични***. След като транслаторът (интерпретатор или компилатор) съобщи за допуснатите грешки, програмистът ги отстранява и отново опитва да транслира програмата. И това се повтаря до изчистването на всички синтактични грешки (*Фиг. 1*).

Фиг. 1.

След като изходният код е правилно преведен до обектен код, специализирана програма, наречена ***свързващ редактор*** (linker), добавя (свързва) към него предварително преведените до обектен код стандартни подпрограми, използвани в изходния код. Свързващият редактор може също така да свърже в една изпълнима програма няколко отделно компилирани модула на сложна програмна система. В резултат от свързването се получава ***изпълнима програма***, която се изпълнява, за да се извърши предвидената обработка на данни.

Около времето на появата на първите терминали се създават и първите ОС с възможности да изпълняват по няколко задачи едновременно. Така вместо да се компилират и изпълняват една след друга перфорирани на карти програми, програмите се въвеждат от терминали в текстови файлове и се компилират веднага. Една многозадачна ОС може да поддържа десетки терминали и така много програмисти могат да работят едновременно на един и същ компютър, като въвеждат програмите си с текстов редактор във файл, записан на магнитен носител и веднага го изпращат за компилиране, свързване и изпълнение. Така не само се уплътнява по-добре времето за транслиране и изпълнение на програми на компютъра, но се създават и редица възможности за облекчаване и интензифициране на труда на програмиста. Не се чака перфориране на програмата и данните, не се чака на опашка за изпълняване на програмата. В рамките на един работен ден програмистът може да извърши стотици корекции в изходния код, да компилира отново програмата, да я свърже и изпълни.

След създаването на програма, която да е работоспособна, не е изключено тя да не дава исканите резултати, т.е. да е логически неправилна. Това се проверява със специално подготвени ***тестови***

данни. Изключително тежкият етап на проверка на програмата и изчистването и от логическите грешки отначало се извършва ръчно. Програмистът чете изходния код и се стреми да намери грешки в логиката на програмирания алгоритъм, което е доста трудно. Тази тежка задача може да бъде улеснена от специална програма – ***дебъгер*** (от англ. debug, буквално „отстраняване на буболечки“, термин приет в информатиката за означаване изчистване на грешки в компютърна програма). Дебъгерите позволяват програмата да се изпълни стъпка по стъпка, да се наблюдават извършваните на всяка стъпка действия и така се намират грешките в логиката и местата в програмата, където са допуснати логически грешки.

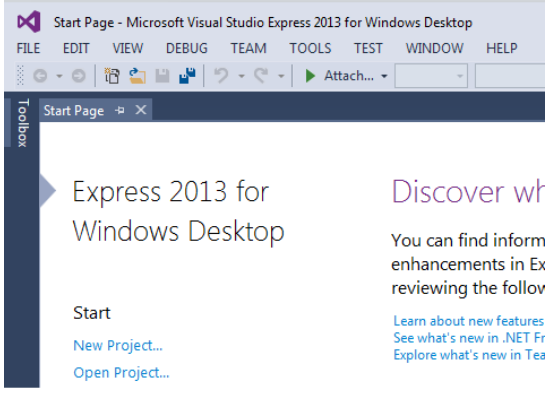
Система за програмиране наричаме съвкупност от инструментални програми, предназначени за създаване на нови програми. Системата за програмиране е основният инструмент на програмистите. Основните компоненти на една система за програмиране са: текстов редактор, компилатор от език за програмиране (или няколко компилатора), библиотека стандартни програми, свързващ редактор, дебъгер и т.н. В наши дни спектърът от инструменти, подпомагащи създаването на програми, включително и много сложни, се развива непрекъснато. Струва си да споменем инструментите за **поддържане на версии**, за **автоматично прекомпилиране** на частите на системата, засегнати от измененията в модул от който зависят, и **автоматично свързване**, програми подпомагащи **документирането**, програми за **автоматизирано генериране на тестови данни** и др.

Интегрирана среда за визуално програмиране

Съвременната тенденция е да се интегрират всички елементи на системата за програмиране, в едно цяло, наричано ***интегрирана среда за програмиране*** (англ. Integrated Development Environment или IDE). Така на програмиста се осигурява средство за бързо и лесно създаване на програми. Много често в рамките на една и съща среда са организирани няколко системи за програмиране, всяка от които е свързана с различен език за програмиране.

Средата Maestro I на немската фирма SoftLab е първата IDE в света. В недалечното минало много използвани бяха интегрираните среди Turbo на фирмата Borland за създаване на програми с буквено-цифров интерфейс на езиците C, C++ и Pascal, които се използват и до днес. В наши дни много популярни са интегрираните среди CodeBlocks (C, C++ и Fortran), Eclipse (поддържаща повече от 20 езика) и др.

Специално място при интегрираните среди за програмиране имат тези от тях, които са предназначени за разработване на програми с графичен интерфейс. В миналото много популярна такава среда беше Delphi на фирмата Borland. Характерно за тези интегрирани среди е използването на ***редактори на графичен интерфейс***, със силно развити библиотеки от графични компоненти. Всички такива среди по принцип поддържат езици за обектно-ориентирано програмиране, тъй като само така може лесно да се използват сложните компютърни обекти, каквито са елементите на графичния интерфейс.



Фиг. 2 Средата Visual Studio Express 2013

Ако използването на буквено-цифрови терминали е бил технологичният скок, благодарение на който са били създадени първите интегрирани среди за програмиране, то технологичният скок, благодарение на който са създадени първите програми с графичен интерфейс и първите среди за създаване на такива програми, е изобретяването на цветните компютърни монитори.

В този учебник ще се запознаем със средата на Microsoft за програмиране на приложения с графичен интерфейс Visual Studio, която поддържа компилатори от няколко езика за програмира-

не, включително езика за ООП C#, на който ще програмираме в процеса на обучение. Илюстрациите в учебника са с безплатно разпространяваната, но ограничена версия на средата, наречена Express 2013. Не е проблем да се използват и други свободно достъпни версии на средата Visual Studio, като Express 2010 или Express 2012, тъй като няма съществени разлики в езика C#, поддържан от тези версии.

Речник

source (code)	сòурс (код)	текст на програма, изходен код
executable	екзекю̀тъбъл	изпълнима програма
environment	инвàйърънмънт	обкръжение, среда (за разработване на софтуер)
development	дивèлъпмънт	развитие; разработване на софтуер

Въпроси и задачи

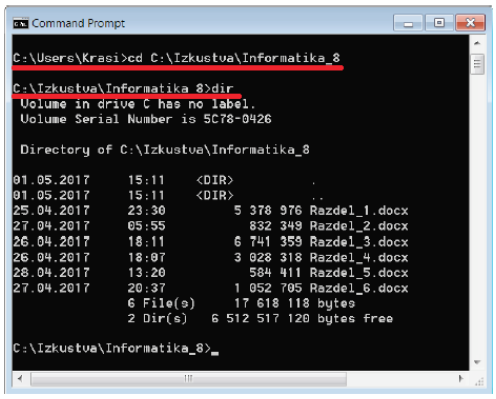
1. *Кое е технологичното решение, довело до създаването на първите интегрирани среди за разработване на софтуер?*
2. *Кое е технологичното решение, довело до създаването на първите интегрирани среди за разработване на софтуер с графичен интерфейс?*
3. *Кое е входно-изходното устройство, освен компютърния монитор, без което създаването и използването на програми с графичен интерфейс е невъзможно? Потърсете информация в интернет за историята на създаването на това устройство.*
4. *Потърсете в интернет информация за някои от най-разпространените среди за програмиране, споменати в урока.*

11 Интегрирана среда за визуално програмиране Visual Studio

В този урок ще се запознаем със средата за програмиране Visual Studio на Microsoft, във версията Express 2013.

Графичен и буквено-цифров интерфейс

За нуждите на изучаваното в следващите уроци е добре, освен графичния интерфейс на ОС Windows да се познава и другата възможна форма на интерфейс с ОС и приложните програми – буквено-цифровият интерфейс**.** При буквено-цифровия интерфейс, както и при графичния, взаимодействието между потребителя и ОС става с подаване на команди и настройване работата на тези команди посредством параметри. При графичния интерфейс подаваме командите с натискане на командни бутони и настройваме параметрите в диалогови прозорци. При буквено-цифровия интерфейс командите са последователности от знаци на клавиатурната азбука, завършващи със знака за нов ред (клавишът Enter). С интервали, командата се разделя на отделни думи.



Фиг. 1. Конзола

Първата дума на командата е нейният идентифициращ ***код***, а останалите думи са параметрите ѝ. Затова командите към ОС много приличат на машинните инструкции и образуват език, който се нарича ***команден език***.

За да подаваме команди към ОС, трябва да отворим специфичен прозорец за буквено-цифров интерфейс, наричан още ***конзола*** (*Фиг. 1*). За целта трябва да натиснем бутона Start и от менюто All programs/Accessories да изберем Command Prompt. На *Фиг. 1* конзолата е показана след изпълнението на две команди (подчертани с червено). Първата команда е C:\Users\Manev>cd C:\Izkustva\Informatika_8. Всеки команден ред започва с името на текущата папка, в случая C:\Users\Manev и знакът >, който означава, че ОС очаква въвеждането на команда. Самата команда е cd C:\Izkustva\Informatika_8. Кодът на командата cd показва, че искаме да сменим текущата папка със зададената като параметър C:\Izkustva\Informatika_8. В резултат на това, както се вижда от следващата команда C:\Izkustva\Informatika_8>dir, текущата папка вече е C:\Izkustva\Informatika_8. Командата dir без аргументи предизвиква извеждане на монитора съдържанието на текущата папка.

Задача. Отворете конзола на компютъра, на който работите, изберете папка от файловата система, преместете се в нея като в текуща и разпечатайте съдържанието ѝ.

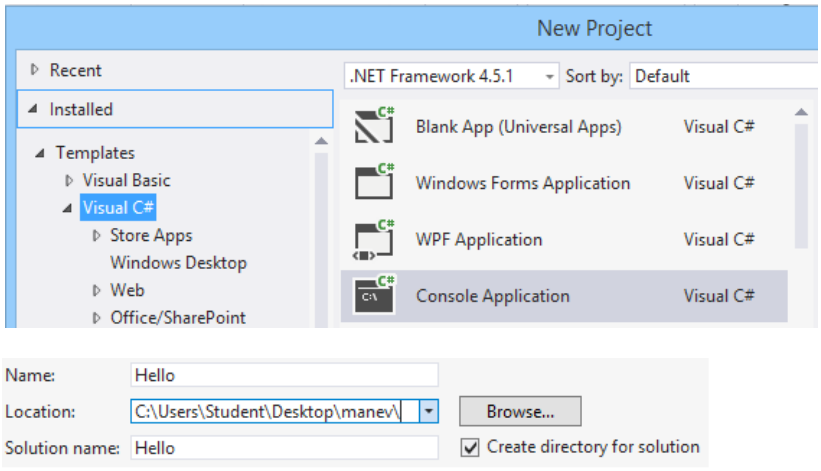
Проект и решение. Конзолно приложение

Важни за работата в средата Microsoft Visual Studio Express 2013 (за да пестим място, от тук нататък ще я наричаме просто ***средата***) са понятията ***проект*** и ***решение***. За създаването на една компютърна програма обикновено са нужни няколко различни файла, създавани от програмиста, както и служебни файлове на средата, с помощта на които се управлява създаването на програмата. Съвкупността от всички тези файлове, се нарича ***проект*** (англ. project).

От файловете на проекта, след подходяща обработка, се получава резултат, приложна програма, който се нарича ***решение*** (англ. solution). Решението може да бъде изпълнима програма, обектен код на подпрограма или цяла библиотека от подпрограми, която да бъде използвана в текущия или други проекти и т.н. Едно решение може да е съставено от няколко различни проекта. Най-простото за създаване решение е изпълнима програма с буквено цифров интерфейс – ***конзолно приложение*** (англ. console application). Затова ще започнем със създаване на такова решение.

Работа с компютър

Стартирайте средата. В стартовия прозорец, непосредствено под лентата с инструменти, щракнете върху текста New Project. След това от диалоговия прозорец изберете Console Application (*Фиг. 2*). След като вече сте в прозореца на програмата, същата команда може да изпълните, като изберете от менюто File подменюто New Project и от него Console Application. Преминаването през няколко менюта докато стигнем до необходимата ни команда, за по-кратко, означаваме с имената на всички менюта, през които преминаваме, разделени със знака / и завършваме с командата – в случая File/New Project/Console Application. В полето Name въведете име на проекта, който ще създадем, например Hello. В диалоговия прозорец на командата може да зададем и мястото на проекта във файловата система (текстовата кутия Location), както и да променим избраното от средата име на решението, което ще създадем (текстовата кутия Solution Name). По подразбиране средата ще постави там името на про-



Фиг. 2. Създаване на конзолно приложение

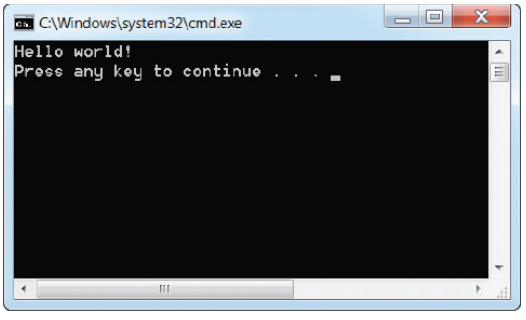
екта. В кутията за избор Create directory for solution по подразбиране е поставена отметка, затова програмата създава папка с името на проекта и новосъздаденият проект ще се запази в тази папка със същото име. В тази папка ще се съхранят и всички други файлове, които се създават автоматично. Това води до по-добра организация на твърдия диск.

След като завършите с настройките, натиснете бутона OK. В резултат средата отваря няколко нови прозореца, с предназначението на които ще се запознаем постепенно. Този, който най-много ни интересува вляво на екрана е прозорецът на редактора, в който средата е приготвила текст за първата ни програма. Сега вече може да създадем конзолното приложение Hello. Програмата, която ще напишем, ще прави нещо много просто – ще отвори конзолния прозорец и ще изпише в него текста `"Hello world!"` – традиционното поздравление на всеки начинаещ програмист към света. За целта добавете текста `Console.WriteLine("Hello world!");` както е показано на *Фиг. 3* (оцветяването на текста по посочения на фигурата начин ще бъде направено автоматично от текстовия редактор на средата). Програмата е готова и трябва да се компилира , т.е. да се преведе на машинен език и да се свърже с необходимите подпрограми – в случая подпрограмата за извеждане на текст в конзолата, за да се получи изпълнимият код.

```
static void Main(string[] args)
{
      Console.WriteLine("Hello world!");
}
```

Фиг. 3.

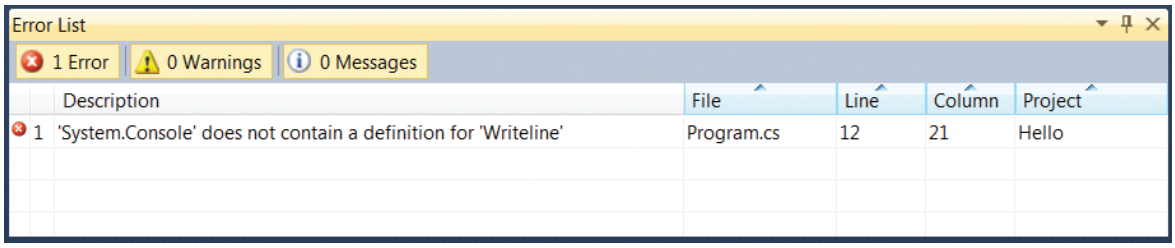
В средата Microsoft Visual Express 2013 създаваме решението с командата Debug/Build Solution или с натискане на клавиша F6. Ако компилирането и свързването е успешно, то в лентата на състоянието, под двата прозореца на средата се появява съобщението Build Succeeded. Командата Build Solution извършва и съхраняване на файла. Това не отменя задължението на програмиста да съхранява отвреме навреме написаното, за да не загуби промените при неочаквани обстоятелства. Можете да разберете кога файлът е бил променян по това, че средата добавя звезда след името – Program.cs*. Стартирайте изпълнимия файл с клавишната комбинация Ctrl+F5. Резултатът от изпълнението е показан в конзолния прозорец на *Фиг. 4*.



Фиг. 4. Изпълнение на конзолна програма

Ако програмата съдържа някакви грешки, компилацията няма да завърши успешно. Променете думата `WriteLine` на `Writeline` и се опи-

тайте да компилирате получения текст. Средата ще отвори прозореца Error List, показан на *Фиг. 5*, в който ще изведе съобщение, описващо допуснатата грешка: 'System.Console' does not contain a definition for 'Writeline', т.е. класът, който управлява работата с конзолата, не притежава метода `Writeline`. Ако щракнете два пъти върху това съобщение, текстовият показалец на редактора се разполага върху реда, който е причинил грешката, а сбърканата дума е подчертана с червено.



Фиг. 5. Прозорец за грешки при компилиране

Направете следната промяна в програмата `Hello` – изтрийте кавичките на текста, който искаме да изведем и се опитайте да създадете отново решение. В прозореца Error List компилаторът ще изведе съобщение за синтактична грешка, което ни подсказва, че текстът, който даваме като аргумент на метода `Console.WriteLine` непременно трябва да бъде поставен в кавички.

```
static void Main(string[] args)
{   Console.WriteLine("What is your name?");
    string s = Console.ReadLine();
    Console.WriteLine("Hello " + s + "!");

}
```

Фиг. 6.

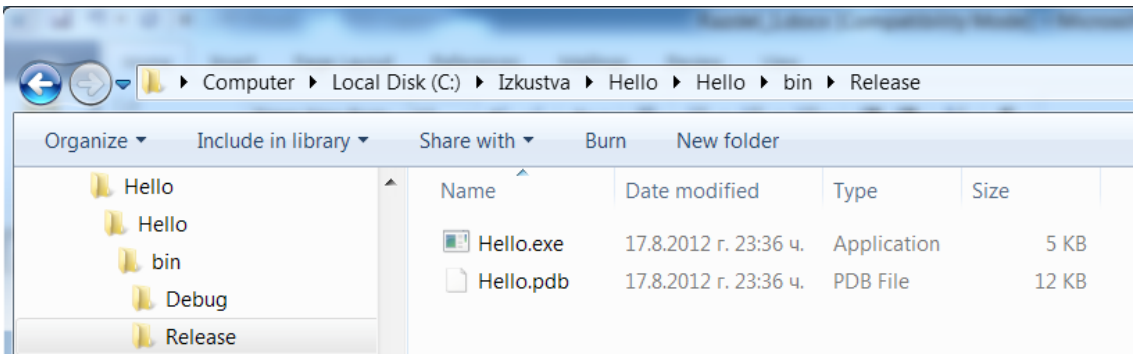
Освен да пишат в конзолата, програмите могат и да четат данни от нея. Да направим конзолно приложение `Hello1`, което при стартиране ни подканва да въведем името си и след това ни отправя поздрав. Промените, които трябва да бъдат направени за целта, са показани на *Фиг. 6*. Създайте приложението и го изпълнете, за да проверите работоспособността му.

Речник

application	апликѐйшън	приложна програма
location	локѐйшън	местоположение
project	прòджект	проект
solution	сълю̀шън	решение

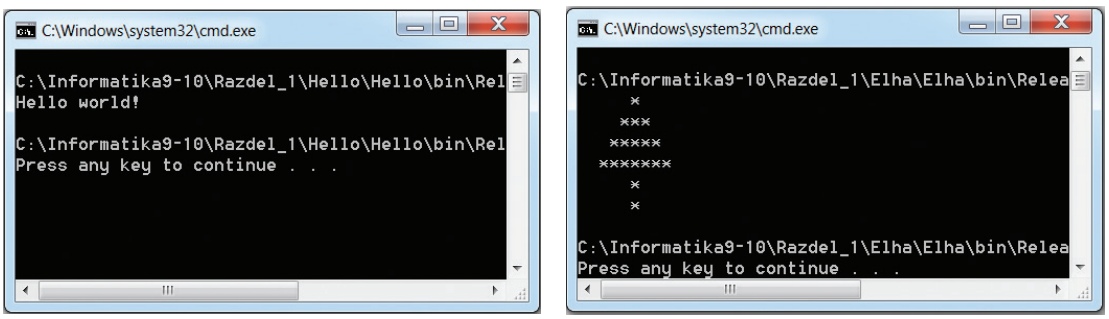
Въпроси и задачи

1. *Посочете основни разлики между буквено-цифровия и графичния интерфейс. При кой от двата вида интерфейс създаването на програми е по-трудоемко?*
2. *Кой от двата вида интерфейс е по-привлекателен за потребителя и защо?*
3. *Намерете във файловата система папката на решението* Hello, *в нея намерете папката на проекта* Hello. *В папката на проекта има папка* bin *с две подпапки* Debug *и* Release. *В папката* Release *ще намерите изпълнимата програма Hello.exe (Фиг. 7). Стартирайте програмата с двойно щракване върху името ѝ. Какво се получи?*
4. *Стартираната в предишното упражнение програма работи за много кратко време и завърши, което доведе до затваряне на конзолния прозорец и невъзможност да се види резултатът от изпълнението. Затова отворете с програмата* Notepad *текстов файл, въведете в*



Фиг. 7. Разполагане на изпълнимата програма във файловата система

него, на първия ред името на изпълнимата програма – Hello, *а на втория –* pause *и съхранете файла под името* Hello.bat *в папката* Release. *Първият от двата реда е команда към ОС да изпълни програмата с име* Hello, *а вторият – да не затваря конзолния прозорец. Щракнете двойно върху името на файла* Hello.bat. *Резултатът ще е подобен на този, който получихме при стартиране на програмата с* Ctrl+F5 *в средата (Фиг. 8.а) но сега конзолата няма да се затвори след като изпълнението на програмата завърши и ще може да видите резултата. Натиснете кой да е клавиш, за да затворите конзолата.*



а. **б.**

Фиг. 8.

5. *Създайте ново конзолно приложение* Elha, *в проект със същото име. Модифицирайте програмата така, че след като я компилирате и изпълните, в конзолния прозорец да се изобрази елхата, показана на Фиг. 8.б.*

12 Програма на C#

Елементи на програмата на C#

Всяка компютърна програма е предназначена да извършва някаква обработка на съхранените в паметта на компютъра данни, посредством множеството от инструкции на централния процесор. В езиците за програмиране данните се представят под фòрмата на ***константи*** – полетата от паметта,

чието съдържание не се мени по време на изпълнение на програмата и ***променливи*** – полета от паметта, чието съдържание се изменя. Една данна може да е много проста – цяло число, дробно число или знак от клавиатурата, но може да е и много сложно структурирана от по-прости данни (спомнете си как един цвят се представя в компютърната памет с тройка цели числа). Затова данните се различават по ***тип***. Типът се определя от **множеството стойности**, които могат да приемат променливите от този тип и множеството операции, присъщи за типа, и се идентифицира с уникално име.

Образ на машинните инструкции в езика за програмиране са ***операторите*** му. Операторите на език за програмиране, в общия случай, са с доста по-сложна структура от машинните инструкции. Един оператор обикновено извършва работата на редица от машинни инструкции. Операторите се групират в ***програмни модули***, носещи различни наименования – подпрограми, процедури, функции и т.н., като всеки модул има определен брой ***аргументи*** от определени типове. Отличаваме модули, които пресмятат резултат от някакъв тип и такива, които не извършват пресмятания или пресмятат много стойности. В езикът C# и едните, и другите, носят името ***функция***.

Всяка функция се идентифицира с ***името*** си, ***броя*** и ***типа на аргументите*** в ***списъка от аргументи***. Списъкът от аргументи на една функция може да е празен, но ако не е, е последователност от двойки „тип – име на променлива“. Имената на функциите, както и другите имена в езика се съставят по правила, с които ще се запознаем в този урок. ***Тялото на функцията*** е съставено от ***описания на данните,*** които тя използва и от ***оператори*** на езика за програмиране, които реализират възложената на програмата обработка. ***Тип на функцията*** е типът на стойността, която тя пресмята. Както вече споменахме, не всяка функция пресмята стойност. Типът на функциите, които не пресмятат стойности, се означава в C# с `void`.

Ще разгледаме два вида решения, написани на езика C# – конзолни приложения, с каквото вече се запознахме в предишен урок и ***приложения с графичен интерфейс***. Всяка програма задължително има поне една функция и точно една ***главна функция***. Това е функцията, която е ***входна точка*** на програмата, т.е. когато стартираме програмата, изпълнението ѝ започва с изпълнение на главната функция. В конзолно приложение на C# името на тази функция е `Main`, а в графично – `Form1`.

Езикът C# е представител на езиците за обектно-ориентирано програмиране (ООП), предназначен предимно за създаване на приложения с графичен интерфейс. Основен елемент в езиците за ООП е класът. Всеки клас е съставен от един или повече екземпляри, обединени от някакви общи свойства, както и програмни модули –наричани в C# методи, с които се извършват обработките.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
namespace ConsoleApplication1
{ class Program
  {
    static void Main(string[] args)
    {   Console.WriteLine("What is your name?");
        string s = Console.ReadLine();
        Console.WriteLine("Hello " + s + "!");
    }
  }
}
```

Фиг. 1.

На *Фиг. 1* е показан изходният код на конзолното приложение, което написахме в края на предния урок. То е съставено от един модул – задължителната главна функция `Main`. Типът ѝ е `void` и има само един аргумент, поставен в малки („кръгли“) скоби след името ѝ. Функцията има три оператора, поставени между две големи („къдрави“) скоби, веднага след списъка с аргументи.

В езика C# всяка функция или данна трябва да е включена в клас. Затова главната функция на нашата програма е включена в клас (`class`) с име `Program` и е негов единствен екземпляр. Класът може да съдържа няколко функции. Не е допустимо в класа да има две функции с еднакво име и еднотипни аргументи. Програмата може да има много класове. Класовете, дефинирани в една програма, се разбиват на ***пространства от имена***, като не е допустимо два елемента, разположени в едно и също пространство с имена, да имат еднакво име. Имената на класовете се оцветяват от редактора в светло синьо.

Пространствата от имена също си имат имена и те трябва да са различни в рамките на една и съща програма. Програмата от примера е разположена в пространство от имена `ConsoleApplication1`. Стандартните класове на езика също са разположени в различни пространства от имена. За да може

програмата да използва стандартен клас, като класа `Console` в нашия пример, тогава името на този клас трябва да бъде указано на програмата в инструкцията `using`.

В нашия пример на програмата са посочени 4 такива пространства от имена. Стандартният клас `Console` се съдържа в служебното пространство от имена на средата `System`. Затова пространството `System` е включено в програмата с директивата `using System;`. Можем да не включим пространството от имена `System` в програмата, но тогава извикването на метода `WriteLine` трябва да изглежда така:

```
System.Console.WriteLine("What is your name?");
```

В автоматично генерирания код са включени и три други пространства, които не са необходими за тази проста програма. Премахнете съответните редове и се убедете в това.

Ключови думи

Езикът C# използва едно множество от думи за свои вътрешни цели. Тези думи наричаме ***служебни*** или ***ключови***. Всяка ключова дума има строго определено предназначение в езика, което не може да се променя. В примера от *Фиг. 1* ключовите думи – `using`, `namespace`, `static`, `void` и `string` са изписаните в ярко синьо.

Ключовите думи на езика C# са 77 и са подредени лексикографски в следната таблица.

abstract	as	base	bool	break
byte	case	catch	cahr	checked
class	const	continue	decimal	default
delegate	do	double	else	enum
event	explicit	extern	false	finally
fixed	float	for	foreach	goto
if	implicit	in	in (genetic)	int
interface	internal	is	lock	lond
namespace	new	null	object	operator
out	out (genetic)	override	params	private
protected	public	readonly	ref	return
sbyte	sealed	short	sizeof	atackalloc
static	string	struct	switch	this
throw	true	try	typeof	uint
ulong	unchecked	unsafe	ushort	using
virtual	void	volatile	white	

Имена (идентификатори)

Вече знаем, че всички елементи на езика и програмите се идентифицират с имената си. ***Имената*** (или ***идентификаторите***) в C#, както и в повечето езици за програмиране, са последователности от латински букви, цифри и знакът за подчертаване (_), започващи с латинска буква или знака за подчертаване. **За имена не могат да се използват ключови думи.** Например, `asd43`, `asd_43`, `a_s_d_43`, `_asd43` и `asd43_` са допустими имена в C#, а `asd-43`, `43asd` и `int` – не са допустими.

В езика C# малката буква и съответната ѝ главна се считат различни. Затова `asd43`, `Asd43` и `ASD43` са различни имена. Добре е да се съобразяваме и със следните препоръки при избор на имена, които са свързани с конвенцията CLS за съвместимост между различните използвани в средите на Microsoft езици:

- не използвайте знака за подчертаване за начало на името – създателите на компилатори използват този знак за специфични цели на компилацията и свързването на компилираните модули в програми;

- не създавайте имена, които се различават едно от друго само по това, че в едното е използвана малка, а в другото съответната главна буква (например `name` и `Name`);
- избирайте имената така, че да подсещат за предназначението на именуваното;
- започвайте името с малка буква;
- ако името е съчетание от няколко думи, то втората и всяка следваща дума да започват с главна буква (например `userName`).

При създаване на обемист програмен код се използват много имена на променливи, функции, методи и класове, което е предпоставка за възникване на конфликти на имена и може да направи кода неясен. Пространство от имена е средството на езика C# за решаването на подобни проблеми.

Коментари

Коментарът е текст в програмния код, който не е част от програмата и не се компилира. Използва се за описване на ролята на променливите, за обяснения на алгоритъма като цяло или на някои от използваните оператори. Понякога като коментар се оформя част от кода, за да не се компилира временно и по този начин може да се тества само останалата част, за да се локализират местата на грешките.

Има два вида коментари:

- коментар на един ред – започва с две наклонени черти (`//`) и продължава до края на реда;
- коментар на повече от един ред – започва със знаците `/*` и завършва с `*/`.

Оформяне на програмата

За компилатора е абсолютно безразлично как ще бъде подреден текстът на програмата, стига да е написан по правилата на езика за програмиране. При въвеждане и редактиране на кода на програмата, обаче, е добре да се спазват няколко общоприети правила, за да стане програмата по-лесна за четене, тестване и внасяне на изменения:

- ако нямате нещо друго предвид, изписвайте всеки оператор на отделен ред, т.е. разполагайте програмата „на дължина“, а не „на ширина“;
- изписвайте всяка затваряща скоба `}` със същото отместване от началото на реда, с което е изписана и съответната ѝ отваряща скоба `{`;
- оформяйте методите и атрибутите на един клас с едно и също отместване от началото на реда;
- при въвеждане на имена на ***членове*** (атрибути и методи) на клас, използвайте менюто с имената на всички членове на класа, което текстовият редактор извежда, когато напишете име на обект от класа и знака точка. Това не само помага за по-бързо писане на кода, а и за избягване на грешките, които се допускат при ръчно писане на имената на членовете;
- използвайте коментари, за да може програмният код да бъде разбираем, както за вас след известно време, така и за други програмисти.

Работа с компютър:

От казаното в този урок става ясно, че програмата Hello, използвана в предния урок, като пример може да бъде опростена доста и, след като ѝ добавим коментари, да добие вида:

```
class Program //програмата трябва да е в клас
{static void Main(string[] args)
{  System.Console.WriteLine("Hello world!"); }}
```

което е минималната форма на конзолно приложение. Редактирайте програмата Hello по указания по-горе начин, компилирайте я и я изпълнете, за да се убедите в работоспособността ѝ. Това не означава, че препоръчваме конзолните приложения да се пишат в този вид – напротив, спазвайте препоръчваното от средата оформяне.

Въпроси и задачи

1. *Разгледайте съдържанието на файла* Program.cs *в програмата* Hello *от предния урок и отбележете ключовите думи в него. В какъв цвят текстовият редактор на средата оцветява ключовите думи?*
2. *Въведете таблицата с ключовите думи в текстов документ. Срещу всяка ключова дума, която сме споменали отбележете предназначението ѝ. Продължавайте да отбелязвате предназначението на всяка ключова дума, щом споменем за нея в поредния урок.*
3. *Може ли като име на променлива да се използва ключова дума? Защо?*
4. *Коя според вас е причината, за имената да се препоръчва да напомнят предназначението им в програмата?*
5. *Коя от следните думи е правилно име в C#:* `Abc`, `клас`, `klas`, `1klas`, `byte`, `22`, `s22`*? За всяка дума, която не може да е име, посочете правилото, което е нарушено.*

13. Изграждане на графичния интерфейс

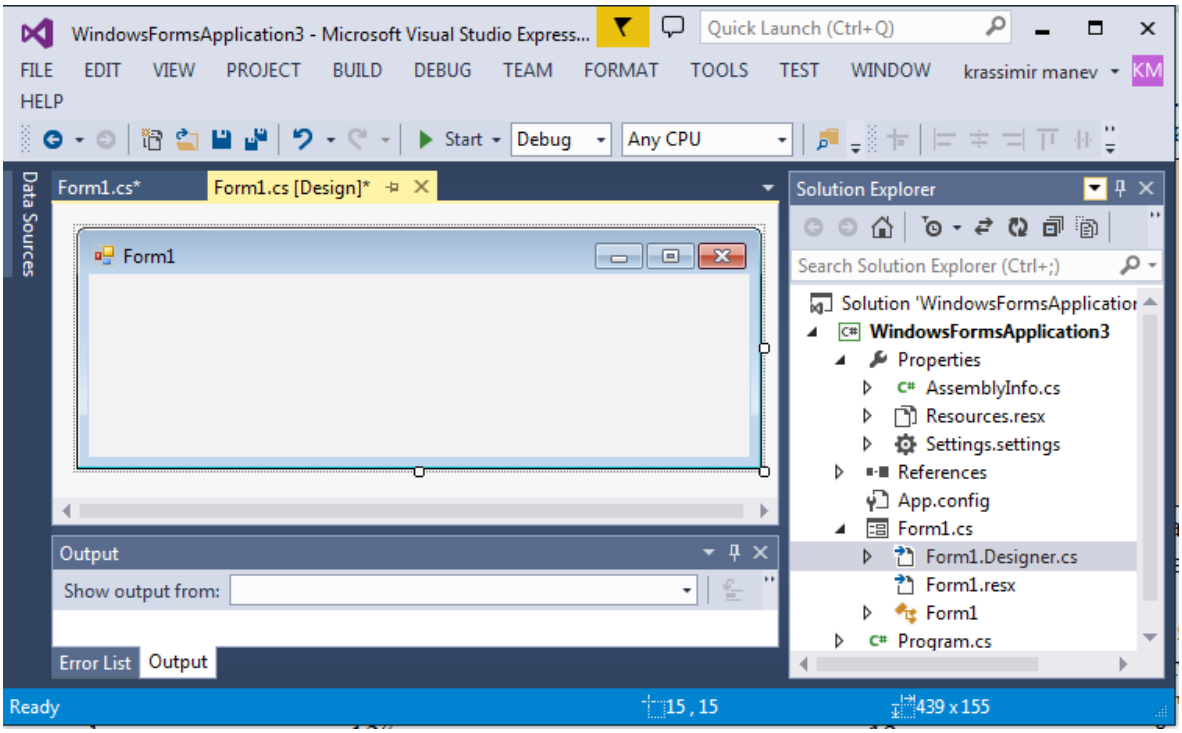
Приложение Windows Forms Application

Вече знаем как се създават конзолни приложения. Време е да се научим да създаваме и приложения с графичен потребителски интерфейс, който днес се счита стандартен за приложните програми. Първите стъпки са подобни на тези, които правим при създаване на конзолно приложение. Стартираме средата и отваряме нов проект. В менюто за вида на приложението, обаче, посочваме Windows Forms Application. Както се вижда на *Фиг. 1*, работното поле на средата вече изглежда по друг начин. Към познатите ни прозорци в списъка на Solution Explorer са се добавили новите Form1.cs и Form1.cs [Design]. Както и при конзолните приложения, съхраняваме новосъздаваното решение под подходящо име, например HelloForm, в избрана от нас папка с командата File/Save All.

Графичен интерфейс

При стартирането на средата за създаване на приложение с графичен интерфейс, в работното поле се отваря прозорецът Form1.cs [Design]. В него е показан графичният елемент ***екранна форма*** (или просто ***форма***), от който се създава основният прозорец на приложението (*Фиг. 1*). Фòрмата е единствен обект на клас, с избрано от средата име `Form1`, който е ***наследник*** на класа `Form`. В ООП, когато един клас наследява друг, това означава, че той получава наготово всички методи на класа родител и в него може да се добавят липсващи в класа-родител възможности. Това е същността на програмирането на приложения с графичен интерфейс – трудните неща са направени в родителския клас, а за неопитния програмист остава да програмира нещо по-просто – функционалността, която иска да реализира.

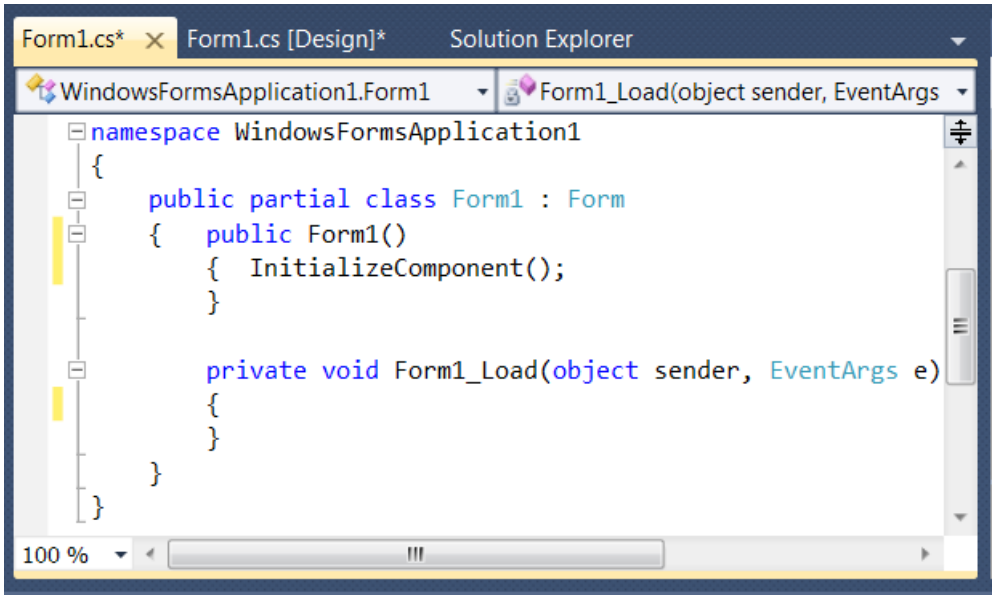
От класа-родител в класа `Form1` се наследяват двата файла с имена `Form1.cs` и `Form1.Designer.cs`. В тях се съхраняват двата изгледа към създавания графичен интерфейс: за програмите и за дизайна, съответно. Дизайнерският изглед (прозорецът `Form1.cs [Design]`, показан на *Фиг. 1*) е предназначен за създаване на графичния интерфейс. Върху изобразената в изгледа форма се разполагат необходимите за работата на приложението елементи на графичния интерфейс. Класът `Form1` наследява и редица характеристики, наричани атрибути или свойства, с променянето на които можем да дадем на фòрмата желания от нас вид.



Фиг.1. Създаване на приложение с графичен интерфейс (Windows Forms Application)

Преминаването от дизайнерския изглед в изгледа с програмите става с натискане на клавиша F7, а преминаването в обратната посока – с клавишната комбинация Shift+F7. В програмния изглед (*Фиг. 2*) средата показва заготовка на програмния код, който реализира функционалността на приложението и където ще изписваме нужния ни код.

Още програмен код има във файловете Form1.Designer.cs и Program.cs. Това са програмните фрагменти, наследени от класа `Form`, които създават и извеждат фòрмата при стартиране на приложението, създават и разполагат елементите във фòрмата и т.н. Този код се създава и управлява от средата,

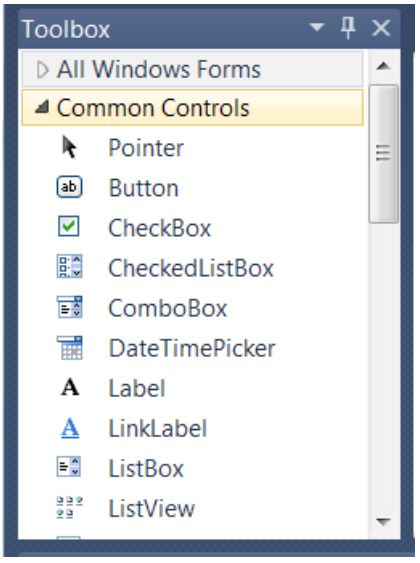


Фиг. 2.

обновява се автоматично, когато се добавят компоненти във фòрмата и се променят свойствата им. Не е желателно да правите изменения в тези файлове засега.

Прозорецът Toolbox

Следващият инструмент на средата, който не сме използвали до момента, е ***кутията с елементи на графичния интерфейс*** Toolbox (англ. кутия с инструменти), които за кратко наричаме ***компоненти*** (*Фиг. 3*) или ***контроли***. Прозорецът Toolbox се отваря с едноименния бутон ( ) или с командата View/Other Windows/Toolbox. По премълчаване средата го поставя неподвижен в лявата част на работното поле. Ако програмистът желае, може да го направи преместваем, като отвори менюто Windows Position с бутона стрелка ( ), намиращ се в горната дясна част на прозореца и избере от него Float (англ. плаващ).



Фиг. 3. Прозорецът Toolbox

Кутията Toolbox съдържа всички компоненти на графичния интерфейс, поддържани от езика C#. С част от тези компоненти и начинът, по който се вграждат в графичния интерфейс на приложните програми, ще се запознаем в следващите няколко урока. В стила на визуалното програмиране най-естественият начин да се постави компонента в екранната форма е да се „хване“ с мишката съответната икона в прозореца Toolbox и да се „влачи“ до желаното място във фòрмата.

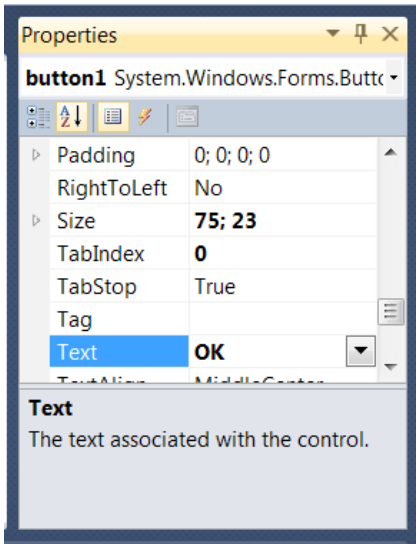
Прозорецът Properties

Последният елемент на средата, на който ще обърнем повече внимание в този урок, е прозорецът Properties (англ. свойства), за който вече говорихме в началото на урока. Средата, обикновено, отваря този прозорец неподвижен, вдясно от основния, но при желание на програмиста и той може да бъде направен преместваем, също както прозорецът Toolbox. Да разгледаме по-подробно този важен за програмирането на графичен интерфейс прозорец.

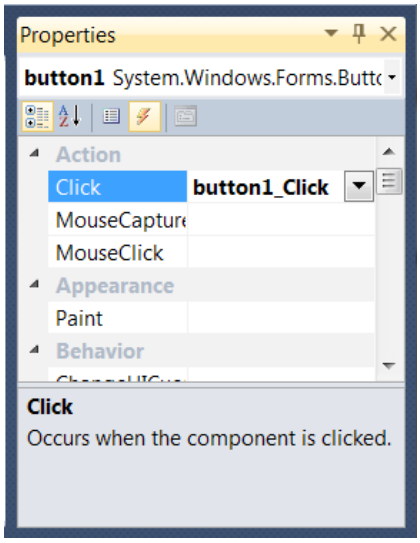
Вече стана дума, че всички обекти на даден клас, включително екранната форма и компонентите, притежават различни ***свойства***, с настройването на които програмистът изгражда интерфейса на приложението. Особеност на класа Form и класовете от компоненти е, че освен свойства, с обектите от тези класове се свързват и ***събития***, които могат да се случат по време на работата на програмата.

Всеки клас има специфични събития, но има и много събития, които се срещат в повечето класове. Такива са, например, щракването с ляв бутон на мишката върху компонентата (Click) и смяната на съдържанието при компоненти, в които може да се пише (TextChanged). Когато щракнем върху компонента, поставена във фòрмата, можем да видим всичките ѝ свойства и събития в показания на *Фиг. 4* прозорец Properties.

Под лентата с името на прозореца е разположена комбинирана текстова кутия, от списъка на която може да се избере обектът, чиито свойства и събития искаме да видим. Под тази кутия има лента с бутони. По премълчаване прозорецът Properties показва свойствата на обекта, подредени в лексикографски ред на имената им (както са показани свойствата на *Фиг. 4*). За да видим свързаните с обекта събития, трябва да натиснем бутона Events (англ. събития). Връщането към списъка със свойствата става с бутона Properties . Освен в лексикографски ред на имената, свойствата и събитията могат да бъдат показани групирани по категории (както са показани събитията на *Фиг. 5*), с



Фиг. 4. Свойства (в лексикографски ред)



Фиг. 5. Събития (по категории)

натискане на бутона Categorized (англ. категоризирани, групирани в категории). Връщането към лексикографска наредба става с бутона . Когато маркирате свойство или събитие в лентата, в долната част на прозореца се описва предназначението на избраното свойство или събитие.

За всяко свойство и събитие в прозореца е отделен един ред с две полета. В лявото поле е изписано името на свойството или събитието. В дясното поле на свойство изписваме стойността на свойството. В дясното поле на събитие, което искаме да обработваме в програмата, се изписва името на съответната функция, която обработва събитието.

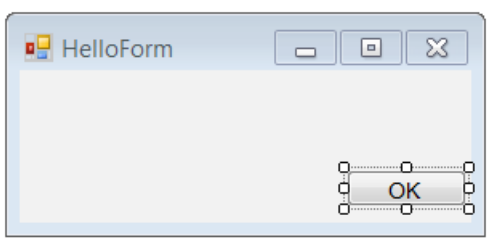
Работа с компютър

Стартирайте средата, отворете ново приложение с графичен интерфейс и го съхранете под името HelloForm, както е указано в началото на урока. След това изпълнете следните задачи:

1. От прозореца Properties променете свойството Text на фòрмата в HelloForm, а размерите ѝ на 300,140.

2. От кутията с компоненти, с хващане и влачене, поставете в долния десен ъгъл на фòрмата бутон. От прозореца Properties променете надписа на бутона (свойството Text) на OK (*Фиг. 6*).

3. С двойно щракване върху бутона отворете прозореца с програмния код, където средата автоматично е добавила функцията `private void button1_Click()`, предназначена да обработи събитието `Click` – щракване върху бутона. Напишете в тялото на функцията оператора `Form.ActiveForm.Close();`, който при щракване на бутона предизвиква затваряне на прозореца и с това прекратява изпълнението на програмата. Компилирайте получената програма и я изпълнете. На екрана ще се отвори създаденият от програмата прозорец. Прекратете изпълнението, като щракнете върху бутона OK.



Фиг. 6.

Въпроси и задачи

1. *Сравнете размерите в сантиметри на фòрмата* HelloForm*, такива каквито се изобразяват на екрана при стартиране на програмата, с размерите, зададени в свойството* Size *и определете колко екранни пиксела съответстват на 1 сантиметър от екрана на вашия компютър.*
2. *Навярно сте забелязали, че и в прозореца* Toolbox *има групиране на компонентите –* Common Controls *(англ. компоненти с общо предназначение),* Containers *(англ. контейнери; компоненти, които съдържат други компоненти),* Menus&Toolbars *и т.н. Групите в двата прозореца са оформени като папки, които се отварят и затварят, както папките във файловата система – с щракване върху триъгълния знак вляво. Отворете всяка от трите групи, споменати по-горе и потърсете компоненти, които са ви познати от уроците по ИТ.* ***Упътване.*** *Ако не разпознавате някоя компонента по иконата или името ѝ, включете я в екранна форма, за да видите как изглежда, когато е в прозорец на програма.*

14. Свойства и методи на графичните компоненти

Общи свойства

В този урок ще се запознаем по-подробно с най-често използваните за създаване на графичен интерфейс компоненти, като посочим основните им свойства и събитията, които обикновено свързваме с тях. Има няколко свойства, които са присъщи на всички компоненти, независимо от вида им. Затова да разгледаме първо тези, общи за всички компоненти свойства:

- Name – всяка компонента, участваща в интерфейса на едно приложение, има уникално име. Както видяхме при екранните форми това свойство винаги се показва най-горе в прозореца. За всяка включена във фòрмата компонента, средата създава служебно име от името на класа, като променя първата буква от главна в малка и добавя уникален пореден номер. Например, button1, button2 и т.н. Програмистът може да промени името, за да избере такова, което подсказва предназначението на компонентата.
- Visible – когато стойността на това свойството е `true`, компонентата е видима, т.е. изобразява се в прозореца, а ако е `false` – не се изобразява. Със задаване на желана стойност на това свойство по време на изпълнение на програмата, програмистът може да променя динамично съдържанието на прозореца, като скрива едни, а показва други компоненти.
- Enabled – промяната на това свойство от `true` във `false` оставя компонентата видима, но я прави недостъпна, т.е. функцията, която тя изпълнява, е блокирана за момента. Смяната от достъпна в недостъпна обикновено се придружава с видимо изменение на изображението на компонентата, например изрисуване в сиво.
- Size – размер на компонентата в пиксели. Свойството е съставно, композирано от две подсвойства – ширина (Width) и височина (Height) на компонентата. Както във файловата система, пред съставните свойства е поставен триъгълния знак, с който се показват или скриват подсвойствата. Стойностите могат да се редактират както в редовете Width и Height, така и в обобщаващия ред Size.
- Location – това свойство също е съставно и съдържа координатите X и Y в пиксели на горния ляв ъгъл на компонентата в съдържащия елемент – форма или друг контейнер. Оста x на екранната координатна система е насочена от ляво надясно, а оста y – отгоре надолу.

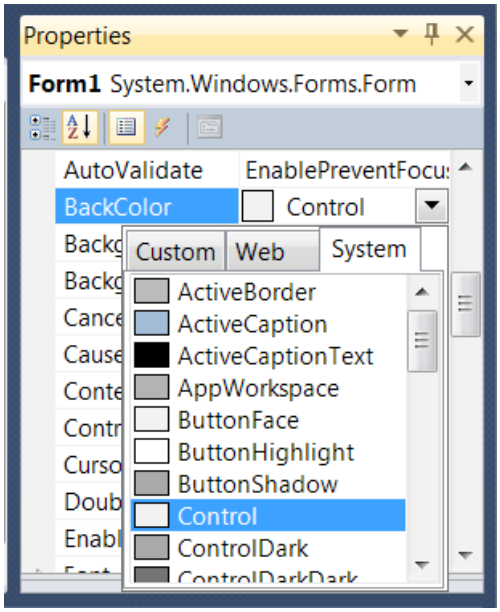
Екранна форма

Вече знаем, че всяко приложение с графичен интерфейс в ОС Windows трябва да има точно един основен прозорец, който се създава като обект от класа `Form`. В него се поставят останалите елементи на главния прозорец на програмата. Празната екранна форма се създава автоматично от средата, само при отваряне на ново приложение с графичен интерфейс. Затова в прозореца Toolbox няма такава компонента.

По премълчаване средата дава на екранната форма името Form1, но без да променя началната буква на името на класа от голяма в малка. За тази компонента свойството Location на практика не съществува, тъй като няма елемент, който да съдържа екранната форма на приложението. Мястото на основния прозорец на програмата върху екрана се задава в свойството StartPosition. По премълчаване средата дава на това свойство стойността WindowsDefaultLocation, т.е. там където ОС прецени за най-подходящо.

По-важни други свойства на фòрмата са:

- Text е свойство, в което се задава заглавието на прозореца, което се изписва в лентата в горния му край. Заглавието най-често съвпада с името на програмата.
- BackColor е цветът на фона на фòрмата, т.е. цветът, в който е оцветена нейната вътрешност. Тъй като цветовете, от които програмистът може да избира са много, дясното поле на свойството, в което трябва да се изпише избраният цвят, е комбинирана текстова кутия, в списъка на която са поставени всички възможни цветове (*Фиг. 1*). Такива комбинирани кутии има в десните полета на много свойства. Не само на тези, за които стойностите са много, но и на тези, за които има само две възможности – `true` и `false`.



Фиг. 1.

- ForeColor е цветът, с който по премълчаване се изписват всички текстове на всички включени във фòрмата компоненти. Ако в някоя от компонентите, обаче, в която има текст, се постави друг цвят в нейното свойство ForeColor, то изписването на текста в тази компонента ще стане с този цвят.
- ControlBox активира/деактивира групата от три бутона  , които управляват показването на прозореца. Тези три бутона познаваме от уроците по ИТ. Припомнете си тяхното предназначение.

Останалите компоненти на прозореца на програмата се избират от прозореца Toolbox и се добавят във фòрмата освен с хващане и влачене, така и като щракнем с левия бутон на мишката върху иконата на компонентата и след това щракнем на избраното за компонентата място във фòрмата. При необходимост, чрез хващане и влачене на компонентите или страните им, може да се променят тяхното местоположението или размер.

Етикет

Основното предназначение на компонентите от класа Label (англ. етикет) е да се поставят надписи в основния прозореца на програмата и останалите контейнери. В етикети се поставят различни заглавия на прозорците и поясняващи надписи за предназначението на останалите компоненти. В етикети могат да се извеждат и стойности, които трябва само да се покажат в прозореца, но не бива

да се изменят от потребителя – например резултати от извършените от програмата пресмятания. В прозореца Toolbox етикетът е представен с реда A Label.

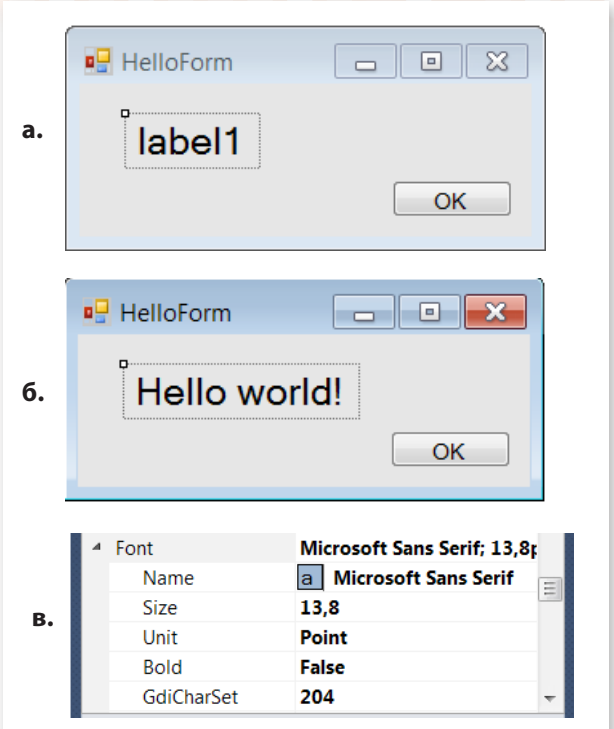
Текстът, който ще се изпише в етикета, се задава в свойството му Text.

При поставяне на етикета в екранната форма, по премълчаване съдържанието на свойството Text съвпада с името на етикета (*Фиг. 2а*).

За да поставим искания от нас текст, трябва да го изпишем в полето за редактиране на свойството (*Фиг. 2б*).

Друго важно свойство на етикета е Font – съставно свойство, в което се задават параметрите на използвания шрифт, с много подсвойства. Най-важните от тях са Name и Size, задаващи вида и размера на шрифта, както и определящите стила – Bold, Italic и Underline (*Фиг. 2в*).

Задача. Променете програмата `HelloForm`, като поставите етикет с надпис, както е показано на *Фиг. 2б*.

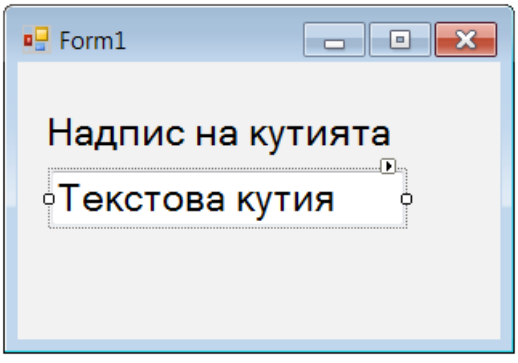


Фиг. 2.

Текстова кутия

Компонентите от класа TextBox (текстова кутия) са предназначени за въвеждане на данни от клавиатурата по време на изпълнение на програмата. Точно това е и основното различие между текстовата кутия и етикета. В останалото, свойствата на тези две компоненти са почти идентични. В прозореца Toolbox текстовата кутия е представена с реда abl TextBox.

Компонентата позволява да се въвеждат данни на един или на много редове, което се управлява от свойството Multiline. След като потребителят е въвел данните в текстовата кутия, те стават достъпни в програмата като съдържание на свойството Text. Като съдържание на това свойство, обаче, по време на проектиране на фòрмата или програмно, може да се постави текст, който ще се покаже при отваряне на прозореца и може да бъде съдържание на кутията по премълчаване или подсказка за потребителя какво да въведе в полето (*Фиг. 3*).

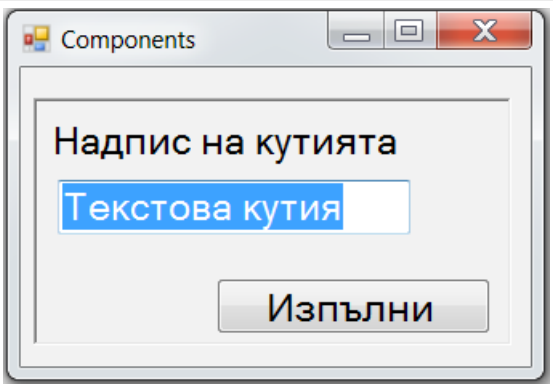


Фиг. 3. Текстова кутия

Компонентите, в които потребителят може да въвежда данни, наричаме активни. Много е важно програмата да „научи“, че в съответната компонента потребителят е въвел данни. Затова, важно събитие за текстовата кутия е TextChanged (англ. текстът е променен). Когато това събитие се случи, програмата би трябвало да го обработи, като съхрани въведеното в кутията в съответна променлива.

Задача. Направете екранната форма, показана на *Фиг. 3*.

Бутон



Фиг. 4.

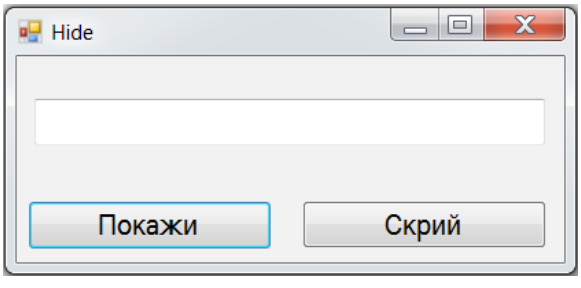
Компонентите от класа Button (бутон) са предназначени за подаване на команди от страна на потребителя към изпълняваната програма, затова са наричани още командни бутони. В прозореца Toolbox бутонът е представен с реда Button. Основно свойство на командния бутон е надписът му, който е стойност на свойството Text и е добре да подсказва командата, която ще се стартира, когато потребителят щракне върху него с мишката (вж. бутона на *Фиг. 4*).

За тази компонента много характерно е и свойството Click, което се генерира в компютъра, когато потребителят натисне и след това бързо отпусне левия бутон на мишката. Затова средата автоматично генерира заготовка на метода, с който това събитие да бъде обработено, когато щракнем върху съответната компонента. В тази функция програмистът изписва програмен код, който изпълнява свързаната с бутона команда.

Панел

Екземплярите от класа Panel (англ. панел) са контейнери, в които се поставят други компоненти, обединени от общо предназначение или близост на функциите им. Характерно за панела е свойство BorderStyle, стойността на което задава вида на рамката. Етикетът, текстовата кутия и бутонът на *Фиг. 4* са събрани в панел, рамката на който е оформена в 3D-стил.

15 Свойства и методи на графичните компоненти – упражнение



Фиг. 1.

Задача 1. Отворете от папка Sources програмата с графичен потребителски интерфейс Hide (файлът Hide.sln). Фòрмата ѝ съдържа текстова кутия и два бутона. Единият бутон е с надпис Покажи, а другият – с надпис Скрий (*Фиг. 1*). При натискане на бутона Покажи в текстовата кутия трябва да се изпише низът `Hello World!`, а при натискане на бутон Скрий, текстовата кутия трябва да се скрива.

Разгледайте кода на това приложение:

```
namespace Hello_World
{ public partial class Form1 : Form
  { public Form1()
   { InitializeComponent(); }
```

```
    private void button1_Click(object sender, EventArgs e)
    { textBox1.Text = "Hello World!";
      textBox1.Visible = true;
    }
    private void button2_Click(object sender, EventArgs e)
    { textBox1.Visible = false; }
  }
}
```

Двете функциите button1_Click и button2_Click са свързани със събитието Click на двата бутона. Заготовката на тези функции – заглавния ред с името, списъкът от параметри, които ние все още не използваме и двете къдрави скоби на тялото се създават автоматично, при двукратно щракване върху съответния бутон или при щракване върху събитието Click в прозореца Properties.

За програмиста остава да допише в заготовките програмните фрагменти, които изпълняват това, което поискахме да стане при „натискането“ на всеки от бутоните.

При натискане на бутона с надпис Покажи (button1) трябва да се напише в текстовата кутия (textBox1) програмисткият поздрав:

```
textBox1.Text = "Hello World";    // показва текст в кутията
```

и след това да осигурим, че кутията е видима, защото при натискане на другия бутон тя се скрива:

```
textBox1.Visible = true;     // прави кутията видима
```

При натискане на бутона с надпис Скрий (button2) трябва само да се направи текстовата кутия невидима:

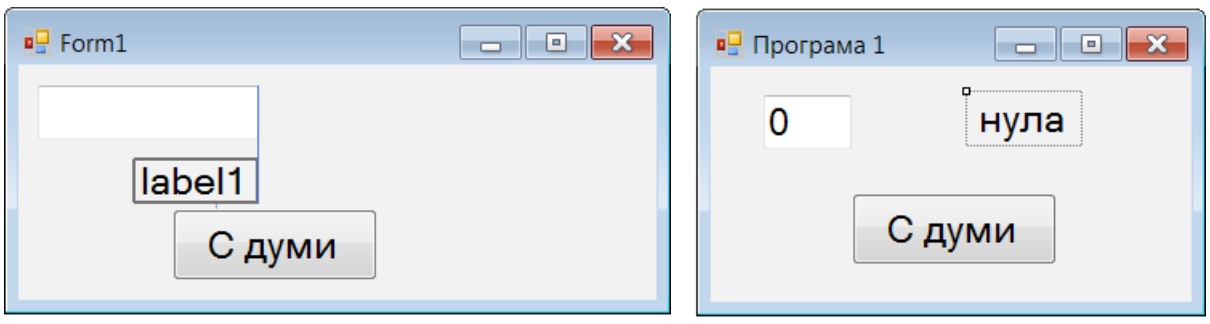
```
textBox1.Visible = false;    // прави кутията видима
```

Компилирайте програмата и я изпълнете, за да проверите работоспособността ѝ. *Забележка.* Макар и с друго предназначение, бутонът F5 също ще стартира програмата.

Задача 2. Напишете програма с графичен интерфейс, която да позволява въвеждане на цифра и след натискане на бутон с надпис С думи да изписва въведената цифра с думи.

От менюто File/New Project изберете Windows Forms Application и изберете име на проекта, например WithWords. Екранната форма се намира в прозореца Form1.cs [Design]. Може да промените размера ѝ по всяко време на проектирането, като хванете с мишката някоя от трите маркирани с бели квадратчета точки по страните на фòрмата и влачите – надолу, надясно или по диагонала.

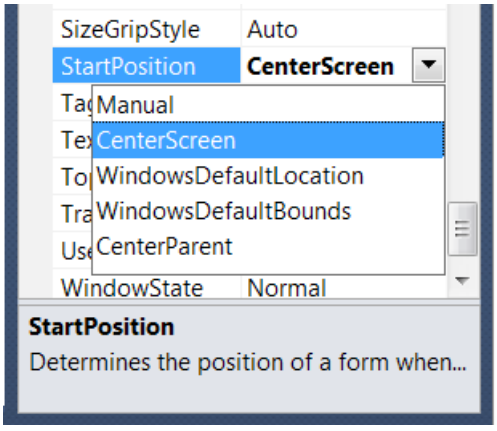
В екранната форма поставете текстова кутия за въвеждане на цифрата, един етикет, в който програмата ще изписва цифрата с думи и един бутон, с който потребителят да стартира изписването. Изберете необходимите компоненти от прозореца Toolbox. Оставете имената на трите компоненти такива, каквито средата ги е поставила – textBox1, label1 и button1. С тези имена ще си служим в указанията по-долу. Разположете трите елемента така, че фòрмата да е добре балансирана. Това не е лека задача. Чувството за добре проектирана и балансирана форма се изработва с много практика. Вляво на *Фиг. 2* е показана една не добре балансирана форма – трите компоненти са струпани в лявата половина на фòрмата, разстоянията между етикета и другите две компоненти са различни,



Фиг. 2.

текстовата кутия е твърде широка за единствената цифра, която ще се въвежда в нея. Фòрмата вдясно на *Фиг. 2* е доста по-добра. По време на местене на компонентите, върху фòрмата се появяват сини линии (виж вляво на *Фиг. 2*). Използвайте тези линии, за да подравнявате компонентите една спрямо друга – хоризонтално и вертикално.

На следващата стъпка от разработване на приложението можете да се заемете с надписите. Всички те се задават в прозореца Properties, в свойството Text на съответните компоненти. Променете свойството Text на фòрмата така, както искате да се нарича програмата. Този надпис трябва да е достатъчно информативен, защото ще се показва в лентата на прозореца и по него потребителят разбира за коя програма става дума. Текстовата кутия и етикетът може да оставите без надписи или пък да сложите цифрата 0 в текстовата кутия и низа нула – в етикета, за да подскажете какво прави програмата (вдясно на *Фиг. 2*).



Фиг. 3.

Позицията на прозореца при стартиране на програмата задаваме в свойството StartPosition на фòрмата. Можете да оставите стойността по премълчаване WindowsDefaultLocation, т.е. ОС да избере подходящо място, където да се отвори прозорецът при стартиране на програмата. Или да посочите своето предпочитание, например CenterScreen, т.е. в средата на екрана (*Фиг. 3*).

Компилирайте програмата и проверете дали при изпълнение фòрмата има вида, който сте очаквали. Затворете прозореца с бутона .

След като стартираме програмата, ще забележим, че тя е готова като интерфейс, но няма нужната функционалност – каквото и да въвеждаме в текстовата кутия, при натискане на бутона С думи съдържанието на етикета остава каквото е било при стартирането. За да получим функционалността, която се иска в задачата, трябва да обработим събитието, свързано с натискане на бутона Click. Програмният код, който ще направи това е:

```
namespace WindowsFormsApplication1
{
  public partial class Form1 : Form
  {
    public Form1()
    {
      InitializeComponent();
    }

    private void label1_Click(object sender, EventArgs e)
    {

    }

    private void button1_Click(object sender, EventArgs e)
    {
      label1.Font = new Font(label1.Font, FontStyle.Regular);
      if (textBox1.Text == "0") label1.Text = "нула";
      else if (textBox1.Text == "1") label1.Text = "едно";
      else if (textBox1.Text == "2") label1.Text = "две";
      else if (textBox1.Text == "3") label1.Text = "три";
      else if (textBox1.Text == "4") label1.Text = "четири";
      else if (textBox1.Text == "5") label1.Text = "пет";
      else if (textBox1.Text == "6") label1.Text = "шест";
      else if (textBox1.Text == "7") label1.Text = "седем";
```

```
            else if (textBox1.Text == "8") label1.Text = "осем";
            else if (textBox1.Text == "9") label1.Text = "девет";
            else label1.Text = "Не знам!";

        }
    }
}
```

Активирайте прозореца със събитията на button1 с  и щракнете двукратно в полето отдясно на събитието Click или направо щракнете двукратно върху самия бутон. Ще се отвори прозорецът за въвеждане на програмния код (Form1.cs). В него средата C# е включила автоматично заготовка на метода за обработване на събитието:

```
private void button1_Click(object sender, EventArgs e) { }
```

В тялото на метода трябва да добавите частта от кода, който проверява съдържанието на TextBox1 и въвежда подходящ текст в Label1. Компилирайте програмата и я тествайте с всяка от цифрите, задавани в произволен ред. Какво става, ако се въведе число, което не е едноцифрено? Опитайте се да обясните действието на оператора. `if...else`.

16. Свойства и методи на графичните компоненти – упражнение

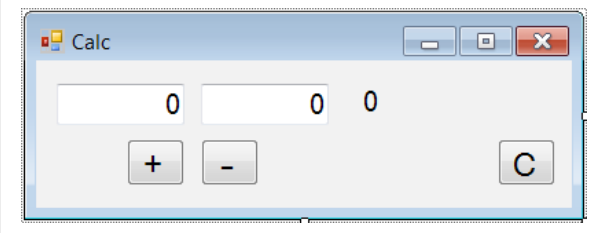
Текстовата кутия е основен елемент на графичния интерфейс и вече видяхме как в нея може да изписваме текстове и да четем текстове оттам. Освен текстове обаче, като входни данни на програма с графичен интерфейс може да се наложи да се задават и числа, а резултатът от програмата може също да е число. За целта трябва да може да превръщаме текст, съдържащ само цифри в число и число – в текст. Целите числа, които искаме да използваме, трябва да бъдат декларирани в програмата с ключовата дума `int`. Превръщането на текста въведен в текстова кутия `textBox1` в цяло число `a` става с оператора:

```
int a = int.Parse(textBox1.Text);
```

а извеждането в текстова кутия `textBox1` или етикет `label1` на цялото число, записано в `int a`, става с операторите:

```
textBox1.Text = a.ToString(); или label1.Text = a.ToString();.
```

Задача 1. Напишете програма `Calc`, която да събира и изважда две цели числа. Числата ще бъдат въвеждани от потребителя в текстови кутии `textBox1` и `textBox2`. Изборът на операция ще става с натискане на един от двата бутона – `button1`, надписан с `+` и `button2`, надписан с `-`. Програмата трябва да изведе в етикета `label1` сбора или разликата на числата, според натиснатия бутон. С бутона `button3`, надписан с `C`, потребителят ще изчиства съдържанието на двете кутии и етикета, когато поиска. На *Фиг. 1* е показан един възможен изглед на прозореца на програмата. Вие може да предпочетете друго подреждане на компонентите.

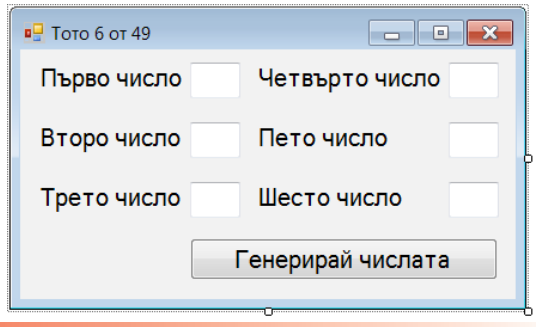


Фиг. 1.

Отворете ново приложение с графичен интерфейс и го озаглавете `Calc`. Поставете компонентите във формата, разположете ги добре в нея и променете размерите ѝ, ако е много малка или много голяма.

Програмата ще трябва да обработи събитието `Click` на всеки от трите бутона. При надписаните с `+` и `–` бутони ще трябва да се превърнат текстовете от двете кутии в числа `a` и `b`, да се извърши избраното аритметично действие, да се превърне полученият в числото `c` резултат в текст и да се запише в етикета. При третия бутон – ще трябва да се изчисти съдържанието на текстовите кутии и етикета, като се изпишат в трите компоненти нули. Получаваме следната програма:

```
using System;
using System.Windows.Forms;
namespace WindowsFormsApplication1
{   public partial class Form1 : Form
    {   int a, b, c;
        public Form1()
        {   InitializeComponent();  }
        private void button1_Click(object sender, EventArgs e)
        {   a = int.Parse(textBox1.Text);
            b = int.Parse(textBox2.Text);
            c = a + b;
            label1.Text = c.ToString();
        }
        private void button2_Click(object sender, EventArgs e)
        {   a = int.Parse(textBox1.Text);
            b = int.Parse(textBox2.Text);
            c = a - b;
            label1.Text = c.ToString();
        }
        private void button3_Click(object sender, EventArgs e)
        {   textBox1.Text = "0";
            textBox2.Text = "0";
            label1.Text = "0";
        }
    }
}
```



Фиг. 2.

Задача 2. Напишете програма `Toto`, която при натискане на бутон в прозореца ѝ ще избира по случаен начин шест цели числа от 1 до 49 и ще ги показва в шест текстови кутии, т.е. симулация на теглене на печелившите числа в тото играта „6 от 49“.

Надяваме се, че изработването на дизайна на приложението (външния вид на прозореца му) вече не е сложна задача. Ще са необходими шест текстови кутии за числата, със съответните шест етикета за надписи и бутон за стартиране на генерирането (*Фиг. 2*).

Надпишете етикетите и бутона, както е показано на фигурата. Подредете компонентите във формата с използването на бутоните за подравняване. Не забравяйте да смените текста на прозореца с такъв, който напомня за предназначението на програмата.

За обработване на събитието `Click` на бутона трябва да се генерират шест случайни цели числа в интервала от 1 до 49, които да се запишат в шестте текстови кутии. Ето как ще направим това генериране. Ще създадем наш метод `GetRandNumber()`, който ще връща цяло число, затова типът му е `int`. Не са необходими входни данни, т.е. списъкът с параметри на метода ще бъде празен. В тялото на метода ще трябва да използваме един клас, който още не познаваме – класа `Random`, предназначен

за генериране на случайни числа. За да започнем генерирането, първо трябва да се създаде обект `r` от този клас с оператора:

```
Random r = new Random();
```

който трябва да поставим в началото на класа `public partial class Form1:Form`.

Самото генериране на случайни числа става с метода `Next()`, който приложен към обект от класа `Random` генерира по случаен начин цяло неотрицателно число. За да ограничим интервала от генерирани числа до [1;49], използваме операцията `x%49`, която дава в резултат остатъка при делението на `x` и `49` – неотрицателно цяло число от 0 до 48. Следователно, генерирането на нужното ни число става с оператора:

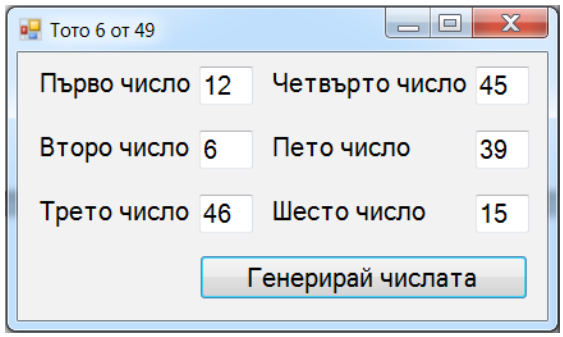
```
int randNumber = r.Next() % 49 + 1;
```

За да може един метод да върне пресметната стойност `X` на мястото, на което е извикан, трябва в края на кода на метода да поставим оператора:

```
return X;
```

Получаваме следния метод:

```
int GetRandNumber()
     {
   return r.Next() % 49 + 1;
}
```



Фиг. 3.

Генерираното число поставяме в текстова кутия с метода `ToString()` за превръщане на число в текст:

```
textBox1.Text = GetRandNumber().ToString();
```

Същото правим и за останалите 5 кутии и обработката на събитието `Click` на бутона ще изглежда така:

```
private void button1_Click(object sender, EventArgs e)
     {  textBox1.Text = GetRandNumber().ToString();
   textBox2.Text = GetRandNumber().ToString();
   textBox3.Text = GetRandNumber().ToString();
   textBox4.Text = GetRandNumber().ToString();
   textBox5.Text = GetRandNumber().ToString();
   textBox6.Text = GetRandNumber().ToString();
}
```

Довършете програмата и я компилирайте. Стартирайте я и натиснете бутона, за да генерира числата (*Фиг. 3*). Натиснете бутона още няколко пъти, като преди всяко натискане изберете 6 ваши числа, за да проверите късмета си. При многократно изпълнение на тази програма може да забележите един недостатък, за избягването на който ни е нужна съответна конструкция на езика C#.

III Програмиране. Основни типове данни

17 Тип низ

Типове данни

Ако можехме да разгледаме едно поле от ОП, щяхме да видим в него само последователност от нули и единици. Каква данна е записана там зависи само от начина, по който ще тълкуваме съдържанието на полето. Начинът, по който тълкуваме записаното в едно поле на паметта, наричаме ***(примитивен) тип*** на данните. Всеки компютър има заложени в себе си няколко примитивни типа данни. Характерно за всеки примитивен тип е ***размерът***, в байтове, на полето от ОП, който всяка данна от този тип заема в паметта, както и ***операциите***, които могат да се изпълняват с данните от този тип. Размерът на типа и начинът, по който се представят данните в полето, еднозначно определят множество от ***възможни стойности*** на типа.

Когато говорихме за представянето на характеристиките на различни обекти като данни в паметта на компютъра, видяхме какво разнообразие в характеристиките на различните обекти съществува. Естествено е, че не можем да очакваме компютърът да предоставя всевъзможни типове данни, които ще ни се наложи да използваме при създаването на компютърни приложения. Затова всеки език за програмиране предвижда механизъм за структуриране на данните за сложни обекти, чрез който от примитивните типове да се създават нови – ***потребителски*** (или ***абстрактни***) ***типове*** данни.

В езика C#, както и в другите езици за програмиране, за обозначаване на примитивните типове се използват ключови думи, а новите потребителски типове се създават като класове от обекти.

Константи и променливи

Всяка конкретна стойност на типа се нарича ***константа***. Всеки език за програмиране има правила за изписването на константите, с който ще се запознаем постепенно.

Данните, необходими за работата на една програма – както входните, така и тези, които се пресмятат по време на работата на програмата, се съхраняват в ***променливи***. Всяка променлива е от някакъв тип и заема съответното количество байтове в паметта. Всяка променлива може да получава стойност само от множество от допустими стойности на типа. За да може една променлива да се използва в програма, тя трябва да бъде ***декларирана***:

Операторът за деклариране на променлива има следния синтаксис:

<име на тип> <име на променлива>`;`

Например, `int x;`, `double pi;`, `string nameAndFamily;` са валидни оператори за деклариране на променливи. Няколко променливи от един и същ тип могат да се декларират с един оператор:

```
int x, y, z;
```

Забележете, че операторът за деклариране на променливи трябва непременно да завършва със знака точка и запетая! Декларирането на променлива означава, че в оперативната памет, там където е разположена програмата, се определя поле с размера, необходим за съхраняване на стойности от посочения в оператора тип. Например, за декларираната променлива `x` от типа `int` ще бъде заделено поле от паметта с размер 4 байта, тъй като това е размерът за този тип. Повече информация за различните типове в езика C# и диапазона на стойностите им ще дадем в следващите няколко урока.

Както и в математиката, така и в програмирането, стойността на една променлива може да е различна във всеки един момент от работата на програмата, откъдето и названието променлива. Затова често говорим не просто за стойността на променливата, а за нейната ***текуща стойност***.

Когато на някое място в програмата поставим **константа** или **име на променлива**, това означава, че на това място искаме да използваме **тяхната стойност**.

Инициализиране на променливи

Преди да използваме една променлива в езика C#, трябва да я инициализираме, т.е. да ѝ зададем начална стойност. Това може да стане или по време на декларирането или в оператор за присвояване, преди променливата да бъде използвана. Ако това не е направено, компилаторът ще изведе съобщението за грешка Use of unassigned local variable <*име_на_променливата*>, а променливата ще бъде подчертана с червена вълнообразна линия.

Инициализирането на променливи може да стане в ***оператора за деклариране*** на променливи

<*име на тип*> <*име_на_променлива*> = <*константа*> ; ,

например `int x=0, y=-1, z=3;` , или по-късно, но непременно преди стойността на променливата да бъде използвана за пръв път, с най-простия вид ***оператор за присвояване на стойност***:

<*име_на_променлива*> = <*константа*>; или <*име на променлива*>=<*променлива*>.

Например:

```
int x, y, z; ... x=0; y=x; z=3;
```

Обърнете внимание на нещо, което вече споменахме в Урока за алгоритми. Ролята на знака за равенство в информатиката е различна от ролята му в математиката. Затова и при четенето на записа тук постъпваме по различен начин. Ако в математиката записът $x = 12$ е означение на факта „x е равно на 12“, то в програмирането записът `x = 12;` е означение на действието „на променливата `x` се присвоява стойност `12`“.

Когато една константа се използва многократно в програмата, по-добре е да ѝ дадем име. Има няколко причини да се даде име на стойност, която няма да се променя в програмата, вместо да се изписва тази стойност навсякъде, където е необходимо. Ако в програмата се налага няколко пъти да изпишем много дълга константа, възможно е да я изпишем погрешно. Освен това, многократното изписване на дълга константа отнема повече време. Най-важната причина е, че ако след време решим да променим стойността на константата, ще бъде изключително трудно да намерим всички места в програмата, където тя се среща. Ако пък в програмата се използват две различни по предназначение константи, които в един момент случайно имат еднаква стойност, тогава на всяка цена трябва да се оформят като именувани константи, защото в противен случай програмата става трудна за четене.

За създаване на ***именувана константа*** използваме оператора:

`const` <*тип*> <*име_на_константата*> = <*константа*> `;`

Например, `const double pi = 3.14159265;` .

Именуваната константа може да бъде използвана в програмата само там, където може да се използва константа, т.е. от дясната страна на оператора за присвояване. Не е възможно да се промени стойността на именуваната константа. При опит да промените стойността на именувана – константа компилаторът ще издаде съобщение за грешка. Освен това, за стойност на именуваната константа не може да се вземе стойността на променлива.

Типовете char и string

В ежедневието много често се налага да работим с текстове. Затова компютърната обработка на текстове е поне толкова важна, колкото и обработката на числови данни. Ако вземем за пример личните данни на един човек ще видим, че по-голямата част от тях са текстови – трите имена; мястото на раждане; името на училището, в което е учил; университетът, който е завършил; фирмите, в които е работил и т.н. За работа с текстове са създадени специални текстообработващи програми, някои от които познаваме от уроците по Информационни технологии. Затова, всеки език за програмиране разполага с възможности за работа с текстови данни. В езика C# това са примитивният тип `char` (***знаци***) и класът от обекти `string` (***низове***). За простота на изложението, класа от обекти `string` ще наричаме също тип.

В езика C# знаците се представят с поредния си номер в двубайтовата таблица Unicode, в която има $2^{16} = 65536$ знака. Стандартът Unicode е създаден в края на 80-те и началото на 90-те години на миналия век, с цел съхраняването на текстови данни на различни езици. Да напомним, че предшественикът му, таблицата ASCII, позволява записването на едва 128 (в 7-битова версия) или 256 знака (в 8-битова версия). За съжаление, това не удовлетворява съвременните изисквания, особено публикуването в интернет, тъй като в 128 позиции могат да се поберат само цифрите, малките и главни латински букви и някои специални знаци, а в 255 – и кирилицата, но това е всичко. Ако се наложи работа с текст на кирилица, латиница и някаква трета азбука, например, 256 знака са крайно недостатъчни. Ето защо все по-често използваме 16-битовата кодова таблица Unicode. Низовете във всеки език за програмиране са последователности от знаци.

Константите от типа `char` се записват, като съответният знак, когато го има на клавиатурата, се постави между два апострофа. Например `'A'`, `'8'` или `'@'`. В противен случай се използва кодът на знака в таблицата Unicode, предшестван от указание за преобразуване `char`, например `(char)1040`. Константните ***низове*** са последователност от знаци от таблицата Unicode, поставени в кавички. Например, `"hello"`, `"Hello"`, `"My name is: "`, `"="` и т.н. Малките и главните букви имат различни кодове, т.е. са различими (низът “`hello`” е различен от низа `"Hello"`). Това обяснява защо е много важно, когато се създава парола, например, да се запомни точно кои от буквите са малки и кои главни.

Декларирането на променливи от тип `char` и `string` става по обичайния начин. Например,

```
char x; string nameAndFamily; char a, b, c;
```

При деклариране на променлива `x` от типа `char` в паметта ще бъде заделено поле с размер 2 байта, тъй като това е размерът за този тип. Количеството памет, заделено за променлива от типа `string`, не зависи от дължината на съхранения в нея низ, защото тя съдържа не самия низ, а адреса на полето от паметта, където той е разположен.

Както вече споменахме, преди да се използва променлива за пръв път, тя трябва да се инициализира. Ако това не е направено, компилаторът ще изведе съобщението за грешка Use of unassigned local variable *<име_на_променливата>*, т.е. Използва се стойността на неинициализираната променлива *<име_на_променливата>*.

Инициализирането на променливи от тип `char` и `string` може да стане с оператор за деклариране:

```
char <променлива от тип char> = <константа>;
string <променлива от тип string> = <константа>;
```

или

```
char  <променлива от тип char > = <променлива от тип char >;
string  <променлива от тип string> = <променлива от тип string >;
```

Например,

```
char x = '*', y = 'a'; char z = x;
string s = "Здравей"; string s = t;
```

Когато създаваме конзолно приложение, за да дадем на потребителя да въведе желан от него низ, използваме метода `Console.ReadLine()`:

```
string s = Console.ReadLine();
```

Когато изпълнението на програмата достигне до извикването на този метод, програмата спира и изчаква потребителя да напише това, което иска и да натисне клавиша Enter. От знаците, въведени от потребителя, методът образува низ. В приложение с графичен интерфейс за инициализиращ низ може да се вземе съдържанието на текстово поле (`textBox1`) или надписа на етикет (`label1`) чрез свойството `Text`:

```
string s1 = textBox1.Text;
string s2 = label1.Text;
```

Извеждането на низ в конзолно приложение става с метода `WriteLine()`, в скобите на който се изписва име на променлива или константа от тип `string`:

```
string s = "Здравей"; Console.WriteLine(s);
Console.WriteLine("Въведи име:");
```

В графично приложение за извеждане на низ най-често се използват интерфейсните компоненти текстово поле и етикет и свойството им `Text` :

```
label1.Text = "My name is:"; textBox1.Text = "Ivan";
```

Естествена операция с низове е ***сливането*** (***конкатенацията***) на два низа. Конкатенацията на низовете α и β е низът $\alpha\beta$. В езика C# конкатенацията на низове е със знак +. Тази операция не е симетрична – има значение кой от операндите е отляво и кой отдясно на знака +. В примера:

```
string s1 = "Иван", s2 = "Борис";
string s3 = s1 + s2;
string s4 = s2 + s1;
```

низовете `s3` и `s4` получават съответно стойностите `"ИванБорис"` и `"БорисИван"`, които очевидно са различни.

Могат да се сливат и повече от два низа, като между всеки два от тях се слага знакът +. Например, ако искаме да обединим съдържанието на две текстови полета в трето, но между тях трябва да има за разделител интервал, то трябва да добавим и низ с интервал `" "` в конкатенацията:

```
textBox3.Text = textBox1.Text + " " + textBox2.Text;
```

В изрази с низове могат да участват и числови стойности, тъй като всяка числова стойност може да се представи с десетични цифри, точка и знаците плюс и минус. Когато е включено в израз с низове, всяко число се преобразува в низа, който е десетичното му представяне:

```
string s = "Иван е на " + 14 + " години";
```

Константната стойност на низа `s` е `"Иван е на 14 години";`.

Дължина на низ и достъп до знаците

Всеки от знаците в низ е стойност от типа `char` и броят на знаците в низ е неговата ***дължина***. Дължината на низа е свойство на класа `string` с името `Length`. Така например дължините на двете константи от тип низ в примера по-горе са: 10 на низа `"Иван е на "` и 7 на низа `" години"`. А дължината на получения в резултат от конкатенацията низ е `s.Length` = 19.

Всеки знак в низа се определя от ***позицията*** (***или индекса***) на знака – цяло число между 0 и `s.Length - 1`. Така, ако `s.Length` е `n`, то `s[0]`, `s[1]`, ..., `s[n-1]` са променливи от тип `char`, всяка от които съдържа поредния знак на низа. Например, за низа `s2 = "Здравей "` `s[0]` е `'З'`, `s[1]` е `'д'`, и т.н. , а `s[7]` е `' '`. Обърнете внимание на една важна особеност: посредством индексиране на променливата от тип `string` може да се използват отделните знаци на низа, но не може да се променят! Т.е. опитът да се присвои нова стойност, например `s[7] = '!'` ще доведе до грешка още при компилиране на кода.

Въпроси и задачи

1. *Кои от следните низове са правилни константи от типа* `string`*:*
   а) `"-1524"`; б) `'1''2''3'`; в) `abcde`; г) `"1000000000000`; д) `" "`?

2. *Кои от следните декларации са правилни:*
   а) `string 1s;` б) `char 'a';` в) `char b, c;`
   г) `string "hello";` д) `string x?`
3. *Кои от следните инициализации на променливи са правилни:*
   а) `string s="123";` б) `string a='hello';` в) `string x='1';`
   г) `char c='c';` д) `char s=*;` е) `char z="@";`
4. *Въведете в програма грешни оператори, както са описани в текста и вижте реакцията на компилатора.*

18. Тип низ – продължение

В този урок ще разгледаме някои операции, които се използват при обработката на низове. Те са оформени като методи на класа `string`.

Преминаване към главни и малки букви

За преобразуване на всички знаци на един низ в малки се използва методът `ToLower()`, а за преобразуване на всички букви в главни – методът `ToUpper()`. Например, ако `s = "Георги"`, след изпълнението на оператора `s1 = s.ToLower();` съдържанието на низа `s1` ще бъде `"георги"`. А след изпълнението на оператора `s1 = s.ToUpperr();` съдържанието на низа `s1` ще бъде `"ГЕОРГИ"`.

Търсене на низ в друг низ

Друга често срещана задача при обработката на низове е да се провери дали един низ *p* (наричан ***образец***) се среща в друг низ *t* (наричан ***текст***), т.е. всички знаци на образеца да се срещат в текста един след друг, така както са в образеца. В такъв случай казваме, че *p* е ***подниз*** на *t*. В езика C# за целта се използват методите `IndexOf(<образец>)` и `LastIndexOf(<образец>)`. Приложен към *<текст>*, всеки от тези методи връща или неотрицателно число, показващо от коя позиция *<образец>* се среща в *<текст>* за първи път, като методът `IndexOf` обхожда низа отляво-надясно, а `LastIndexOf` – отдясно-наляво. Ако *<образец>* не е подниз на *<текст>*, тогава и двата метода връщат стойност –1.

Например, ако стойността на променливата `t` от тип `char` е `"алабаланица"`, тогава:

- след изпълнение на `int d = t.IndexOf("ала");` стойността на `d` ще бъде 0;
- след изпълнение на `int d = t.LastIndexOf("ала");` стойността на `d` ще бъде 4;
- след изпълнение на `int d = t.IndexOf("ааа");` и `int d = t.LastIndexOf("ааа");` стойността на `d` ще бъде –1.

Всеки от двата метода може да бъде извикан и с втори аргумент, показващ от коя позиция в низа да започне търсенето. Например, ако

```
string str = "C# Programming Course";
```

тогава `str.IndexOf("r");` ще върне 4, но `str.IndexOf("r",5);` ще върне 7, а `str.IndexOf("r",8);` ще върне 18.

Извличане на част от низ

В практиката често се налага да извличаме подниз от даден низ. За целта трябва да се укаже от коя позиция започва поднизът и неговата дължина или пък от коя позиция започва поднизът и в коя завършва. За целта езикът C# предоставя методът `Substring(<начало>,<дължина>)` с два аргумента – позицията на първия елемент на подниза и дължината му.

Следният програмен фрагмент:

```
string s = "Плевен", s1;
Console.WriteLine(s);
s1 = s.Substring(1, 3);
Console.WriteLine(s1);
```

ще завърши с присвояването на променливата поднизът с дължина 3, започващ в позиция 1 на низа в променливата, т.е. `"лев"` (*Фиг. 1*) и извеждане на този подниз в конзолата.

s	п	л	е	в	е	н
Позиция	0	1	2	3	4	5

Фиг. 1.

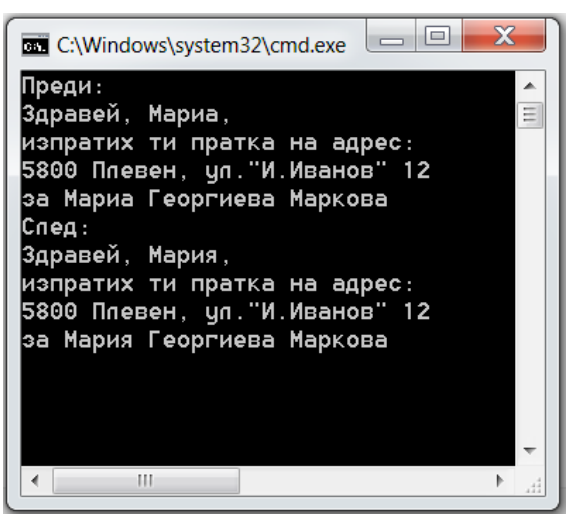
Замяна на подниз с друг

Тази операция също е често срещана в практиката. Например, в дълъг текст е въведено грешно името на човек – Мария вместо Мария, и трябва да бъде заменено всяко грешно срещане с правилното.

Използва се методът `Replace(<търсен низ>,<заменящ низ>)` с два аргумента – низът, който трябва да се замени и низът, с който ще бъде заменен. В резултат, методът построява нов низ, в който всички срещания на *<търсен низ>* в текста се заместват със *<заменящ низ>*. Например:

```
static void Main(string[] args)
{ string s = "Здравей, Мария,\n"+
 "изпратих ти пратка на адрес:\n"+
 "5800 Плевен, ул.\"И.Иванов\" 12\n"+
 "за Мария Георгиева Маркова";
 string s1;
 Console.WriteLine("Преди:");
 Console.WriteLine(s);
 s1 = s.Replace("Мария","Мария");
 Console.WriteLine("След:");
 Console.WriteLine(s1);
 Console.ReadLine();
}
```

Обърнете внимание на факта, че операторът `s.Replace("Мария","Мария")`, сам по себе си не променя претърсвания низ, а изработва нов. Затова построеният от метода низ трябва да се присвои на друга променлива. Забележете също, че се заменят всички поднизове, а не само първият! Резултатът от изпълнение на програмата е показан на *Фиг. 2*. Напомняме, че управляващият знак `\n` предизвиква преминаване на нов ред.

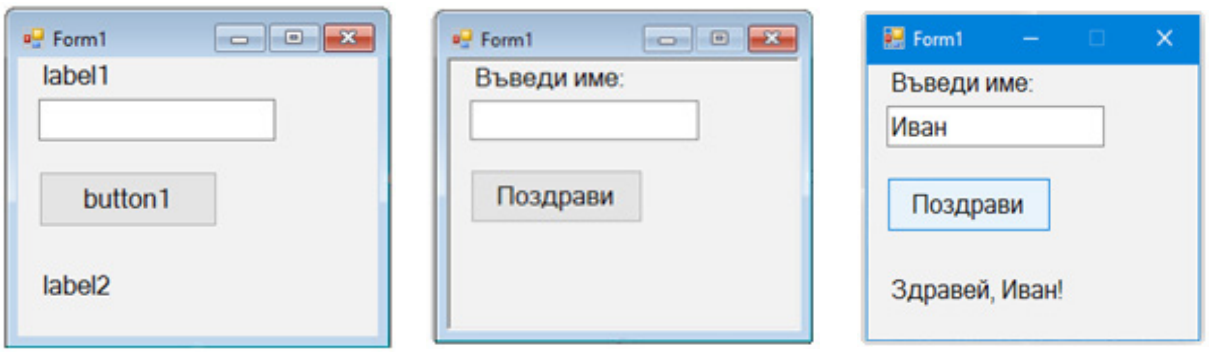


Фиг. 2.

Задача. Напишете малки конзолни приложения, в които да използвате представените в този урок методи.

19 Тип низ – упражнение

Задача 1. Направете графично приложение с име Hello, в което се въвежда име на човек в текстово поле и след натискане на бутон се извежда в етикет поздрав, съдържащ възклицателно изречение, съставено от думите „Здравей“ и въведеното име.



а. б. в.

Фиг. 1.

Решение:

1. Съставяме интерфейса на програмата (*Фиг. 1.а*), който се състои от четири компоненти: текстово поле с име textBox1, два етикета с име label1 и label2 и бутон с име button1.

2. Настройваме компонентите:

а. Забраняваме на формата Form1 да се максимизира и преоразмерява чрез свойствата ѝ MaximizeBox=False и FormBorderStyle=Fixed3D;

б. Променяме надписа на първия етикет на Въведи име: чрез свойството му Text.

в. Изтриваме надписа на втория етикет в свойството му Text.

г. Променяме надписа на бутона на Поздрави чрез свойството му Text.

д. Свързваме програмния код

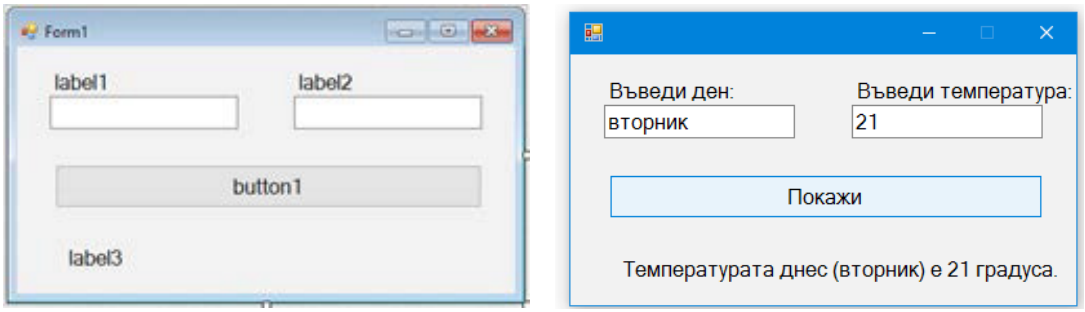
```
private void button1_Click(object sender, EventArgs e)
{
 string name = textBox1.Text;
 label2.Text = "Здравей, " + name + "!";
}
```

със събитието button1_Click, като щракваме двукратно върху компонентата и написваме кода на функцията в отворилия се прозорец.

Стартираме програмата с Ctrl + F5, за да проверим работоспособността ѝ.

Задача 2. Направете графично приложение с име Weather, което се състои от две текстови полета за входните данни на програмата, два етикета за надписи на полетата, бутон и един етикет за резултата от програмата (*Фиг. 2*). Програмата трябва при натискане на бутона да извежда в етикета следното съобщение: „Температурата днес (*<ден от седмицата>*) е *<число>* градуса.“, където денят от седмицата се взима от първото текстово поле, а числото – от второто текстово поле.

Задача 3. Разгледайте кода от *Фиг. 3*. В резултат на изпълнението му, в трите етикета ще се отпечатат три числа. Въведете кода от *Фиг. 3* в графично приложение с три етикета label1, label2 и label3. Компилирайте го и се убедете в работоспособността му.



Фиг. 2.

а) В кой етикет програмата ще изведе отрицателно число? Защо?

б) Колко ще е стойността на `d1`, записана в `label1` след изпълнение на програмата от *Фиг. 3*, ако изменим декларацията на `s` в:

```
s = "e-mail of Ivo Kolev:kolev@kolev.com";
```

в) Колко ще е стойността на `d3`, записана в `label3` след изпълнение на програмата от *Фиг. 3*, ако изменим декларацията на `s` в:

```
s = "e-mail of Ivo Kolev:kolev@kolev.bg\";
```

г) Проверете правилността на отговорите на въпросите от подзадачи а), б) и в), като променяте програмата по указания начин.

д) Променете `s1` така, че стойността на променливата `d1` да е 0 след изпълнението на програмата.

```
public Form1()
{ InitializeComponent();
  string s,s1,s2;
  s = "E-мейлът на Иво Колев е
kolev@kolev.com";
  int i, d1, d2, d3;
  s1="kolev", s2="KOLEV";
  d1 = s.IndexOf(s1);
  d2 = s.LastIndexOf(s1);
  d3 = s.LastIndexOf(s2);
  label1.Text = d1;
  label2.Text = d2;
  label3.Text = d3;
}
```

Фиг. 3.

Въпроси и задачи

1. *Изберете такива стойности за променливите* `p` *и* `q`*, че при изпълнение програмният фрагмент*

```
string s = "Езикът C# е обектно-ориентиран";
s1 = s.Substring(p, q);
label1.Text = s1;
```

да изведе в етикета текста `"C#"`.

2. *След изпълнение на програмния фрагмент*

```
s = "Георги Георгиев";
s1 = s.Replace("Георги", "Петър");
```

стойността на променливата `s1` *ще бъде:*

а) `"Георги Георгиев"` б) `"Петър Георгиев»` в) `"Петър Петърев»`.

3. *Даден е низ, който е пълно име на файл – пътят от корена до съдържащата файла папка и собственото име на файла. Направете графично приложение, в което да се въвежда в текстово поле пълното име и да се извежда в етикет собственото име на файла.*

ПРИМЕР:

Вход	**Изход**
d:/Izkustva/klas_8/Urok_01/Image1.jpg	Image1.jpg.

20 Целочислени типове данни

Освен с текстови данни, много от характеристиките на обектите се описват и с числови данни – цели или реални числа. Например, скорост, височина, дължина, температура, заплата, брой, среден успех, резултат и др. Някои характеристики като ЕГН и номер на GSM също изглеждат числови, но в действителност са текстови данни, защото могат да започват с 0 и ако бъдат съхранени като числови, то ще се загубят водещите нули. За съхранение и обработка на числови данни в компютъра е необходимо всеки език за програмиране да има и числови типове данни.

Целочислени типове

За представянето на целите числа в компютъра езиците за програмиране използват двоична бройна система и по този начин всяко цяло число се представя като последователност от битове, т.е. последователност от нули и единици. Съхраняването на неотрицателните и отрицателните числа се различава по същността на алгоритъма, който ги превръща в двоична бройна система, което налага да има различни типове данни за тях. От друга страна, тъй като голяма част от числата имат различна дължина като брой битове, то би било неефективно всички числа да се съхраняват в еднакъв обем оперативна памет (например човешката възраст в години е много по-малка от броя на населението на която и да е държава). Затова в повечето езици за програмиране има няколко целочислени типове данни, които се различават по големината си и по това дали в тях могат да се съхраняват и отрицателни числа. Тези две характеристики определят и множеството от допустими стойности на типа.

В езика C# има 8 целочислени типа данни, които са показани в таблицата:

Име	Вид	Големина	Множество от допустими стойности
`sbyte`	Знаков	1B	$[-2^7, 2^7 - 1] = [-128, 127]$
`byte`	Беззнаков	1B	$[0, 2^8 - 1] = [0, 255]$
`short`	Знаков	2B	$[-2^{15}, 2^{15} - 1] = [-32768, 32767]$
`ushort`	Беззнаков	2B	$[0, 2^{16} - 1] = [0, 65535]$
`int`	Знаков	4B	$[-2^{31}, 2^{31} - 1] = [-2147483648, 2147483647]$
`uint`	Беззнаков	4B	$[0, 2^{32} - 1] = [0, 4294967295]$
`long`	Знаков	8B	$[-2^{63}, 2^{63} - 1] =$ $[-9223372036854775808, 9223372036854775807]$
`ulong`	Беззнаков	8B	$[0, 2^{64} - 1] = [0, 18446744073709551615]$

Разнообразието от типове дава възможност на програмиста да избира подходящ тип за различните видове характеристики на обектите. В нашите програми ние няма да се възползваме често от тази възможност, защото в началото ще използваме много малко променливи и не е необходимо да се съобразяваме с използваната оперативна памет. Освен това има възможност за допускане на повече грешки при използването им в различни аритметични изрази и методи.

Константи за целочислените типове са целите числа, които в езика C#, както и в математиката, изписваме в десетична бройна система – с последователност от десетични цифри. Например `123`, `+2048`, `007`, `-1734`. Поставянето на водещи нули е допустимо и не променя стойността на числото, както и знакът +, но след съхраняването им в някой от целочислените типове се изтриват. Знакът минус може да се добавя в началото само за типовете със знак – `sbyte`, `short`, `int` и `long`.

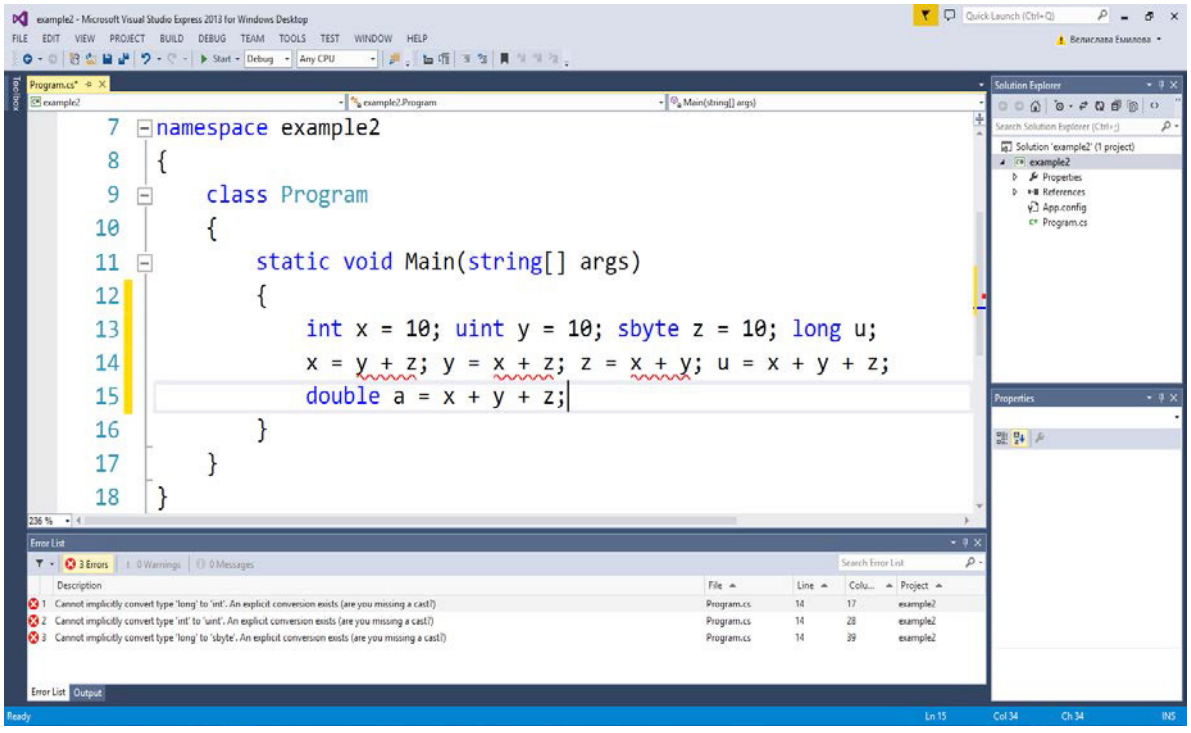
Операции с целочислени данни

С целите числа могат да се извършват няколко аритметични операции, резултатът от които също винаги е цяло число. Това са познатите от математиката операции събиране (+), изваждане (–), умножение (*) , целочислено деление (/) и остатък при целочислено деление (%).

```
int x = 20, y = 3, z = 4;
int ans;
ans = x + y; //Стойността на ans е 23
ans = x - y; //Стойността на ans е 17
ans = x * y; //Стойността на ans е 60
ans = x / y; //Стойността на ans е 6
ans = x % y; //Стойността на ans е 2
ans = x % z; //Стойността на ans е 0
ans = y / z; //Стойността на ans е 0
ans = y % z; //Стойността на ans е 3
```

Инициализация на целочислени променливи

Инициализация на целочислените променливи можем да направим както и на другите видове променливи – в оператор за деклариране, в оператор за присвояване или чрез въвеждане на стойност (в конзолата или от контрола на графичния интерфейс). При инициализация на променливите в оператор компилаторът следи за това константите, които изписваме отдясно на знака за присвояване, да са измежду допустимите стойности на типа на променливата отляво от знака. Например, да не са твърде големи, да не се присвояват отрицателни стойности на целочислени променливи без знак или пък числа с дробна част на целочислени променливи (*Фиг. 1*).



Фиг. 1. Грешки при инициализация на променливи

Преобразуване на низ в цяло и обратно

Понякога е необходимо да запишем в променлива от целочислен тип число, зададено като низ от цифри. Тогава трябва да използваме, както в предишен урок, вградения метод `Parse(<низ>)` за преобразуване на низ в число, който се намира в класа `int` и преобразува зададения му като аргумент низ в цяло число:

```
int side = int.Parse(textBox1.Text);
```

Разбира се, не всеки низ може да се превърне в цяло число и ако това е невъзможно, програмата генерира грешка по време на изпълнение (***изключение*** от класа `System.FormatException`). Това може да се случи, ако низът съдържа знаци, които не са цифри.

В езика C#, както вече знаем, е предвидена и обратната възможност – да се преобразува цяло число в низ. Това винаги е възможно и става чрез вградения метод `ToString()`, който е част от всеки един клас и може да се използва с всеки обект:

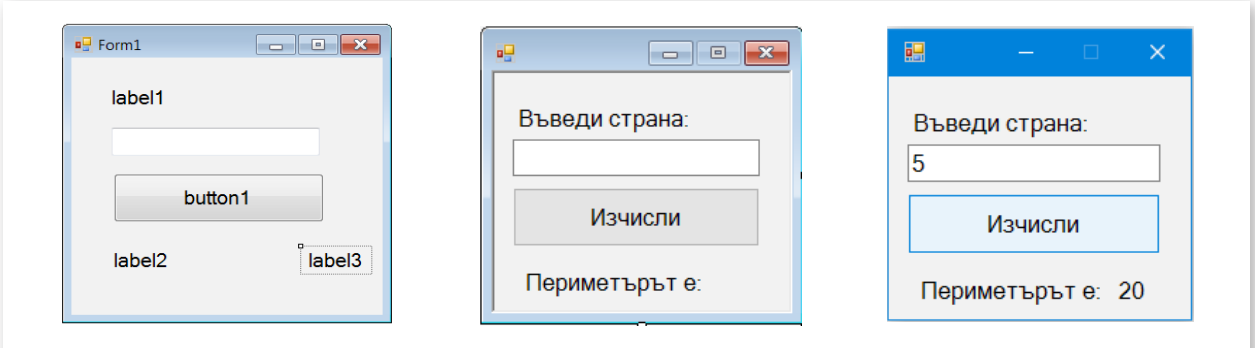
```
int P = 20;
label3.Text = P.ToString();
```

Въпроси и задачи

1. *Кои от следните декларации на променливи са правилни?*
   а) `int a;` б) `int byte x;` в) `string 1s;` г) `byte 'a', b, c;`
2. *Кои от следните низове са правилни константи от типа* `uint`*?*
   а) `-1524` ; б) `123E4` ; в) `000` ; г) `1000000000000` ; д) `1.`

21 Целочислени типове данни – упражнение

Задача. Направете графично приложение с име `Square1`, в което се въвежда страната на един квадрат и се извежда периметърът му, като повторите дизайна на формата от *Фиг. 1*.



Фиг. 1.

1. Съставяме интерфейса на програмата, който се състои от пет компоненти: текстово поле с име `textBox1`, три етикета с име `label1` , `label2` и `label3` и бутон с име `button1`.

2. Извършваме настройки на компонентите:

а) Забраняваме на формата да се максимизира и преоразмерява чрез свойствата ѝ `MaximizeBox=False` и `FormBorderStyle=Fixed3D` и изтриваме надписът ѝ в свойството `Text`.

б) Променяме надписа на `label1` и `label2` съответно на „Въведи страна:“ и „Периметърът е:“ чрез свойството `Text`.

в) Изтриваме надписа на `label3` в свойството му `Text`.

г) Променяме надписа на бутона на `"Изчисли:"` чрез свойството му `Text`.

3. Свързваме програмния код

```
private void button1_Click(object sender, EventArgs e)
{
   int side = int.Parse(textBox1.Text);
    int P = side * 4;
   label3.Text = P.ToString();
}
```

със събитието `button1_Click`, като щракваме двукратно върху компонентата и написваме кода на функцията в отворилия се прозорец.

4. Стартираме програмата с Ctrl + F5, за да проверим работоспособността ѝ.

Въпроси и задачи

1. *Добавете в програмата, която създадохте по време на урока, възможността да изчислява и лицето на квадрата.*
2. *Направете графично приложение с име* `Rectangle`*, в което се въвеждат двете страни на един правоъгълник и се извеждат лицето и периметъра му.*

22 Реални типове данни

Реални типове

При работа с числови данни в информатиката са необходими както цели, така и дробни числа. Вече знаем, че представянето и на дробните числа в паметта на компютъра става в двоична бройна система. Знаем също, че процедурата за превръщане на дробно число от десетичен в двоичен вид не винаги завършва, т.е. има дробни числа, (например `0,1`), които са безкрайни дроби в двоична бройна система. Но тъй като паметта на компютъра е крайна, то такива числа се съхраняват в отделената част от паметта до определен брой цифри след двоичната запетая в двоична бройна система. Това означава, че при взимането им от паметта, за да бъдат използвани в изрази и променливи, стойността им вече ще се различава от първоначалната и може да се загуби точност след определен знак след десетичната запетая в десетична бройна система. Например, ако съхраняваме мантисата на десетичното дробно число `0,1` в 1 байт, т.е. в 8 бита, то стойността ѝ в двоична бройна система ще бъде `0,00011001` (виж *Фиг. 1*).

```
0 , 1 x 2 =
0 | 2 x 2 =
0 | 4 x 2 =
0 | 8 x 2 =
1 | 6 x 2 =
1 | 2 x 2 =
0 | 4 x 2 =
0 | 8 x 2 =
1 | 6
```

Фиг. 1.

Но $0,00001100_{(2)} = 1.2^{-4} + 1.2^{-5} + 1.2^{-8} = 1/16 + 1/32 + 1/256 = 0,0625 + 0,03125 + 0,00390625 = 0,09765625_{(10)} \neq 0,1_{(10)}$.

От този пример става ясно, че при работа с някои дробни числа ще има загуба на точност. Затова е добре дробните типове да са с колкото може повече битове. В повечето езици за програмиране има два реални типа за работа с дробни числа, които използват представяне с плаваща запетая. Има още един реален тип данни, който съхранява дробните числа в десетична бройна система с фиксирана запетая и това дава възможност да е много по-точен. Работата с него е много бавна и затова се използва само в задачи, в които се изисква голяма точност – финансови изчисления и др. В показаната таблица са дадени реалните типове данни в C#:

Име	Големина	Точност	Обхват
`float`	4B	До 7-8 десетични знака	$\pm1,5 \times 10^{-45}$ до $\pm3,4 \times 10^{38}$
`double`	8B	До 15-16 десетични знака	$\pm5 \times 10^{-324}$ до $\pm1,7 \times 10^{308}$
`decimal`	16B	До 28-29 десетични знака	$\pm1 \times 10^{-28}$ до $\pm7,9 \times 10^{28}$.

Основната разлика между реалните типове `float` и `double`, и тези от тип `decimal` е в точността на пресмятане и степента, до която се закръглят пресмятаните стойности. Типът `double` може да работи с много големи стойности и такива много близки до нулата, но за сметка на това точността му е малка поради внесени грешки от закръгляне. Типът `decimal` работи в по-малък обхват, но гарантира много голяма точност.

Константи и инициализация

Константи за реалните типове `float` и `double` са дробните числа, с или без знак, като десетичният знак в C# е точка. Например, `123.2048`, `0.07`, `-1.734`. За дробните константи е допустимо да се представят и в експоненциален вид – с мантиса и десетичен порядък, при това за мантисата не е нужно да се грижим да е подравнена по начина, по който ще се представи в паметта. Мантисата и порядъкът се разделят с голямата латинска буква `E`. Например, дробната константа 3.14 може да се представи в програма на C# като `314E-2` или като `0.314E1`.

В графичната среда десетичният знак за дробните константи зависи от настройките на операционната система и може да бъде или точка, или запетая!

Инициализирането на променлива от реален тип не се различава по същество от инициализирането на целочислените типове и може да бъде извършено както по време на деклариране, така и по-късно чрез оператор за присвояване, но винаги преди да бъде използвана в израз:

```
double x = 3.14;
```

или

```
double x;
x = 3.14;
```

На променлива от реален тип може да се присвоява целочислена стойност, тъй като множеството на целите числа е част от множеството на реалните числа и всяко цяло число може да бъде представено като дробно, но обратното не е възможно – на променлива от целочислен тип не може да се присвои дробна константа. Затова `double x = 3;` или `double x = 3.0;` е допустимо и по същество е едно и също – числото 3 ще бъде съхранено като дробно число (по съответния алгоритъм), но `int x = 3.14;` е синтактична грешка!

Преобразуване на низ в дробно число и обратно

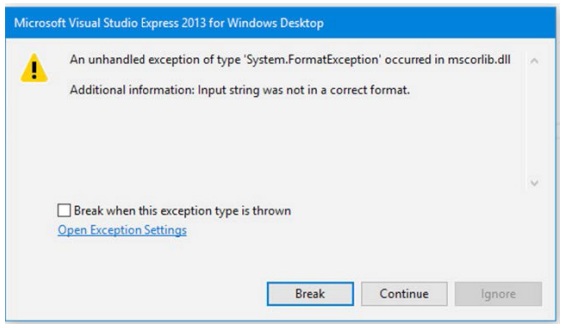
Ако е необходимо, можем да запишем в променлива от реален тип число, зададено в низ. Трябва да използваме вградения метод `Parse(`*<низ>*`)` за преобразуване на низ в число, който се намира в класа `double` и преобразува зададения му като аргумент низ в дробно число:

```
double x = double.Parse(textBox1.Text);
```

Разбира се, не всеки низ може да се превърне в дробно число и ако това е невъзможно, програмата генерира грешка по време на изпълнение (изключение от класа `System.FormatException`, виж *Фиг. 2*).

В езика C# е предвидена и обратната възможност – да се преобразува дробно число в низ. Това винаги е възможно и става чрез вградения метод `ToString()`, който е част от всеки един клас и може да се използва с всеки обект:

```
double x = 3.14;
label3.Text = x.ToString();
```



Фиг. 2.

Аритметични операции с дробни числа

С дробните числа също могат да се извършват аритметичните операции събиране (+), изваждане (–), умножение (*) и деление (/), като ако поне един от операндите е от реален тип, то резултатът също е от реален тип:

```
double x = 20, y = 3.5, z = 4;
double ans;
ans = x + y; //Стойността на ans е 23.5
ans = x - y; //Стойността на ans е 16.5
ans = x * y; //Стойността на ans е 70.0
ans = x / y; //Стойността на ans е 5.71428571428571
ans = y / z; //Стойността на ans е 0.875
ans = x / z; //Стойността на ans е 5.0
ans = x / 6; //Стойността на ans е 3,33333333333333
int a = 20, b = 6;
ans = a + b; //Стойността на ans е 26.0
ans = a - b; //Стойността на ans е 14.0
ans = a * b; //Стойността на ans е 120.0
ans = a / b; //Стойността на ans е 3.0
ans = b / a; //Стойността на ans е 0.0
int c = x + y;
```

Последният ред предизвиква синтактична грешка: `"Cannot implicitly convert type 'double' to 'int'"`. Това е така, защото на променлива от целочислен тип се опитваме да присвоим дробен резултат, тъй като и `x`, и `y` са от реален тип!

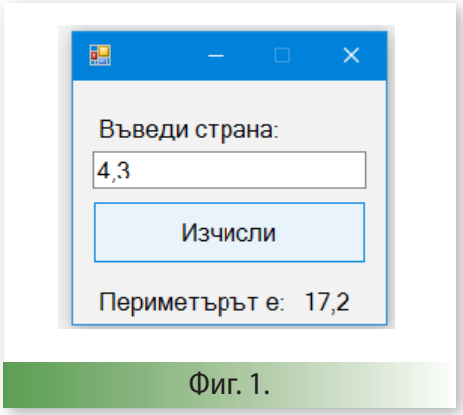
Въпроси и задачи

1. Кои от следните низове са правилни константи от типа `double`*?*

а) `-1524`;
б) `123E4`;
в) `000.`;
г) `.1000000000000`;
д) `1.1`.

23 Реални типове данни – упражнение

Задача 1. Стартирайте графичното приложение с име Square1, което създадохме в предишен урок и въведете за страна на квадрата дробно число (като първо проверите с кой десетичен знак работи вашата операционна система). Какво стана? Редактирайте програмата, така че да може да се въвежда и дробно число по модела, показан на *Фиг. 1*.

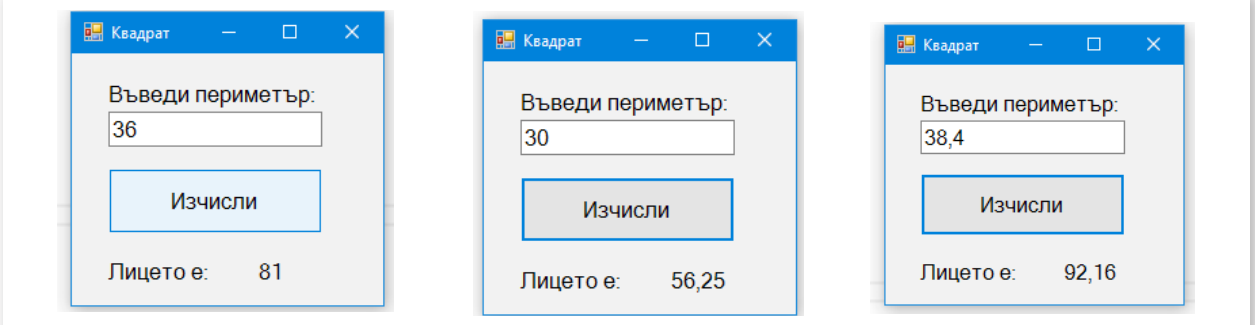


Фиг. 1.

Решение: Програмата прави изключение, т.е. грешка по време на изпълнение, тъй като се опитва да превърне в цяло число текст, в който има знаци, които не са цифри. За да работи и с дробни числа (целите числа са част от дробните!), то трябва да се смени както типа на данните, така и класа на метода Parse в кода на бутона:

```
private void button1_Click(object sender,
EventArgs e)
{
   double side = double.Parse(textBox1.Text);
   double P = side * 4;
   label3.Text = P.ToString();
}
```

Задача 2. Създайте графичното приложение с име Square2 по модела, показан на *Фиг. 2*, в което се въвежда периметър на квадрат и се извежда лицето му. Програмата трябва да може да работи и с дробни числа.



Фиг. 2.

24 Аритметични изрази. Приоритет на операциите

Операции, операнди, знаци на операциите

Както споменахме вече в предишен урок, за всеки тип данни са присъщи и съответни ***операции***. Всяка операция се прилага върху един или няколко ***операнда*** (или ***аргумента***) и връща като ***резултат*** някаква стойност. В зависимост от броя на операндите си, операциите се делят на такива с един операнд (***едноаргументни***), с два операнда (***двуаргументни***), с три операнда (***триаргументни***) и т.н.

Обичайно е за всяка операция да имаме ***знак на операцията*** и да разполагаме аргументите около знака. Така например, за едноаргументните операции имаме две възможности за поставяне на знака:

префиксен запис на операцията: <*знак_на_операцията*> <*операнд*>;
постфиксен запис на операцията: <*операнд*> <*знак_на_операцията*>.

Едноаргументна операция, например, е операцията за обръщане на положително число в отрицателно, с поставяне на знака минус пред числото. Префиксен или постфиксен е този запис? Тъй като знаците на клавиатурата, които можем да използваме за знаци на операции не са много, в някои случаи за означаване на операции се използват комбинации от няколко знака.

Едноаргументната операция може да бъде записана и във ***функционален*** запис. Такава едноаргументна операция е, например, операцията за определяне броя на байтовете, които заема стойност от някакъв тип в оперативната памет – `sizeof`. При функционалния запис аргументът, в случая някой от възможните типове, се поставя в кръгли скоби след името на операцията `sizeof(`<*тип*>`)`.

За двуаргументните операции, каквито са повечето от операциите в математиката, общоприет е инфиксният запис, при който знакът на операцията се поставя между двата операнда: <*операнд*> <*знак_на_операцията*> <*операнд*>.

Пример за двуаргументни операции са събирането, изваждането, умножението и деленето.

Операции в C#

Езикът C#, както и езикът C, от който той произхожда, предлагат голямо разнообразие от операции. Операциите в C# могат да бъдат разделени в няколко категории:

- Двуаргументните ***аритметични*** операции – събиране, изваждане, умножение и деление – са ни добре познати от математиката и от използването им за съставяне на формули в програмите за работа с електронни таблици. По-малко популярна е операцията ***намиране на остатък*** при делене на цели числа със знак `%`. С използването на аритметичните операции ще се занимаем по-нататък в този урок.
- От ***операциите за присвояване*** за момента, познаваме само най-простото присвояване с инфиксен знак `=`. **Много важно е да подчертаем, че присвояването е операция и като всяка операция тя има резултат – стойността, присвоена на променливата в лявата страна.**
- ***Операция за сравняване*** проверява дали двата ѝ аргумента са в зададено съотношение или не. Резултатът е една от двете логически стойности – `true`, ако проверяваното съотношение е в сила или `false`, когато съотношението не е в сила. Знаците им са `<`, `<=`, `>`, `>=`, `==`, `!=`.
- ***Логическите операции*** се прилагат върху логически стойности и дават в резултат логически стойности. Логическите операции ***конюнкция*** („и“), ***дизюнкция*** („или“) и ***отрицание*** („не е вярно, че“) са ни познати от формулите в електронни таблици. Знаците им са `&&`, `||`, и `!`.
- ***Побитовите операции*** са аналози на логическите, само че се извършват между съответните битове на целочислените типове, като 0 се приема за `false`, а 1 – за `true`.
- Друга ***операция***, която е трудно да бъде класифицирана в някаква група – например е операцията ***за слепване*** *(конкатенация)* на два низа, знакът на която е `+`.

Таблица 1 съдържа всички операции на езика C# (с x и y в таблицата са означени някакви променливи, с e – израз, с t – тип, а с s_1 и s_2 – низове).

Аритметични операции и изрази

Аритметичните изрази са ни познати от математиката и работата с електронни таблици. Построяваме ги от числови стойности, зададени в променливи или като константи, и аритметичните операции. Три от аритметичните операции – събиране (инфиксен знак `+`), изваждане (инфиксен знак `-`) и умножение (инфиксен знак `*`) се извършват еднотипно, независимо какви са аргументите им.

Приоритет	Операции	Асоциативност
най-висок	x, y, f(x), a[x], x++, x--, new, typeof(x), checked, unchecked, ->	отляво надясно
	++x, --x, +e, -e, !e, ~e, (t)x, &x, sizeof(t), true, false	отдясно наляво
	*, /, %	отляво надясно
	s_1 + s_2	отляво надясно
	+, -	отляво надясно
	<<, >>	отляво надясно
	<, >, <=, >=, is, as	отляво надясно
	==, !=	отляво надясно
	&	отляво надясно
	^	отляво надясно
	|	отляво надясно
	&&	отляво надясно
	||	отляво надясно
	e ? *e* : *e*	отляво надясно
най-нисък	=, *=, /=, %=, +=, -=, <<=, >>=, &=, ^=, |=, ??	отдясно наляво

Таблица 1. Операции в C#

Извършването на операцията деление (с инфиксен знак /), когато аргументите ѝ са цели числа, дава в резултат само цялата част на резултата. Например, резултатът от делението 3/2 е 1, а не 1.5. За да бъде компенсирана неточността на целочислено деление, в компютрите е реализирана операцията за пресмятане на ***остатъка при целочислено деление*** (знак %). Например, 3%2 е 1. Остатъкът при деление на 2 често използваме, за да проверим дали едно цяло число е четно или нечетно.

На всеки аритметичен израз еднозначно се съпоставя число, което наричаме ***стойност*** на израза. За да може еднозначно да пресметнем стойността на аритметичен израз, в който имаме повече от една операция, са необходими правила за реда, в който се прилагат операциите. Такъв ред може да зададем с поставянето на кръгли скоби. Например, ((3*5)-(7+2))+((4*2)-1). Правилото е, че най-напред изпълняваме операция, която е поставена в скоби и в скобите няма други операции в скоби. Ако има няколко такива операции в скоби, няма значение от коя ще започнем. Затова изразът от примера пресмятаме в следния ред

((3*5)-(7+2))+((4*2)-1)=(15-9)+(8-1)=6+7=13.

Както се вижда от примера, ако редът за прилагане на операциите се определя по този начин, изразите се получават претрупани със скоби. За да се опрости изписването на изразите, се въвеждат допълнителни правила, като ***приоритет и асоциативност*** на операциите:

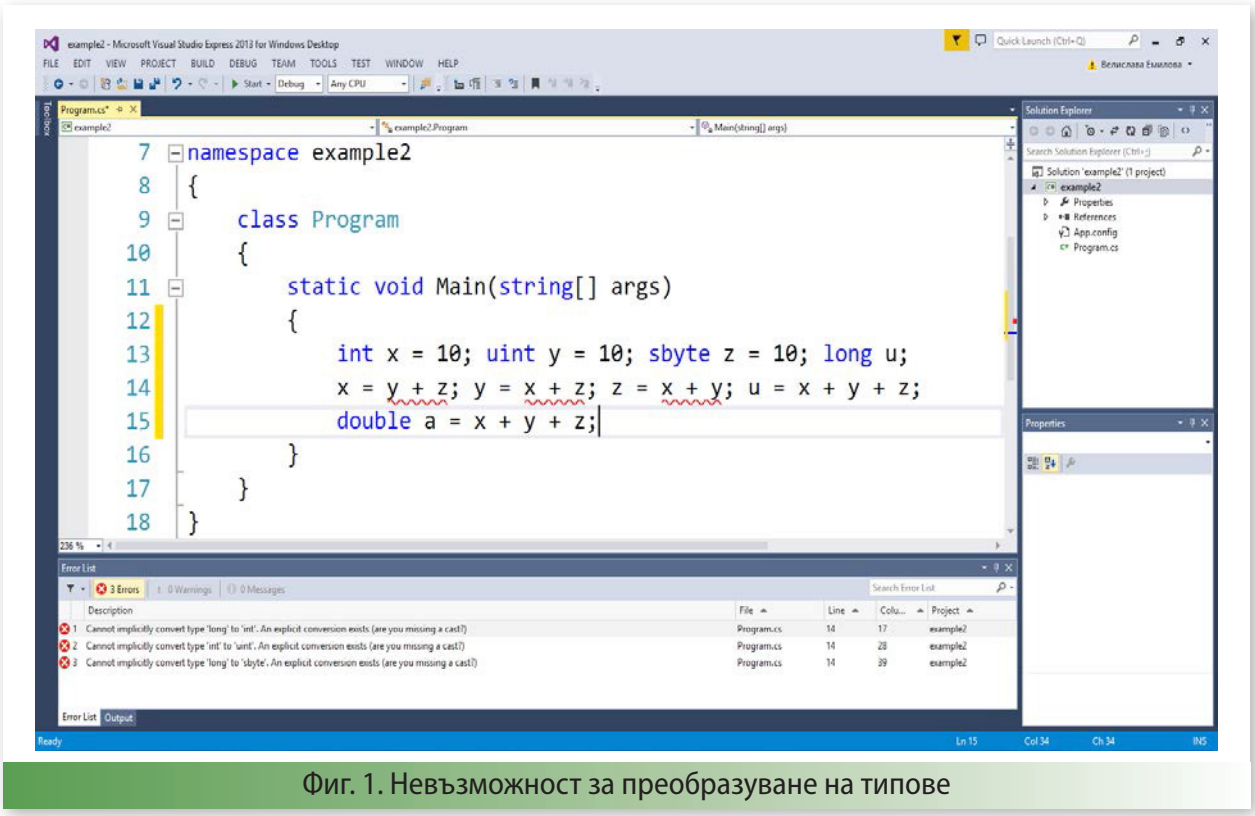
- на всяка операция се определя приоритет, като между две операции с различни приоритети по-рано се изпълнява тази с по-висок приоритет;
- за множество от операции с еднакъв приоритет се определя асоциативност – от ляво на дясно или отдясно на ляво. Когато в израза имаме няколко операции с еднакъв приоритет, те се изпълняват от ляво надясно или от дясно наляво, в зависимост от асоциативността им.

Аритметичните операции умножение, деление и намиране на остатъка при целочислено деление, и в математиката и в информатиката, имат по-голям приоритет от събирането и изваждането.

Асоциативността и на едните и другите е от ляво надясно. Затова изразът от примера можем да запишем по-просто така `3*5-(7+2)+4*2-1`. Последните скоби не може да премахнем, защото пресмятайки според асоциативността от ляво надясно ще трябва от произведението `3*5` да извадим 7 и да прибавим 2, което е различно от това да извадим (7 + 2).

В Таблица 1 операциите в един ред са с еднакъв приоритет, като приоритетът по редове намалява от горе надолу. В най-дясната колона е посочена асоциативността на операциите в съответния ред.

Много важно за еднозначното пресмятане на израз е как компилаторът постъпва, когато в израза участват **променливи и/или константи от различни типове**. Правилото е, че в операция с два операнда от различен тип, преобразуване на единия тип в другия се допуска само ако стойностите на първия тип са измежду стойностите на втория. В такъв случай казваме, че вторият тип е ***по-мощен*** от първия. Ако такова включване на стойности не е възможно в нито една от двете посоки, тогава двата типа са ***несравними***. Изразът се пресмята само ако в него има стойност от тип, който е сравним с останалите и е по-мощен от всички тях.



Фиг. 1. Невъзможност за преобразуване на типове

Например, стойност от беззнаковия тип `byte` може да се преобразува както в беззнаковия `uint`, защото интервалът от [0,255] се съдържа в интервала [0,65535], така и в знаковия `int`, защото [0,255] се съдържа и в [–32768,32767]. Знаковият `int` и беззнаковият `uint` не могат да се преобразуват един в друг, защото са несравними. Дробните типове `float` и `double` са по-мощни от всеки целочислен тип, т.е. преобразуването ще бъде допуснато от компилатора, но може да се загуби точност. Например, така е, ако присвоите много голяма цяла стойност на променлива от тип `float`. Във всеки случай, когато пресмятането на израза е невъзможно заради несъвместимост на типове, ще получите предупреждение още от редактора на средата (синтактична грешка) (*Фиг. 1*).

Общ вид на оператор за присвояване

След като вече знаем какво е израз, можем да дадем по-точна дефиниция на оператора за присвояване.

Синтаксисът на **оператора за присвояване** е:

<променлива> = *<израз>* ;

Действието на оператора е следното: пресмята се изразът отдясно на знака за присвояване и стойността му се присвоява на променливата отляво на знака.

Когато изразът се пресмята в оператор за присвояване, това което ще бъде присвоено на променливата отляво на знака зависи и от нейния тип. Правилото е следното. Ако пресметнатата стойност на израза е от същия или по-слаб тип от типа на променливата вляво, тогава операторът е допустим. Ако променливата вляво е от по-слаб тип или двата типа са несъвместими, операторът се счита за неправилен.

Автоинкрементна и автодекрементна операция

Едноаргументната ***автоинкрементна операция*** е със знак ++. **Аргумент на операцията може да бъде само целочислена променлива.** Операцията е в два варианта – постфиксен `a++` и префиксен `++a`. Когато операцията се използва самостоятелно в оператор `a++;` или `++a;`, тогава двете форми на операцията водят до един и същ резултат – увеличаване на стойността на променлива `a` с 1. В този случай всеки от двата оператора е еквивалентен на оператора за присвояване `a = a + 1;`. Когато операцията е включена в израз, например `a++ + b` или `++a + b`, тогава разликата между префиксната и постфиксната форма е съществена. В префиксната форма първо се изпълнява присвояването `a = a + 1` и получената стойност на `a` участва в пресмятането на израза, а при постфиксната – текущата стойност на `a` участва в пресмятането на израза, а след това се изпълнява присвояването `a = a + 1`. Например, ако `a` е 2, а `b` е 3, то след присвояването `c = a++ + b` стойността на `c` ще бъде 5, а стойността на `a` ще бъде 3. Докато след присвояването `c = ++a + b` стойността на `a` също ще бъде 3, но стойността на `c` ще бъде 6.

Автодекрементната операция със знак `--` е напълно аналогична на автоинкрементната, като вместо увеличаване на съдържанието на променливата аргумент с 1, се извършва намаляване на стойността ѝ с 1. Например, ако `a` е 2, а `b` е 3, то след присвояването `c = a-- + b` стойността на `c` ще бъде 5, а стойността на `a` ще бъде 1. Докато след присвояването `c = --a + b` стойността на `a` също ще бъде 1, но стойността на `c` ще бъде 4.

Автоинкрементната и автодекрементната операция не са задължителни, но изписването им е по-кратко от алтернативната възможност. Нещо повече, тези две операции се компилират в съответните автоинкрементна и декрементна инструкция на машинния език, които са много по-бързи от инструкциите за събиране и изваждане.

Въпроси и задачи

1. Какво ще бъде изведено в двата етикета `label1` *и* `label2`*, след изпълнение на програмите?*

а.
```
public Form1()
 { InitializeComponent();
   int a = 3;
   int b;
   b = a++;
   label1.Text = a;
   label2.Text = b;
 }
```

б.
```
public Form1()
 { InitializeComponent();
   int a = 3;
   int b;
   b = ++a;
   label1.Text = a;
   label2.Text = b;
 }
```

в.
```
public Form1()
 { InitializeComponent();
   int a = 3;
   int b = 5;
   a--;
   b++;
   b=a++ + b;
   label1.Text = a;
   label2.Text = b;
 }
```

г.
```
public Form1()
 { InitializeComponent();
   int a = 3;
   int b = 5;
   a--;
   ++b;
   a = a + b;
   label1.Text = a;
   label2.Text = b;
 }
```

д.
```
public Form1()
 { InitializeComponent();
   int a = 0, c = 12;
   int b = 0, d = 5;
   a = c / d;
   b = a + b;
   label1.Text = a;
   label2.Text = b;
 }
```

е.
```
public Form1()
 { InitializeComponent();
   int a = 0, c = 12;
   int b = 10, d = 5;
   a = c % d;
   b = b / a;
   label1.Text = a;
   label2.Text = b;
 }
```

25 Аритметични изрази – упражнение

Задача 1. Създайте от всеки от фрагментите в раздела Въпроси и задачи на предния урок програма с графичен интерфейс, компилирайте я и проверете правилно ли сте определили какво извежда програмата в съответните етикети.

Задача 2. Разгледайте двете функции от *Фиг. 1*. Опитайте се да определите какво ще бъде изведено в етикета при изпълнение на съответните програми.

а.
```
public Form1()
{  InitializeComponent();
   double x;
   int y = 10, z = 4;
   x = y / z;
   label1.Text = x;
}
```

б.
```
public Form1()
{  InitializeComponent();
   double x, y = 10;
    int z = 4;
    x = y / z;
    label1.Text = x;
}
```

Фиг. 1.

Задача 3. Създайте от всяка от програмите в предната задача графично приложение, компилирайте го и проверете правилно ли сте определили какво извежда програмата в етикета.

Задача 4. Направете графично приложение с име Number, в което след натискане на бутон, на променливата x се присвоява стойност < 1000, която се взима от текстово поле, а след това на променливата suma се присвоява сборът от цифрите на трицифреното число, въведено в променливата x и се показва в етикет.

26 Вградени математически функции

Класът Math

В предишен урок имахме възможност да споменем ролята на стандартните подпрограми за бързото и качествено програмиране на често решавани задачи. Редица математически функции са трудни за програмиране от неопитни програмисти и затова е по-добре да бъдат включени в програмата като стандартни подпрограми. В езика C# има клас Math, който съдържа няколко метода за пресмятане на математически функции.

Атрибути

В C# имената на атрибутите на клас се поставят след името на класа и точка. Класът Math има два атрибута:

- Math.PI – именувана константа със стойност числото π – съотношението между дължината на окръжността и нейния диаметър – представено с максималната допустима от типа double точност: const double PI = 3.14159265358979;
- Math.E – именувана константа със стойност числото *e* – основа на натуралния логаритъм – представено с максималната допустима от типа double точност: const double E = 2.71828182845905.

Методи

Класът Math предлага голям брой методи за пресмятане на математически функции. Когато една и съща математическа функция може да се приложи към аргументи от различен тип, тогава в ООП се създава по един метод за всеки от типовете, като всички методи носят едно и също име. За програмиста нещата изглеждат така, сякаш има само един метод, който може да се прилага към аргументи от различен тип. Затова всички едноименни методи ще описваме като един, и ще посочваме множеството от възможните стойности на аргументите – константи или променливи.

Ето някои от най-често използваните методи на класа Math:

- Math.Abs(<*числова стойност/променлива*>) – връща абсолютната стойност на аргумента;
- Math.Ceiling(<*дробна стойност/променлива*>) – връща най-близката цяла стойност, по-голяма или равна на аргумента. Например Math.Ceiling(3.14) е 4;
- Math.Floor(<*дробна стойност/променлива*>) – връща най-близката цяла стойност, по-малка или равна на аргумента. Например Math.Floor(3.14) е 3;
- Math.Max(<*числова стойност/променлива, числова стойност/променлива*>) – връща по-голямата от двете стойности на аргументите;
- Math.Min(<*числова стойност/променлива, числова стойност/променлива*>) – връща по-малката от двете стойности на аргументите;

- `Math.Sqrt(`<*дробна променлива/стойност от типа* `double`>`)` – връща квадратния корен на аргумента;
- `Math.Pow(`<*дробна променлива/стойност от типа* `double`, *дробна променлива/стойност от типа* `double`>`)` – връща първия аргумент, повдигнат на степен втория аргумент.
- `Math.Round(`<*дробна променлива/стойност от типа* `double`, *целочислена стойност/променлива*>`)` – връща първия аргумент, закръглен до толкова знака след десетичната точка, колкото е стойността на втория аргумент.

За използването на функциите/методите, които пресмятат стойности, е важно да напомним следното правило:

Когато в израз поставим **извикване на функция/метод, която/който пресмята стойност**, тогава извикването на функцията/метода се заменя с пресметнатата стойност и тя участва по-нататък в пресмятането на стойността на израза. Извикването на функция/метод се счита за операция с най-висок приоритет в израза.

Примери

Основна част от работата на всяка програма е пресмятането на изрази. Ето няколко примера за пресмятане на изрази, с използване на методите на класа `Math`.

1. На променливата `s` да се присвои лицето на кръг с радиус `r`.

Решение:
```
s = Math.PI * r * r;
```

2. На променливата `c` да се присвои дължината на окръжност с радиус `r`.

Решение:
```
c = 2 * Math.PI * r;
```

3. Стена има правоъгълна форма с височина, съдържаща се в променливата `a` и дължина, съдържаща се в променливата `b`. С една кутия боя може да се покрият `c` квадратни единици от стената. На променливата `ans` да се присвои броят кутии боя, които трябва да се закупят, за да се боядиса цялата стена.

Решение:
```
ans = Math.Celling((a * b) / c);
```

4. На променливата `x` да се присвои закръгленото до цяло на положително дробно число `y`. Например, ако `y` е 3.567, то съдържанието на `x` трябва да бъде 4, а ако `y` е 3.499, тогава съдържанието на `x` трябва да бъде 3.

Решение:
```
x = Math.Floor(y + 0.5);
```

Въпроси и задачи

1. *Създайте графични приложения за всеки от примерите в края на урока, в които чрез натискане на бутон въвеждате променливите от примерите в текстови полета и извеждате получения резултат в етикет.*
2. *Метален прът с цилиндрична форма има дължина, съдържаща се в променливата* `a`. *От него трябва да се изрежат метални цилиндри с височина, съдържаща се в променливата* `b`. *Направете графично приложение с име* MetalRod, *в което се въвеждат конкретни стойности на променливите в текстови полета, пресмята се и се извежда в етикет при натискане на бутон броят на цилиндрите, които могат да се изрежат от дадения прът.*
3. *Направете графичното приложение с име* Square3, *в което се въвежда лице на квадрат и се извежда периметърът му. Програмата трябва да може да работи и с дробни числа и да извежда периметъра, закръглен до втория знак след десетичната точка.*

27 Форматиране на извежданите данни

Както вече знаем, общуването на потребителя и работещата програма може да стане по различен начин – с графичен интерфейс или с буквено-цифров интерфейс (както при конзолните приложения). Разбира се, програмите с графичен интерфейс са по-удобни за обикновения потребител, по-привлекателни, и постепенно изместиха програмите с буквено-цифров интерфейс. Въпреки това, конзолата си остава незаменимо средство за общуване с програмата в някои случаи.

Един от тези случаи е необходимостта от малки програми, със сравнително прост вход, за еднократна употреба, когато е необходимо вниманието да се насочи към конкретния проблем, който решаваме, а не към атрактивно представяне на резултатите. Друг случай, в който конзолата е полезна е, когато тестваме част от кода на голяма програма. Поради простотата на работа на конзолното приложение, може да се изолира тази част от кода лесно и удобно, без да се налага да се преминава през сложен потребителски интерфейс и поредица от прозорци, за да се стигне до желания код за тестване.

Форматиран изход

За да се направи конзолното приложение по-удобно за потребителя, е необходимо извежданите данни да са форматирани по подходящ начин. За разлика от въвеждането, където всички данни попадат в програмата във вид на низ и трябва да се преобразуват с метода `Parse`, при извеждането могат да се извеждат директно константи, променливи и стойности на изрази от всички познати типове чрез познатите ни методи `Console.WriteLine(`*<израз>*`)` и `Console.Write(`*<израз>*`)`. Да напомним разликата между двата метода: методът `Write` извежда на конзолата стойността на израза, докато методът `WriteLine` прави същото, след което преминава на нов ред.

Да разгледаме ситуация, при която се налага да се изведат на един ред на конзолата данни от различни типове. Например, в променливата `years` е пресметната възрастта на потребителя и програмата трябва да изведе съобщението: `"Вие сте на ... години"`, където на мястото на точките трябва да се постави съдържанието на променливата `years`. Вече знаем два начина да направим това:

Първият начин е да разделим текста на три части, които да изведем с три отделни извиквания на методи:

```
Console.Write("Вие сте на ");
Console.Write(years);
Console.WriteLine(" години");
```

Вторият начин е да използваме операцията сливане на низове:

```
Console.WriteLine("Вие сте на " + x + " години");
```

Методите `WriteLine` и `Write` имат следния по-общ вид

`WriteLine(`*<форматиращ низ>*`, `*<списък от изрази>*`)`

`Write(`*<форматиращ низ>*`, `*<списък от изрази>*`)`

където

- *<списък от изрази>* се състои от няколко израза, разделени един от друг със запетаи, номерирани с 0, 1, 2 и т.н. в реда, по който се срещат в списъка;
- *<форматиращ низ>* е константен низ, съдържащ текст, който ще се извежда и който съдържа ***форматиращи елементи***. Най-простият вид на форматиращ елемент е {*<номер на израз>*}. Това означава, че на мястото на форматиращия елемент ще бъде изведена стойността на израза, с посочения във форматиращия елемент номер в списъка от изрази.

Сега ще покажем и трета възможност да изведем подобни съобщения, като покажем специалните възможности на методите `WriteLine` и `Write` за форматиране на изхода.

Например, ако в променлива `lev` сме запомнили левовата част на някаква сума пари, а стотин-

ките – в променливата st и искаме програмата да изведе съобщение за тази сума, тогава можем да напишем оператора:

```
Console.WriteLine("Сумата е {0} лева и {1} стотинки", lev,st);
```

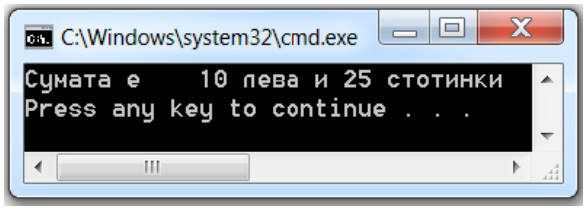
Така, на мястото на форматиращия елемент {0} ще бъде изведена стойността на променливата lev, на мястото на форматиращия елемент {1} ще бъде изведена стойността на променливата st.

Ако във форматиращия елемент, след номера на израз и разделено със запетая от него, е изписано още едно число, то задава броя позиции, в които да се изведе стойността на посочения израз. Ако извежданата стойност заема по-малко позиции, тогава в излишните позиции ще се изведат интервали. Ако броят позиции е недостатъчен, ще бъдат заделени автоматично толкова позиции, колкото трябва.

Например, резултатът от изпълнение на програмата:

```
static void Main(string[] args)
{ int lev=10, st=25;
 Console.WriteLine("Сумата е {0,5} лева и {1,1} стотинки", lev,st);
}
```

е показан на *Фиг. 1*. Тъй като променливата lev има стойност, записваща се в 2 знака, вляво остават 3 интервала. Стойността на променливата st пък не се събира в един знак и затова програмата отделя за нея необходимите 2 знака.



Фиг. 1.

Освен в конзолни приложения, изходът може да се форматира и в графични приложения чрез метода Format на класа String, който има същите възможности като метода WriteLine на класа Console:

```
double temp = 20.4;
string s = String.Format("Температурата е {0}°C.", temp);
label1.Text = s;
```

Управляващи знаци

Управляващите знаци са с кодове от 0 до 31 в таблицата Unicode. Някои от тях се използват за форматиране на изхода при методите за извеждане. Тъй като тези знаци нямат свое графично изображение, прието е да се изписват с някакъв знак, напомнящ за предназначението им при форматиране на изхода, предшестван от знака обратна наклонена черта (\). В следната таблица са дадени някои управляващи знаци:

'\n' – премини на нов ред (new line)	'\'' – изведи апостроф
'\a' – издай звук (system beep)	'\"' – изведи кавички
'\b' – изтрий последния знак (backspace)	'\{' – изведи лява фигурна скоба
'\r' – върни в началото на реда (return)	'\}' – изведи дясна фигурна скоба
'\t' – хоризонтална табулация (tab)	'\\'– изведи обратна наклонена черта

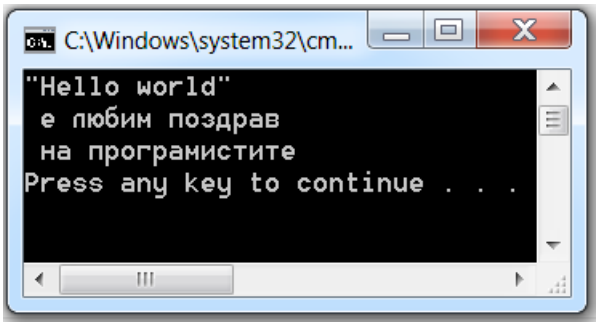
Така например, извикването Console.Write("{0}\n",x) е еквивалентно на извикването Console.WriteLine(x).

Забележете, че кавичките, апострофите и фигурните скоби играят специална роля при оформяне на форматиращия низ. Затова, когато се налага да изведем някой от тези знаци на конзолата – трябва да отменим специалното му значение. Това става, като пред него напишем знака обратна наклонена черта (\). В тази си роля знакът обратна наклонена черта става специален и затова, когато трябва да го изведем на конзолата, изписваме две обратни наклонени черти.

Например, ако искаме да го изведем на конзолата текста `"Hello world" е любим поздрав на програмистите`, не можем да я изпишем в този вид между двойка кавички. Опитайте да го направите и вижте какво ще се случи. Низът трябва да се изпише така, както е показано на *Фиг. 2.а* и тогава резултатът ще бъде този, който искаме (*Фиг. 2.б*).

```
class Program
{  static void Main(string[] a)
   {
     Console.WriteLine("\"Hello
     world\"\n е любим поздрав\n
     на програмистите");
   }
}
```

а.



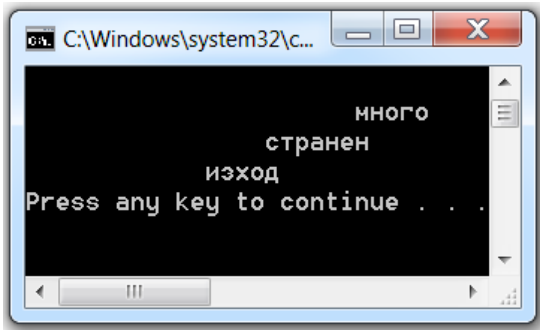
б.

Фиг. 2.

Ако пред форматиращия низ се постави знакът `'@'`, то зададеният текст ще се изведе така, както е зададен в редактора, дори и да е на повече от един ред, заедно с включените интервали, знаци за табулация и т.н. Например, това което ще изведе програмата от *Фиг. 3.а,* е показано на *Фиг. 3.б*.

```
static void Main(string[] args)
{
   Console.WriteLine(@"
            много
       странен
   изход");
        }
```

а.



б.

Фиг. 3.

Работа с компютър

Да напишем програма, която разменя стойностите на две променливи. Това може да бъде извършено по няколко начина. Двата оператора: `a = b; b = a;` на пръв поглед правят това, но ако се вгледаме по-внимателно и си припомним същността на операцията присвояване, ще разберем, че тези два оператора не водят до искания резултат. Напишете конзолно приложение с име Swap, което въвежда стойности в `a` и `b`, изпълнява оператори: `a = b; b = a;` и показва получения резултат. Причината за неуспеха е, че когато присвоим на `a` стойността на `b`, изгубваме стойността на `a`.

От тези разсъждения се вижда, че за да не загубим стойността на `a`, ще ни трябва още една променлива `c`, която преди присвояването `a = b;` да съхрани стойността на `a`, и от която след това ще прехвърлим първоначалната стойност на `a` в `b`. Въведете получената програма (*Фиг. 4*), компилирайте я и я изпълнете, за да проверите работоспособността ѝ.

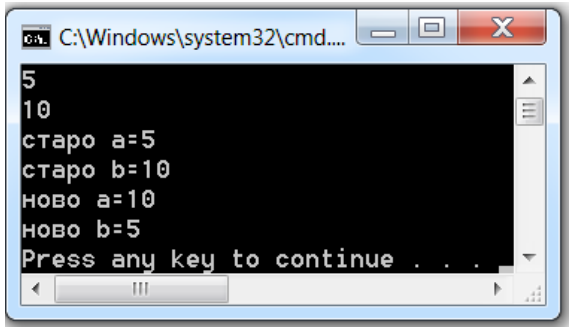
Ако правилно сте въвели програмата, резултатът от изпълнението трябва да бъде този, който е показан на *Фиг. 5*. Има и друг интересен начин за размяна стойностите на две променливи, при

```
using System;
class Program
{  static void Main(string[] args)
   {
        int a, b, c; string s;
       s=Console.ReadLine(); a = int.Parse(s);
       s=Console.ReadLine(); b = int.Parse(s);
        Console.WriteLine("старо a={0}/n старо b={1}", a, b);
        c = a; a = b; b = c;
      Console.WriteLine("ново a={0}/n ново b={1}", a, b);
   }
}
```

Фиг. 4.

който не се използва допълнителна променлива, а операторите за присвояване: `a = a + b; b = a - b; a = a - b;`.

Напишете нова версия на програмата с име Swap1, която реализира този подход, компилирайте я и проверете работоспособността ѝ. Този подход, обаче, не винаги е за предпочитане, тъй като е малко по-бавен, а при големи стойности на променливите `a` и `b` може да доведе до препълване на типа, т.е получаване на по-голяма стойност от максималната, която променлива от използвания тип може да съдържа (в кой от трите оператора може да стане това?) !



Фиг. 5.

Въпроси и задачи

1. Направете конзолно приложение с име MySchool, *което извежда на екрана името на вашето училище, оградено с кавички.*

2. Довършете оператора

```
Console.WriteLine(„Всеки . . . идентифицира.“);
```

като замените многоточието с текст така, че в конзолата да се изведе:

```
Всеки гражданин има
ЕГН\ЛНЧ, чрез който се идентифицира.
```

3. Напишете конзолно приложение с име Example1, *което да изведе на екрана следния текст: След изпълнението на операторите:*

```
int a = 5;
if (a > 0) { a = a + 5; }
else { a = a - 5; }
```

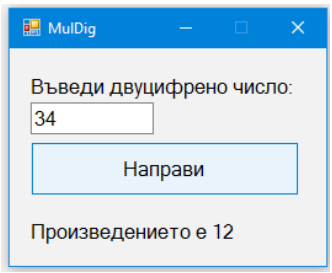
стойността на променливата `‘a’` *ще бъде 10.*

4. Допишете програмния фрагмент

```
int a = 10, b = 12, c = 14;
Console.WriteLine(. . . . .);
```

така, че, стойностите на трите променливи да бъдат изведени в един ред на конзолата, разделени със знака за хоризонтално табулиране, т.е. `10  12  14.`

28 Форматиране на извежданите данни – упражнение



Фиг. 1.

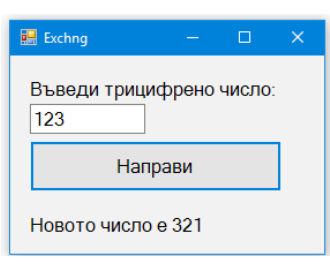
Задача 1. Да се направи графично приложение с име Muldig по модела от *Фиг. 1*, в което след натискане на бутон се въвежда от текстово поле с подходящ надпис едно положително двуцифрено число и се извежда в етикет произведението на цифрите му.

Решение: Единствената трудност на задачата е отделянето на двете цифри на въведеното число. От урока за представяне на числата в позиционни бройни системи знаем, че цифрата на единиците на двуцифрено число, представено в десетична бройна система, е остатъкът от целочисленото деление на числото на 10, а цифрата на десетиците – частното от това деление.

Програмен код:

```
namespace Muldig
{
   public partial class Form1 : Form
   {
      public Form1()
      {
         InitializeComponent();
      }
      private void button1_Click(object sender, EventArgs e)
         {   string s;
             s = textBox1.Text;
             int number;
             number = int.Parse(s);
             int pr;
             pr = (number % 10) * (number / 10);
             label2.Text = String.Format("Произведението е {0}",pr);
         }
   }
}
```

Въведете програмата, компилирайте я и проверете работоспособността ѝ с няколко примера. Какъв е резултатът от изпълнението на програмата, ако зададете едноцифрено число? А ако зададете трицифрено?



Фиг. 2.

Задача 2. Направете графично приложение с име Exchng по модела от *Фиг. 2*, което след натискане на бутон въвежда трицифрено цяло положително число *N* от текстово поле с подходящ надпис и извежда в етикет числото, което се получава, когато разменим местата на първата и последната цифра на *N*.

Решение: Първата и последната цифра на трицифрено число можем да получим както и в Задача 1 като частното от целочисленото деление на числото на 100 и остатъка от такова деле-

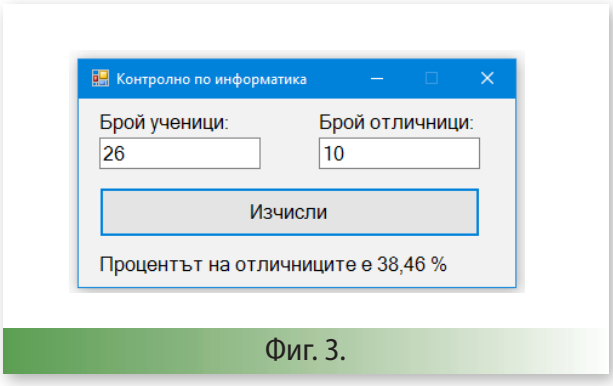
ние на числото на 10, съответно. Проблемът в този алгоритъм е отделянето на втората цифра, която ще ни трябва за построяване на новото число. За целта отделяме двуцифреното число, съставено от първите две цифри като частното от деленето на 100, а от него отделяме цифрата на единиците му като остатък при деление на 10.

Програмен код:

```
public partial class Form1 : Form
{
   public Form1()
   {
   InitializeComponent();
   }
    private void button1_Click(object sender, EventArgs e)
   {  string s = textBox1.Text;
      int number = int.Parse(s);
      int ed, des, sto;
      ed = number % 10;    //цифрата на единиците
      des = (number / 10) % 10; //цифрата на десетиците
      sto = number / 100;     //цифрата на стотиците
      int swap;     //размяна на цифрите
      swap = ed; ed = sto; sto = swap;
      int newNumber;
      newNumber = sto * 100 + des * 10 + ed; //новото число
      label2.Text = String.Format("Новото число е {0}", newNumber);
   }
}
```

Въведете програмата, компилирайте я и проверете работоспособността ѝ с няколко примера. Какъв е резултатът от изпълнението на програмата, ако зададете двуцифрено число? А ако зададете едноцифрено?

Задача 3. Направете графично приложение с име Excellent по модела от *Фиг. 3*, в което се въвежда броят на учениците в един клас и броят на тези ученици, които са получили отлична оценка на контролна работа по информатика, след което се намира и показва процента на отличниците, закръглен до втората цифра след десетичната точка.



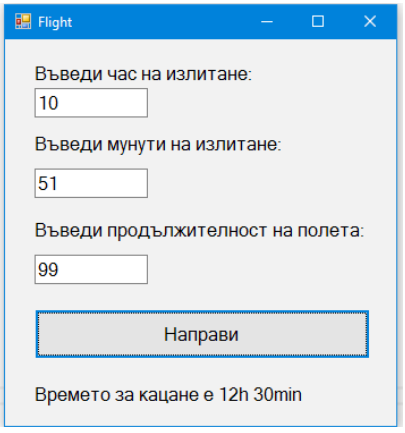
Фиг. 3.

Решение: Трудността в тази задача е изрязването на дробно число до втория знак след десетичната точка. За целта може да се ползва методът `Round` от класа `Math` или това да стане в метода `Format` на класа `String` (вижте в кода). Друга особеност е да се използва за поне една от двете входни променливи реален тип, за да може да се извърши точно деление при пресмятане на процента. За ограничаване броя на цифрите след десетичния знак ще използваме форматиращия елемент `#.##`. Броят на знаците `#` след десетичния знак означава, че дробната част трябва да има точно толкова цифри след знака, а броят на знаците `#` преди десетичния знак означава, че цялата част трябва да има поне толкова цифри.

Програмен код:

```
private void button1_Click(object sender, EventArgs e)
{
    int students = int.Parse(textBox1.Text);
    double excellentStudents = double.Parse(textBox2.Text);
    double percent = excellentStudents / students * 100;
```

```
        //percent = Math.Round(percent, 2);
        label3.Text = String.Format("Процентът на отличниците е {0:#.##} %", percent);
    }
```



Фиг. 4.

Задача 4. Направете графично приложение с име Flight по модела от *Фиг. 4*, в което се въвеждат часа и минутите от началото на часа на излитането на самолет, както и продължителността на полета му в минути. Програмата трябва да изведе часа и минутите, когато самолетът трябва да кацне, при условие, че кацането става в деня на излитане.

Решение: Решението използва обичайната техника за работа с мерни единици, когато всички данни се превръщат в най-малката от използваните мерни единици – в случая минути, извършват се пресмятанията, а резултатът отново се преобразува в началния вид, в случая – час и минути от началото на часа.

Програмен код:

```
private void button1_Click(object sender, EventArgs e)
{  string s = textBox1.Text;
   int hours = int.Parse(s);
   s = textBox2.Text;
   int minutes = int.Parse(s);
   s = textBox3.Text;
   int x = int.Parse(s);
   int allMinutes = hours * 60 + minutes + x;
   int newHours = allMinutes / 60;
   int newMinutes = allMinutes % 60;
   label4.Text = String.Format("Времето за кацане е {0}h {1}min", newHours,
   newMinutes);
}
```

IV Програмиране на разклонени и циклични алгоритми

29 Булев тип данни

Типът Boolean

За да могат да бъдат програмирани разклонени и циклични алгоритми, езикът за програмиране предоставя механизъм за вземане на решения в коя от две възможни посоки да продължи изпълнението на програмата. Когато програмата трябва да вземе някакво решение, това става с помощта на ***логическо условие*** (или просто ***условие***). Условието е израз, стойността на който е булева – `true` (вярно) или false (невярно). Затова езикът за програмиране поддържа типа `bool`, при който възможни константи са само двете булеви стойности.

Декларирането на променливи от тип `bool` не се различава от декларирането на променливи от кой да е друг тип, като в една декларация може да обявим типа за една или повече променливи:

```
bool b; bool check, flag1, flag2;
```

На променливи от този тип можем да присвояваме булеви стойности:

```
flag1 = true; flag2 = false;
```

и да извеждаме съдържанието им в конзолата или в компоненти на графичния интерфейс. Така например, операторът:

```
Console.Write(flag1 + " " + flag2);
```

ще изведе в конзолата ред със съдържание

```
True False
```

Забележете как при извеждане на булеви стойности, програмата промени първите букви на двете константи в главни. Ако, обаче, напишете в програмата булева константа с главна буква, компилаторът ще ви даде съобщение за грешка. Операторът:

```
Console.Write(sizeof(bool));
```

ще изведе в конзолата 1, тъй като булевите стойности се записват в 1 байт.

Типът Boolean

Най-простият начин за построяване на условие е с помощта на операциите за сравняване, които познаваме от уроците за електронни таблици. На *Фиг. 1* са показани знаците на шестте операции за сравняване в езика C#, като знаците на две от операциите се различават от използваните в MS Excel. Кои са те?

Двата аргумента на операция за сравняване може да са произволни изрази, стойностите на които могат да се сравняват. Например, може да се сравняват стойностите от кои да са два числови израза, но не може да се сравняват числови стойности с низове. Стойностите от тип `char` се сравняват както числа, защото в действителност се сравняват техните кодове в таблицата Unicode.

Знак	Значение
==	Равно (=)
>	По-голямо (>)
<	По-малко (<)
>=	По-голямо или равно (≥)
<=	По-малко или равно (≤)
!=	Не равно (≠)

Фиг. 1. Сравнения

Изразите от тип `string` в C# могат да участват само в сравнения от типа `==` и `!=`. Както знаем от уроците за електронни таблици, другите сравнявания на низове също имат смисъл, когато се използва лексикографската наредба. Причината да не са позволени в C# е, че в таблицата Unicode има всевъзможни азбуки и резултатът от сравняването може да е безсмислен.

Нека `int a = 12, b = 6, c=20; bool flag`. Тогава операторът `flag = a>b;` проверява дали 12 > 6, което е вярно. Затова на булевата променлива `flag` се присвоява стойността `true`. Операторът `flag = (a+b)>=c;` проверява дали сумата на `a` и `b` е по-голяма или равна на стойността на `c`, т.е. дали 12 + 6 >= 20, което не е вярно. Затова на булевата променлива `flag` се присвоява стойността `false`.

Логически операции

При компютърните пресмятания често се налага да се комбинират няколко аритметични отношения едновременно. Например, ако искаме едно число да е в интервала от 0 до 10, то трябва да е едновременно по-голямо или равно на 0 **и** по-малко или равно на 10. В друг случай числото трябва да е извън този интервал и тогава то трябва да е по-малко от 0 **или** по-голямо от 10. Понякога се налага да използваме противното на дадено твърдение. Например: твърдението „числото е в интервала от 0 до десет“ означава **не е вярно**, че числото е по-малко от 0 **и** по-голямо от 10.

x	y	x && y	x || y
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	1

а.

x	!x
0	1
1	0

б.

Фиг. 2.

В такива случаи се използват логическите операции. Техните операнди са булеви стойности, а резултатът е също булева стойност. Езикът C# предоставя логическите операции ***конюнкция***, ***дизюнкция*** и ***отрицание***. Резултатът от прилагане на всяка от операциите за всички възможни комбинации от стойности на двата операнда е даден в таблиците на *Фиг. 2*.

Операциите конюнкция и дизюнкция са двуаргументни, а операцията отрицание – едноаргументна. Операцията конюнкция (наричана още логическо умножение) се означава с `&&` и се чете „и“. Резултатът от конюнкцията е `true` само в случай, че и двата операнда имат стойност `true`. Операцията дизюнкция (наричана още логическо събиране) се означава с `||` и се чете „или“. Резултатът от дизюнкцията е `false` само в случай, че и двата операнда имат стойност `false`. Операцията отрицание се означава с `!` и се чете „не“. Стойността на отрицанието е `true`, когато операндът ѝ е `false` и обратно.

Най-висок приоритет от логическите операции има отрицанието, а приоритетът на конюнкцията е по-висок от този на дизюнкцията.

Логически изрази

Изрази, в които участват булеви стойности (например пресметнати при сравняване) и логически операции, наричаме логически изрази (или условия). Например, при изпълняване на фрагмента:

```
int a;  bool b = (a >= 0) && (a <= 10);
bool c = !((a< 0) || (a >10));
```

булевите променливи `b` и `c` ще имат една и съща стойност в зависимост от това дали числото `a` е в интервала от 0 до 10 (`true`) или извън него (`false`).

В езика C# логическите операции са с по-нисък приоритет от сравненията. Затова горният фрагмент може да се запише по-просто:

```
bool b = a >= 0 && a <= 10;
   bool c = !(a< 0 || a >10);
```

Въпроси и задачи

1. Нека `int a = 3; int b = 2, c = 4;`. *Каква е стойността на логическите изрази?*

а) `a < b && 3 < 0 ;`

б) `a < b || 3 < 0 ;`

в) `!((a-b) > c);`

2. Запишете като логически изрази условията:

а) x е по-голямо или равно на 1 и по-малко или равно на 5;

б) x не е по-голямо или равно на 1 нито по-малко или равно на 5;

в) x не е по-голямо или равно на 1 нито по-голямо или равно на 5;

г) x е по-голямо или равно на 1 и по-малко или равно на –5.

3. Каква ще бъде стойността на всеки от изразите в задача 2, ако x = 7?

30. Условен оператор

Операторът if

Представете си, че трябва да напишете програма, която по дадени координати на горен ляв и долен десен връх на правоъгълника, извеждаше координатите на другите два върха. Вие сте направили вярно алгоритъма за намиране на търсените абсциси и ординати, но дали резултатът винаги ще е верен? Представете си, че въведените координати не са коректни – тогава и изходът няма да е верен. Още по-неприятно ще се получи, ако очакваме в контрола `TextBox` да има въведено число, а между цифрите има буква – тогава програмата ще спре и ще изведе съобщение за грешка.

Иначе казано, за да работи безотказно, програмата трябва „да проверява" входните данни и само когато са коректно зададени, да намира и извежда резултат. В противен случай програмата трябва да изведе подходящо съобщение. В този урок ще разберем как става това.

Задача: Напишете програма, която по зададена страна *A* на квадрат намира лицето му.

Входна данна за програмата е страната на квадрат. Не съществуват квадрати с отрицателни страни. Затова алгоритъмът трябва да съдържа проверка, дали зададената дължина на страната *A* е неотрицателно число. Ако проверката ви е „Ако A > 0, намери лицето", обаче, алгоритъмът отново е непълен. Какво прави програмата, ако A е отрицателно? Затова правилното действие е „Ако A > 0 – намери лицето на квадрата, иначе изведи съобщение за грешен вход."

Нека запишем този алгоритъм във вида:

`Ако A>0` ▶ `намери лицето,`

`иначе` ▶ `отпечатай, че има грешен вход.`

Сега обърнете внимание на конструкцията

`ако условие` ▶`направи нещо` ▶ `иначе` ▶ `направи друго нещо.`

Когато условието е изпълнено, т.е. е вярно, се изпълняват инструкциите в синьо, иначе – инструкциите в червено. Същата конструкция има и в езика C#, като думата `ако` се заменя с `if`, а думата `иначе` – с `else`. Получаваме:

```
if (<логическо условие>)
{
   тук пишем какво да се прави, ако условието е вярно
}
else
```

```
{
  тук пишем какво да се прави, ако условието НЕ е вярно
}
```

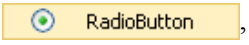
Тази конструкция се нарича ***условна***, а операторът започващ с `if` – ***условен оператор***. В условния оператор може да се пропусне частта, започваща с `else`. Големите (къдрави) скоби се пишат задължително, само когато действията в съответния случай изискват повече от един оператор. Ако операторът е само един – скобите могат да се пропуснат, но не препоръчваме на неопитен програмист да го прави.

Логическото условие се поставя след ключовата дума `if` и е винаги в малки (кръгли) скоби! Например:

```
if (a>15),     if (c<10 || a>b),   if ( (a>b || b>c) && (d<20) ).
```

Условният оператор не се използва само за проверка на коректността на входни данни. Опитайте се да напишете оператор, който проверява дали числото A е четно или нечетно и извежда в `TextBox` съответно съобщение.

Контроли RadioButton и CheckBox

В C# има компонента `RadioButton` 

, която познаваме от използваните приложни програми. ***Радио бутоните*** винаги са по няколко в група, като само един от тях е избран (има точка в него). Казваме, че радиобутонът с точката е ***активен***. Когато натиснем неактивен бутон от групата, активният ще бъде размаркиран и активен ще стане бутонът, който сме натиснали.

Радио бутонът има много свойства, но очевидно това, което показва дали е активен или не, е най-важно. То се нарича `Checked`. Друго свойство на радио бутона, което ще се наложи да използваме, е `Text`. Не е трудно да се досетите че това е надписът на бутона. Пример за това как се използва свойството `Checked` е показан в следния фрагмент:

```
if (RadioButton.Checked) { <оператори> }
```

Друга полезна графична компонента е `CheckBox` (кутия за отметка). Тя също притежава свойствата `Checked` и `Text`, които са аналогични на същите свойства на компонентата `RadioButton`. Разликата е, че кутиите за отметка не се събират в групи и може да имаме едновременно активни няколко кутии за отметка, или никоя кутия да не е активна. Във формата на *Фиг. 1* са активни радио бутонът с надпис Момичета и двете кутии за отметка с надписи 8 а клас и 8 в клас.

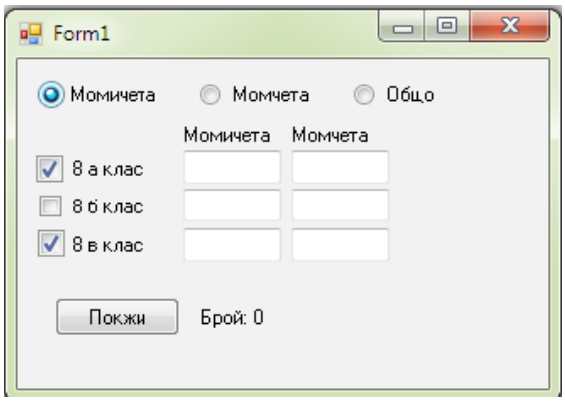
Работа с компютър

Задача 1. Напишете отново програмата Square за намиране на периметър и лице на квадрат, като добавите обсъдената в урока проверка за коректност на входа. Компилирайте получената програма и проверете дали наистина правилно контролира входа. Използвайте следния програмен фрагмент:

```
a = int.Parse(s);
if (a > 0)
{   Console.Write("Периметърът е ");
    Console.WriteLine(4 * a);
    Console.WriteLine("");
    Console.Write("Лицето е ");
    Console.WriteLine(a * a);
    Console.WriteLine("");
}
else
   Console.WriteLine("Грешно въведена страна !");
```

Задача 2. Напишете програма с графичен интерфейс по модела от *Фиг. 1*, която изработва справка за броя на момчетата в 8-те класове на училището, за броя на момичетата в 8-те класове или за

общия брой ученици в 8 клас. Програмата трябва да може да прави и справка за отделен клас или за два избрани класа За целта потребителят трябва да може да избира чрез радио бутоните какво да се сумира, а чрез кутиите за отметки – за кои класове да се сумира.



Фиг. 1.

Решение: Отворете нов проект и го озаглавете `Klas`. Поставете контролите, както е показано на *Фиг. 1* и поставете надписващия текст в съответното свойство.

За реализацията ще ни трябват 6 променливи, в които да запишем броя на момичетата и момчетата от всеки клас. Отделно ще ни трябват две променливи, в които ще сумираме тези бройки, които са маркирани в кутиите за отметки. Опитвайте се да давате на променливите имена, които да напомнят за предназначението им. Нека `bD` и `bM` са имената на тези променливи, в които ще намираме броя на момичетата и броя на момчетата – в случая първата буква е `b` (брой), втората – `D` (дами) или `M` (мъже). По аналогичен начин ще именуваме 6-те променливи, в които ще се въвеждат данните по класове: `bDa`, `bMa`, `bDb`, `bMb`, `bDv` и `bMv` за броя на момичета и момчета от трите класа, като добавената трета буква е буквата на паралелката. По-удобно ще бъде, ако свържем и имената на контролите с данните – например `RadioButtonM` (мъже), `RadioButtonD` (дами) и `RadioButtonO` (общо), както и `CheckBox`*X* за кутиите за отметки и `EditM`*X* или `EditD`*X*, за текстовите кутии за въвеждане на данните, където *X* е една от буквите `A`, `B` или `V`.

Как трябва да работи програмата? Потребителят ще въведе данни в текстовите кутии. При натискане на бутона `Покажи` програмата трябва да извърши изчисленията и резултатът да се покаже в етикета вдясно от бутона (на картинката в него е записана 0). В началото променливите `bD` и `bM` трябва да са инициализирани с 0. След това се проверява кой `CheckBox` е активен. Нека, например, това е само полето за отметка на 8а клас. Тогава се пресмята `bD = bD + bDa` и `bM = bM + bMa`. Аналогично се пресмятат сумите, ако потребителят е избрал и някой от другите два класа. Накрая програмата трябва да провери коя от сумите трябва да се покаже. В зависимост от активния радиобутон, в етикета до бутона `Покажи` програмата трябва да изведе `bD`, или `bM` или `bD + bM`.

Възможно е да се напише по-кратък код, ако вместо да изваждаме съдържанието от кутиите и го записваме в трибуквените променливи `bMa`, `bDa` и т.н., вземаме стойностите за броя на момичетата и момчетата направо от текстовите кутии за съответния клас.

Въпроси и задачи

1. *Разгледайте фрагментите от програми по-долу. За всеки от първите три определете стойността на* `a`*, за четвъртия – какво ще бъде изведено на конзолата.*

```
a = 10;
if (a > 0)
{
  a = a+10;
}
else
a = a-2;
```

```
a = -10;
b = 5;
if (a > 0)
{
  a = a+b;
  a = a*2;
}
a = a-2;
```

```
a = -10;
b = 5;
if (a > 0)
  a = a+b;
else
{  a = a*2;
   a = a-2;
}
```

```
a = 10;
if (a > 0)
{
  Console.Write(a+2);
}
else
Console.WriteLine(a-2);
```

2. *Разгледайте внимателно трите фрагмента от програми по-долу. За всеки от тях определете стойността на променливата* c *и я сравнете с посочената в обяснението стойност.*

```
int a, b, c;

a = 12; b = 6;
if (a>b)
    a=b;
c=a+b;
```

В реда `if (a>b) a=b` се проверява дали a>b. Тъй като 12>6 е вярно, се изпълнява a=b. Променливата **a** приема стойността на **b**. На следващия ред c=a+b променливата **c** става равна на 6+6=12.

```
int a, b, c;

a = 12; b = 6;
if (a<20 && b>20)
  a = b;
c = a + b;
```

В реда `if (a<20 && b>20) a=b` условието **НЕ е вярно**, защото 6>20 и операторът a=b; НЕ се изпълнява. На следващия ред c=a+b променливата **c** става равна на 12+6=18.

```
int a, b, c;

a = 12; b = 6;
if (a<b || b>10)
  a = b;
else
  a = 20;
c= a + b;
```

Отново условието **НЕ е вярно**. Тогава се изпъл-нява оператора след **else**. Променливата **a** става 20. На следващия ред **c** става равна на 20+6=26.

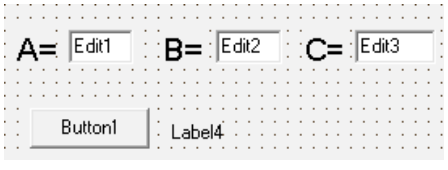
31. Условен оператор – упражнение

Задача 1. Напишете конзолно приложение, което въвежда цяло число и извежда в конзолата подходящо съобщение за това дали числото е по-малко от, равно на или по-голямо от нула.

Задача 2. Напишете приложение, което въвежда цяло число и извежда Да, ако числото е четно, или Не в противен случай.

Задача 3. Напишете приложение, което въвежда показанията на електронен часовник – текущ час и минути от началото на текущия час – и проверява дали са въведени правилно. Часът трябва да е в интервала от 0 до 23, а минутите – от 0 до 59.

Задача 4. Подът на стая ще се покрива с паркет. Дадени са размерите на стаята и колко е цената на 1 кв. м. паркет. Напишете приложение, което въвежда данните и изчислява стойността на необходимия паркет. Програмата трябва да прави проверка и при некоректни входни данни да извежда в конзолата подходящо съобщение.



Фиг. 1.

Задача 5. Напишете фрагмент от програма, която при натискане на Button1 (*Фиг. 1*) показва в Label4 подходящо съобщение, което е отговор на въпроса:

а) Кое от числата *A*, *B* и *C* е най-голямо?

б) Могат ли числата *A*, *B* и *C* да са дължини на страни на триъгълник?

32 Вложени условни оператори

Вложен условен оператор

Понякога се налага в `if` частта, в `else` частта или и в двете да се провери друго условие. Тогава на това място трябва да се изпълни нов условен оператор. В такъв случай казваме, че имаме ***вложен условен оператор***. Да разгледаме няколко примера, в които е използван вложен условен оператор (наричан за по-кратко вложен `if`).

Пример 1:

```
a = -3; b = 3; c = 0;
if (a > 0)
{
   if (b > 0)
   {   c = 20;  }        // ще се изпълни когато a>0 и b>0
   else
   {  c = -20;  }       //ще се изпълни когато a>0 и b<=0
}
Console.WriteLine(c);     //резултат: c=?
```

Опитайте се да определите какво ще изведе в конзолата програма, съдържаща този програмен фрагмент. След това въведете фрагмента и направете от него конзолно приложение. Компилирайте го и проверете дали предположението ви е правилно.

Пример 2:

```
a = -3; b = 3; c = 0;
if (a > 0)
  if (b > 0)
    c = 20;            // ще се изпълни когато a>0 и b>0
  else
    c = -20;        // ще се изпълни когато a>0 и b<=0
Console.WriteLine(c);
```

Пример 2 се различава от Пример 1, само с отсъствието на скоби. Тази разлика обаче е несъществена. Вече споменахме, че ако в `if` частта или `else` частта трябва да изпишем само един оператор, тогава големите скоби не са нужни. Тук си струва да отбележим, че в езика C# няколко поставени в къдравите скоби оператори наричаме ***блок*** (***от оператори***). Блокът от оператори се разглежда като един оператор и може да се поставя навсякъде, където трябва да напишем няколко оператора, а е допустимо поставянето само на един – както е в `if` частта и `else` частта. Другият важен факт, който се илюстрира от този фрагмент е, че `if` частта и `else` частта образуват един оператор, а не два, макар че и двете части завършват с точка и запетая!

Пример 3:

```
a = -3; b = 3; c = 0;
if (a > 0)
  if (b > 0)
    c = 20;
else                  // else се отнася до най-близкия if
  c = -20;
Console.WriteLine(c);
```

Пример 3 се отличава от Пример 2 само по това, че частта `else` не е написана под съответната ѝ част `else`, а по-наляво. Това не бива да ви заблуждава. Отместването на редовете се прави, за да се подобри четимостта на програмата, но не може да измени смисъла ѝ. Правилото е, че всяка `else` част се отнася до най-близката до нея предшестваща я `if` част.

Пример 4:

```
a = -3; b = 3; c = 0;
if (a > 0)
{
   if (b > 0)
      c = 20;          //този ред ще се изпълни когато a>0 и b>0
}
else
   c = -20;              // ще се изпълни когато a<=0
Console.WriteLine(c); // резултат: c=?
```

Фрагментът от Пример 4 се различава съществено от предните три. Обяснете защо. Опитайте се да определите какво ще изведе в конзолата програма, съдържаща този програмен фрагмент. След това въведете фрагмента и направете от него конзолно приложение. Компилирайте го и проверете дали предположението ви е правилно.

Елиминиране на вложен условен оператор

Ако в частта `if` без `else` има вложен `if` оператор без `else` част, тогава вложеният `if` оператор може да бъде елиминиран, като двете условия се съберат в едно с операцията конюнкция. Например, кодът вляво на *Фиг. 1* може да се упрости до кода вдясно на фигурата.

```
int a, b, c;
a = -3; b = 3; c = 0;
if (a < 0)
{
   if (b > 0) { c = 20; }
}
Console.WriteLine(c);
```

```
int a, b, c;
a = -3; b = 3; c = 0;
if (a < 0 && b > 0)
{
   c = 20;
}
Console.WriteLine(c);
```

Фиг. 1.

Работа с компютър

Да напишем програма с графичен интерфейс, която намира периметъра и лицето на правоъгълник така, че да не допуска пресмятането на периметъра и лицето, ако въведената дължина на една от двете страни не е положителна. Проверката за коректност на входните данни в този случай е по-сложна, защото трябва да се проверяват стойностите на две променливи:

```
въвеждане на дължината на страната a
ако a > 0
   въвеждане на дължината на страната b

   ако b > 0
      извеждане на периметъра и лицето          изпълнява се когато a > 0 и b > 0
   иначе
      извеждане на съобщение за некоректно b    изпълнява се когато a>0 И b<=0
иначе
   извеждане на съобщение за некоректно a       изпълнява се когато a<=0
```

Фиг. 2.

Първо да отбележим, че няма смисъл потребителят да въвежда дължината на втората страна, ако е въвел грешно дължината на първата. Или по-точно: ако потребителят въведе некоректна дължина на първата страна, програмата трябва да изведе съобщение за грешка и да преустанови работа. Когато, обаче, първата дължина е коректна, от потребителя се иска да въведе дължината на втората страна. За нея също трябва да се провери дали е положителна – ако да, т.е. и двете дължини на страни са коректни, програмата може да пресметне и изведе резултата. В противен случай програмата ще трябва да съобщи, че втората дължина е грешна и да спре работа.

Получаваме алгоритъма, показан на *Фиг. 2*.

Въпроси и задачи

1. *Дадени са следните фрагменти (Фиг. 2). Определете кои от тях ще довеждат до една и съща стойност на променливата c, независимо от входните данни:*

```
c=5;
if(a > 0)
   if(b>0)
      c=0;
```

```
c=5;
if(a>0 &&b>0)
   c=0;
else    c=3;
```

```
c=5;
if (a>0 && b>0)
   c=0;
```

```
c=5;
if (a>0)
{   if (b>0)
       c=0;
}
else
   c=3;
```

```
c=5;
if (a>0)
   if (b>0)
      c=0;
   else
      c=3;
```

```
c=5;
if (a>0)
{   if (b>0)
       c=0;
    else
       c=3;
}
```

Фиг. 2.

2. *Напишете приложение за подреждане на три числа a, b и c във възходящ ред, използвайки вложен* `if`.

33 Вложени условни оператори – упражнение

Задача. Напишете приложение с графичен интерфейс Figures, което по зададени страни да може да намира лицето и/или периметъра на избрана от потребителя фигура – квадрат, правоъгълник или триъгълник.

Анализ. „Намира лицето И/ИЛИ периметъра“ означава, че програмата трябва да може да извежда периметъра на фигурата, лицето ѝ или и двете. Затова потребителят трябва да може да избере по някакъв начин какво да се пресметне и покаже. Контролата, която позволява такъв избор, е `CheckBox`. Следователно ще имаме две такива контроли във формата.

Второто изискване на задачата е потребителят да може да избира фигурата, за която ще се извършат пресмятанията. Това означава, че трябва да може да се избере само една от фигурите. Контролите, с които може да осъществим този избор, са три радио бутона.

За въвеждане на страните на фигурите е най-естествено да се използват контроли `TextBox`. Въпросът е колко? Изглежда, че ще ни трябват 6 текстови кутии: една за въвеждане страна на квадрата,

две за страните на правоъгълника и три за страните на триъгълника. Можем ли да намалим броя на текстовите кутии? Отговорът е „Да“. С неголеми усилия можем да намалим броя на необходимите ни текстови кутии до три.

За въвеждане на страните на триъгълника винаги ще ни трябват три текстови кутии и затова, когато потребителят избере радио бутона на триъгълника, и трите текстови кутии трябва да са видими във формата. Ако се избере радио бутонът на правоъгълника, видими трябва да останат само две от текстовите кутии, а ако активен е радио бутонът на квадрата – видима трябва да е само една от кутиите. Видимостта на контролата се задава от стойността на атрибута `Visible`, която може да е `true` – това означава, че контролата трябва да се вижда, или `false` – за да не се вижда.

Нека да именуваме радио бутоните за квадрат, правоъгълник и триъгълник с `rbK`, `rbP` и `rbT`, съответно. Нека трите `TextBox`-а, в които ще въвеждаме страните A, B и C да са с имена `tbA`, `tbB` и `tbC`, а надписите им – `lbA`, `lbB` и `lbC`. Всички променливи, в които ще съхраняваме дължините на страните (`A`, `B` и `C`), лицата (`sK`, `sP` и `sT`) и периметрите (`pK`, `pP` и `pT`) на фигурите, ще бъдат от тип `double`. На кутиите за отметки, чрез които избираме периметър или лице, ще дадем имена `cbP` и `cbS`, съответно.

Когато е активен радио бутонът `rbK` – ще се вижда само кутията `tbA`, при активен `rbP` ще се виждат кутиите `tbA` и `tbB`, при активен `rbT` – ще се виждат и трите текстови кутии. Ще ни е нужен и един бутон, с който потребителят да стартира пресмятането, когато е избрал фигурата, лице или периметър и е въвел данните. Нека този бутон е с име `bStart` и е надписан с думата `Пресметни`.

Сега трябва да напишем код за следните стъпки:

При избор на фигура. По-долу е даден кодът за обработка на събитието `OnClick` на `rbP`. Когато потребителят натисне радио бутона за правоъгълника, трябва да скрием `tbC` и нейния етикет, а да направим видими `tbA`, `tbB` и техните етикети. Това се реализира чрез следния фрагмент (за останалите два случая кодът е подобен):

```
tbA.Visible=true; tbB.Visible=true; tbC.Visible=false;
lbA.Visible=true; lbB.Visible=true; lbC.Visible=false;
A=B=C=0.;
tbA.Text= tbB.Text= tbC.Text="";
```

Забележете, че нулираме трите променливи, в които ще съхраняваме въведените дължини и изчистваме съдържанието на текстовите полета, за да види потребителят, че не са въведени данни.

Избор на лице или периметър. Тук не са нужни никакви обработки. Достатъчно е да направим поне една от двете кутии за избор активна, за да е зададена една от трите обработки – пресмятане на лице, пресмятане на периметър или и двете.

Прочитане на данните. За всяка една от видимите кутии пишем код за обработване на събитието `OnClick`, в който присвояваме на съответната променлива въведеното от потребителя число. Например, `A = double.Parse(tbA.Text);`

При натискане на бутона `bStart`. Изчисляваме периметъра и обиколката на избраната фигура:

```
pK = 4 * A; sK = A * A;   // периметър и лице на квадрата
```

или

```
pP= 2 * (A + B); sP = A*B;   // периметър и лице на правоъгълника
```

или

```
// периметър и полупериметър на триъгълника
pT = A + B + C; p2 = pT/2;
// лице на триъгълника
sP= Math.Sqrt(p2 * (p2 - a) * (p2 - b) * (p2 - c));
```

Последната формула навярно ви е непозната. Тя се нарича Херонова формула за лице на триъгълника, зададен с трите си страни. С `double p2;` в нея е означен полупериметърът на триъгълника, а `Math.Sqrt` е стандартната функция за намиране на квадратен корен. След като сме пресметнали лицето и периметъра на избраната фигура извеждаме това, което е посочено в кутиите за избор `cbP` и `cbS`.

34 Циклични алгоритми. Оператор за цикъл с брояч

Цикли

От урока за алгоритми знаем, че една от най-често използваните конструкции в алгоритмиката е ***цикличната***, при която няколко действия се повтарят многократно, докато е изпълнено някакво условие. Например, цикъл има в алгоритъма за събиране на числа в позиционна бройна система, като при всяка стъпка на цикъла трябва да се съберат поредните две цифри и преносът, остатък от деленето на тази сума на основата на бройната система да се запомни като поредна цифра на сумата, а частното да стане пренос. Припомнете си и други алгоритмични процедури, които познавате и в които има повтарящи се стъпки.

Цикличната алгоритмична конструкция се реализира в програма чрез ***оператори за цикъл***. Всеки език за процедурно програмиране притежава поне един такъв оператор. В езика C# има три оператора за цикъл – операторът `for`, операторът `while` и операторът `do...while`. В този урок ще се запознаем с един от тях – операторът за цикъл `for` (англ. за). Предназначението му е за създаване на цикли, в които тялото на цикъла трябва да се изпълни **за зададени стойности** на променлива или променливи.

Операторът for

Операторът за цикъл `for` има следния синтаксис:

```
for (<инициализация>; <условие>; <обновяване>)
{   <тяло на цикъла> }
```

Изпълнението на оператора започва с изчисляване на израза *<инициализация>*. В него обикновено се задава начална стойност на променливата, която управлява изпълнението на цикъла. След това започва повтаряне на следните три действия:

- изчислява се логическият израз *<условие>*. Ако стойността му е `false` – изпълнението на оператора за цикъл се прекратява. Ако стойността му е `true`, се продължава със следващите две стъпки;
- изпълнява се блокът от оператори, наричан *<тяло на цикъла>*;
- изчислява се изразът *<обновяване>*. В него, обикновено, се променя стойността на променливата, управляваща изпълнението на цикъла. След това отново се изчислява изразът *<условие>* и т.н.

Забележете, че трите израза са поставени в кръгли скоби, както условието на условния оператор, и са разделени един от друг с точка и запетая. Тялото на цикъла, както в условния оператор, е блок от оператори, и когато съдържа повече от един оператор – трябва да се поставя в къдрави скоби, а когато се състои само от един оператор – тогава скобите не са задължителни. Препоръчваме скобите да се поставят, за да се чете по-лесно кодът и да не се допуска грешка при добавяне на втори оператор, който без скобите няма да влезе в тялото на цикъла.

Операторът има два много често използвани варианта:

```
for([<тип>] <променлива> = <начална стойност>;
           <променлива> <= <крайна стойност>;
    <променлива> = <променлива> + <стъпка>
      {<блок от оператори>}
```

и

```
for([<тип>] <променлива> = <начална стойност>;
           <променлива> >= <крайна стойност>;
    <променлива> = <променлива> - <стъпка>
      {<блок от оператори>}
```

Променливата, която управлява изпълнението на цикъла, може да бъде от кой да е числов тип. Началната стойност е произволна константа, която в първия вариант трябва да е по-малка или равна на крайната стойност, а във втория вариант – по-голяма или равна на крайната стойност. Типът на променливата, управляваща цикъла, не трябва да се задава, ако променливата е декларирана извън цикъла. Когато променливата е декларирана в цикъла, тя е локална и не е използваема извън цикъла.

Да разгледаме действието на оператора в първия вариант. Изпълнението на цикъла започва с инициализация на променливата на цикъла, на която се присвоява началната стойност. Проверява се дали е изпълнено условието, т.е. дали стойността на променливата е по-малка от или равна на крайната стойност. Ако е така, се изпълнява блокът от оператори. След това променливата се увеличава със стъпката и всичко се повтаря – отново се проверява дали условието е изпълнено, и ако е така отново се изпълнява блокът от оператори. Операторът спира действието си, когато променливата стане по-голяма от крайната стойност.

Във втория вариант действието е същото, с разликата, че променливата всеки път намалява със стъпката и операторът спира действието си, когато променливата стане по-малка от крайната стойност. Най-често използваме оператор `for`, при който началната и крайната стойност са цели числа, а стъпката е 1.

Ако блокът от оператори се състои само от един оператор, може да не се пишат скобите `{}`, но препоръчваме това да не се прави.

Пример 1.

```
int i;for (i=1; i<=10; i++) { <тяло на цикъла> }
```

В този пример в тялото на цикъла променливата `i` ще приеме стойностите 1, 2, 3, …, 10. Защо обръщаме внимание на „в тялото на цикъла“? Защото когато `i` стане 10, изразът `i++` се пресмята отново, вследствие на което `i` става 11. Сравнението `i <= 10` вече не е истина, изпълнението на цикъла се прекратява, а стойността на `i` остава 11. Когато говорим за стойност на променливата, управляваща цикъл, ще имаме предвид стойността ѝ в тялото на цикъла.

Пример 2.

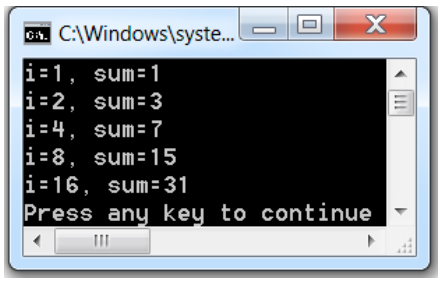
```
int i;for (i=10; i>=1; i--) { <тяло на цикъла> }
```

В тялото на този цикъл променливата `i` ще приема стойностите 10, 9, 8, …, 1. Стойността на `i` след свършване на цикъла ще бъде 0. И в двата примера стъпката на цикъла е 1. Ако искаме стъпката да е различна от 1, трябва да променим оператора по следния начин:

```
for (int i=1; i<=10; i=i+k) { <тяло на цикъла> }
```

където новата стъпка е `k`. Например, в цикъла `for (int i=1; i<=10; i=i+2)`, променлива `i` ще приеме стойностите 1, 3, 5, 7 и 9, а в `(int i=30; i>=1; i=i-5)` стойностите на `i` ще са 30, 25, 20, 15, 10 и 5.

Други възможности



Фиг. 1.

Всъщност разгледаните до тук възможности на оператора `for` са много специални случаи. Както се вижда от дефиницията на оператора, трите израза може да са най-различни и с един оператор могат да се извършат много работи. Следващият пример е цикъл, в който се инициализират и променят на всяка стъпка две променливи. Правилото е, че на мястото на всеки от изразите може да се напишат няколко израза, разделени със запетаи:

```
for (int i = 1, sum = 1; i <= 16; i = i * 2, sum = sum + i)
        Console.WriteLine("i={0}, sum={1}", i, sum);
```

Резултатът от изпълнението на цикъла е показан на *Фиг. 1*. В този пример и двете променливи растат, но е възможно едната променлива да расте, а другата – да намалява. Всъщност тялото на ци-

къла `for` може да е празно. Така например цикълът, който намира сумата на числата от 1 до 10, може да се изпише и без тяло:

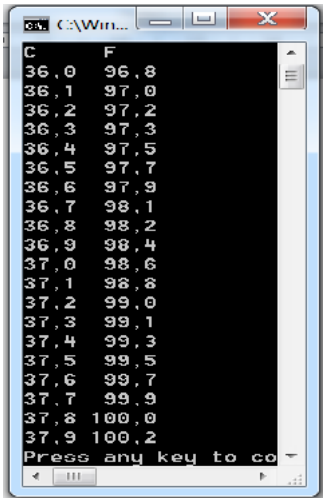
```
for(int sum=0, i = 1; i <= 10; sum = sum + i, i++);
```

Нещо повече, нито един от изразите в оператора не е задължителен – инициализацията може да се направи вън от оператора за цикъл, а проверката на условието и обновяването на управляващите променливи – в тялото на цикъла. В такъв случай, изпълнението на цикъла трябва да се прекрати явно с оператора `break;`:

```
int i = 1, sum = 0;
for (;;) { sum = sum + i; i++; if (i > 10) break; }
```

Работа с компютър

Задача 1. За упражнение нека напишем функция, която съставя таблица за преминаване от температурната скала на Целзий, към температурната скала на Фаренхайт. Формулата за преизчисляване на температурата е $f = 9/5.c + 32$, където c е температурата по скалата на Целзий, а f – съответната ѝ температура по Фаренхайт. В таблицата ще включим температури от 36^0C до 38^0C, през една десета от градуса. Дробната променлива `c`, в която ще получаваме поредната температура по Целзий, ще бъде управляваща на цикъла. Стойността, с която всеки път променяме управляващата променлива, наричаме стъпка на цикъла.



Фиг. 2.

Ще ни е необходима още една дробна променлива `f` за температурата по Фаренхайт. На променливата `c` ще даваме последователно стойностите 36.0, 36.1, 36.2 и т.н. докато стигнем до 37.9. и за всяка стойност на `c` ще пресмятаме съответната стойност на `f` по формулата. За целта ще съставим цикъл `for`, в израза-инициализация на който ще дадем на `c` началната стойност `c = 36.0`. Вторият израз ще бъде условието, при което цикълът да продължи да се изпълнява: `c < 38.0`, а в израза-обновяване ще увеличаваме всеки път стойността на управляващата променлива със стъпката: `c = c + 0.1`. В тялото на цикъла ще пресмятаме в `f` температурата по Фаренхайт и ще извеждаме двете стойности в таблицата:

```
static void Main(string[] args)
    {   double c, f;
        Console.WriteLine("C     F");
        for (c = 36.0; c < 38.0; c = c + 0.1)
        {   f = (9.0 / 5.0) * c + 32.0;
            Console.Write("{0,4:##.0} ", c);
            Console.Write("{1,5:##.0}\n", f);
        }
    }
```

Задача 2. Напишете приложение CelFar, в което да поставите тази главна функция. Компилирайте го и го изпълнете. Резултатът трябва да е този, показан на *Фиг. 2.* Проверете какво ще стане с резултата, ако вместо форматиращите елементи `{0,4:##.0}` и `{1,5:##.0}` напишете `{0,4:##.#}` и `{1,5:##.#}`. Опитайте се да обясните разликата в използване на `#` и `0`, при задаване броя на знаците след десетичната точка на извежданото дробно число.

35 Оператор за цикъл с брояч – упражнение

Задача 1. Напишете конзолно приложение maxval, което намира най-голямото от зададени цели числа. Програмата първо трябва да въведе от клавиатурата ред с броя *n* на числата, $n > 0$, а след това по едно от числата на всеки от следващите *n* реда. На конзолата програмата трябва да изведе най-голямото от *n*-те числа.

ПРИМЕР:

Вход:	Изход:
4	23
6	
12	
5	
23	

Решение: Ще намерим в променливата `maxN` най-голямото от въведените числа. Ще инициализираме тази променлива с най-малката стойност на типа `int`, която е зададена в атрибута `MinValue` на класа `int`. След това ще въведем *n* и ще направим цикъл с брояч от 1 до *n*, който въвежда зададените числа и за всяко число проверява дали не е по-голямо от намерения до момента в `maxN` максимум. Ако това е така, ще заменяме съдържанието на `maxN` с новото по-голямо число. Програмата е показана на *Фиг. 1*.

```
static void Main(string[] args)
{ int n; string s;
  s= Console.ReadLine();
  n = int.Parse(s);
  int maxN=int.MinValue;
  for (int i = 1; i <= n; i++)
  {  s = Console.ReadLine();
     int number = int.Parse(s);
     if (number > maxN)
        maxN = number;
  }
  Console.WriteLine("Max:{0}",maxN);
}
```

Фиг. 1.

Задача 2. Направете необходимите промени в програмата maxval така, че да намира най-малкото от зададените числа.

Задача 3. Напишете конзолно приложение sumval, което да намира сумата на зададени цели числа. Програмата първо трябва да въведе от клавиатурата ред с броя *n* на числата, $n > 0$, а след това *n* реда, с по едно от числата на всеки от тях. На конзолата програмата трябва да изведе сумата на *n*-те числа.

ПРИМЕР:

Вход:	Изход:
5	62
6	
12	
5	
23	
16	

```
static void Main(string[] args)
{   int n; string s;
    s= Console.ReadLine();
    n = int.Parse(s);
    int sum=0;
    for (int i = 1; i <= n; i++)
    {   s = Console.ReadLine();
        int number = int.Parse(s);
        sum = sum + number;
    }
    Console.WriteLine("Sum={0}",sum);
}
```

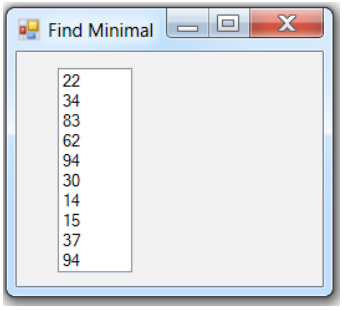
Фиг. 2.

Решение: Също както в предната задача трябва да организираме цикъл, който въвежда *n*-те числа и добавя всяко от числата в променливата за натрупване на сумата, която предварително е инициализирана с нула (*Фиг. 2*)!

Компонента ListBox

След като вече знаем какво е циклична конструкция в алгоритъм и предназначението на оператора `for`, ще разгледаме още един пример за употреба на този оператор, но в приложение с графичен интерфейс. За целта ще се запознаем с един нов елемент на графичния интерфейс – ***списъчната кутия*** (`ListBox`, 

). Тази компонента служи за показване на списък от елементи (числа, текст и др.) върху екрана (*Фиг. 3*). Елементите на списъка се съхраняват в колекцията `Items` (англ. елементи) на класа `ListBox`. За това упражнение ще ни трябват два от методите на колекцията:



Фиг. 3. Компонента ListBox

- методът `Add(`*<елемент>*`)` добавя зададения като аргумент елемент в списъка на кутията;
- методът `Clear()` е без аргументи и изтрива всички добавени до момента елементи в списъка, ако има такива. В резултат списъкът на кутията ще бъде празен.

Задача 4. Напишете програма, която генерира десет случайни числа в интервала от 1 до 100, показва ги в списъчна кутия, след което намира минималното измежду генерираните числа. Потребителят трябва да може, с натискане на съответен бутон, да генерира многократно числа и с натискане на друг бутон да намира минималното измежду генерираните числа.

Да започнем със създаването на списъчната кутия и генериране на числата за списъка. Създайте празно приложение с графичен интерфейс с име FindMin. Във формата, от която ще се създаде прозореца на приложението, добавете една списъчна кутия `listBox1` и един бутон button1, надписан с текста Генериране.

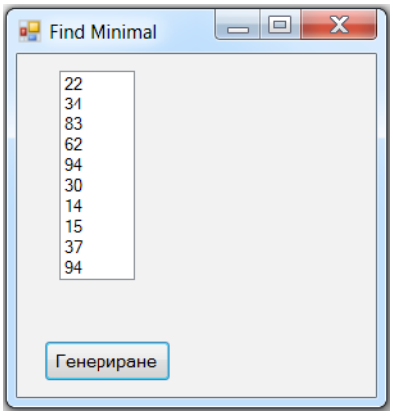
В метода `button1_Click` ще поставим код, който ще извърши генерирането на десетте числа и добавянето им в списъка на кутията listBox1. Заготовката за метода е в страницата Form1.cs, която се отваря с двукратно щракване върху бутона. За генериране на случайно число използвайте класа `Random`. Създайте обект `r` от класа `Random`, за да генерирате чрез него случайни числа:

```
Random r = new Random();
```

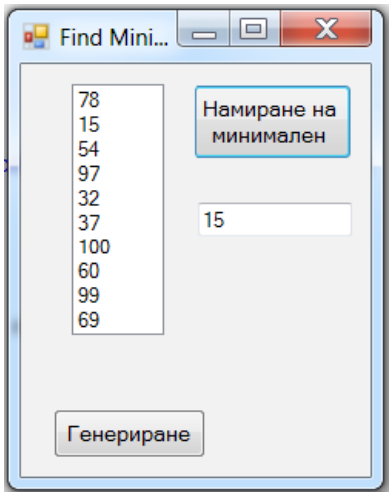
Сега можете да извикате 10 пъти метода `Next()` на обекта `r`, за да създаде 10 случайни числа, но очевидно по-бързо и по-лесно за изписване е, да използвате цикъл с брояч от 1 до 10. В тялото на този цикъл трябва да включите две действия, които да се изпълняват на всяка стъпка – генериране на случайно число от 1 до 100 и добавяне на това число в колекцията Items на списъчната кутия `listBox1`. За добавянето ще използваме метода Add(*<елемент>*) на колекцията. Така получавате кода:

```
private void button1_Click(object sender, EventArgs e)
{  Random r = new Random();
   for (int i = 1; i <= 10; i++)
   {  int number = r.Next()%100+1;
      listBox1.Items.Add(number);
   }
}
```

Поставете този код в програмата, компилирайте я и проверете работоспособността ѝ (*Фиг. 4*). Ако списъчната кутия не е достатъчно висока, за да събере десетте числа, влезте в страницата Form1.cs [Design], и с влачене на мишката увеличете височината на контролата доколкото е необходимо, отново компилирайте и проверете дали кутията вече е достатъчно висока.



Фиг. 4.



Фиг. 5.

Сигурно сте забелязали, че ако натиснете бутона Генериране повторно, то новогенерираните числа се добавят след предишните. Това, разбира се, е нежелателно и трябва да бъде направено съответно изменение в кода. Всеки път, когато потребителят натисне бутона Генериране, съдържанието на списъчната кутия трябва да се изчисти. За целта използвайте метода `Clear()` на колекцията Items. Извикайте метода: `listBox1.Items.Clear();` преди началото на цикъла, в който се генерират числата. Компилирайте и проверете работоспособността на новата версия.

Следващата стъпка е да се добави една текстова кутия и един бутон. При щракване върху този бутон, програмата трябва да намери и покаже в текстовата кутия най-малкото от случайно генерираните числа. Влезте в страницата Form1.cs [Design] и добавете във формата бутон с име button2, надписан с текста Минимален елемент, и текстова кутия textBox1.

Най-добре е намирането на най-малкото число да стане още по време на генерирането. Както вече постъпихте в предишен урок, декларирате променлива `minN` в рамките на класа `Form1`, а не на метода на button1, за да е видима и в метода `button2_Click`. В тази променлива, сравнявайки съдържанието ѝ с всяко от генерираните числа, ще намерите най-малкото от тях. Променливата трябва да инициализирате преди началото на цикъла с най-голямото число от тип `int` – `int.MaxValue`:

```
int minN;
private void button1_Click(object sender, EventArgs e)
{  Random r = new Random();
   listBox1.Items.Clear();
   minN = int.MaxValue;
   for (int i = 1; i <= 10; i++)
   {  int number = r.Next()%100+1;
      listBox1.Items.Add(number);
      if (number < minN)
        minN = number;
   }
}
```

Сега вече може да напишете код за метода `button2_Click()`, в който на свойството `Text` на текстовата кутия `textBox1` да се присвоява стойността на променливата `minN`. Тъй като свойството `Text` е от тип `string`, а стойността на `minN` е от тип `int`, то при присвояването, числовата стойност трябва да се преобразува в низ с метода `ToString()` на класа `int`.

```
private void button2_Click(object sender, EventArgs e)
{  textBox1.Text = minN.ToString();
}
```

Програмата е готова. Компилирайте я и проверете нейната работоспособност (*Фиг. 5*).

Въпроси и задачи

1. *Какво ще изведе в конзолата всеки от следните програмни фрагменти?*
   а)
```
for(int i=1; i<=10; i=i+2)
Console.WriteLine(i);
```

б) `for(int i = 1, x = 20; i <= 128; i = i * 2, x--)`
`{ Console.WriteLine(„i={0}, x={1}“, i, x); }`

2. *Напишете програма, която въвежда от клавиатурата цяло положително число и проверява дали е просто, като извежда в конзолата подходящо съобщение. Упътване. Едно число е просто, ако се дели без остатък само на 1 и на себе си.*
3. *Напишете програма, която да извежда на конзолата степените на числото 2 от 2^0 до 2^{49}. Упътване. Използвайте метода* `Math.Pow()`.

36 Оператори за цикъл с условие

Вече се запознахме с оператора за цикъл `for`. Той е подходящ за случаите, в които тялото на цикъла трябва да се изпълни за определени стойности на някаква управляваща променлива или определен брой пъти. За случаи, различни от споменатите, езикът C# предоставя две други възможности.

Оператор while

Операторът `while` (англ. докато) проверява стойността на логическия израз и ако тя е `true`, тогава изпълнява тялото на цикъла. Операторът продължава да прави това, **докато** условието е изпълнено. Когато за пръв път стойността на израза стане `false`, операторът прекратява работа и предава управлението на следващия оператор. Тъй като в този оператор условието се проверява преди да се изпълни тялото на цикъла, за оператора казваме, че е с ***предусловие***.

Синтаксисът на оператора `while` е:
`while (`*<логически израз>*`) {`*<тяло на цикъла>*`}`

Всъщност възможностите на оператора `while` не са различни от тези на оператора `for`. На *Фиг. 1* вляво е показан схематично оператор `for`, а в дясно – оператор `while`, който изпълнява абсолютно същите действия.

```
for (<инициализация>; <условие>; <обновяване>)
{
    <тяло на цикъла>
}
```

```
<инициализация>;
while (<условие>)
{   <тяло на цикъла>
    <обновяване>
}
```

Фиг. 1.

Работа с компютър

Задача. Фирма произвела през 2010 г. 130 тона продукция. През всяка от следващите две години производството спадало с 10%. Напишете конзолно приложение, което да определи през коя година то ще е за първи път под 10 тона.

```
x = 130.0
g = 2010
докато x >= 10
   x = 0.9 * x
   g = g + 1
изведи g
```

Фиг. 2.

Решение: Ще използваме две променливи – една дробна `x`, за да помним до колко е спаднала продукцията и една цяла `g` – за да помним поредната година На всяка стъпка в тялото на цикъл ще намаляваме `x` с 10%, а ще увеличаваме `g` с 1. Ще изпълняваме цикъла, **докато** съдържанието на променливата `x` стане по-малко от 10. На *Фиг. 2* алгоритъмът е описан на ограничен естествен език. Кодирайте го на езика C# и създайте от него конзолно приложение с име Firma.

Оператор do...while

Третият оператор за организиране на цикъл в C# е операторът `do...while`. Разликата между него и оператора `while` е, че условието за край на цикъла се проверява след всяко изпълнение на тялото на цикъла. Затова в този случай казваме, че операторът за цикъл е с постусловие.

Синтаксисът на оператора `do...while` е:

```
do {  <тяло на цикъла>  } while (<условие>);
```

Не е невъзможно да се моделира работата на кой да е оператор за цикъл с някой от другите оператори за цикъл, макар че понякога това ще бъде доста изкуствено. На *Фиг. 3* е показан схематично оператор `while`, който изпълнява същото, което и операторът `do...while`. Използва се специалният оператор `break`, предназначен да прекъсва изпълнението на кой да е от операторите за цикъл. Както се вижда, решението е да организираме безкраен цикъл с предусловие логическата константа `true`, който да прекратим с оператора `break`, когато постусловието, при което продължаваме изпълнението на цикъла, вече не е истина. За целта от постусловието на `do...while` правим неговото отрицание.

Друг полезен оператор при организацията на цикли е операторът `continue;`. При изпълнението

```
while (true)
{
  <тяло на цикъла>
  if(!<условие>) break;
}
```

Фиг. 3.

```
while (true)
{
  <тяло на цикъла>
  if(<условие>) continue;
  else break;
}
```

Фиг. 4.

на този оператор се преустановява изпълнението на тялото и се преминава към следваща стъпка на цикъла. Например, моделирането на цикъла с предусловие чрез цикъл с постусловие може да стане и по начина, показан на Фиг. *4*.

Работа с компютър

Задача. Напишете конзолно приложение `sqr`, което да изчислява квадратен корен от зададено цяло число.

Решение: Да означим с *a* зададеното цяло число. Ще използваме изчислителна процедура, която доста се различава от алгоритмите, които вече познаваме. Такава процедура се нарича ***числен ме-***

тод. Ще пресмятаме последователно елементите на безкрайната редица $x_1, x_2, \ldots, x_i, \ldots$ по следното правило: първият член на редицата е половината на a, т.е. $x_1 = a/2$, а всеки следващ член на редицата x_{i+1} се получава от предишния x_i по формулата $x_{i+1} = (x_i + a/x_i)/2$, т.е. $x_2 = (x_1 + a/x_1)/2$, $x_3=(x_2 + a/x_2)/2$ и т.н. Такъв начин на задаване на елементите на безкрайна редица се нарича ***рекурентна зависимост***. Доказано е, че всеки следващ елемент на такава редица е все по-близо и по-близо да квадратния корен на a.

Очевидно е, че елементите на редицата трябва да бъдат пресмятани в цикъл. Проблемът при такава изчислителна процедура е, **кога да се прекрати изпълнението на цикъла**. Тъй като стойностите на редицата непрекъснато намаляват (опитайте се да докажете това!), както и разликата между всеки два последователни елемента на редицата, цикълът може да спре, когато разликата между два съседни члена на редицата стане по-малка или равна на някакво много малко число. Това число се нарича ***приближение*** или ***точност*** на метода. С това ограничение върху броя на стъпките на цикъла получаваме алгоритъма от *Фиг. 5*.

```
1) нека eps = 0.001
2) въведи a
3) x = a/2.0
4) повтаряй
5) xs = x
6) x = (x + a/x)/2
7) докато (xs - x) >= eps
8) изведи x
```

Фиг. 5.

По традиция точността на числения метод се означава с гръцката буква ε (епсилон) и затова ще дадем на съответната променлива името `eps`. Последователните стойности на елементите на редицата ще съхраняваме в променливата `x`.

Забележете ролята на променливата `xs` – в първия оператор от тялото на цикъла в нея се съхранява предишната стойност на `x`, а в следващия оператор в `x` се изчислява новата стойност. Цикълът трябва да продължи да се изпълнява, докато разликата в две последователни стойности стане по-малка от `eps`. Да проследим изпълнението на алгоритъма за `a` = 9:

`eps` = 0.001
`a` = 9
`x` = 9/2=4.50
Първа стъпка на цикъла:
`xs` = 4.50
`x` = (4.50 + 9/4.50)/2 = 3.25
4.50 – 3.25 = **1.25 > 0.001** → връщаме се на ред 4.
Втора стъпка на цикъла:
4. `xs` = 3.25
5. `x` = (3.25 + 9/3.25)/2=3.0096153846....
6. 3.25 – 3.0096153846.... = **0.24... > 0.001**? → връщаме се на ред 4.
Трета стъпка на цикъла:
4. `xs` = 3.0096153846 ...
5. `x` = (3.0096153846 ... + 9/3.0096153846 ...)/2 = 3.00001536 ...
6. 3.0096153846... – 3.00001536 = **0.0096... > 0.001**? → връщаме се на ред 4.
Четвърта стъпка на цикъла:
4. `xs` = 3.0000153846...
5. `x` = (3.0000153846... + 9/3.0000153846...)/2 = 3.00000000003932
6. 3.0000153846 – 3.00000000003932 = **0.00001536...<0.001**? → край
Затова
извеждаме `x` = 3.00000000003932.

Забележете колко бързо стойността в променливата `x` се доближава до квадратния корен на 9. Напишете конзолно приложение, което реализира този алгоритъм. Компилирайте го и проверете работоспособността му. Както при всеки друг числен метод, стойността на резултата е намерена **приблизително**, или **с точност** `eps`. Дайте по-малка стойност на `eps`, например 0.0001 или 0.00001 и вижте как се променя резултатът.

Въпроси и задачи

1. *Дайте примери за изречения на български език, в който се използва предусловие и следусловие. Например, „Докато вали дъжд няма да излизам навън”.*
2. *Напишете логически израз в C#, който съответства на следното твърдение:*
   а) числата a, b и c са положителни;
   б) числото x е в числовия интервал [3;10];
   в) числото p е четно или положително;
   г) числата a, b и c са равни помежду си;
   д) числата a, b и c са различни;
   е) поне две от числата a, b и c са положителни.
   ж) нито едно от числата a, b и c не е отрицателно;
   з) точно две от числата a, b и c са равни помежду си;
   и) числата x и y са координати на точка от първи квадрант на координатната система.
3. *Колко пъти ще се изпълни тялото на цикъла от Фиг. 2?*
4. *Променете програмата* sqr, *така че да показва и междинните резултати, както е в текста на урока.*

37. Оператори за цикъл с условие – упражнение

```
ако (a<b) разменете ги
докато (a % b != 0)
{   r = a % b
    a = b
    b = r
}
```

Фиг. 1.

Задача 1. На *Фиг. 1* е показана версия на алгоритъма на Евклид за намиране на НОД. Напишете програмата GCD с графичен интерфейс, използваща този алгоритъм. Обмислете добре каква проверка за коректност на въведените данни трябва да направите. Компилирайте програмата и проверете работоспособността ѝ. Използвайте същия алгоритъм за написване на програма за намиране на най-малкото общо кратно на две числа.

Задача 2. Направете конзолно приложение, което да превръща температура, зададена в градуси по Фаренхайт, в градуси по Целзий. Температурите по Фаренхайт да са в интервала от 0^0F до 200^0F със стъпка 5. Вместо обичайния за такава задача оператор за цикъл for, използвайте някой от операторите за цикъл с условие. *Упътване.* Формулата за преобразуване от Фаренхайт в Целзий изведете от формулата за преобразуване от Целзий във Фаренхайт, която използвахме в предишен урок.

Задача 3. Реколтата от ябълки през 1990 година е била 20 тона. След това, на всеки две години реколтата намалява с 20%. Направете приложение, което решава следните задачи:

а) намира годината, през която за първи път реколтата е била по-малка от 5 тона;

б) намира годината, през която сумарното количество ябълки ще превиши 90 тона.

38 Файлове

Потоци от данни

Едно от важните приложения на компютрите в ежедневието е за бързо обработване на огромно количество информация. Но данните за такава обработка трябва да дойдат отнякъде и да бъдат въведени в компютъра. Ако те са голямо количество, не е подходящо да се използва графично приложение, тъй като трябва да се създадат твърде голям брой текстови полета. От друга страна, ако се използва интерфейса на конзолно приложение, то въвеждането на данните може да се наложи да се направи многократно, тъй като има вероятност да се допусне грешка, която не може да бъде поправена по никакъв начин, освен чрез започване на въвеждането отново. И двата варианта, разбира се, са много неприятни и неприемливи. Затова в езиците за програмиране е предвидена възможността програмата както да чете входните си данни от файл, така и да извежда изходните си данни във файл. Това не е винаги необходимо, но си представете колко много данни има за учениците от едно училище и колко по-лесно би било, ако те са записани във файлове, които компютрите да обработват.

От изучаваните до този момент приложни програми знаем, че те работят с файлове и имат менюта за отваряне, съхраняване и затваряне на файл. В програми на езика C# можем да направим същото, като за целта файловете се разглеждат като ***потоци*** – подредени последователности от байтове, които се изпращат от едно устройство и се получават в друго устройство (подобен процес наблюдаваме, когато гледаме видеоклип от YouTube, например). Потоците позволяват само последователен достъп до данните.

Всички потоци в езика C# се намират в пространството от имена `System.IO`, което означава, че ако искаме да използваме файлове, в началото на програмите трябва да добавяме директивата:

```
using System.IO;
```

Езикът C# предлага различни потоци, подходящи за различни дейности (`BufferedStream`, `MemoryStream`, `Stream`, `FileStream`, `NetworkStream` и др.), но в повечето програми се използват текстови потоци, които позволяват да се четат и извеждат данни в текстови файлове.

Четене от текстов файл

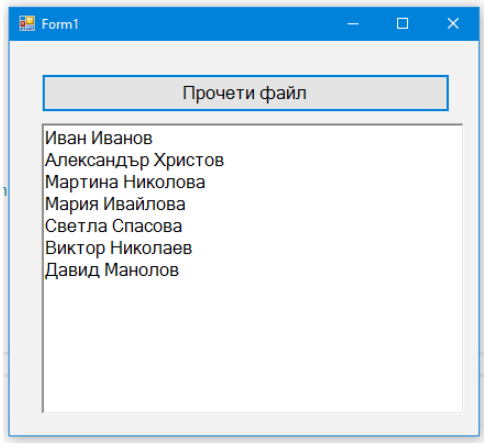
За да накараме компютърът да чете входните данни от текстов файл, първо трябва да го „запознаем“ с файла, т.е. да свържем специална променлива от класа `StreamReader` (***логически файл***, т.е. образ на файла в програмата) с реалния файл, разположен на носител на данни (***физически файл***). Това става по време на инициализация на тази променлива с помощта на специален метод, който се нарича конструктор и се пише след ключовата дума `new` и има същото име като класа:

```
StreamReader r = new StreamReader(<име на файл>,
      Encoding.GetEncoding("windows-1251"));
```

Този метод има два аргумента – първият е името на физическия файл и е от тип низ, а вторият аргумент задава кодирането на файла, за да можем да четем кирилицата от него. Ако файлът не се намира в текущата папка на изпълнимия файл на програмата, тогава трябва да се зададе и пътят до него, като е препоръчително този път да започва от папката на изпълнимия файл – това прави програмата по-независима при евентуалния ѝ пренос на друг компютър.

Самото четене на данни от файла става с помощта на променливата `r`, която играе ролята на логически файл и може да стане по два начина. При първия начин програмата прочита целия текстов файл чрез метода `ReadToEnd()` на класа `StreamReader` наведнъж като един низ и след това може да бъде обработван или показан в интерфейсния елемент `RichTextBox`:

```
string textFile = r.ReadToEnd();
richTextBox1.Text = textFile;
```



Фиг. 1.

Този вариант се предпочита, когато файлът не е много голям (под 1 MB) и не се налага обработка ред по ред.

При втория вариант файлът се чете ред по ред чрез метода `ReadLine()` на класа `StreamReader`, който работи по същия начин като метода `ReadLine()` на класа `Console`. За целта трябва да се използва оператор за цикъл. Най-подходящ е операторът `while`, тъй като обикновено не се знае от колко реда е файлът. Когато се достигне края на файла се прочита неговия знак за край `null` и това може да се използва за условие за край на цикъла. Променливата, която ще се използва в цикъла трябва да бъде от тип низ, тъй като текстовият файл е последователност от редове от тип низ:

```
string line = "";
while(line!=null)
{
        line = r.ReadLine();
        richTextBox1.Text = richTextBox1.Text+line + "\n";
}
```

Този вариант се използва, когато искаме да обработваме входните данни ред по ред. След като приключим работата с файла трябва да го затворим чрез метода `Close()` на класа `StreamReader`:

```
r.Close();.
```

Ако искаме да започнем да четем файла отначало, трябва да го отворим отново с метода-конструктор.

Записване в текстов файл

Понякога се налага изходните данни от една програма да бъдат записани във файл – за да бъдат разпечатани, препратени до друг компютър, да бъдат използвани като входни данни в друга програма и т.н. Записването в текстов файл също е последователно. В езика C# се използват методите `WriteLine()` и `Write()` на класа `StreamWriter`, който работят по същия начин като методите `WriteLine()` и `Write()` на класа `Console`. Първоначално трябва да свържем променлива от класа `StreamWriter` с физическия файл, в който ще записваме. Това отново става по време на инициализация на тази променлива с помощта на специален метод, който се нарича конструктор и се пише след ключовата дума `new` и има същото име като класа:

```
StreamWriter w = new StreamWriter(<име на файл>,<true/false>,
                    Encoding.GetEncoding("windows-1251"));
```

Този метод има три аргумента. Първият е името на физическия файл и е от тип низ. Ако е необходимо, може да се зададе и път до файла, за да бъде посочено мястото за неговото създаване. Ако не се зададе път, то файлът се създава в текущата папка на изпълнимия файл на програмата. Вторият аргумент е от логически тип и указва дали, ако файлът вече съществува, данните да бъдат добавени в края на файла (`true`) или файлът да бъде презаписан (`false`), като се изтрият всички предишни данни в него. Третият аргумент задава кодирането на файла, за да можем да пишем на кирилица в него. Ако файлът не съществува, то той се създава автоматично на указаното място.

Записването на поредния ред във файла става с извикване на метода `WriteLine`, като за аргумент задаваме низа, който трябва да се запише във файла:

```
w.WriteLine("Николай Петков");
```

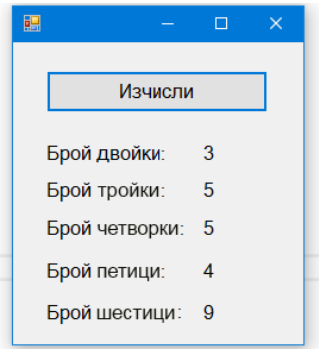
След приключване на записването във файл, задължително трябва да ***затворим*** файла чрез метода `Close()` на класа `StreamWriter`:

```
w. Close();
```

По този начин във файла се добавя знак за край на файла и се извършва съхраняването му. Ако не затворим файла, тогава няма гаранция, че изпратеното действително е записано в него.

Работа с компютър

Да направим графично приложение с име Marks, в което да прочетем от файл с име students.txt оценките на учениците от един клас от контролно по информатика и да извеждаме в пет етикета броя на слабите, средните, добрите, много добрите и отличните оценки.



Фиг. 2.

Решение:

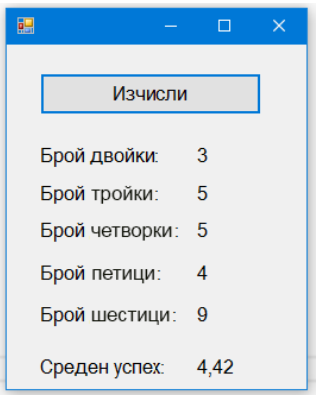
1. Съставяме интерфейса на програмата, който се състои от 10 етикета и един бутон с име `button1` и извършваме настройки на свойствата на формата и компонентите според модела, показан на *Фиг. 2*.

2. Активираме събитието `Click` на бутона, като свързваме с него следния код:

```
private void button1_Click(object sender, EventArgs e)
     {
        StreamReader r = new StreamReader("students.txt",
                              Encoding.GetEncoding("windows-1251"));
           int num2 = 0, num3 = 0, num4 = 0, num5 = 0, num6 = 0;
           string line = "";
           while(line!=null)
           {
                line = r.ReadLine();
                if (line == "2") { num2++; }
                if (line == "3") { num3++; }
                if (line == "4") { num4++; }
                if (line == "5") { num5++; }
                if (line == "6") { num6++; }
           }
           r.Close();
           label2.Text = num2.ToString();
           label4.Text = num3.ToString();
           label6.Text = num4.ToString();
           label8.Text = num5.ToString();
           label10.Text = num6.ToString();
     }
```

Обяснения: Оценките се четат от файла ред по ред и в цикъла всеки път се проверява дали прочетеният ред съвпада с някоя от оценките от 2 до 6 (сравнение на низове). Използват се 5 променливи-броячи, в които се натрупва броят на оценките от всеки вид. След края на цикъла се затваря файлът и се записват съответните оценки в съответните етикети за изход.

39 Циклични алгоритми за работа с файлове – упражнение

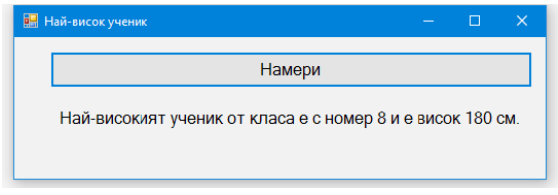


Фиг. 1.

Задача 1. Създайте графично приложение с име Average по модела от *Фиг. 1*, което чете от файл с име students.txt оценките от контролно по информатика на учениците от един клас и извежда в етикет средния успех на учениците от това контролно с точност до втория знак след десетичната точка.

Упътване: Разгледайте внимателно дадения по-долу код, предназначен да обработва събитието `Click` на бутона. Обяснете действието му.

```
private void button1_Click(object sender, EventArgs e)
    {
       StreamReader r = new StreamReader("students.txt",
                          Encoding.GetEncoding("windows-1251"));
          int num = 0;
          double sum = 0;
          string line = "";
          while(line!=null)
           {
                line = r.ReadLine();
                if (line != null)
                {
                     sum = sum + double.Parse(line);
                     num++;
                }
          }
           r.Close();
           sum=Math.Round(sum/num,2);
           label12.Text = sum.ToString();
    }
```



Фиг. 2.

Задача 2. Създайте графично приложение с име Height по модела, показан на *Фиг. 2*, което чете от файл с име students1.txt височината в сантиметри на учениците от един клас, подредени по номера и извежда в етикет номера и височината на най-високия ученик в класа. Ако има повече от един най-висок ученик, програмата извежда този с най-малък пореден номер.

Упътване: Разгледайте внимателно дадения по-долу код, предназначен да обработва събитието `Click` на бутона. Обяснете действието му.

```
private void button1_Click(object sender, EventArgs e)
  {
      StreamReader r = new StreamReader("students1.txt",
                       Encoding.GetEncoding("windows-1251"));
      int num = 0, maxHeight = 0, cnt = 0;
      string line="";
      while((line=r.ReadLine())!=null)
      {
          int height = int.Parse(line);
          cnt++;
          if (height > maxHeight)
          {
              maxHeight = height;
              num = cnt;
          }
      }
      label1.Text = "Най-високият ученик от класа е с номер "
                  +  num + " и е висок " + maxHeight + " см.";
  }
```

40 Изчертаване на графични примитиви

Компютърна графика

С компютърната графика се срещнахме в уроците по ИТ. От тези уроци знаем как компютърът построява различни графични изображения. Компютърният монитор е правоъгълна таблица от квадратчета – ***пиксели***, наречена ***растер***. Таблицата може да има повече или по-малко редове и стълбове. Тази характеристика на графичното поле наричаме ***разделителна способност***. Колкото по-голяма е разделителната способност, толкова по-добри изображения получаваме.

Всеки пиксел може да „свети“ в един от няколко милиона цвята. Компютърното изображение се получава като за всеки пиксел се избере подходящ цвят. Съществуват два принципно различни начина за създаване и съхраняване във файл на компютърна графика:

- **Растерна графика.** Растерното графично изображение (bitmap) е съставено от цветните стойности на всеки отделен пиксел. Растерното изображение е обемисто и след като веднъж е изработено, не може лесно да се промени. При опит да се увеличи или намали, качеството му силно се влошава. Растерни изображения се създават и обработват, например, с познатата ни от уроците по ИТ програма Paint.
- **Векторна графика.** Векторното изображение е съставено от отделни графични елементи – части от прави и криви линии, различни фигури и т.н., всеки от които е представен с математическо описание. Затова представянето е с малък обем. Когато векторното изображение трябва да се представи на екрана, математическото описание се трансформира в множество от цветни точки, които изменят съответните пиксели на екрана. Затова векторните изображения се увеличават или намаляват без да се губи качеството им. Векторната графика също ни е позната от работата с програмите на пакета Microsoft Office.

Езикът C# предлага инструментариум за създаване на векторна графика, включваща и текст – класът `Graphics`, дефиниран в пространството от имена `System.Drawing`. Средата включва автоматично в програмния код на Windows Forms Application това пространство. Класът `Graphics` е им-

плементация на системата за създаване на графични изображение GDI+ (Graphics Device Interface), която е развита версия на WGDI (Windows Graphics Device Interface).

Създаване на приложение с графика

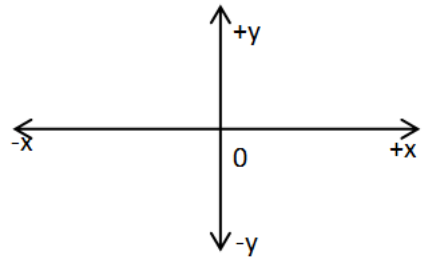
Написването на програма, в която ще създаваме графични изображения, започва с отварянето на приложение с графичен интерфейс. След това трябва задължително **да напишем** в класа `Form1` специален, защитен и предефиниран (`protected override`), метод за обработка на събитието OnPaint с един аргумент – обект `event` от класа `PaintEventArgs`. От свойството `Graphics` на `event` се създава обект `g` от класа `Graphics`, в който можем да създадем нашето графично изображение:

```
protected override void OnPaint(PaintEventArgs e)
{  Graphics g = e.Graphics;...}
```

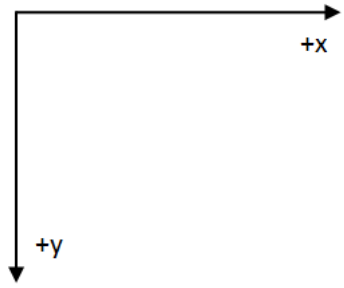
Този метод се изпълнява при отварянето на формата и съобщава на програмата, че вътрешността на екранната форма, няма да се използва за разполагане на компоненти, а за създаване на графика, чрез програмния код, написан за `g`. Изчертаването на графичните обекти трябва да стане също в този метод, на мястото на трите точки.

Чертожно поле

В математиката представянето на обекти в равнината става в ***правоъгълна координатна система***, наричана още ***Декартова***, в чест на френския математик и философ Рьоне Декарт (1596–1650 г). Негова е заслугата за въвеждането на координатна система в равнината и създаването на аналитичната геометрия, която е математическата основа на компютърната графика.



Фиг. 1.



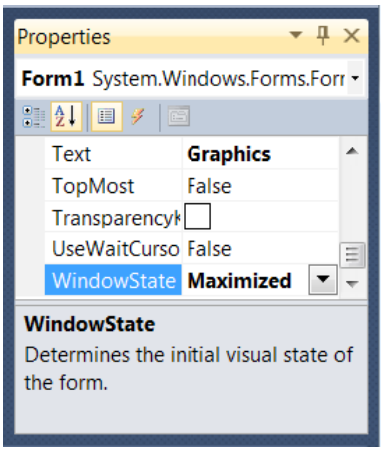
Фиг. 2.

Правоъгълната координатна система (*Фиг. 1*) се дефинира от две перпендикулярни прави в равнината, наричани ***координатни оси*** – ***хоризонтална*** (или ***абсцисна***) ос *Ox* и ***вертикална*** (или ***ординатна***) ос *Oy*. Пресечната точка *O* на двете оси се нарича ***начало*** на координатната система. Точките на всяка от осите съответстват на реалните числа, като началото е нулата на всяка от осите. На абсцисната ос, положителните числа са надясно от нулата, а отрицателните – наляво. На ординатната ос, положителните числа са нагоре от нулата, а отрицателните – надолу. С въвеждането на координатна система, всяка точка *P* на равнината се представя с наредена двойка реални числа (*x*,*y*) – ***координати*** на *P*. Координатата *x* – ***абсцисата*** – е реалното число, съответно на ортогоналната проекция на *P* върху оста *Ox*, а координатата *y* – ***ординатата*** – реалното число, съответно на ортогоналната проекция на *P* върху оста *Oy*. Началото *O* на координатната система е с координати (0,0).

При работа с компютър се използва различна координатна система. В компютърната графика равнината е ограничена до т.н. чертожно поле – правоъгълната част от екранния растер с *r* реда и *c* стълба, която заема вътрешността на екранната форма (*Фиг. 2*). Екранната координатна система, за разлика от Декартовата, е дискретна – екранните точки са пикселите и координатите им са само цели

положителни числа. Пикселът в горния ляв ъгъл на чертожното поле е началото на координатната система – координати (0,0). *x*-координати са целите числа от 0 до *c* – 1 и растат от ляво надясно. *y*-координатите са целите числа от 0 до *r* – 1 и растат от горе надолу (тъй като обновяването на изображението на монитора става с обхождане на пикселите от горе надолу и от ляво надясно).

Реалните размери *r* и *c* на чертожното поле зависят от това, колко голям е прозорецът на програмата. Най-голямо чертожно поле ще получим, разбира се, ако максимизираме прозореца. Това може да стане програмно със задаване стойност Maximized на свойството WindowState на формата (*Фиг. 3*). Текущите размери на чертожното поле на формата се съдържат в свойствата `ClientSize.Width` – ширина и `ClientSize.Height` – височина на чертожното поле.



Фиг. 3.

Създаване на писалка

Когато чертаем върху хартия, използваме молив, химикалка, флумастер или друг пишещ инструмент. Така постъпваме и в компютърната графика. Инструментът, който избираме, се характеризира основно с цвета и дебелината на следата, която оставя в чертожното поле. Тези и други характеристики, са обобщени в класа `Pen` (писалка). Затова, преди да започнем изчертаването, трябва да създадем обект от класа `Pen`:

```
Pen p = new Pen(<цвят>, <дебелина>);
```

Цветът е обект от класа `Color` с 140 свойства, всяко от които е някакъв цвят и се избира от списъчна кутия, която се отваря след написване на знака точка след името на класа. Дебелината на линията е число от тип `float`, и може да считаме, че е закръглено до цяло число. То означава брой пиксели. Всяка стойност на дебелината по-малка от 1 се приема за дебелина 1. Операторът `Pen p = new Pen(Color.Red,-3);`, например, създава писалка с червен цвят и дебелина 1 пиксел.

Ако искаме да създадем писалка с цвят, който не е измежду включените като свойства на класа `Color`, трябва да използваме статичния метод `Color.FromArgb(`<червен>, <зелен>, <син>`)`, който задава цвят, базиран на модела RGB (red-green-blue), т.е. чрез интензитетите (цели числа от 0 до 255) на трите образуващи цвята – червено, зелено и синьо:

```
Pen p = new Pen(Color.FromArgb(49,226,29),2);
```

Трите интензитета на избрания цвят могат да се вземат от цветовата палитра на програмата Paint, като се отвори палитрата и се избере интересуващият ни цвят.

Фонов цвят на чертожното поле

Фоновият цвят на чертожното поле се задава като стойност на свойството BackColor. Друг начин за смяна на фоновия цвят на полето е с метода `Clear` на класа `Graphics`. Той има един параметър, който е обект от класа `Color`:

```
g.Clear(Color.Red);   или
g.Clear(Color.FromArgb(20,140,60));
```

Работа с компютър

Ще напишем програма, която отваря екранна форма, трансформира я в чертожно поле и с оператор за цикъл променя многократно цвета на фона с генериран по случаен начин цвят, при което се получава интересен ефект:

```
protected override void OnPaint(PaintEventArgs e)
{  Graphics g = e.Graphics;
   int a = 256, freq = 10;
   Random c = new Random();
   for (int i = 1; i <= 40; i++)
   {  g.Clear(Color.FromArgb(c.Next()%a,c.Next()%a,c.Next()%a));
      System.Threading.Thread.Sleep(1000);
   }
}
```

Такъв подход се използва при създаването на анимационни ефекти, например. Методът `System.Threading.Thread.Sleep(`*<време>*`)`, който ще използваме при създаването на компютърна графика, забавя изпълнението на следващия оператор за времето в милисекунди, зададено като параметър. Всяко извикване `Next()` на обекта `c` от класа `Random` пък връща едно случайно число, което превръщаме в интензитет (от 0 до 255), като вземем остатъка му по модул 256. Напишете тази програма, компилирайте я и проверете работоспособността ѝ.

41. Изчертаване на линии

Изчертаване на права линия

За изчертаване на права линия между две екранни точки се използва методът `DrawLine` на класа `Graphics`, с параметри писалка и четири цели числа:

```
void DrawLine(Pen p, int x1, int y1, int x2, int y2)
```

Писалката *p* задава цвета и дебелината на линията. Първата двойка числа са координатите на първата точка, а втората двойка числа – на втората. `DrawLine` чертае линия от първата точка до втората точка включително. Например:

```
Pen p = new Pen(Color.Black, 0);
g.DrawLine(p, 0, 0, 15, 15);
```

ще оцвети в черно 16-те пиксела с координати (0,0), (1,1), и т.н., до (15,15), така че в чертожното поле те се виждат като оцветена в черно отсечка. Редът на задаването на двете точки няма значение, затова извикването:

```
g.DrawLine(p, 15, 15, 0, 0);
```

чертае същата отсечка. При задаване на съвпадащи две точки `DrawLine` не чертае нищо.

Работа с компютър

Задача 1. Напишете програма, която изчертава равностранен триъгълник със страна 200 пиксела. Променете програмата така, че дължината на страната да бъде задавана като стойност на променлива (*Фиг. 1*).

Решение: Ще решим втората подзадача. За да се изчертае триъгълник, трябва да се изчертаят трите му страни. За целта, трябва да се намерят координатите на върховете му, тъй като те ще бъдат параметри на метода `DrawLine`. Нека променливите `x1` и `y1` са инициализирани с координатите на долния ляв връх, а променливата `side` с дължината на страната:

```
int x1=200, y1=500, side=200;
```

За да бъде триъгълникът равностранен, стойностите за координатите на другите два върха трябва да бъдат пресметнати математически. Така за координатите x_2 и y_2 на долния десен ъгъл се получава:

```
int x2 = x1 + side, y2 = y1;
```

Малко по-сложно се намират координатите `x3` и `y3` на третия връх. Неговата абциса съвпада със средата на основата на триъгълника:

```
int x3 = (x1 + x2) / 2;
```

Фиг. 1.

а ординатата му се получава от ординатата на левия долен ъгъл, като я намалим с височината на триъгълника. Височината на равностранен триъгълник със страна `side` може да се намери с Питагоровата теорема и е равна на `side.` $\sqrt{3}/2$. Затова да пресметнем стойността на височината и да я преобразуваме в целочислен тип, чрез поставянето на (`int`) пред дробния израз, след което да я извадим от `y1`:

```
int h = (int)(side * Math.Sqrt(3) / 2); y3 = y1 - h; .
```

Сега вече сме готови да изчертаем триъгълника, като изчертаем поотделно всяка от трите му страни:

```
g.DrawLine(p, x1, y1, x2, y2);
g.DrawLine(p, x2, y2, x3, y3);
g.DrawLine(p, x3, y3, x1, y1);
```

Ето и целия програмен код, събран в тялото на метода `OnPaint`:

```
protected override void OnPaint(PaintEventArgs e)
{
   Graphics g = e.Graphics;
   Pen p = new Pen(Color.Black, 2);
   int side=200, x1=200, y1=500, x2, y2, x3, y3;
   x2 = x1 + side; y2 = y1; x3 = (x1 + x2) / 2;
   int h = (int)(side * Math.Sqrt(3) / 2);
   y3 = y1 - h;
   g.DrawLine(p, x1, y1, x2, y2);
   g.DrawLine(p, x2, y2, x3, y3);
   g.DrawLine(p, x3, y3, x1, y1);
}
```

Класът Point

Тъй като в компютърната графика много често работим с точките на екранната координатна система, в пространството от имена `System.Drawing` е дефиниран класът `Point`. Най-важните свойства на обект от класа са абсцисата *X* и ординатата *Y* на точката. За изчертаване на права линия между две екранни точки се използва и версията на метода `DrawLine`, в която вместо четири цели координати се задават две точки:

```
g.DrawLine(Pen p, Point a, Point b).
```

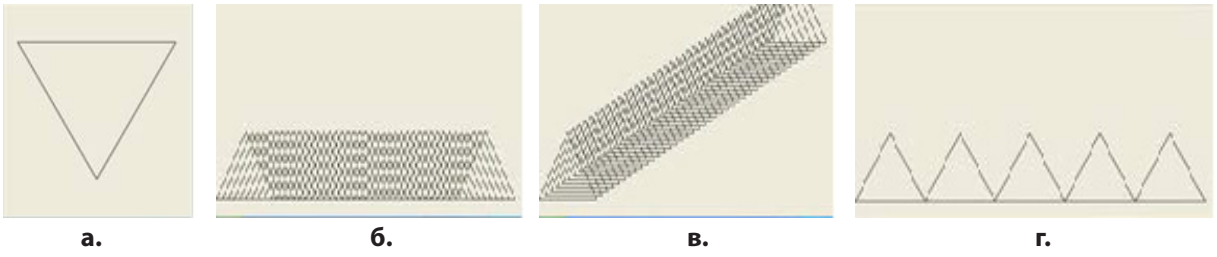
За да стане това, трябва предварително да конструираме точките `a` и `b` от координатите им:

```
Pen p = new Pen(Color.DarkOrchid, 2);
Point a = new Point(123, 24), b = new Point(12, 242);
g.DrawLine(p, a, b);
```

Работа с компютър

Задача 1. Експериментирайте с програмния код на решената в урока задача, като променяте големината на страната в `side` и началното положение на координатите `x1` и `y1`. Променя ли се при това размерът на триъгълника и неговото местоположение? А остава ли триъгълникът равностранен?

Задача 2. Какво трябва да се промени в кода на програмата, която чертае равностранен триъгълник, за да се изчертае триъгълникът обърнат надолу (*Фиг. 2а*)? *Упътване.* Пресметнете отново ординатата на третия връх.



а. **б.** **в.** **г.**

Фиг. 2.

Задача 3. Как бихте направили картинките от *Фиг. 2б, 2в* и *2г*?

Решение: Сигурно се досещате, че при изчертаването е използван оператор за цикъл! Ето как трябва да се промени кодът на програмата от урока, за да се получи втората картинка:

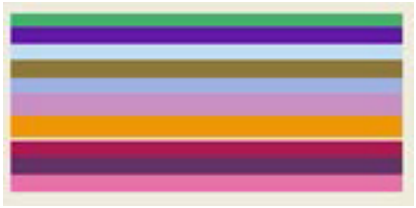
```
protected override void OnPaint(PaintEventArgs e)
{  Graphics g = e.Graphics;
   Pen p = new Pen(Color.Black, 2);
   int side=200, x1=0, y1=500, x2, y2, x3, y3;
   for (; x1 <= 800; x1 += 20, y1-=10)
   {  x2 = x1 + side; y2 = y1;
      x3 = (x1 + x2) / 2;
      int h = (int)(side * Math.Sqrt(3) / 2);
      y3 = y1 - h;
      g.DrawLine(p, x1, y1, x2, y2);
      g.DrawLine(p, x2, y2, x3, y3);
      g.DrawLine(p, x3, y3, x1, y1);
   }
}
```

Опитайте се сами да направите другите две картинки. Може да пробвате да направите и собствени, като, например, променяте в тялото на цикъла и дължината на страната на триъгълника.

Задача 4. Напишете програма, която изчертава 10 хоризонтални линии с дължина 500, разположени плътно една до друга, със случайна дебелина и цвят (*Фиг. 3*).

Решение:

```
protected override void
OnPaint(PaintEventArgs e)
{  Graphics g = e.Graphics;
   Random r=new Random();
   int h=100, red, green, blue;
   for (int i=1; i<=10; i++)
   {  int w=r.Next()%20+10;
      red = r.Next() % 256;
```



Фиг. 3.

```
            green = r.Next() % 256;
            blue = r.Next() % 256;
            Pen p = new Pen(Color.FromArgb(red,green, blue),w);
            g.DrawLine(p, 100, h, 600, h);
            h = h + w / 2;
        }
    }
```

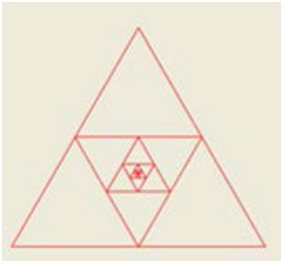
Задача 5. Напишете отново всички програми от урока, като замените извикванията на `DrawLine`, при които като параметри се задават четирите цели координати, с извиквания, при които параметри са началната и крайната точка на изчертаваната отсечка.

42. Изчертаване на линии – упражнение

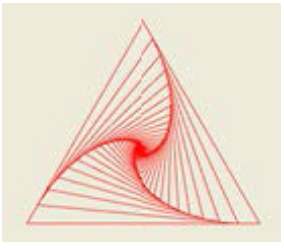
Задача 1. Да се изчертаят 6 вписани един в друг равностранни триъгълници (*Фиг. 1*).

Решение:

```
protected override void OnPaint(PaintEventArgs e)
{  Graphics g = e.Graphics;
   Pen pen = new Pen(Color.Red, 2);
   //изчертаване на първия триъгълник
   int x1 = 100, y1 = 500, side = 500;
   int x2 = x1 + side, y2 = y1;
   int h = (int)(side * Math.Sqrt(3) / 2);
   int x3 = (x1 + x2) / 2, y3 = y1 - h;
   g.DrawLine(pen, x1, y1, x2, y2);
   g.DrawLine(pen, x2, y2, x3, y3);
   g.DrawLine(pen, x3, y3, x1, y1);
   //изчертаване на вписаните триъгълници
   for (int i = 1; i <= 6; i++)
   { int x = x1, y = y1;
     x1 = (x1 + x2) / 2; y1 = (y1 + y2) / 2;
     x2 = (x2 + x3) / 2; y2 = (y2 + y3) / 2;
     x3 = (x3 + x) / 2;  y3 = (y3 + y) / 2;
     g.DrawLine(pen, x1, y1, x2, y2);
     g.DrawLine(pen, x2, y2, x3, y3);
     g.DrawLine(pen, x3, y3, x1, y1);
   }
}
```



Фиг. 1.



Фиг. 2.

Задача 2. Да се изчертае картинката от *Фиг. 2*:

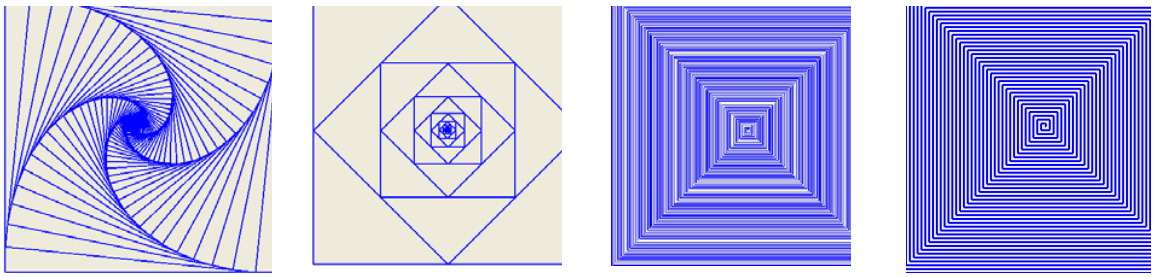
Решение:

```
protected override void OnPaint(PaintEventArgs e)
{  Graphics g = e.Graphics;
     Pen p = new Pen(Color.Blue, 2);
     //задаване на пропорции
     int n = 1, m = 30;
     //изчертаване на първия триъгълник
     int x1 = 100, y1 = 500, side = 500;
     int x2 = x1 + side, y2 = y1;
     int h = (int)(side * Math.Sqrt(3) / 2);
```

```
    int x3 = (x1 + x2) / 2, y3 = y1 - h;
    g.DrawLine(pen, x1, y1, x2, y2);
    g.DrawLine(pen, x2, y2, x3, y3);
    g.DrawLine(pen, x3, y3, x1, y1);
//изчертаване на вписаните триъгълници
    for (int i = 1; i <= 80; i++)
    {  int x = x1, y = y1;
         x1 = (n*x1 + m*x2) / (n+m);
         y1 = (n*y1 + m*y2) / (n + m);
         x2 = (n*x2 + m*x3) / (n + m);
         y2 = (n*y2 + m*y3) / (n + m);
         x3 = (n*x3 + m*x) / (n + m);
         y3 = (n*y3 + m*y) / (n + m);
         g.DrawLine(pen, x1, y1, x2, y2);
         g.DrawLine(pen, x2, y2, x3, y3);
         g.DrawLine(pen, x3, y3, x1, y1);
    }
}
```

Въпроси и задачи

1. *Да се изчертае картинката от Фиг. 3а. Упътване: Променете решението на Задача 2.*
2. *Да се изчертае картинката от Фиг. 3б.*



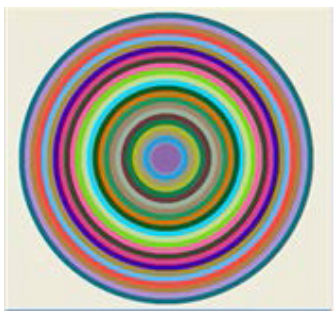
а. б. в. г.

Фиг. 3.

3. *Да се напише програма, която да изчертае квадратна развиваща се спирала (Фиг. 3в и 3г) със страни, успоредни на координатните оси и начална точка в средата на екрана;*

- *Първата страна е с дължина 1 и е ориентирана по положителната посока на абсцисата;*
- *Всяка следваща страна е с дължина с две по-голяма от предишната и е перпендикулярна на нея.*
- *Посоката на развиване на спиралата е обратна на часовниковата стрелка.*
- *Изчертаването продължава, докато се достигне до точка, която е извън чертожното поле.*

4. *Напишете програма, която да изрисува няколко концентрични окръжности, с център средата на чертожното поле, със случайно генерирани радиуси и цветове (Фиг. 4).*



Фиг. 4.

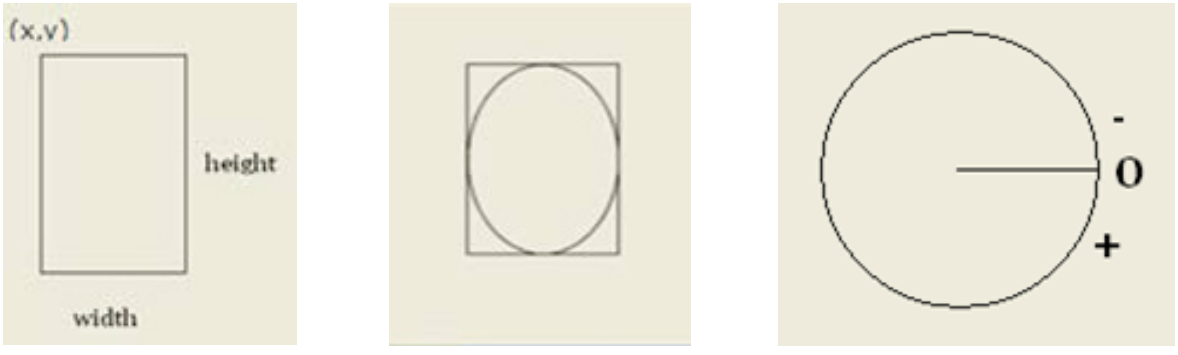
43 Изчертаване на правоъгълник и елипса

Правоъгълник със страни успоредни на координатните оси

Правоъгълник може да се изчертае с четири извиквания на метода `DrawLine` (направете го за упражнение), но тъй като е най-разпространеният обект, то има специален метод за чертане на правоъгълници:

```
void DrawRectangle(Pen p, int x, int y, int width, int height);
```

Писалката, както при чертане на линии, задава цвета и дебелината на линията на контура. Параметрите `x` и `y` съдържат координатите на горния ляв ъгъл на правоъгълника, а параметрите `width` и `height` – задават ширината (дължината на страната, успоредна на абсцисата) и височината (дължината на страната, успоредна на ординатата) на правоъгълника. Ясно е, че ако зададем ширина, която е равна на височината, методът ще изрисува квадрат. Ширината и височината се измерват в пиксели и трябва да са положителни числа (*Фиг. 1а*).



а. **б.** **в.**

Фиг. 1.

Изчертаване на елипса

За изчертаване на елипса се използва методът `DrawEllipse` със същите параметри, както при изчертаване на правоъгълник с метода `DrawRectangle`.

```
void DrawEllipse(Pen p, int x, int y, int weidth, int height);
```

Това е така, защото се задава правоъгълникът, в който е вписана елипсата (Фиг. *1б*). Ако правоъгълникът е квадрат, тогава методът `DrawEllipse` ще изчертае окръжност.

Изчертаване на дъга от елипса

За да се изчертае дъга от елипса, трябва да се зададат същите параметри, които са необходими и за изчертаване на елипса, но допълнително трябва да се посочи къде започва и къде завършва дъгата. По тази причина методът за изчертаване на дъга от елипса `DrawArc` има следния вид:

```
void DrawArc(Pen p, int x, int y,
             int weidth, int height,
             int startAngle, int endAngle);
```

Първите пет параметъра са същите като на метода `DrawEllipse`. Двата допълнителни параметъра са ъглите, които задават началото на дъгата и нейната дължина. Ъглите, които могат да бъдат положителни или отрицателни, се измерват по посока на часовниковата стрелка в градуси, от началото на хоризонталната ос надясно от центъра на елипсата (*Фиг. 1в*). На *Фиг. 2* са показани две дъги, заедно с началния и крайния ъгъл на изчертаването.

`g.DrawArc(p,200,100,300,250,0,90)`

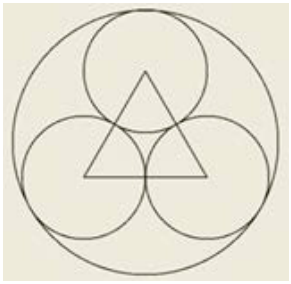
а.

`g.DrawArc(p,200,100,300,250,0,180)`

б.

Фиг. 2.

Работа с компютър



Фиг. 3.

Задача. Напишете програма, която да изчертава чертежа, показан на *Фиг. 3* – равностранен триъгълник, разположен в центъра на чертожното поле, с три взаимнодопирателни окръжности с центрове в трите върха на триъгълника и окръжността, до която са вътрешно допирателни тези три окръжности. *Упътване.* За изчертаване на равностранния триъгълник използвайте програмата, която правихме в предишен урок. Докажете, че диаметърът на малките окръжности е равен на страната на триъгълника, а радиусът на голямата е сума от радиуса на малката окръжност и 2/3 от височината на триъгълника.

44. Изчертаване на правоъгълник и елипса – упражнение

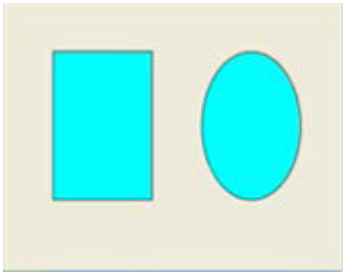
Методи за запълване

За създаването на по-интересни графики, е необходимо да запълваме изчертаните фигури с цвят. Някои от методите на `Graphics`, разгледани дотук, дефинираха затворени области, но чертаеха само контура на областта и не запълваха вътрешността ѝ. Освен тези методи – с префикс на името `Draw`, които изчертават затворени области – съществуват и методи, имената на които започват с `Fill` (запълвам), които запълват вътрешността на съответната фигура (*Фиг. 1*). Първият параметър на тези методи е четката – обект от класа `Brush`, с която се запълва затворена от контур на фигура област. Ето как се създава нова четка:

```
Brush b = new SolidBrush(<цвят на четката>);
```

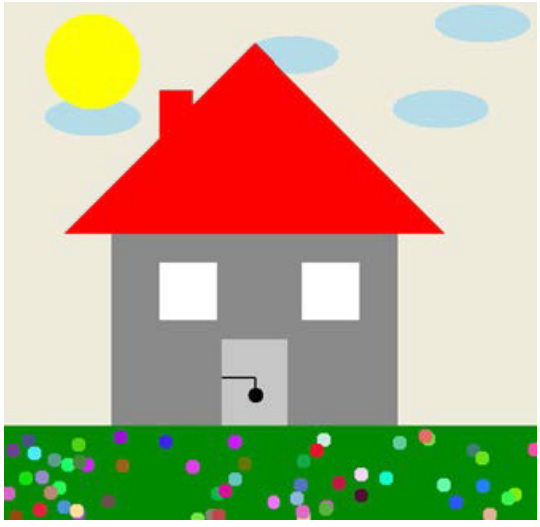
Препоръчително е извикването на метод за запълване да се поставя преди извикването на метода за изчертаване контурите на съответната затворена фигура. Параметрите на метода с префикс на името `Fill` трябва да се същите като параметрите на съответния му метод `Draw`.

```
protected override void OnPaint(PaintEventArgs e)
{  Graphics g = e.Graphics;
   Pen p = new Pen(Color.Black, 2);
   Brush b = new SolidBrush(Color.Aqua);
   g.FillRectangle(b, 100, 100, 200, 300);
   g.DrawRectangle(p, 100, 100, 200, 300);
   g.FillEllipse(b, 400, 100, 200, 300);
   g.DrawEllipse(p, 400, 100, 200, 300);
}
```



Фиг. 1.

Задача. Напишете графични приложения с имена Picture1 и Picture2, които изрисуват графики, подобни на тези от *Фиг. 2* и *Фиг. 3*.



Фиг. 2.

Фиг. 3.

V Съставни типове данни

45 Тестване и верификация на програми

В този раздел програмите ще станат по-трудни и създаването на работоспособна програма ще включва и такава важна дейност като ***тестването***.

Програмистът носи отговорност не само за съставянето на програма за решаване на поставена задача, но и за удовлетворяване на някои изисквания към нея – да е разбираема, правилна, достатъчно бърза, да реализира исканите функционалности, да има дружелюбен потребителски интерфейс и т.н. Съвкупността от свойствата на програмата определят нейното ***качество***.

Качество на програмите трудно се дефинира, но лесно се разпознава при използването им. Качеството трябва да се вгражда в програмите през всички етапи на създаването им, започвайки още от възникването на идеята за някаква програма. Това се постига чрез дейностите ***верификация*** и ***валидация***. Верификацията трябва да отговори на въпроса дали разработваме правилно програмата, а валидацията – дали разработваме правилния продукт. В по-нататъшното изложение ще се придържаме към следната дефиниция: ***Качествена*** е тази програма, която удовлетворява формулираните към нея изисквания.

Всяко отклонение от изискванията ще наричаме ***дефект***. Дефектите обикновено се дължат на една или няколко ***грешки***. Грешка е всяка неправилна стъпка при създаване на програмата, волно или неволно преиначаване на обект или процес. В зависимост от това къде се откриват, грешките могат да бъдат грешки в заданието, в проекта, в самата програма, в документацията, в тестовите данни и др. В този урок ще се спрем само на грешките в програмите, дейностите за откриването и отстраняването им и средствата, подпомагащи тези дейности.

Тестване и настройване

Причините за грешки в програмите могат да бъдат различни:

а) ***алгоритмични*** – неправилно избран или неточно програмиран алгоритъм;

б) ***програмни*** – неправилен избор или използване на конструкция на езика за програмиране;

в) ***технологични*** – неправилно въведена програма или неправилно подготвени входни данни;

г) ***системни*** – несъобразяване с използваната технологична среда (хардуер, ОС, система за управление на базата от данни и т.н.).

Двете основни дейности, свързани с откриването и отстраняването на грешки в програмата са тестването и настройване. ***Тестване*** **(*testing*)** наричаме внимателното наблюдение на работата на програмата върху реални или подходящо избрани, близки до реалните, данни за установяване на съ-

Фиг. 1.

ответствието ѝ с различни правила и изисквания. ***Настройване* (*debugging*)** наричаме локализиране и отстраняване на установени грешки.

Дейностите настройване и тестване се различават по основното си предназначение, по използваните методи и по нивото на сложност на откриваните грешки. Когато настройването завърши, е ясно, че програмата решава някаква задача. Предназначението на тестването е да провери точно това ли е задачата, която е трябвало да бъде решена. Връзката между тестването, настройването и поправянето на грешки може да се илюстрира чрез схемата на *Фиг. 1*.

Тестването се осъществява в три основни стъпки: планиране, реализация и отчитане. При ***планирането*** на тестването се определя целта на тестването, какво да се тества, кога, с какви данни, как и кой да го извършва. ***Реализацията*** на тестването се описва чрез сценарии за тестване. ***Отчитането*** се извършва чрез анализ на документираните резултати и прилагане на критерии за изчерпателността и обхвата на тестването.

Има два основни подхода за тестване: ***структурен*** и ***функционален.*** Целта на ***структурното тестване*** е да се изберат такива тестови данни, че да се осигури преминаване през всички програмни части на системата. При ***функционалното тестване*** се проверява правилността на реализираните основни функции.

В зависимост от това кой извършва тестването, то може да бъде:

- ***вътрешно тестване*** – от самите разработчици. Всеки програмист проверява правилността на създадените от него програми. Преимущество на този подход е, че не е необходимо допълнително разучаване на програмите. Недостатък е, че програмистите имат склонност към надценяване на възможностите си и увереността, че не допускат грешки, пречи на систематичното тестване. Освен това, програмистът несъзнателно избягва да тества функционалност, в изправността на която не е вътрешно уверен.
- ***независимо тестване*** – от независими потребители или от експерти, които не са участвали в разработването на програмите, които ще направят това тестване добросъвестно. Независимият тестер не знае какво има в програмата и често измисля такива входни данни, за обработването на които програмистът дори не е и помислял. Затова, за тестването на по-сложни програми е необходим и независим тестер.

Инструментът дебъгер

Създадени са множество инструментални средства, които подпомагат дейностите по откриване и отстраняване на грешки. Чрез автоматизация на настройването и тестването се постига систематичност, подобряване на организацията на тестване, повишаване на надеждността на програмната система, документиране на извършваните дейности, съхраняване на тестовите данни, автоматично измерване на обхвата на тестването.

В зависимост от предназначението си, програмните средства за тестване могат да бъдат анализатори на програми (статични и динамични), генератори на тестови данни, помощни средства и среди за тестване.

Ще разгледаме по-подробно инструменталните средства, подпомагащи настройването, които се наричат ***дебъгери.*** Основните възможности на дебъгерите са следните:

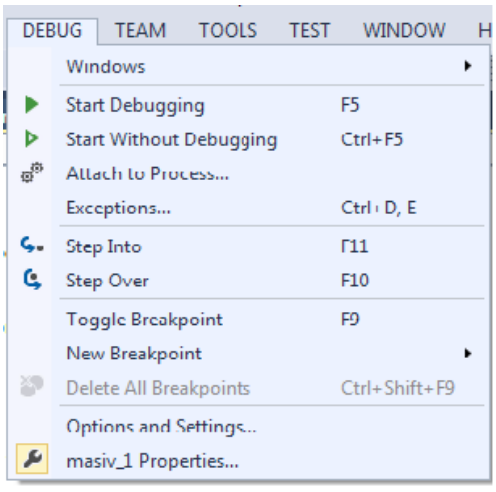
а) ***разглеждане на текста*** на програмата на различни нива (показване само имената на функциите или процедурите, показване съдържанието на отделни програмни фрагменти). Съвременните дебъгери реализират многопрозоречно визуализиране на интересуващите ни програмни части.

б) ***трасиране на изпълнението*** – показване на операторите в реда на изпълнението им при конкретните тестови данни;

в) възможност за дефиниране на ***контролни точки*** и определяне на действията при достигане на съответната контролна точка. Тези действия могат да бъдат:

- преустановяване на изпълнението на програмата;
- разпечатване съдържанието на определени области от паметта или на указани променливи;
- продължаване на изпълнението на програмата от указан оператор.

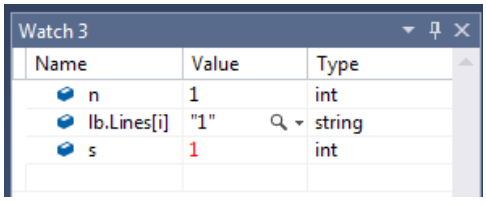
Дебъгерът на средата Visual Studio



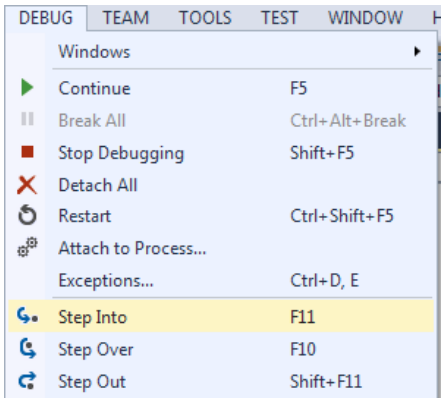
Фиг. 2.

```
int i,n,s;
int[] a = new int[111];
n = s = 0;
for (i=0; i < lb.Lines.Length; i++)
{   string ss = lb.Lines[i];  n++;
    if(ss!="")
        a[n] = int.Parse(lb.Lines[i]);
    s = s + a[n];
}
label1.Text = s.ToString();
```

Фиг. 3.



Фиг. 4.



Фиг. 5.

Програмата дебъгер на средата Visual Studio Express 2013 се управлява от менюто DEBUG (*Фиг. 2*). Първото нещо, което трябва да направим, за да започнем настройката, е да изберем една или няколко контролни точки (breakpoint) – редове в изходния код, на които искаме програмата да спре, за да наблюдаваме какво е направила до момента.

За да поставим контролна точка върху един ред на програмата, трябва да го маркираме и след това да изпълним командата DEBUG/Togle Breakpoint. Встрани от реда, на който сме поставили контролна точка, редакторът поставя червена точка (*Фиг. 3*).

С командата Start Debugging или с клавиша F5 стартираме дебъгера. Той изпълнява операторите до срещане на първия ред, който е контролна точка и спира изпълнението. Естественото желание на извършващия настройката в този момент е да провери какви са съдържанията на някои от променливите.

Затова е необходимо да се отиде към някой от прозорците за наблюдаване съдържанието на променливите Watch или Locals (*Фиг. 4*).

В първия свободен ред на таблицата изписваме в колоната Name името на променливата, която искаме да наблюдаваме, а дебъгерът изписва в другите две колони съдържанието на тази променлива в момента и нейния тип.

След като сме наблюдавали съдържанието на променливите при достигане на една контролна точка, имаме два начина да продължим. Или постъпково, като изпълним командата Step Over (клавиша F10) и тогава дебъгерът ще изпълни поредния оператор, или да продължим, без да спираме, до следващата контролна точка, като изпълним командата Continue (*Фиг. 5*).

Процесът продължава дотогава, докато установим причината за грешката и я поправим.

При спряла на контролна точка програма, можем да поставим допълнителни контролни точки или да премахнем контролна точка, която вече не ни е нужна, с щракване върху червената точка в реда ѝ.

С командата Stop Debuging прекратяваме настройването.

Работа с компютър

За пример на настройване да разгледаме програмата Calculator от компактния диск. В нея има грешки. Например, ако въведем като втори аргумент числото 0 и опитаме да извършим операцията намиране на остатък при делене, ще получим в етикета неразбираемия за средния потребител надпис NuN. Да настроим тази програма, за да избегнем непонятния за крайния потребител завършек. Поставяме контролна точка на реда

```
private void button5_Click(object sender, EventArgs e),
```

входна точка на функцията, която обработва събитието `Click` на бутона за намиране на остатък при делението. Стартираме настройката. Отваря се прозореца на програмата. Въвеждаме в първото текстово поле число различно от 0, а във второто 0 и натискаме бутона **%** за намиране на остатъка при делене. Дебъгерът спира върху контролната точка. Задаваме в прозореца за наблюдение Watch променливите `arg1` и `arg2` и започваме да изпълняваме програмата постъпково с подаване на команди Step Over. След изпълнение на оператора

```
ShowResult(arg1 % arg2);
```

програмата извежда в прозореца си непонятния текст. Очевидно нещо е станало при пресмятане на остатъка при деление `arg1 % arg2`. Прозорецът Watch показва, че стойността на делителя е 0 и това ни подсказва, че причината за непонятния текст е, че делим на 0. Затова извършваме поправяне на грешката, като заменяме проблемния код с:

```
private void button4_Click(object sender, EventArgs e)
{ GetNumbers();
 if(arg2!=0) ShowResult(arg1 % arg2);
 else MessageBox.Show("Делене на нула!");
}
```

който проверява дали вторият аргумент е 0 и ако е така избягва проблемното деление.

Малко по-добре се държи програмата при деление на 0 – извежда текста + безкрайност, който, обаче, също може да уплаши някои потребители. Трябва при въвеждане на втори аргумент 0 и за тази операция, да се отвори прозорец с подходящо съобщение и да не се извършва операцията, която математически е невъзможна. Поправете по същия начин и метода, който обработва събитието Click на бутона за деление. Сега, ако се въведе 0 като втори аргумент, при натискане на тези два бутона няма да стане нищо нередно!

Въпроси и задачи

1. *Тествайте програмата* `FractionCalculator`*, която ще намерите в компактния диск и при наличие на грешки извършете нужното настройване.*

46 Едномерен масив

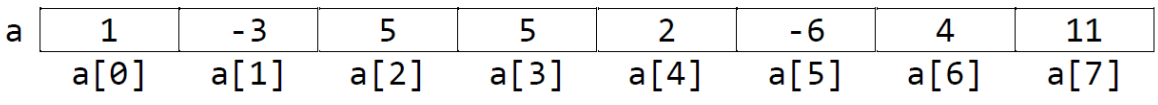
Да се опитаме да решим с програма следната, не лека за изпълнение на ръка, задача: Дадени са средните дневни температури, измерени в град София, за 2011 г. Да се намерят десетте най-високи стойности. Вече сме решавали задача, при която се въвеждат определен брой числа и се намира най-голямото от тях, чрез преглеждане на всички числа. В случая обаче се иска да изведем не най-голямото число, а десетте най-големи, което налага да се прегледат дадените 365 числа 10 пъти. **Не можем да искаме от потребителя да въвежда данни за програмата повече от веднъж! Ако ня-**

кои от въведените данни ни трябват повече от един път, те трябва да се съхраняват в паметта. Абсолютно неразумно би било също така, да декларираме 365 променливи – `a1`, `a2`, …, `a365` и да пишем отделен код за обработката на всяка от тях. Тъй като описаната ситуация е много характерна, всеки език предоставя средства за справяне с проблема, с които ще се запознаем в този урок.

Масиви

За решаването на тази, и подобни на нея задачи, в езиците за програмиране се използват ***масиви***. Масивът е област от паметта (структура), съставена от еднотипни полета с еднакъв размер, в които могат да се съхранят редица от стойности от един и същ тип. Масивът има ***име***, което е общо за всички полета и ***дължина*** – броят на полетата. Всяко поле от съответния на масива ***тип***, наричано ***елемент*** на масива, може да се разглежда и използва в програмата като променлива от същия тип. Елементите на масива се различават един от друг по своя индекс – така наричаме поредния номер на елемента в редицата. Номерирането на елементите в C# започва от нула. На *Фиг. 1* е показан масив `a` с 8 елемента от тип `int`.

В езика C# името на елемент на масив се образува, като след името на масива се постави индексът му, ограден в квадратни (средни) скоби: `a[0]`, `a[1]`, …, `a[i]`, …



Фиг. 1.

Деклариране на масив

Както променливите, така и масивите трябва да се декларират.

Синтаксисът на оператора за **деклариране на масив** е
<тип>`[]` *<име на масива>*;
Например, `int[] a; double[] point; string[] names;`. В един оператор могат да се декларират няколко масива. Например, `int[] a, b, c;`.

За разлика от декларирането на променливи, **декларацията на масив не означава автоматично заделяне на памет за масива** по време на изпълнение на програмата, както е с обикновените променливи. Това се подсказва и от факта, че никъде в оператора за деклариране на масив не се указва размера му. Затова, преди да се използва един масив, той трябва да се разположи в паметта с помощта на оператора за заделяне на памет.

Синтаксисът на оператора за **заделяне на памет** за деклариран масив е
<име на масива> = `new` *<тип>*`[`*<размер>*`]`;
Например, `a = new int[8]; point = new double[2];`. Подобно на инициализацията на скаларните променливи, заделянето на памет може да стане в оператора за деклариране на масива. Например, `int[] a = new int[8];`.

След като е указана дължината на масива, например n, в програмата могат да се използват само елементи с индекси от 0 до $n - 1$!

Инициализация на масив, въвеждане и извеждане на елементите

Инициализацията на масив се състои в инициализация на елементите му. По подразбиране, елементите на целочислените масиви се инициализират с нули, масивите от тип `char` – с малки латински букви `'a'`, а от тип `string` – с празни низове. Ако потребителят иска друга инициализация, има няколко начина да се направи това.

- **В оператора за деклариране.** Например, `int[] a = {1, -3, 5, 5, 2, -6, 4, 11};`. При този начин, стойностите на елементите се разделят един от друг със запетая и се заграждат в къдрави скоби. В този случай няма нужда от служебната дума `new` и задаване на размера. За масива се отделя толкова място в паметта, колкото са елементите в инициализиращия списък.
- **В кода на програмата.** В този случай даването на стойност става с оператор за присвояване за всеки елемент поотделно и не се различава от даването на стойност на проста променлива. Например, `a[0] = 1; a[1] = -3; a[2] = 5;` и т.н. или в цикъл `for(i=0;i<n;i++) a[i] = i;` .
- **Въвеждат се от потребителя**, по време на изпълнение на програмата. За образец може да се използва следният програмен фрагмент:

```
int n, i;
n = int.Parse(Console.ReadLine());
int[] array = new int[n];
for (int i = 0; i < n; i++)
{ array[i] = int.Parse(Console.ReadLine()); }
```

Забележете възможността, която се предоставя на програмиста в този случай – да остави на потребителя сам да определи размера на масива по време на изпълнението. В примера, броят на елементите на масива се въвежда в променливата `n` и след това се декларира самият масив, с размер `n`.

Извеждането на елементите на масив също не се отличава от извеждането на коя да е променлива от същия тип. За образец може да се използва следният програмен фрагмент:

```
for (int i = 0; i < 5; i++)
{ Console.WriteLine(a[i]); }
```

Работа с компютър:

Задача. Зададени са 10 цели числа в определен ред. Да се въведат в компютъра и да се изведат на конзолата в ред, обратен на реда на въвеждането им.

Решение: Очевидно е, че числата трябва да се съхранят в масив. Задачата може да се реши по 2 начина. Първата възможност е да се въведат числата в масива с цикъл, в който управляващата променлива се мени от 0 до 9, а при извеждането – да намалява от 9 до 0. Вторият начин е да се постъпи обратно: при въвеждането управляващата променлива на цикъла да се мени от 9 до 0, а при извеждане – от 0 до 9. Програмният фрагмент, с който реализираме първият подход е:

```
int i, n=10;
int[] array = new int[n];
for (i = 0; i < n; i++)
{ array[i] = int.Parse(Console.ReadLine());
}
for (i = n-1; i >= 0; i--)
{ Console.WriteLine(array[i]);
}
```

Задача. Напишете съответно конзолно приложение, компилирайте го и проверете работоспособността му.

Въпроси и задачи

1. *За всеки от програмните фрагменти:*

а)
```
int[] a = {1,2,3,4,5,6}, b;
b = a[2] + a[3];
```

б)
```
int[] a = {1,2,3,4,5,6}, b;
b = a[2] + a[6];
```

определете стойността на b *след изпълнението на фрагмента.*

2. *Дадени са n цели числа. Напишете конзолно приложение, което въвежда от клавиатурата броя на числата и самите числа и извежда в конзолата само отрицателните числа, в ред обратен на реда на въвеждането им.*

ПРИМЕР:

Вход	Изход
3	-2
-4	-4
1	
-2	

3. *Дадени са две редици от по n числа. Напишете конзолно приложение, което въвежда броя n и елементите на редиците, а извежда на конзолата* YES*, ако двете редици са съставени от едни и същи числа, в един и същ ред, или* NO *в противен случай.*

ПРИМЕР 1:

Вход	Изход
3	YES
1	
1	
-2	
1	
1	
-2	

ПРИМЕР 2:

Вход	Изход
3	NO
1	
1	
-2	
1	
-2	
1	

47 Едномерен масив – упражнение

Масивите са важни за програмирането. За много задачи данните трябва да се разполагат в масив, за да могат да се обработват ефективно. Вече знаем как се декларира едномерен масив, как се въвеждат и извеждат елементите му. В този и следващи уроци ще покажем редица техники за работа с масиви, които са необходими за решаване на по-сложни задачи, като например:

- да се намери най-големият и най-малкият елемент (ако елементите са от тип, който позволява сравняване по големина);
- да се търси елемент с конкретна стойност;
- да се добавя елемент;
- да се премахва елемент/елементи по даден признак;
- да се подреждат (сортират) елементите по даден признак, и т.н.

Задача 1. Дадена е редица от *n* цели числа. Напишете конзолно приложение, което да намира и извежда в конзолата най-голямото и най-малкото число на редицата.

Решение: Вече знаем как се намира най-голямото число на редица, която въвеждаме от клавиатурата, без да използваме масив. Най-малкото число намираме по същия начин – приемаме първото число за текущо най-малко, а на всяка стъпка на цикъла заменяме текущо най-малкото с нововъведеното, ако то се окаже по-малко от текущото. Причината да поискаме числата да се съхранят в масив

е, че за някои по-сложни задачи може да се наложи числата да се прегледат отново, а не е добре да искаме от потребителя на програмата да въвежда числата втори път.

Програма:

```
static void Main(string[] args)
{ int i, n, maxN, minN;
    n = int.Parse(Console.ReadLine());
    int[] array = new int[n];
    array[0] = int.Parse(Console.ReadLine());
    maxN = array[0]; minN = array[0];
    for (i = 1; i < n; i++)
    { array[i] = int.Parse(Console.ReadLine());
      if (maxN < array[i]) maxN = array[i];
      if (minN > array[i]) minN = array[i];
    }
    Console.WriteLine(„Max =“ + maxN);
    Console.WriteLine(„Min =“ + minN);
}
```

Създайте конзолно приложение MaxMin. Въведете програмата, компилирайте я и проверете работоспособността ѝ. Ако програмата не работи както очаквате, използвайте дебъгера, за да я настроите.

Тестване: Когато тествате програма, в която трябва да се въвеждат дълги редици от числа, никога не задавайте голяма стойност за *n*. Защо? Да допуснем, че в програмата има грешка. Стартирате я с $n = 10$ и въвеждате 10 числа. Забелязвате грешката, поправяте я и отново стартирате програмата. Пак въвеждате 10 числа. А ако в програмата все още има грешки …? Много по-добре е да тествате с малко *n,* например 4 числа вместо 10, докато се убедите че сте изчистили всички грешки. Много е вероятно програмата, която работи правилно за $n = 4$, да работи правилно и за $n = 10$, $n = 100$, $n = 1000$ и т.н. Това не значи, че не трябва да направите и 1-2 по-големи теста.

Задача 2. Дадена е редица от *n* цели числа. Напишете конзолно приложение, което да намира и извежда в конзолата най-малкото и най-голямото число на редицата и по колко пъти се срещат в нея тези две числа.

Решение: За решаването на тази задача трябва само да се допълни кодът от решението на Задача 1. Необходимо е още едно обхождане на елементите на масива с цикъл, в който ще се преброява колко пъти се срещат запомненият в променлива `maxN` максимален елемент и запомненият в променлива `minN` минимален елемент на масива. За целта са необходими две нови променливи `brMin` и `brMax` – броячи, които ще увеличаваме с 1 всеки път, когато срещнем най-малкото или най-голямото число, съответно. Както винаги, когато декларираме променлива брояч, не трябва да забравяме да я инициализираме с нула.

Създайте ново конзолно приложение `brMinMax`, като копирате в него кода от Задача 1 и добавите в края на функцията `Main` следните допълнителни редове:

```
int brMin = 0, brMax = 0;
for (i = 0; i < n; i++)
{ if (minN == array[i]) brMin++;
  if (maxN == array[i]) brMax++;
}
Console.WriteLine("brMin=" + brMin);
Console.WriteLine("brMax=" + brMax);
```

Компилирайте програмата и проверете работоспособността ѝ.

Задача 3. Съставете алгоритъм за решаване на Задача 2**,** който не използва масив. Напишете конзолно приложение, което реализира този алгоритъм.

Решение: Да се спрем само на намиране на броя на срещанията на най-малкото число, защото за най-голямото ще постъпим по подобен начин. На пръв поглед изглежда невъзможно да се реши

задачата без използване на масив, защото ще знаем кое е най-малкото число, едва когато завършим с въвеждането на всички числа. Но не е така! Да се възползваме от идеята, която при решаването на *Задача 1* нарекохме текущо минимално. След като в променливата `minN` запомняме най-малкото намерено до момента число на редицата, тогава нека паралелно с това в променливата `brMin` да съхраняваме броя на срещанията на това текущо най-малко число. Когато на поредната стъпка на цикъла срещнем текущото минимално, тогава трябва да увеличим съдържанието на `brMin` с 1. А когато срещнем число, което е по-малко от текущо минималното? Тогава просто трябва да започнем броенето отново, като запомним новото минимално и дадем на `brMin` стойност 1.

Програма:

```
static void Main(string[] args)
{ int i, n, a, minN, brMin;
 n = int.Parse(Console.ReadLine());
 minN = int.Parse(Console.ReadLine());
 brMin=1;
 for (i = 1; i < n; i++)
 { a = int.Parse(Console.ReadLine());
   if (minN > a) { minN = a; brMin = 1; }
   else if (minN == a) brMin++;
 }
 Console.WriteLine("Min =" + minN);
 Console.WriteLine("brMin=" + brMin);
}
```

Компилирайте програмата и проверете работоспособността ѝ.

Въпроси и задачи

1. *Дадена е редица от n цели числа. Напишете конзолно приложение, което въвежда числата и извежда в конзолата най-голямото число на редицата и на коя позиция (броейки от 0) то се среща за първи път.*
2. *Добавете към решението на предната задача код, който да извежда най-малкото число на редицата и позицията, на която то се среща за първи път.*
3. *Променете решението на Задача 3 от урока така, че да намира и броя на срещанията на най-голямото число в редицата.*
4. *Напишете конзолно приложение, в което се въвеждат оценките от контролна работа по информатика на 13-те ученици от една група и се намира броят на учениците, които имат оценки, по-големи от средния успех на групата (представили са се добре!).*

48 Използване на списъчна кутия – упражнение

Задача. Даден е целочислен масив `A` и две цели числа `P` и `Q`. Напишете програма на C#, която да намери броя на числата от `A`, които са в интервала `[P;Q]`. Стойностите на масива да се въведат в списъчна кутия, а стойностите на `P` и `Q` – в TextBox. Резултатът да се изведе в етикет Label.

Анализ. В предишни часове използвахме списъчната кутия ListBox за въвеждане на числата, като броят им беше предварително фиксиран. Обаче в ListBox може само да се изписва текст. Следова-

телно, трябва да използваме друга контрола, която е подобна на списъчната кутия, но в нея може да въвеждаме данни. Такава е контролата RichTextBox. Тя прилича на ListBox, но вместо `Items`, в нея се използва свойството Lines. Броят на редовете в контролата е свойството Count на Lines и се изписва по този начин:

```
RichTextBox1.Lines.Count().
```

`Lines`, както и `Items`, е едномерен масив от тип `string` и достъпът до елементите му става по познатия начин: `RichTextBox1.Lines[i]= … .` Първият ред от контролата се записва в RichTextBox1.Lines[0], следователно ако контролата има 10 реда, т.е. `RichTextBox1.Lines.Count()=10`, то последният ред ще се запише в `RichTextBox1.Lines[9]`.

Нека `n` е променлива, в която ще въведем броя на числата в списъчното поле `RichTextBox`, а `i` – индексът на поредното число от текстовата кутия. В началото i=0. Ще обхождаме редовете на контролата, всеки път ще увеличаваме `i` с 1 и `A[i] = RichTextBox1.Lines[i]`. Ето частта от кода за обработката на събитието `Click` на бутона, в която въвеждаме данните от текстовата кутия в масива `A`:

```
n = RichTextBox1.Lines.Count();
for (i = 0; i < n; i++)
{
     A[i] = int.Parse(RichTextBox1.Lines[i]);
}
```

Преди цикъла прочитаме в променливата `n` броя на елементите на масива `Lines[i]`, защото е по-удобно да се работи с променлива, отколкото със стойността, връщана всеки път от метода RichTextBox1.Lines.Count().

След като прочетем данните в масива `A`, можем да преброим колко от тях са в интервала, зададен от целите числа P и Q, определени от съдържанията на текстовите кутии `tbP` и `tbQ`:

```
int P = int.Parse(tbP.Text); int Q = int.Parse(tbQ.Text)
```

а това ще стане с фрагмента:

```
int br = 0;
for (i = 0; i < n; i++)
{
   if (A[i] >= P && A[i] <= Q) br++;
}
```

Тази програма ще даде грешка при изпълнение на този код, ако потребителят въведе празен ред между две от числата. Причината е че, при преобразуването на текстовете от кутията `RichTextBox1` в цели числа, методът `int.Parse()` не може да преобразува празни низове в цели числа.

За да предотвратим такава грешка е необходима допълнителна проверка:

```
for (i = 0; i < n; i++)
{ if (RichTextBox1.Lines[i] != "")
      A[i] = int.Parse(RichTexBox1.Lines[i]);
  else break;
}
```

Операторът `break;` ще прекъсне изпълнението на цикъла при срещане на празен низ.

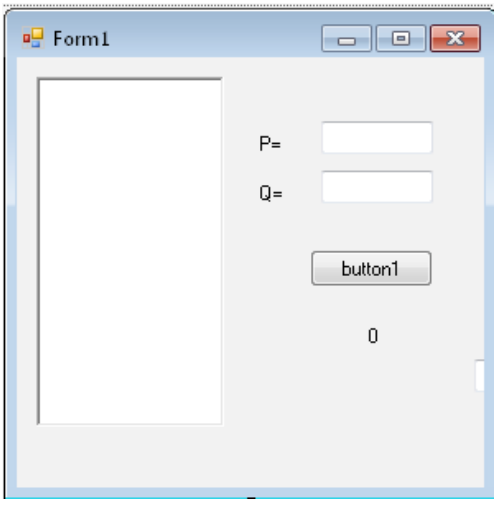
Проверете какво ще стане ако стойността, въведена в кутията `tbP` е по-голяма от стойността, въведена в `tbQ`. Едва ли е интересно такова поведение на програмата. Направете необходимата проверка, за да предотвратите такава ситуация.

На *Фиг. 1* е дадена примерна форма за програмата. В етикета, в който ще се показва резултатът, има записана 0. Дайте подходящо име на бутона и подредете контролите по ваш избор.

След като напишете програмата, я тествайте и се убедете, че работи вярно.

По подобен начин могат да бъдат решени и други задачи:

- Да се изведе броят на четните числа в масива.
- Да се изведе сумата на числата в масива.
- Да се изведе броят на отрицателните числа в масива.
- Да се изведат числата в масива, които са извън интервала [P;Q].
- Да се изведат числата в масива, които са равни на P.



Фиг. 1.

- Да се изведе най-малкият индекс на елемент от масива, чиято стойност е равна на Q. Ако няма такава, в етикета да се изведе –1.
- Да се изведе най-малкият и най-големият елемент на масива.

За да спестите време, ви препоръчваме да копирате част от програмния код за обработката на първия бутон и да го поставите в кода за обработката на втори бутон и след това да направите нужните промени в кода така, че да решават новата задача. Освен това, не е необходимо при натискане на всеки бутон да се създава масивът. По-добре е да се добави бутон `"Масив"`, който след като се въведат стойности в списъчното поле RichTextBox, да ги запише в масива `A`. След това може да се напише обработката на събитието `Click` за бутоните на задачите. По този начин програмата ще е по-кратка.

49 Изчертаване на полигон

Полигоните са затворени фигури с три или повече страни, като триъгълници, четириъгълници, петоъгълници и др., които не са непременно изпъкнали. Методът `DrawPolygon` на класа `Graphics` се използва за чертаене на полигони в C#. Списъкът от параметри на този метод е по-различен от списъците на останалите разгледани до сега методи:

```
void DrawPolygon(Pen p, Point[] point);
```

Новото тук е масивът `point` от точки – елементи на класа `Point`. В този масив се задават координатите на върховете на многоъгълника в реда на обхождането им. Фигурата се затваря автоматично от отсечка, която свързва последната точка с първата.

Задача 1. Даден е масив от четирите върха на полигон – елементи на класа `Point`. Напишете програма, която да изчертава образувания от тях полигон.

Например, да разгледаме следния масив с четири елементи на класа `Point`:

```
Point[] point = { new Point(100, 100), new Point(200, 200),
      new Point(180, 300), new Point(120,250) };
```

Той съдържа описанието на четири точки от чертожното поле. Ако извикаме метода `DrawPolygon()` с аргумент този масив:

```
g.DrawPolygon(pen, point),
```

той ще изчертае четириъгълника от *Фиг. 1*.

Фиг. 1.

Задача 2. Да се изчертае звездата (петолъчка) от *Фиг. 2*.

Решение: Тъй като намирането на координатите на петте върха на звездата е трудна математическа задача, няма да се занимаваме с нея тук. Съответните формули са програмирани в цикъла `for`, с който се създава масивът `point[]` с 5 елемента от класа `Point`. След края на цикъла се извиква методът `DrawPolygon()`, който изрисува фигурата:

```
protected override void OnPaint(PaintEventArgs e)
{ Graphics g = e.Graphics;
  Pen p = new Pen(Color.Black, 2);
  int cx = ClientSize.Width;
  int cy = ClientSize.Height;
  Point[] point = new Point[5];
  for (int i = 0; i < 5; i++)
  { double angle=(i*0.8-0.5)*Math.PI;point[i] =
    new Point((int)(cx*(0.25+0.24*Math.Cos(angle))),
    (int)(cy*(0.5+0.48*Math.Sin(angle))));
  }
  g.DrawPolygon(p, point);
}
```

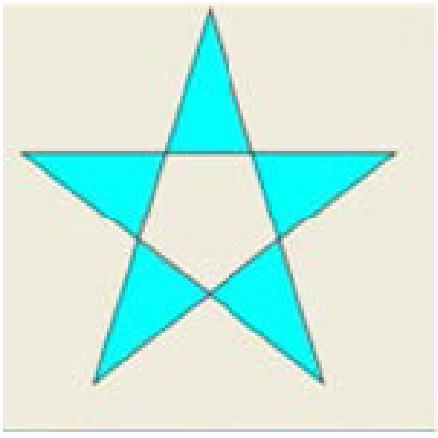
Фиг. 2.

Ако се налага да се запълни получената фигура, трябва да се дефинира съответната четка и да се извика съответният метод за запълване, преди метода за изчертаване:

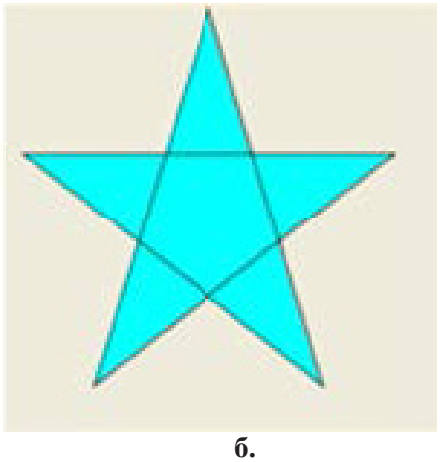
```
Brush b = new SolidBrush(Color.Aqua);
g.FillPolygon(b, point);
g.DrawPolygon(p, point);
```

Ако многоъгълникът не е изпъкнал, както е петолъчката, резултатът от запълването е неочакван (виж *Фиг. 3а*). Ако искате цялостно запълване (*Фиг. 3б*), трябва да се добави трети аргумент на метода за запълване:

```
g.FillPolygon(b,point,System.Drawing.Drawing2D.FillMode.Winding);
```



а.



б.

Фиг. 3.

50 Обработване на данни от файл – упражнение

При обработване на данни, записани във файл, е естествено да се използват масиви. Затова ще решим няколко задачи за обработка на данните от файл. За целта, първо да си осигурим файл с необходимите данни.

Задача 1. Направете програма с графичен интерфейс, която по зададено цяло положително число *N* и име на текстов файл, генерира *N* случайни цели числа и извежда в текстов файл със зададеното име, *N* и *N*-те генерирани числа.

Упътване.

1. Направете екранна форма с две текстови кутии – една за въвеждане на броя на числата, а другата – за въвеждането на името на текстовия файл, както и бутон, с който да стартирате генерирането на числата. Преценете каква да бъде ширината на всяка от двете текстови кутии, в зависимост от това какво ще бъде съдържанието им.

2. Свържете със събитието `Click` на бутона функция за генериране на числата и съхраняването им в масив по модела, използван в друга задача, в която трябваше да се генерират случайни числа.

3. Отворете текстов файл със зададеното име и изведете в него броя на числата и самите числа. Ако в папката, в която създавате текстовия файл вече има файл с това име, програмата да изтрие старото съдържание и да запише във файла само генерираните числа.

След като сте създали файла от Задача 1, решете Задача 2.

Задача 2. Направете програма с графичен интерфейс, която по зададено име на текстов файл, въвежда от текстовия файл записаните в него числа, след което намира броя на числата, които са по-големи от средното аритметично на тези числа и записва този брой в етикет.

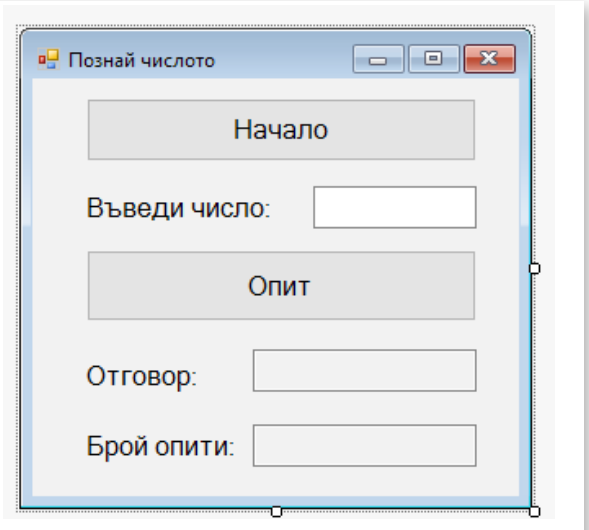
Задача 3. Във всеки от редовете на текстов файл са записани по една двойка координати на неизвестен брой точки в равнината. Направете програма с графичен интерфейс, която по зададено име на файл изчертава полигона, образуван от тези точки.

Упътване. Създайте предварително текстов файл с данни, с които да тествате програмата.

VI Създаване на софтуерен продукт

51 Проект Познай числото

Направете проект на графично приложение на игра с име GuessTheNumber, в което, след натискането на бутона Начало, компютърът генерира случайно цяло число в интервала от 1 до 100 включително, а играчът се опитва да го познае с минимален брой опити. При всеки опит на играча програмата трябва да връща едно от следните съобщения: `нагоре` (търсеното число е по-голямо от предположението на играча), `надолу` (търсеното число е по-малко от предположението на играча) и `позна` (когато играчът въведе генерираното от компютъра число). След като играчът познае числото, програмата не трябва да позволява повече опити и извежда в прозореца броя на направените опити. Реализирайте този проект.



Фиг. 1.

Решение: Примерен интерфейс на играта е показан на *Фиг. 1,* а примерен програмен код е даден по-долу.

```
namespace GuessTheNumber
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
            button2.Enabled = false;
        }

        int compNum, cnt;

        private void button1_Click(object sender, EventArgs e)
        {
            Random r = new Random();
            compNum = r.Next(1, 101);
            button2.Enabled = true;
            button1.Enabled = false;
            cnt = 0;
            textBox1.Clear();
            textBox2.Clear();
            textBox3.Clear();
        }

        private void button2_Click(object sender, EventArgs e)
        {
            int myNum = int.Parse(textBox1.Text);
            cnt++;
```

```
            textBox3.Text = cnt.ToString();
            textBox1.Focus();
            textBox1.SelectAll();
            if (myNum<compNum)
            {
                textBox2.Text = "нагоре";
            }
            if (myNum > compNum)
            {
                textBox2.Text = "надолу";
            }
            if (myNum == compNum)
            {
                textBox2.Text = "позна";
                button2.Enabled = false;
                button1.Enabled = true;
            }
        }

        private void textBox1_KeyPress(object sender,KeyPressEventArgs e)
        {
            if (e.KeyChar<'0' || e.KeyChar>'9')
            {
                if (e.KeyChar!=8)
                {
                    e.Handled = true;
                }
            }
        }
    }
}
```

52. Проект Калкулатор

Задача. В този урок ще продължим работата по проекта, зададен в предишен урок за самостоятелна работа. Ще създадем отново калкулатор с графичен интерфейс, но с повече функции и стриктен контрол върху въвежданите данни. Калкулаторът ще е сравнително прост, само с аритметичните операции – събиране, изваждане, умножение, деление и намиране на остатъка при деление. По-късно, ако желаете, може да разширите възможностите му и с други операции!

Конструкцията try...catch...finally

Мощен инструмент на езика C# за контрол на въвежданите от потребителя данни е операторът `try ... catch ... finally`. С негова помощ могат да бъдат „прихващани“ и обработвани различни неприятни ситуации, наричани изключения (англ. exeption), които ако не бъдат обработени, програмата ще спре аварийно.

Синтаксисът на операторът `try...catch...finally` е:

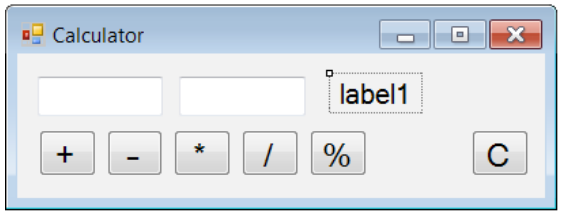
`try {` *< оператори, при изпълнение на които очакваме изключение >* `}`
`catch (`*< вид на изключението >*`)`
`{` *< оператори, които ще се изпълнят при възникване на изключение>* `}`
`finally {` *< оператори, които ще се изпълнят винаги >* `}`

Когато достигне този оператор, програмата се опитва да изпълни операторите в блока `try` (англ. опитвам). В частта `catch () ... finally` операторът прилича на `if () ... else`, но разликата е значителна. Ролята на условие играе възникването на някакво очаквано изключение, като видът на изключението се поставя в скоби, както условието. Ако изключението се случи, се изпълнява кодът след `catch` (англ. хващам, улавям), а след това – кодът след `finally` (англ. накрая). Ако очакваното изключение не се случи, се изпълнява само кодът след `finally`. Както частта `catch`, така и частта `finally`, може да липсва. Ще използваме този оператор за контрол на въведените от потребителя данни.

Проектиране на формата

Отворете ново приложение с графичен интерфейс и го съхранете под името Calculator. За проектиране на формата можете да използвате примера от предишен урок, в който за самостоятелна работа трябваше да направите калкулатор с две операции (*Фиг. 1*). Променете свойството Text на формата в Calculator. Добавете и три нови бутона с надписи *****, **/** и **%**, за стартиране на операциите умножение, деление и намиране на остатъка при деление. Подредете елементите така, че да са добре разположени по цялото поле на формата и ги подравнете с командите за подравняване.

Програмиране на приложението



Фиг. 1.

Ясно е, че методите за обработка на събитието `Click` на всеки от бутоните ще са еднотипни. Ще трябва да се вземат въведените в двете текстови кутии числа, зададени като низове, да се превърнат с метода `int.Parse()` в числа, като се контролира с оператора `try...catch...finally` наличието на неразрешени знаци, да се извърши операцията, свързана с бутона и да се запише резултатът в етикета, като се превърне отново в низ с метода `ToString()` на класа `int`. И това трябва да се направи пет пъти, като всеки път променяме в кода само един знак на операция!

За да минимизираме повтарянето на код, ще напишем два собствени метода на класа Form1. Първият метод ще взима двата низа от текстовите кутии и ще ги превръща в числа. Вторият метод ще записва резултата в етикета. За метод, обработващ събитието `Click` на бутон на операция ще остане само да извика първия метод, за да получи операндите, да извърши операцията и да извика втория метод, за да покаже резултата.

Да започнем с първия метод – той няма да връща стойност, а ще извършва действия, т.е ще взима съдържанието на двете текстови кутии и ще опита да ги превръща в числа, с които после да се извършват операциите. Затова типът му ще бъде `void`. За да може да се използват резултатите от него, две дробни числа ще трябва да бъдат декларирани извън всички методи на класа, за да бъдат достъпни във всеки метод. Методът ще се казва `GetNumbers` и списъкът му от параметри ще бъде празен:

```
double arg1, arg2, result;
void GetNumbers()
{  try { arg1 = double.Parse(textBox1.Text); }
```

```
        catch (FormatException) { arg1 = 0.0; textBox1.Text = "0"; }
        try { arg2 = double.Parse(textBox2.Text); }
        catch (FormatException) { arg2 = 0.0; textBox2.Text = "0";}
    }
```

Вторият метод ще записва резултата в етикета label1. Тъй като не връща стойност, типът му също ще е `void`, но ще има един параметър – пресметнатият резултат. Да наречем този метод `ShowResult`:

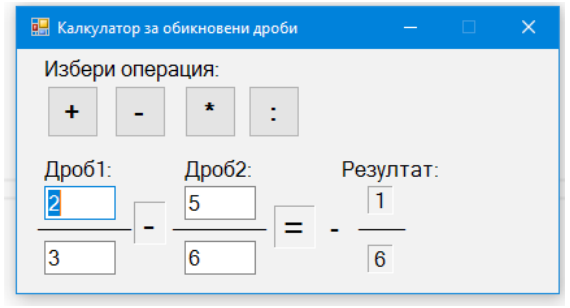
```
    void ShowResult(double result)
    {  label1.Text = result.ToString(); }
```

Вече може да напишем и методите, с които ще обработвате събитията Click на петте бутона. Да започнем с бутона за събиране:

```
    private void button1_Click(object sender, EventArgs e)
    {  GetNumbers(); ShowResult(arg1 + arg2);  }
```

Напишете сами останалите четири метода. Компилирайте програмата и я тествайте с различни входни данни, включително и некоректни.

53. Проект Дробен калкулатор



Фиг. 1.

Направете проект на графично приложение с име FractionCalculator, в което се въвеждат две обикновени дроби и се дава възможност на потребителя да избере една от четирите аритметични операции – събиране, изваждане, умножение, деление. След избора на операция, програмата трябва да извежда резултата от нея, като операндите ѝ са двете въведени обикновени дроби. Реализирайте този проект.

Решение:

Примерен интерфейс на програмата е показан на *Фиг. 1.*, а примерен програмен код е даден по-долу:

```
namespace FractionCalculator
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
        }
        private void button1_Click(object sender, EventArgs e)
        {
            label9.Text = button1.Text;
            Calculate();
        }

        private void button2_Click(object sender, EventArgs e)
        {
            label9.Text = button2.Text;
            Calculate();
```

```
}

private void button3_Click(object sender, EventArgs e)
{
    label9.Text = button3.Text;
    Calculate();
}

private void button4_Click(object sender, EventArgs e)
{
    label9.Text = button4.Text;
    Calculate();
}

private void Calculate()
{
    try
    {
        textBox1.Focus();
        textBox1.SelectAll();
        label10.Visible = false;
        label11.Text = "";
        label12.Text = "";
        int num1 = int.Parse(textBox1.Text);
        int denom1 = int.Parse(textBox2.Text);
        int num2 = int.Parse(textBox3.Text);
        int denom2 = int.Parse(textBox4.Text);
        int num3 = 0, denom3 = 0;
        if(label9.Text=="+")
        {
            num3 = num1 * denom2 + num2 * denom1;
            denom3 = denom1 * denom2;
        }
        if (label9.Text == "-")
        {
            num3 = num1 * denom2 - num2 * denom1;
            denom3 = denom1 * denom2;
        }
        if (label9.Text == "*")
        {
            num3 = num1 * num2;
            denom3 = denom1 * denom2;
        }
        if (label9.Text == ":")
        {
            num3 = num1 * denom2;
            denom3 = denom1 * num2;
        }
        int nod = Nod(num3, denom3);
        num3 /= nod;
        denom3 /= nod;
        if (denom3 == 0) throw new Exception();
```

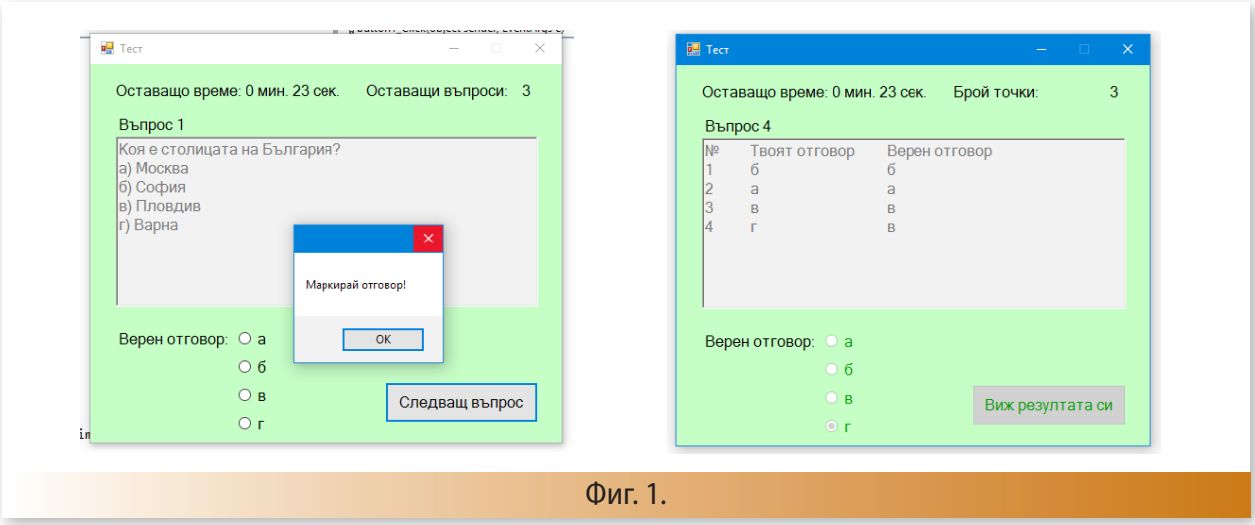
```
                if (denom3 < 0) { num3 = -num3; denom3 = -denom3; }
                if (num3<0)
                {
                    label10.Visible = true;
                    num3 = -num3;
                }
                label11.Text = num3.ToString();
                label12.Text = denom3.ToString();
            }
            catch(Exception)
            { }
        }
        private int Nod(int x, int y)
        {
            x = Math.Abs(x);
            y = Math.Abs(y);
            while(x!=0 && y!=0)
            {
                if (x > y) { x %= y; }
                else { y %= x; }
            }
            return x + y;
        }
        private void textBox1_KeyPress(object sender,KeyPressEventArgs e)
        {
            if (e.KeyChar < '0' || e.KeyChar > '9')
            {
                if (e.KeyChar != 8)
                {
                    e.Handled = true;
                }
            }

        }
    }
}
```

Забележка: Обработката на събитието `KeyPress` на `textBox1` е закачено и за `textBox2`, `textBox3` и `textBox4` в прозореца `Properties`, страницата `Events`.

54. Проект Тест

Направете проект на графично приложение с име Test, което чете от файл с име questions.txt няколко въпроса и по 4 възможни отговора за всеки въпрос (между всеки два въпроса има празен ред), а от друг файл с име answers.txt чете верните отговори ред по ред (направете файловете скрити и само за четене!). Програмата трябва да показва въпросите един след друг, като на следващ въпрос се преминава само когато е посочен отговор на текущия въпрос. След изтичане на предварително зададено време или преминаване през всички въпроси (интерфейсът трябва да показва броя на оставащите въпроси), програмата трябва да покаже резултатите от теста – брой точки и отговорите за

всеки въпрос (както на изпитвания, така и верните). Освен това програмата трябва да съхрани тези данни във файл с име results.txt и да дава възможност този файл да бъде отворен и разглеждан в интерфейса! Реализирайте този проект.



Фиг. 1.

Решение:

Примерен интерфейс на програмата е показан на *Фиг. 1*, а примерен програмен код е даден по-долу:

```
namespace Test
{
    public partial class Form1 : Form
    {
        public Form1()
        { InitializeComponent();
        }
        string[] questions;
        int numQuestions, currQuestion, points=0, time;
        string[] answers;
        StreamWriter w = new StreamWriter("results.txt", true,
        Encoding.GetEncoding("windows-1251"));
        private void Form1_Load(object sender, EventArgs e)
        {
            StreamReader r1 = new StreamReader("questions.txt",
            Encoding.GetEncoding("windows-1251"));
            questions = new string[100];
            string line = ".", q="";
            numQuestions = 0;
            while (line != null)
            {
                q = '';
                line = ".";
                while (line!="" && line!=null)
                {
                    line = r1.ReadLine();
                    q = q + line + "\n";
                }
                numQuestions++;
                questions[numQuestions] = q;
```

```
            }
            r1.Close();
            currQuestion = 1;
            richTextBox1.Text = questions[currQuestion];
            label5.Text = (numQuestions - currQuestion).ToString();
            w.WriteLine("№\tТвоят отговор\tВерен отговор");
            label3.Text = "Въпрос " + currQuestion;
            time = numQuestions * 10; //времето в секунди
            StreamReader r2 = new StreamReader("answers.txt",
            Encoding.GetEncoding("windows-1251"));
            answers = new string[100];
            for(int i=1; i<=numQuestions; i++)
            {
                line = r2.ReadLine();
                answers[i] = line;
            }
            r2.Close();
        }

        private void button1_Click(object sender, EventArgs e)
        {
            if (button1.Text=="Виж резултата си")
            {
                button1.Enabled = false;
                StreamReader r = new StreamReader("results.txt",
    Encoding.GetEncoding("windows-1251"));
                richTextBox1.Text = r.ReadToEnd();
                r.Close();
                return;
            }
            if(button1.Text=="Край на теста")
            {
                timer1.Stop();
                label4.Text = "Брой точки:";
                label5.Text = points.ToString();
                w.Close();
                button1.Text = "Виж резултата си";
                return;
            }

            if(!radioButton1.Checked && !radioButton2.Checked &&
!radioButton3.Checked && !radioButton4.Checked)
            {
                MessageBox.Show("Маркирай отговор!");
                return;
            }
            string ans="";
            if (radioButton1.Checked) { ans = "а"; }
            if (radioButton2.Checked) { ans = "б"; }
            if (radioButton3.Checked) { ans = "в"; }
            if (radioButton4.Checked) { ans = "г"; }
            if (radioButton1.Checked && answers[currQuestion] == "а")
```

```
            { points++; }
        if (radioButton2.Checked && answers[currQuestion] == "б")
            { points++; }
        if (radioButton3.Checked && answers[currQuestion] == "в")
            { points++; }
        if (radioButton4.Checked && answers[currQuestion] == "г")
            { points++; }
        w.WriteLine("{2}\t{0}\t\t\t{1}", ans, answers[currQuestion],
                    currQuestion);
        if (currQuestion == numQuestions)
        {
            button1.Text = "Край на теста";
            radioButton1.Enabled = false;
            radioButton2.Enabled = false;
            radioButton3.Enabled = false;
            radioButton4.Enabled = false;
            return;
        }
        currQuestion++;
        richTextBox1.Text = questions[currQuestion];
        label3.Text = "Въпрос " + currQuestion;
        label5.Text = (numQuestions - currQuestion).ToString();
        radioButton1.Checked = false;
        radioButton2.Checked = false;
        radioButton3.Checked = false;
        radioButton4.Checked = false;

    }
    private void timer1_Tick(object sender, EventArgs e)
    {
        time--;
        label2.Text = time / 60 + " мин. " + time % 60 + " сек.";
        if (time==0)
        {
            timer1.Stop();
            button1.Text = "Край на теста";
            radioButton1.Enabled = false;
            radioButton2.Enabled = false;
            radioButton3.Enabled = false;
            radioButton4.Enabled = false;
        }
    }
  }
}
```

Обяснения: Използван е един нов интерфейсен компонент `Timer` – разучете какви свойства и методи има той и как работи. С негова помощ е добавено ограничение за времето. Може ли да се досетите защо в конструктора на класа `StreamWriter` вторият параметър е направен `true`? Не е хубаво да се правят повторни опити с този тест, нали?

ИНФОРМАТИКА 8. КЛАС
ОБЩООБРАЗОВАТЕЛНА ПОДГОТОВКА

Информатика за 8. клас
общообразователна подготовка

Авторски колектив:
Красимир Манев,
Нели Манева,
Велислава Христова

Редактор на издателството Ясена Христова
Предпечатна подготовка и графичен дизайн Кирил Мавров
Художник Киро Мавров
Формат 60x84/8. Печатни коли 17,5.
Първо издание, 2017
ISBN 978-619-7243-21-5

Печат: „Хеликспрес“ ЕООД, Варна, **тел.**: 052 684800
web: www.helixpress.com

Издателство „Изкуства“, 1700 София, ул. „Д-р Йордан Йосифов“ № 8Б,
тел.: 02 943 4724, 02 943 4785 **факс**: 02 943 4397
e-mail: izkustva@yahoo.com, office@izkustva.net
web: www.izkustva.net

издателство изкуства