Довідник по C#-y

*Літерал* – це явно вказане значення декого типу. Розрізняють такі типи літералів:

|  |  |  |
| --- | --- | --- |
| *тип літерала* | *опис* | *приклади* |
| цілочисловий | у десятковій системі числення – звичне нам ціле число | 125, -89, 108; |
| у вісімковій системі числення – починається з 0 | 023, 075, 0416; |
| у шістнадцятковій системі числення – поичанається з 0х або 0Х | 0х4А, 0х6FE2, 0xABC |
| дійсного типу | у форматі з фіксованою крапкою – у записі числа є крапка, що розділяє цілу і дробову частити | 25.69, 2.0, 145.058 |
| у форматі з плаваючою крапкою – у записі використано символ «е» або «Е», що розділяє мантису від порядку | 3.5е5, 3е12, 678е2 |
| символьний | заданий у явному вигляді – символ записується у одинарних лапках | ‘Z’, ‘a’, ‘E’ |
| прості ескейп-послідовності – службові символи починаються з символу «\» | \n -перехід на новий рядок,  \t -горизонтальна табуляція,  \’ - апостороф, і т. д. |
| ескейп-послідовності Unicode – символи «\u», за якими вказують код символу з чотирьох цифр у шістнадцятковій системі числення | ‘\u0123’ , ‘\u3А58 ’ |
| рядковий | дослівний – починається із символу «@», за яким у подвійних лапках вказується рядок символів (при цьому ескейп-послідовності сприймаються як символи) | @”Я люблю \t С#” |
| регулярний – у подвійних лапках вказується рядок символів (ескейп-послідовності обробляються) | ”Я люблю \t С#” |
| логічний | може приймати два значення | true,  false |
| порожньої адреси (нульовий літерал) | використовується у випадку, коли покажчик не містить жодної адреси | null |

**Стандартні «прості» типи змінних**

|  |  |  |  |
| --- | --- | --- | --- |
| **Тип** | **Опис** | **Область значень** | **Приклад** |
| **object** | **Базовий клас для всіх інших типів** |  | **object obj = new Stack();** |
| **string** | **Рядковий тип, послідовність символів Unicode** |  | **string sz = "Привіт";** |
| **sbyte** | **8-розрядне ціле число з знаком** | -128 до 127 | **sbyte sbVal = 5;** |
| **short** | **16-розрядное ціле число з знаком** | -32768 до 32767 | **short sVal = 5;** |
| **int** | **32-розрядне ціле число з знаком** | -2147483648 до 2147483647 | **int iVal = 5;** |
| **long** | **64-розрядне ціле число з знаком** | -9223372036854775808 до 9223372036854775807 | **long lVval = 5; long Val = 5L;** |
| **byte** | **8-розрядне ціле число без знака** | 0 до 255 | **byte bVal = 5; byte Val = 5U;** |
| **ushort** | **16-розрядне ціле число без знака** | 0 до 65535 | **ushort usVal = 5; ushort Val = SU;** |
| **uint** | **32-розрядне ціле число без знака** | 0 до 4294967295 | **uint uVal = 5; uint Val - 5U;** |
| **ulong** | **64-розрядне ціле число без знака** | 0 до 18446744073709551615 | **ulong ulVal = 5; ulong Val = 5U; ulong V - 5L; ulong VI = 5UL;** |
| **float** | **Число з плаваючою крапкою**  4 байти, точність — 7 розрядів | ±1,5\*10-45 до ±3,4\*1033 | **float Val = 3.1416926F;** |
| **double** | **Число з плаваючою крапкою**  8 байт, точність —16 розрядів | ±5\*10-324 до ±1,7\*10306 | **double dVal = 1.23; double Val = 8.9D;** |
| **bool** | **Логічний тип** | true або false | **Bool bVal = true;** |
| **char** | **Тип символу Unicode** | U+0000 до U+ffff | **char cVal = ' z' ;** |
| **decimal** | **Тип десяткового числа**  12 байт, точність — 28 розрядів |  | **decimal dVal = 3.1415926M;** |

**Структура консольної програми**

|  |  |
| --- | --- |
| *Загальний вигляд* | *Приклад* |
| class <назва класу >  {  static void Main()  {  <Тіло програми>  }  } | class Hello  {  static void Main()  {  Console.WriteLine(“Hello!”);  Console.ReadKey();  }  } |

**Введення/виведення даних у консольних програмах**

|  |  |
| --- | --- |
| Приклад | На екрані |
| Console.WriteLine(“Hello!”); | Hello! |
| int i=25;  Console.WriteLine(i); | 25 |
| int i=25;  Console.WriteLine(“i=”+i); | i=25 |
| int j=3;  double d=2.8;  Console.WriteLine("d="+d+" j="+j); | d=2.8 j=3 |
| int i=25;  Console.WriteLine(“i={0}”,i); | i=25 |
| int j=3;  double d=2.8;  Console.WriteLine("d={0} j={1}",d,j); | d=2.8 j=3 |
| int i=25;  Console.WriteLine(“i={0:d5}”,i); | i=00025 |
| double d = 2.853;  Console.WriteLine("d={0:f2} ", d); | d=2.85 |

**Введення символьних даних**

|  |  |
| --- | --- |
| Приклад | На екрані |
| int c;  c =Console.Read();  Console.WriteLine(c); | A  65 |
| char c;  c =(char)Console.Read();  Console.WriteLine(c); | A  A |

**Введення числових даних**

Приклади.

|  |  |
| --- | --- |
| З використанням методу Parse | З використанням класу Convert |
| int c;  Console.Write("c=");  c=int.**Parse**(Console.ReadLine()); | int c;  Console.Write("c=");  c=**Convert.ToInt16**(Console.ReadLine()); |
| double d;  Console.Write("d=");  d=double.**Parse**(Console.ReadLine()); | double d;  Console.Write("d=");  d=**Convert.ToDouble**(Console.ReadLine()); |

**Арифметичні та логічні вирази**

Бінарні операції:

|  |  |  |
| --- | --- | --- |
| Операція | Позначення | Приклад |
| + | додавання | z=x+y |
| – | віднімання | z=x-y |
| \* | множення | z=x\*y |
| / | ділення | z=x/y |
| % | остача від ділення | z=x%y |

Унарні операції:

|  |  |
| --- | --- |
| Операція | Аналог |
| i++ або ++i | i=i+1 |
| i-- або --i | i=i-1 |

|  |  |
| --- | --- |
| Операція | Аналог з бінарними операціями |
| int i = 5;  int j = ++i; // j=6 i=6 | int i = 5;  i=i+1;  int j = i; // j=6 i=6 |
| int i = 5;  int j = i++; // j=5 i=6 | int i = 5;  int j = i;  i=i+1; // j=5 i=6 |
| int i = 5;  int j = --i; // j=4 i=4 | int i = 5;  i=i-1;  int j = i; // j=4 i=4 |
| int i = 5;  int j = i--; // j=5 i=4 | int i = 5;  int j = i;  i=i-1; // j=5 i=4 |

**Побітові операції**

|  |  |  |
| --- | --- | --- |
| Операція | Позначення | Приклад |
| & | побітове «і» | int x=10; //x=1010(2)  int y=7; //y=0111(2)  z=x&y // z=2=0010(2) |
| | | побітове «або» | int x=10; //x=1010(2)  int y=7; //y=0111(2)  z=x|y //z=15=1111(2) |
| ^ | Побітове «виключаюче або» | int x=10; //x=1010(2)  int y=7; //y=0111(2)  z=x^y //z=13=1101(2) |

**Бінарні операції зсуву**

|  |  |  |
| --- | --- | --- |
| Операція | Позначення | Приклад |
| >> | зсув розрядів вправо  (змінна) =(змінна)**>>**(кільк. розрядів) | int i=4; //i=100(2)  i=i>>1; //i=2=10(2) |
| << | зсув розрядів вліво  (змінна) =(змінна)<<(кільк. розрядів) | int i=4; //i=100(2)  i=i<<2; // i=16=10000(2) |

**Операції присвоєння**

|  |  |  |
| --- | --- | --- |
| Операція | Операція | Аналог з бінарними операціями |
| += | int i = 5;  i += 3; // i=8 | int i = 5;  i=i+3; // i=8 |
| -= | int i = 5;  i -= 3; // i=2 | int i = 5;  i=i-3; // i=2 |
| \*= | int i = 5;  i \*= 3; // i=15 | int i = 5;  i=i\*3; // i=15 |
| /= | int i = 6;  i /= 3; // i=2 | int i = 6;  i=i/3; // i=2 |
| >>= | int i = 5;  i >>= 1; // i=2 | int i = 5;  i=i>>1; // i=2 |
| <<= | int i = 5;  i <<= 1; // i=10 | int i = 5;  i=i<<1; // i=10 |
| &= | int i = 5;  int j = 7;  i &= j; // i=5 | int i = 5;  int j = 7;  i=i&j; // i=5 |
| ^= | int i = 5;  int j = 7;  i ^= 1; // i=2 | int i = 5;  int j = 7;  i=i^j; // i=2 |
| |= | int i = 5;  int j = 7;  i |= 1; // i=7 | int i = 5;  int j = 7;  i=i|j; // i=7 |

***Логічні вирази***

|  |  |  |
| --- | --- | --- |
| Операція | Позначення | Приклад |
| == | рівність | x==y |
| > | Більше | x>y |
| < | Менше | X<y |
| >= | більше або рівно | x>=y |
| <= | менше або рівно | X<=y |
| != | не рівно | x!=y |

У арифметичному виразі можуть також використовуватися логічні операції

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| x | y | x&&y  (логічне «і») | x || y  (логічне «або») | !x  (заперечення) |
| false | false | false | false | true |
| false | true | false | true | true |
| true | false | false | true | false |
| true | true | true | true | false |

**Умовний оператор**

**Повна форма**

|  |  |  |
| --- | --- | --- |
| **Програмна структура** | **Аналог на мові блок-схем** | **Приклад** |
| if (<умова>)  <оператор1>;  else  <оператор2>; | умова  Оператор 1  Оператор 2  +  – | if(x>y)  max=x;  else  max=y; |

**Скорочена форма**

|  |  |  |
| --- | --- | --- |
| **Програмна структура** | **Аналог на мові блок-схем** | **Приклад** |
| if (<умова>)  <оператор1>; | умова  Оператор 1  +  – | if(x!=0)  z=1/x; |

Якщо в умовному операторі при виконанні чи невиконанні умови необхідно виконати декілька операторів, то необхідно ці оператори помістити в складений оператор.

**Повна форма**

|  |  |  |
| --- | --- | --- |
| **Програмна структура** | **Аналог на мові блок-схем** | **Приклад** |
| if (<умова>)  {  <оператор1.1>;  . . . . . . . . . . .  <оператор1.N>;  }  else  {  <оператор2.1>;  . . . . . . . . . . .  <оператор2.M>;  } | умова  Оператор 1.1  +  –  Оператор 1.N  Оператор 2.1  Оператор 2.M | if(x>y)  {  max=x;  min=y;  }  else  {  max=y;  min=x;  } |

**Скорочена форма**

|  |  |  |
| --- | --- | --- |
| **Програмна структура** | **Аналог на мові блок-схем** | **Приклад** |
| if (<умова>)  {  <оператор 1>;  . . . . . . . . . . .  <оператор N>;  **}** | умова  Оператор 1  +  –  Оператор N | if(x>0)  {  z=1/x;  l=y/x;  } |

**Умовний оператор (?:)**

|  |  |
| --- | --- |
| Загальний вигляд умовного оператора **(?:)** | Аналог з використанням умовного оператора if |
| <змінна>=**<умова>?<значення1>:<значення 2>;** | if (<умова>)  <змінна>=**<значення 1>;**  else  <змінна>=**<значення 2>;** |

Приклад. Знайти максимальне із двох дійсних чисел.

|  |  |
| --- | --- |
| З використанням умовного оператора **(?:)** | З використанням умовного оператора if |
| max=(x>y)? x : y ; | if (x>y)  max=x;  else  max=y; |

**Оператор вибору** switch

|  |  |
| --- | --- |
| **Загальна форма** | **Приклад.** Вводиться оцінка – цифра, вивести оцінку прописом (селектор вибору цілого типу). |
| **switch (<селектор вибору>)**  **{**  **case <константа 1> : <оператор 1>;**  **break;**  **case <константа 2> : <оператор 2>;**  **break;**  **………………………………………….**  **case <константа N> : <оператор N>;**  **break;**  **default : <оператор N+1>;**  **break;**  **}** | using System;  using System.Collections.Generic;  using System.Linq;  using System.Text;  namespace ConsoleApplication3  {  class Program  {  static void Main(string[] args)  {  int mark;  Console.Write("Mark = ");  mark = int.Parse(Console.ReadLine());  **switch (mark)**  **{**  **case 2: Console.WriteLine("Незадовільно");**  **break;**  **case 3: Console.WriteLine("Задовільно.");**  **break;**  **case 4: Console.WriteLine("Добре");**  **break;**  **case 5: Console.WriteLine("Відмінно");**  **break;**  **default: Console.WriteLine("Неправильна**  **оцінка.");**  **break;**  **}**  Console.ReadKey();  }  }  } |

**Оператори циклу**

**Оператор циклу while:**

|  |  |  |
| --- | --- | --- |
| **Програмна структура** | **Аналог на мові блок-схем** | **Приклад.** Знайти суму перших n чисел. |
| while (<умова>)  <оператор>; | умова  Оператор  +  – | int sum=0;  int i=1;  while(i<=n)  sum=sum+i++; |

Якщо тіло циклу складається з більше ніж одного оператора, то необхідно використати складений оператор.

|  |  |  |
| --- | --- | --- |
| **Програмна структура** | **Аналог на мові блок-схем** | **Приклад.** Знайти суму і добуток перших n чисел. |
| while (<умова>)  {  <оператор 1>;  . . . . . . . . . . .  <оператор N>;  **}** | умова  Оператор 1  +  –  Оператор N | int sum=0;  int mult=1;  int i=1;  while(i<=n)  **{**  sum=sum+(i);  mult=mult\*(i++);  **}** |

**Оператор циклу do-while**

Оператор циклу do-while відрізняється від оператора while тим, що перевірка умови виконується не до, а після виконання інструкції (оператора).

|  |  |  |
| --- | --- | --- |
| **Програмна структура** | **Аналог на мові блок-схем** | **Приклад.** Знайти суму перших n чисел. |
| do  {  <оператор>;  }  while (<умова>); | умова  Оператор  +  – | int sum=0;  int i=1;  do  {  sum=sum+i++;  }  while(i<=n); |

Загальна форма оператора наступна

for (<вираз ініціалізації>;<логічна умова>;<ітераційна частина>)

<оператор>;

Схематично роботу оператора поки виконується умова можна зобразити так

Початкова ініціалізація

Умова продовження циклу

Оператор

Ітераційна частина

for( <вираз ініціалізації>;<умова продовження>;<ітераційна частина>)

<оператор>;

**Початок**

**Структурний підхід до розробки алгоритмів. Функції.**

|  |  |
| --- | --- |
| **Загальна форма** | **Приклад**. Знайти максимальне з трьох цілих чисел |
| **static <тип функції> <ім’я функції>(<список форм. парам.>)**  {  //тіло методу  } | **static int Max(int c1,int c2,int c3)**  **{**  **int m = c1;**  **if (c2 > m)**  **m = c2;**  **if (c3 > m)**  **m = c3;**  **return m;**  **}** |

Проілюструємо застосування структурного підходу на прикладі. Обчислити значення виразу 

|  |  |
| --- | --- |
| Без функцій | З функціями |
| using System;  using System.Text;  namespace ConsoleApplication3  {  class Program  {  static void Main(string[] args)  {  int a,b,c;  Console.Write("a=");  a = int.Parse(Console.ReadLine());  Console.Write("b=");  b = int.Parse(Console.ReadLine());  Console.Write("c=");  c = int.Parse(Console.ReadLine());  int max1 = a;  if (-b > max1)  max1 = -b;  if (3 > max1)  max1 = 3;  int max2 = a;  if (2\*b > max2)  max2 = 2\*b;  if (c > max1)  max2 = c;  int max3 = -a;  if (b > max3)  max3 = b;  if (7 > max3)  max3 = 7;  int S = max1 \* max2 - max3;  Console.WriteLine("S={0}",S);  Console.ReadKey();  }  }  } | using System;  using System.Text;  namespace ConsoleApplication3  {  class Program  {  **static int Max(int c1, int c2, int c3)**  **{ int m = c1;**  **if (c2 > m)**  **m = c2;**  **if (c3 > m)**  **m = c3;**  **return m;**  **}**  static void Main(string[] args)  {  int a,b,c;  Console.Write("a=");  a = int.Parse(Console.ReadLine());  Console.Write("b=");  b = int.Parse(Console.ReadLine());  Console.Write("c=");  c = int.Parse(Console.ReadLine());  int max1 = Max(a,-b,3);  int max2 = Max(a, 2\*b, c);  int max3 = Max(-a, b, 7);  int S = max1 \* max2 - max3;  Console.WriteLine("S={0}",S);  Console.ReadKey();  }  }  } |

**Формальні параметри методів**

* ref
* out
* params

|  |  |  |
| --- | --- | --- |
| **Передача за значенням**  **(немхє ніяких модифікаторів)**  **(інціалізація змінної n обов’язкова)** | **Передача за вказівником**  **(модифіктор ref)**  **(інціалізація змінної n обов’язкова)** | **Аргумент як вихідне значення (модифіктор out)**  **(інціалізація змінної n не обов’язкова)** |
| class Program  {  static void Fun1(int m)  {  m = 1;  }  static void Main(string[] args)  {  int n = **25**;  Fun1(n);  Console.WriteLine("n= {0}",n);  Console.ReadKey();  }  } | class Program  {  static void Fun1(**ref** int m)  {  m = 1;  }  static void Main(string[] args)  {  int n = **25**;  Fun1(**ref n**);  Console.WriteLine("n= {0}",n);  Console.ReadKey();  }  } | class Program  {  static void Fun1(**out** int m)  {  m = 1;  }  static void Main(string[] args)  {  int n;  Fun1(**out n**);  Console.WriteLine("n= {0}",n);  Console.ReadKey();  }  } |
| **Вивід програми:**  n=25 | Вивід програми:  n=1 | Вивід програми:  n=1 |
| **Змінні в пам’яті ЕОМ:**  До виклику:    **Під час роботи функції:**  **(для змінної m виділяється нова пам’ять)**    **Після роботи функції:** | **Змінні в пам’яті ЕОМ:**  До виклику:    **Під час роботи функції:**  **(для змінної m нова пам’ять не виділяється а викорстовується пам’ять змінної n)**    **Після роботи функції:** | **Змінні в пам’яті ЕОМ:**  До виклику:    **Під час роботи функції:**  **(для змінної m нова пам’ять не виділяється а викорстовується пам’ять змінної n)**    **Після роботи функції:** |

**Масиви**

**Одновимірні масиви**

*Опис одновимірного масиву*

**<тип елем.>[ ]** <ім’я типу масиву>;

Ім’я типу масиву задається у формі ідентифікатора.

**Приклад**.

**byte [ ] b; //** Опис одновимірного масиву **b** типу **byte**

**double [ ] d; //** Опис одновимірного масиву **d** типу **double**

char [ ] r; // Опис одновимірного масиу r типуchar

*Виділення пам’яті*

Виділення пам’яті здійснюється з використанням оператора new

**<тип елем.>** [ ] <ім’я масиву> **= new <тип елем.> [<кількість елементів>];**

**Приклад**

byte [ ] **b = new byte [20]; //**Виділення пам’яті для 20 елементів типу **byte**

double [ ] **d = new double[50]; //**Виділення пам’яті для 50 елементів типу **double**

char [ ] r=new char[15]; **//**Виділення пам’яті для 15 елементів типу char

Зауважимо, що кількість елементів може бути не тільки константою чи літералом, а й значення деякої змінної

Приклад.

int n; // n - кількість елементів

Console.Write("n=");

n = int.Parse(Console.ReadLine()); // Введення кількості елементів з клавіатури

int [ ] a = new int[n]; // Виділення пам’яті для n елементів масиву a

*Звертання до елементів масиву*

<ім’я масиву> [ <індекс> ]

**Приклад.**

b [ 0 ] = 2;

d [ 4 ] = 2.4;

r [ 1 ] = ‘М’ ;

*Початкова ініціалізація. Ініціалізатори масивів*

**Приклад.**

byte [ ] b = new byte [ 4 ] {11, 25, 23, 14};

double [ ] d=new double [ 3 ] { 2.3, 2.5, 8 };

byte [ ] b = {11, 25, 23, 14};

byte [ ] b = new byte [ ] {11, 25, 23, 14};

byte [ ] b = new byte [ 4 ];

b[0]=11;

b[1]=25;

b[2]=23

b[3]=14;

*Введення елементів масиву*

for(<параметр>= 0; <параметр> < <кільк.елем.>; <параметр>++)

{

Console.Write (“<ім’я мас.>[ {0} ]= ”,<параметр>);

<ім’я мас.>[ <параметр> ] =<тип елементів>.Parse(Console.ReadLine());

}

**Приклад.**

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

**{**

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

**a[i] = double.Parse(Console.ReadLine());**

**}**

*Виведення елементів масиву*

for(<параметр>= 0; <параметр> < <кільк.елем.>; <параметр>++)

{

Console.WriteLine(“<ім’я мас.>[ {0} ]={1} ”,<параметр>,<ім’я мас.>[ <параметр>]);

}

**Приклад.**

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

**{**

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

**}**

**Приклад**. Дано одновимірний масив з  елементів. Знайти суму елементів масиву.

static void Main(string[] args)

{

int n;

Console.Write("n="); //Введення кількості елементів n

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

double[] a = new double[n]; //Виділення пам’яті для масиву a

for (int i = 0; i < n; i++) //Введення елементів масиву a

{

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

a[i] = double.Parse(Console.ReadLine());

}

double Sum = 0;

for (int i = 0; i < n; i++) //Знаходження суми елементів масиву a

{

Sum+= a[i];

}

//Виведення суми на екран

Console.WriteLine("Sum={0}",Sum);

Console.ReadLine();

}

# Багатовимірні масиви

## Прямокутні масиви

*Опис двовимірного масиву*

<тип елементів> [ , ] <ім’я масиву>;

**Приклад.**

int [ , ] A;

double [ , ] d;

*Виділення пам’яті*

<тип елементів> [ , ] <ім’я масиву>=new <тип елементів>[кільк. рядків, кількі. стовпців];

Приклад.

int [ , ] A= new int[ 30,20];

double [ , ] d = new double [5 , 3];

int n; //Введення кількості рядків 

Console.Write("n=");

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

int m; //Введення кількості стовпців 

Console.Write("m=");

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

int[,] A = new int[n, m]; //Виділення пам’яті

*Звертання до елементів масиву*

<ім’я масиву> [ <індекс1> , <індекс2> ]

або, що те ж саме,

<ім’я масиву> [ <номер рядка > , <номер стовпця > ]

Приклад.

А[2,7]=21;

d[1,5]=34;

*Ініціалізатори масивів*

Прилад.

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

*Введення*

int n; //Введення кількості рядків 

Console.Write("n=");

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

int m; //Введення кількості стовпців 

Console.Write("m=");

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

int[,] A = new int[n, m]; //Виділення пам’яті

for (int i = 0; i < n; i++) // Введення масиву з клавіатури

{

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

{

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

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

}

}

*Виведення елементів масиву*

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

{

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

{

Console.WriteLine("A[{0},{1}]={2}", i, j,A[i, j]);

}

}

**Приклад.** Дано матрицю . Необхідно помножити кожен елемент матриці на її найбільший елемент.

*Розв’язання*

static void Main(string[] args)

{

int n;

Console.Write("n="); //Введення кількості рядків 

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

int m;

Console.Write("m="); //Введення кількості стовпців 

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

int[,] A = new int[n, m]; //Виділення пам’яті для масиву 

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

{

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

{

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

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

}

}

//Знаходження максимального 

int max = A[0, 0];

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

{

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

{

if (max < A[i, j])

{

max = A[i, j];

}

}

}

//Множення елементів на масиву на максимальний

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

{

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

{

A[i, j] \*= max;

}

}

//Виведення елементів масиву на екран

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

{

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

{

Console.WriteLine("A[{0},{1}]={2}", i, j, A[i, j]);

}

}

Console.ReadLine();

}

**Результати роботи програми**

n=2

m=3

A[0,0]=1

A[0,1]=2

A[0,2]=3

A[1,0]=4

A[1,1]=5

A[1,2]=6

A[0,0]=6

A[0,1]=12

A[0,2]=18

A[1,0]=24

A[1,1]=30

A[1,2]=36

***Багатовимірні масиви з різною кількістю елементів у рядку***

*Опис*

<тип елементів> [ ] [ ] <ім’я масиву>;

**Приклад.**

**double [ ] [ ] d;**

*Виділення пам’яті*

Приклад 1.

**double [ ] [ ] d = new double[3] []; //Виділення пам’яті для посилань на рядки**

d[0]=new double [3]; **//Виділення пам’яті для елементів рядків**

d[1]=new double [7];

d[2]=new double [5];

Як бачимо, кількість елементів у кожному з рядків різна.

Приклад 2.

**double [ ] [ ] d = new double[50] []; //Виділення пам’яті для посилань на рядки**

**for (int i=0; i<50; i++)**

**d[ i ] = new** double**[ i+1 ];**

*Звертання до елементів*

<ім’я масиву> [ <індекс1> ][ <індекс2> ]

або, що те ж саме,

<ім’я масиву> [ <номер рядка > ] [ <номер стовпця > ]

Приклад.

d[1][2]=34;

**Основи об’єктно-орієнтованого програмування у C#**

**1.** **Класи та об’єкти**

**1.1. Поняття класу та об’єкта**

Прямокутник

Сторона *a*

*Дані*

Сторона *b*

Дія *знаходження периметру*

Дія *знаходження площі*

*(з іншим прямокутником)*

*Функціональні можливості*

Дія *порівняння*

Дія *масштабування*

**1.2. Опис класів**

**class** <Ім’я класу>

**{**

*Поля класу*

[специфікатори] [const] <тип поля> <ім'я поля 1> [= <початкове значення>];

[специфікатори] [const] <тип поля> <ім'я поля 2> [= <початкове значення>];

. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .

[специфікатори] <Ім’я класу> ([<спис.форм. параметрів>])

*Конструктори*

{

<тіло конструктора 1>

}

[специфікатори] <Ім’я класу> ([<спис.форм. параметрів>])

{

*Члени класу*

<тіло конструктора 2>

}

. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .

*Методи класу*

[специфікатори] <тип результату> <ім’я методу 1> ([<спис.форм.парам.>])

{

<тіло методу 1>

}

[специфікатори] <тип результату> <ім’я методу 2> ([<спис.форм.парам.>])

{

<тіло методу 2>

}

. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .

*Деструктор*

~ <Ім’я класу> ()

{

<тіло деструктора>

}

* **private** – закриті члени класу. Доступні тільки у методах цього ж класу;
* **public** – відкриті члени класу. Доступ до них може бути здійснений поза межами класу;
* **protected** – захищені члени класу. Доступні у методах цього ж класу та методах його нащадків.

**Приклад.** Опишемо розглядувану логічну структуру “Прямокутник” за допомогою класу у C#.

| **Об’єкт предметної області** | **Програмна структура** class |
| --- | --- |
| **Прямокутник**  **Сторона *a***  **Сторона *b***  **Дія *створення трикутника***  **(конструктор)**  **Дія *знаходження периметру***  **Дія *знаходження площі***    **Дія *порівняння***  ***(з іншим прямокутником)***  **Дія *масштабування***  **Дія *знищення трикутника***  **(деструктор)** | class Rectangle  {  public double a;  public double b;  public Rectangle (double a, double b)  {  this.a = a;  this.b = b;  }  public double Perimeter ()  {  return 2 \* (a + b);  }  public double Square ()  {  return a \* b;  }    public bool IsEqual (double a, double b)  {  return (this.a == a && this.b == b)  || (this.a == b && this.b == a);  }  public void Scale (double k)  {  a = k \* a;  b = k \* b;  }  ~Rectangle ()  {  }  } |

**1.3. Конструктори. Створення об’єктів**

**1.3.1. Конструктор без параметрів**.

**Приклад**. Для класу Rectangle наведемо приклад реалізації конструктора без параметрів, який ініціалізує поля одиничними значеннями.

class Rectangle

{

. . . . . . . . .

**public Rectangle()**

**{**

**a = b = 1;**

**}**

. . . . . . . . .

}

**1.3.2. Конструктори з параметрами**.

**Приклад**. У нашому класі Rectangle вже є конструктор з двома параметрами. Для прикладу наведемо ще одну можливу реалізацію конструктора з одним параметром для класу Rectangle.

class Rectangle

{

. . . . . . . . . . . . .

public Rectangle(double a)

{

this.a = a;

b = 1;

}

. . . . . . . . . . . . .

}

**Приклад**. Модифікуємо у нашому класі Rectangle конструктор з двома параметрами наступним чином:

class Rectangle

{

. . . . . . . . . . . . .

public Rectangle(double a)

{

this.a = a;

b = 1;

}

public Rectangle (double a, double b)**:this(a)**

{

**this.b = b;**

}

. . . . . . . . . . . . .

}

**1.3.3. Конструктор копіювання**.

**Приклад**. Наведемо приклад реалізації конструктора копіювання для нашого класу Rectangle.

class Rectangle

{

. . . . . . . . . . . . .

public Rectangle(Rectangle r)

{

this.a = r.a;

this.b = r.b;

}

. . . . . . . . . . . . .

}

**1.3.4. Створення об’єктів**.

|  |
| --- |
| <Ім’я класу> <ім’я об’єкта> = new <Ім’я\_класу>(список\_параметрів); |

Приклади:

|  |
| --- |
| Rectangle r1 = new Rectangle() { a=2, b=7 };  Rectangle r2 = new Rectangle(2) { b=7 }; |

**1.4. Доступ до членів класу**

| **Загальне правило** | **Приклад** |
| --- | --- |
| <Ім’я об’єкта>**.**<Ім’я поля>  <Ім’я об’єкта>**.**<Ім’я методу> | Rectangle r = new Rectangle();  **r.a**=4;  **r.b**=5;  Console.WriteLine("S={0}", **r.Square()**);  Console.WriteLine("P={0}", **r.Perimeter()**); |

**Приклад**. Розглянемо детально реалізацію конструктора:

public Rectangle **(double a, double b)**

{

Формальні параметри

this.a = a;

this.b = b;

Поля, описані у класі

}

**1.5. Статичні поля та методи**

<Ім’я класу>.<Ім’я статичного поля>

**Приклад.** Опишемо у класі статичне поле, у якому будемо зберігати кількість створених об’єктів даного класу. У момент створення чергового об’єкта значення цього поля повинно збільшуватись на 1.

| **Приклад опису статичного поля** | **Приклад звертання до статичного поля** |
| --- | --- |
| class Rectangle  {  **public static int Count;**  . . . . . . . . . . . .  public Rectangle()  {  **Count++;**  a = b = 1;  }  . . . . . . . . . . . .  } | Rectangle r1 = new Rectangle();  Rectangle r2 = new Rectangle();  Console.WriteLine(**Rectangle.Count**); // 2 |

**Приклад**. Наведемо приклад реалізації статичного конструктора для нашого класу Rectangle, у якому статичне поле Сount буде ініціалізуватися нулем.

class Rectangle

{

. . . . . . . . . .

**static Rectangle()**

**{**

**Count = 0;**

**}**

. . . . . . . . . .

}

*Статичні методи*

<Ім’я класу>.<Ім’я статичного метода>

**Приклад.** Реалізуємо у класі Rectangle статичний метод знаходження площі прямокутника.

| **Приклад опису статичного методу** | **Приклад звертання до статичного методу** |
| --- | --- |
| class Rectangle  {  . . . . . . . . . . . .  **public static double Square(Rectangle r)**  **{**  **return r.a \* r.b;**  **}**  . . . . . . . . . . . .  } | Rectangle r1 = new Rectangle(3,4);  Console.WriteLine(**Rectangle.Square(r1)**); |

**1.6. Деструктор**

Загальний вигляд деструктора:

~<Ім’я класу>()

{

<тіло деструктора>

}

**Приклад**. У нашому класі Rectangle перед знищенням об’єкта потрібно зменшити значення відповідного лічильника об’єктів. Це можна зробити у деструкторі.

class Rectangle

{

. . . . . . .

**~Rectangle ()**

**{**

**Count--;**

}

. . . . . . .

}

1. **Перевантаження операторів**

**2.1. Унарні оператори**

+ - ! ~ ++ -- true false

Загальне правило перевантаження унарного оператора:

public static <тип результату> operator <унарний оператор>(<Ім’я класу> <аргумент>)

{

<тіло оператора>

}

| **Приклад перевантаження** | **Приклад використання** |
| --- | --- |
| class Complex  {  public double a;  public double b;  public Complex(double a, double b)  {  this.a = a;  this.b = b;  }  public static Complex operator -(Complex c)  {  return new Complex(-c.a, -c.b);  }  public static Complex operator ~(Complex c)  {  return new Complex(c.a, -c.b);  }  public static double operator !(Complex c)  {  return Math.Sqrt(c.a\*c.a + c.b\*c.b);  }  public static Complex operator ++(Complex c)  {  return new Complex(++c.a, ++c.b);  }  public static bool operator true(Complex c)  {  return c.b == 0;  }  public static bool operator false(Complex c)  {  return c.b != 0;  }  } | Complex c1 = new Complex(1,2); // 1+2i  // протилежне комплексне число  Complex c2 = -c1; // -1-2i  // спряжене комплексне число  Complex c3 = ~c1; // 1-2i  // модуль комплексного числа  double a = !c1; //  // префіксний та постфіксний інкримент  Complex c4 = ++c1; // 2+3i  Complex c5 = c1++; // 2+3i  if(c1)  Console.WriteLine("Уявна частина = 0.");  else  Console.WriteLine("Уявна частина != 0."); |

**2.2. Бінарні оператори**

+ - \* / % & | ^ << >> == != > < >= <=

Загальне правило перевантаження бінарного оператора:

public static <тип результату> operator <бін. опер.>(<Тип1> <арг.1>, <Тип2> <арг.2>)

{

<тіло оператора>

}

| **Приклад перевантаження** | **Приклад використання** |
| --- | --- |
| class Complex  {  . . . . . . . . . . . . . . . . . . . . . .    public static Complex operator +(Complex c1, Complex c2)  {  return  new Complex(c1.a + c2.a, c1.b + c2.b);  }  public static Complex operator +(Complex c, double d)  {  return new Complex(c.a + d, c.b);  }  public static Complex operator +(int i, Complex c)  {  return new Complex(c.a + i, c.b);  }  public static bool operator ==(Complex c1, Complex c2)  {  return c1.a == c2.a && c1.b == c2.b;  }  public static bool operator !=(Complex c1, Complex c2)  {  return !(c1==c2);  }  } | Complex n1 = new Complex(1,2); // 1+2i  Complex n2 = new Complex(3,-4); // 3-4i  // додаваня двох комплексних чисел  Complex s1 = n1 + n2; // 4-2i  // додаваня комплексного та дійсного чисел  double d1 = 0.5;  Complex s2 = n1 + d1; // 1.5 + 2i  // додаваня цілого та комплексного чисел  int i1 = 1;  Complex s3 = i1 + n1; // 2 + 2i  // перевірка на рівність двох комлексних чисел  if(n1 == n2)  Console.WriteLine("Числа рівні");  else  Console.WriteLine("Числа різні"); |

**2.3. Операції перетворення типів**

Загальне правило перевантаження операції *неявного* перетворення типів:

public static **implicit** operator <цільовий тип>(<Тип> <аргумент>)

{

<тіло оператора>

}

Загальне правило перевантаження операції *явного* перетворення типів:

public static **explicit** operator <цільовий тип>(<Тип> <аргумент>)

{

<тіло оператора>

}

| **Приклад перевантаження** | **Приклад використання** |
| --- | --- |
| class Complex  {  . . . . . . . . . . . . . . . . . . . . . .    public static implicit operator Complex(double d)  {  return new Complex(d,0);  }  public static explicit operator double(Complex c)  {  return c.a;  }  } | // неявне перетворення дійсного числа у  // комплексне  double d1 = 1.5;  Complex c1 = d1; // 1.5+0i  // явне перетворення комплексного числа  // у дійсне  Complex c2 = new Complex(-3,0);  double d2 = (double)c2; // d2=-3; |

**3.** **Інкапсуляція. Властивості та індексатори**

**3.1. Інкапсуляція**

Rectangle r1 = new Rectangle();

r1.a=-2;

r1.b=-9;

**Приклад.** Опишемо клас Rectangle, у якому поля для збереження довжин сторін прямокутника будуть закритими.

class Rectangle

{

// *Опис закритих членів класу*

**private double a\_;**

**private double b\_;**

public double Get\_a() { return a\_; } // *Зчитування* а\_

public void Set\_a(double a) // *Встановлення* а\_

{

if (a>0) a\_= a; // *Перевірка коректності значення* а

}

public double Get\_b() { return b\_; }// *Зчитування* b\_

public void Set\_b(double b) // *Встановлення* b\_

{

if (b>0) b\_= b; // *Перевірка коректності значення* b

}

public Rectangle (double a, double b)

{

// *Виклик функцій для встановлення значень закритих полів*

**Set\_a(a);**

**Set\_b(b);**

}

}

Rectangle r = new Rectangle(3,8);

r.**Set\_a**(2); r.**Set\_b**(9); // *Встановлення значень*

double s= r.**Get\_a**()\*r.**Get\_b**(); // *Зчитування значень*

**3.2. Властивості**

Загальний синтаксис опису властивості:

[специфікатор доступу] <тип властивості> <Ім'я властивості>

{

get

{

*// тіло функції зчитування значення*

return <результат>;

}

set

{

*// тіло функції встановлення значення*

}

}

**Приклад**. Наведемо приклад опису класу Rectangle, у якому реалізуємо принцип інкапсуляції із використанням властивостей.

class Rectangle

{

//*Опис закритих членів класу*

private double a\_;

private double b\_;

**public double a**

**{**

**get { return a\_; }** //*Зчитування* а\_

**set** //*Встановлення* а\_

**{**

**if (value > 0) a\_ = value;** //*Перевірка коректності значення* а

**}**

**}**

**public double b**

**{**

**get { return b\_; }** //*Зчитування* b\_

**set** //*Встановлення* b*\_*

**{**

**if (value > 0) b\_ = value;** //*Перевірка коректності значення* b

**}**

**}**

public Rectangle (double a, double b)

{

*//Звертання до властивостей для встановлення значень закритих полів*

this.a = a;

this.b = b;

}

}

**Приклад.** Нехай r є об’єктом класу Rectangle.

| **Звертання до властивості** | **Дія** | **Пояснення** |
| --- | --- | --- |
| r.a = 5; | *встановлення значення* | Викликається блок set властивості a. Значення 5 передається у блок set через неявний параметр value. |
| double d = r.a; | *зчитування значення* | Викликається блок get властивості a. |

**3.3. Індексатори**

Загальний синтаксис опису індексатора:

[специфікатор доступу] <тип індексатора> this[<список індексів>]

{

get

{

*//тіло функції одержання значень за індексами*

return <результат>;

}

set

{

*//тіло функції встановлення значень за індексами*

}

}

**Приклад**. Розглянемо приклад опису класу Array для роботи із одновимірним масивом, елементи якого повинні знаходитись у діапазоні [1,100].

| **Приклад опису одновимірного індексатора** | **Приклад звертання до індексатора** |
| --- | --- |
| class Array  {  private int[] a;  public Array(params int[] a)  {  this.a = new int[a.Length];  for (int i = 0; i < a.Length; i++)  this[i] = a[i];  }  // *властивість тільки для читання*  // *кількості елементів*  public int Length  {  get  {  return a.Length;  }  }  // *індексатор для доступу до елементів*  // *закритого поля-масиву*  **public int this[int i]**  **{**  **get**  **{**  **if (i >= 0 && i < this.Length)**  **return a[i];**  **else**  **return 0;**  **}**  **set**  **{**  **if (i >= 0 && i < this.Length &&**  **value >= 0 && value <= 100)**  **a[i] = value;**  **}**  **}**    } | Array m = new Array(1,2,3);  int v = m[2];  m[1] = 5; |

1. **Успадкування**

Загальний синтаксис опису класу-нащадка:

[список специфікаторів] class <Ім'я класу-нащадка> : <Ім’я класу-предка>

{

...

}

**Приклад.** Опишемо клас, який моделює прямокутний паралелепіпед RParallelepiped, як нащадок від класу Rectangle.

public class Rectangle

{

private double a\_;

private double b\_;

public double a

{

get { return a\_; }

set

{

if (value > 0) a\_ = value;

}

}

public double b

{

get { return b\_; }

set

{

if (value > 0) b\_ = value;

}

}

public double Perimeter()

{

return 2 \* (a + b);

}

}

public class **RParallelepiped** : Rectangle

{

private double c\_;

public double c

{

get { return c\_; }

set

{

if (value > 0) c\_ = value;

}

}

public double Volume()

{

return a \* b \* c;

}

}

**4.2. Успадкування та конструктори**

[специфікатори] <Ім’я класу> ([<спис. форм. парам.>]) : **base** ([<спис. факт. парам.>])

{

<тіло конструктора класу-нащадка>

}

**Приклад**. Проілюструємо особливості опису конструкторів при успадкуванні на прикладі наших класів Rectangle та RParallelepiped.

public class Rectangle

{

. . . . . . . . . . . . . .

public Rectangle(double a, double b)

{

this.a = a;

this.b = b;

}

}

public class RParallelepiped : Rectangle

{

. . . . . . . . . . . . . .

public RParallelepiped(double a, double b, double c) **: base(a, b)**

{

this.c = c;

}

}

**4.3. Перевизначення методів**

base.<Iм'я прихованого методу класу-предка>([<cпис. факт. параметрів>]);

public class Rectangle

{

. . . . . . . . . . .

public double Square()

{

return a \* b;

}

}

public class RParallelepiped : Rectangle

{

. . . . . . . . . . . . . .

public **new** double Square()

{

return Perimeter() \* c + 2 \* **base**.Square();

}

}

**4.4. Заборона успадкування**

**Приклад**. Заборонимо створювати класи-нащадки від нашого класу RParallelepiped:

public **sealed** class RParallelepiped: Rectangle

{

...

}

**5.** **Поліморфізм**

**5.1. Поняття поліморфізму**

Розрізняють 3 типи поліморфізму:

* *Параметричний* *поліморфізм* − дозволяє в межах одного класу описувати декілька методів з однаковим іменем але різним списком формальних параметрів. Відповідні методи називають *поліморфними методами класу*. Мова C# підтримує параметричний поліморфізм методів (згадайте правила опису функцій).
* *Простий поліморфізм* − дозволяє перевизначити методи при успадкуванні. Відповідні методи називають *статичними поліморфними*. Мова C# підтримує простий поліморфізм методів (див. п. 4.3.).
* Складний поліморфізм.

**5.2. Складний поліморфізм (віртуальні методи)**

RParallelepiped rp = new RParallelepiped(2, 3, 4);

Rectangle r = rp; // *тут діє правило неявного висхідного перетворення типів*

**Приклад**. Модифікуємо описання класу Rectangle, описавши метод знаходження площі прямокутника Square як віртуальний. При описанні класу-нащадка RParallelepiped визначимо метод знаходження площі повної поверхні паралелепіпеда Square, із використанням специфікатора override.

public class Rectangle

{

. . . . . . . . . . .

public **virtual** double Square()

{

return a \* b;

}

}

public sealed class RParallelepiped : Rectangle

{

. . . . . . . . . . . . . .

public **override** double Square()

{

return Perimeter() \* c + 2 \* base.Square();

}

}

Нехай у програмі описано два об’єкти r і rp.

Rectangle r = new Rectangle (3, 5);

RParallelepiped rp = new RParallelepiped(2, 3, 4);

Тоді:

r.Square() Обчислення площі прямокутника (описано в Rectangle)

rp.Square() Обчислення площі поверхні паралелепіпеда (описано в RParallelepiped)

Надамо змінній r значення rp.

r = rp;

**5.3. Перетворення типів при успадкуванні**

Rectangle r = new Rectangle(3, 5);

RParallelepiped rp1 = new RParallelepiped(2, 3, 4);

r = rp1;

// *тут діє правило явного низхідного перетворення типів*

RParallelepiped rp2 = (RParallelepiped)r;

Rectangle r = new Rectangle(3, 5);

object o = r;

// *тут виникне помилка часу виконання*

RParallelepiped rp = (RParallelepiped)o;

Для запобігання подібних ситуацій у мові C# визначено оператори is та as. Загальний синтаксис цих операторів наступний:

<об’єкт> as <ім’я\_класу>

<об’єкт> is <ім’я\_класу>

Rectangle r = new Rectangle(3, 5);

object o = r;

// *тут вже не виникне помилка часу виконання*

if (o is RParallelepiped) // або if (o as RParallelepiped != null)

{

RParallelepiped rp = (RParallelepiped)o;

}

**5.4. Абстрактні методи**

**Приклад.** Опишемо клас Figure, який є предком класів Rect (представляє квадрат) та Circle (представляє круг). Обидві геометричні фігури характеризуються однією величиною Data (сторона квадрата, чи радіус круга) та методом відображення цієї величини Show\_Data, однак способи знаходження площі цих фігур відрізняються.

public **abstract** class Figure // *клас абстрактний*

{

private double Data\_;

public double Data

{

get{ return Data\_; }

set

{

if(value > 0) Data\_ = value;

else Data\_ = 1;

}

}

public Figure(double data)

{

Data = data;

}

public void Show\_Data()

{

Console.WriteLine("Data={0}", Data);

}

public **abstract** double Square()**;** // *метод абстрактний (реалізація не наводиться)*

}

public class Rect : Figure

{

public Rect(double a) : base(a)

{

}

public **override** double Square() // *абстрактний метод перевизначається*

{

return Data \* Data;

}

}

public class Circle : Figure

{

public Circle(double r) : base(r)

{

}

public **override** double Square() // *абстрактний метод перевизначається*

{

return Math.PI \* Data \* Data;

}

}

**6.** **Інтерфейси**

**6.1. Поняття інтерфейсу. Опис інтерфейсів**

[специфікатор доступу] interface <Ім’я інтерфейсу> [:<Список інтерфейсів-предків>]

{

<тип результату> <ім’я методу 1> ([<спис.форм.парам.>]);

<тип результату> <ім’я методу 2> ([<спис.форм.парам.>]);

. . . . . . . . . . . . . . . . . . . . . . . . . . . .

}

public interface IMeasurable

{

double Perimeter();

double Square();

}

public interface ICircumcircleIncircle

{

double R { get; } // *радіус описаного кола*

double r { set; } // *радіус вписаного кола*

}

**6.2. Реалізація інтерфейсів**

[специфікатор доступу] class<Ім’я класу>:<Ім’я класу-предка>,

**<Ім’я інтерфейсу-предка1>,<Ім’я інтерфейсу-предка2>, ..., <Ім’я інтерфейсу-предкаN>**

{

**<реалізація членів інтерфейсу-предка1>**

**<реалізація членів інтерфейсу-предка2>**

**. . . . . . . . . . . . . . . . . . . .**

**<реалізація членів інтерфейсу-предкаN>**

<опис інших членів класу>

}

**public** <тип результату> <Ім’я методу> ([<спис.форм.парам.>])

{

<реалізація методу>

}

**Явна реалізація**.

<тип результату> **<Ім'я інтерфейсу>.**<Ім’я методу> ([<спис.форм.парам.>])

{

<реалізація методу>

}

**Приклад.** Опишемо клас Rect (геометрична фігура квадрат) та наведемо в цьому класі реалізацію інтерфейсів IMeasurable та ICircumcircleIncircle.

public class Rect : **IMeasurable,** **ICircumcircleIncircle**

{

private double a\_;

public double a

{

get { return a\_; }

set

{

if (value > 0) a\_ = value;

else a\_ = 1;

}

}

public Rect(double a)

{

this.a = a;

}

// ***неявна реалізація методу Perimeter інтерфейсу IMeasurable***

public double Perimeter()

{

return 4 \* a;

}

// ***явна реалізація методу Square інтерфейсу IMeasurable***

double IMeasurable.Square()

{

return a \* a;

}

// ***неявна реалізація властивості R інтерфейсу ICircumcircleIncircle***

public double R

{

get

{

return (Math.Sqrt(2) / 2) \* a;

}

}

// ***явна реалізація властивості r інтерфейсу ICircumcircleIncircle***

double ICircumcircleIncircle.r

{

get

{

return a / 2;

}

}

}

**6.3. Доступ до функціональних можливостей** **інтерфейсу**

У випадку неявної реалізації члена інтерфейсу є можливість доступу до нього на рівні об’єктів. Наприклад:

Rect r1 = new Rect(2); // квадрат зі стороною 2

**IMeasurable i1 = (IMeasurable)r1;**

Console.WriteLine("Периметр квадрата:{0}", i1.Perimeter());

Console.WriteLine("Площа квадрата:{0}", i1.Square());

Rect r1 = new Rect(2); // квадрат зі стороною 2

Console.WriteLine("Радіус описаного кола:{0}", **r1.R**);

## DataGridView

|  |
| --- |
| public DataGridViewRowCollection Rows { get; } |

|  |
| --- |
| Назва\_елемента\_управління.Rows[номер\_рядка].Cells[номер\_стовпця].Value |

Для доступу до комірок можна також використовувати наступний індексатор класу DataGridView:

|  |
| --- |
| public DataGridViewCell this[int columnIndex, int rowIndex] { get; set; } |

Розмірність таблиці можна регулювати через наступні властивості класу DataGridView:

|  |  |
| --- | --- |
| public int RowCount { get; set; } | Повертає або задає кількість рядків для даного екземпляру DataGridView |
| public int ColumnCount { get; set; } | Повертає або задає кількість стовпців для даного екземпляру DataGridView |

|  |  |
| --- | --- |
| public event DataGridViewCellEventHandler CellValueChanged; | Настає при зміні значення комірки |

У класі DataGridView також реалізовано багато методів та властивостей, які дозволяють налаштувати зовнішній вигляд відповідного елемента управління. Нижче наведено деякі з цих методів.

|  |  |
| --- | --- |
| public bool AllowUserToAddRows { get; set; } | Повертає або задає значення, що вказує, чи відображається для користувача параметр додавання рядків |
| public bool AllowUserToResizeColumns {get;set;} | Повертає або задає значення, що вказує, чи може користувач змінювати розмір стовпців |
| public bool AllowUserToResizeRows { get; set; } | Повертає або задає значення, що вказує, чи може користувач змінювати розмір рядків |
| public bool ColumnHeadersVisible { get; set; } | Повертає або задає значення, що вказує, чи відображається рядок заголовків стовпців |
| public bool RowHeadersVisible { get; set; } | Повертає або задає значення, що вказує, чи відображається стовпець, що містить заголовки рядків |
| public bool ReadOnly { get; set; } | Повертає або задає значення, що вказує, чи може користувач змінювати комірку елемента управління DataGridView |
| public Color BackgroundColor { get; set; } | Повертає або задає колір фону для об'єкта DataGridView |

|  |  |
| --- | --- |
| public DataGridViewCellStyle Style {get; set;} | Повертає або задає стиль окремої комірки |

|  |
| --- |
| *Визначити позиції всіх екстремальних елементів цілочислової матриці, згенерованої випадковим чином.* |
| namespace Unbound  {  public partial class MyForm : Form  {  int Min = int.MaxValue;  int Max = int.MinValue;  public MyForm()  {  InitializeComponent();  }  private void rowCount\_ValueChanged(object sender, EventArgs e)  {  Matrix.RowCount = (int)rowCount.Value;  }  private void colCount\_ValueChanged(object sender, EventArgs e)  {  Matrix.ColumnCount = (int)colCount.Value;  }  private void Form1\_Load(object sender, EventArgs e)  {  Matrix.RowCount = 1;  Matrix.ColumnCount = 1;  }  private void btnRandom\_Click(object sender, EventArgs e)  {  Min = int.MaxValue;  Max = int.MinValue;  Random r = new Random();  for (int i = 0; i < Matrix.RowCount; i++)  for (int j = 0; j < Matrix.ColumnCount; j++)  Matrix[j,i].Value = r.Next(-50, 50);  }  private void btnRun\_Click(object sender, EventArgs e)  {  for (int i = 0; i < Matrix.RowCount; i++)  for (int j = 0; j < Matrix.ColumnCount; j++)  {  Matrix[j,i].Style.BackColor = Matrix.DefaultCellStyle.BackColor;  if ((int)Matrix[j,i].Value == Max)  Matrix[j,i].Style.BackColor = Color.Green;  else  if ((int)Matrix[j,i].Value == Min)  Matrix[j,i].Style.BackColor = Color.Red;  }  }  private void Matrix\_CellValueChanged(object sender, DataGridViewCellEventArgs e)  {  ((DataGridView)sender)[e.ColumnIndex,e.RowIndex].Style.BackColor = ((DataGridView)sender).DefaultCellStyle.BackColor;  if ((int)((DataGridView)sender)[e.ColumnIndex,e.RowIndex].Value < Min)  Min = (int)((DataGridView)sender)[e.ColumnIndex,e.RowIndex].Value;  if ((int)((DataGridView)sender)[e.ColumnIndex,e.RowIndex].Value > Max)  Max = (int)((DataGridView)sender)[e.ColumnIndex,e.RowIndex].Value;  }  }  } |

***Зв’язаний режим.***

|  |  |
| --- | --- |
| public Object Current { get; } | Повертає поточний елемент списку |
| public int Position { get; set; } | Повертає або задає індекс поточного елемента в базовому списку |
| public virtual int Add(object value); | Додає існуючий елемент у кінець списку |
| public virtual void Insert(int index, object value); | Вставляє елемент у список за вказаним індексом |
| public virtual void Clear(); | Видаляє всі елементи зі списку |
| public void RemoveCurrent(); | Видаляє поточний елемент зі списку |
| public virtual void RemoveAt(int index); | Видаляє елемент зі списку за вказаним індексом |

|  |
| --- |
| *Нехай задано список, який містить інформацію про книжки. Відомості про кожну книжку − це автор, назва та рік видання. Розробити рішення, графічний інтерфейс якого дає можливість виконувати наступні задачі:*   * *додавання нової інформації про книжки;* * *редагування інформації про наявні книжки;* * *вилучення інформації про книжки;* * *збереження даних списку у файлі та завантаження інформації про книжки із файлу;* * *вивід інформації про книги за назвою.* |
| namespace Bound  {  public partial class Form1 : Form  {  List<Book> BookList;  BindingSource BindSource;  public Form1()  {  InitializeComponent();  }  private void Form1\_Load(object sender, EventArgs e)  {  BookList = new List<Book>();  BindSource = new BindingSource();  BindSource.DataSource = BookList;  BookInfo.DataSource = BindSource;  BookInfo.Columns["Author"].HeaderText = "Автор";  BookInfo.Columns["Title"].HeaderText = "Назва";  BookInfo.Columns["Year"].HeaderText = "Рік видання";  }  private void btnAdd\_Click(object sender, EventArgs e)  {  Book item = new Book(textAuthor.Text, textTitle.Text, int.Parse(textYear.Text));  BindSource.Add(item);  }  private void toolStripMenuItem4\_Click(object sender, EventArgs e)  {  Application.Exit();  }  private void toolStripMenuItem3\_Click(object sender, EventArgs e)  {  if (saveFileDialog.ShowDialog() == DialogResult.OK)  {  FileStream fs = new FileStream(saveFileDialog.FileName, FileMode.Create, FileAccess.Write, FileShare.ReadWrite);  BinaryFormatter bf = new BinaryFormatter();  bf.Serialize(fs, BookList);  fs.Close();  }  }  private void toolStripMenuItem2\_Click(object sender, EventArgs e)  {  if (openFileDialog.ShowDialog() == DialogResult.OK)  {  FileStream fs = new FileStream(openFileDialog.FileName, FileMode.Open, FileAccess.Read, FileShare.Read);  BinaryFormatter bf = new BinaryFormatter();  BindSource.Clear();  foreach (Book b in (List<Book>)bf.Deserialize(fs))  BindSource.Add(b);  fs.Close();  }  }  private void btnSearch\_Click(object sender, EventArgs e)  {  textResultSearch.Text = "";  foreach (DataGridViewRow row in BookInfo.Rows)  if ((string)row.Cells["Title"].Value == textSearch.Text)  textResultSearch.Text += (string)row.Cells["Author"].Value + Environment.NewLine;  }  }    [Serializable]  class Book  {  public string Author { get; set; }  public string Title { get; set; }  public int Year { get; set; }  public Book(string \_author, string \_title, int \_year)  {  Author = \_author;  Title = \_title;  Year = \_year;  }  }  } |