**Технологія .Net Framework. Середовище CLR**

Технологія .Net Framework представляє собою середовище, яке дає можливість розробляти і виконувати програми не залежно від платформи (операційної системи). Вона дозволяє розробляти проекти, частини яких можуть бути виконані у різних мовах програмування. Основною мовою програмування цієї технології є мова С#. У цій технології єдиний спосіб опису програмного коду (метадані), одне середовище виконання (Common Language Runtime) і одна базова бібліотека (Frameworks).

Міжмовна інтеграція досягається за рахунок того, що програма, написана на мові високого рівня (С# чи Basic, чи C++, чи ін.) не одразу компілюється у машинний код, а спочатку відбувається компіляція у команди (псевдокод) єдиної проміжкової мови Intermediate Language (IL).

Незалежність від платформи досягається за рахунок того, що під час виконання програми команди проміжкової мови перетворюються у машинні команди у середовищі Common Language Runtime (CLR). за допомогою відповідних компіляторів (JIT-компіляторів).

С#

С++

Basic

Інші мови програмування

Проміжкові мова

(IL)

Машинний код

**JIT компілятор**

**CLR**

Отже, технологія .Net Framework реалізована в середовищі Common Language Runtime (CLR). CLR можна інтерпретувати, як деякий препроцесор, який перетворює команди проміжкової мови Intermediate Language (IL) в команди процесора. При цьому використання CLR для розробки програм забезпечує:

* міжмовну інтеграцію
* автоматичне керування пам’яттю (збирання сміття)
* контроль типів
* контроль версій програм

Більшість з цих можливостей можливі завдяки тому, що CLR реалізовано з використанням метаданих. Метадані представляють собою об’єднання програм і даних в одному файлі формату Porrtable Executable (PE). Програма на проміжковій мові містить не тільки сам програмний код, але і метадані, які надають середовищу виконання інформацію про всі типи, що використовуються, всіх сигнатурах, всіх елементів цих типів і іншу інформацію, яка в програмах з використанням СОМ поміщалась в бібліотеки типів і системний реєстр. Таким чином, програма і вся необхідна їй інформація розміщена в одному файлі, що суттєво полегшує установку програми, оскільки виключає реєстрацію в системному реєстрі і пошук відповідних бібліотек.

Як було зазначено, С# є основною мовою програмування технології .Net Framework. Програма може розроблятися і на інших мовах програмування, але при цьому необхідно дотримуватися певних правил, що визначено у середовищі CLR. Так правила описання і використання типів у середовищі CLR визначаються системою віртуальних об’єктів (NGWS Virtual Object System або ж VOS). Використання системи VOS спрощує міжмовну інтеграцію і забезпечує безпечність типів без суттєво зниження роботи програми. В мові використовуються чотири основні складові VOS:

1. Система типів – призначена для уніфікації типів даних, які використовуються різними мовами програмування.
2. Метадані – містять описання і посилання на типи, які визначені в системі.
3. Common Language Specification – специфікація єдиної мови.

Virtual Execution System -- віртуальне середовище виконання. Це середовище представляє собою реалізацію VOS і відповідає за завантаження і виконання програм, які використовують CLR.

**Мова програмування. Поняття програми**

Як було сказано раніше, ***мова програмування –*** фіксована система позначень та правил для опису структур даних та алгоритмів, призначених для виконання обчислювальними машинами.

З формальної точки зору ***мова програмування***  – це набір вихідних символів *(алфавіт)* разом із системою правил утворення з цих символів фор­мальних конструкцій *(синтаксис)* та системою правил їх тлумачення (інтерпре­тації) *(семантика),* за допомогою яких описуються алгоритми. **Програма –** це алгоритм записаний за допомогою мови програмування.

**Алфавіт, синтаксис та семантика** — це три основні складові частини будь-якої мови програмування. До **алфавіту** мови програмування, як правило, входять літери латинського алфавіту, арабські цифри, знаки арифметичних операцій, розділові знаки, спеціальні символи. Із символів алфавіту будують послідовності, які називають *словами (лексемами).* Кожне слово у мові програмування має своє змістове призначення. **Правила синтаксису** пояснюють, як потрібно будувати ті чи інші мовні повідомлення для опису всіх понять мови, здійснення описів та запису вказівок. **Правила семантики** пояснюють, яке призначення має кожен опис та які дії повинна виконати обчислювальна машина під час виконання кожної із вказівок. Вказівки на виконання конкрет­них дій називають ще **командами** або **операторами** мови.

Усі слова або ж лексеми, поділяють на

* *службові* (заре­зервовані або ж ключові) слова;
* *стандартні ідентифікатори;*
* *ідентифікатори користувача;*
* *знаки операцій;*
* *літерали;*
* *розділові знаки*.

*Службові слова* мають наперед визначене призначення і використовують­ся для формування структури програми, здійснення описів, позначення опе­рацій, формування керуючих конструкцій (вказівок). Наприклад, службовими словами для мови С# є: abstract, do, in, protected, as, public, try, base, else, interface, readonly, typeof, enum, internal, ref, break, event, is, return, byte, explicit, lock, unchecked, case, extern, sealed, unsafe, catch, namespace, finally, new, sizeof, using, checked, fixed, alloc, virtual, class, const, for, operator, volatile, continue, foreach, out, struct, while, goto, override, switch, default, if, params, this, delegate, implicit, private, throw.

Імена (позначення) для програмних об'єктів (типів даних, констант, змінних, функцій і т. п.) формують у вигляді ідентифікаторів. *Ідентифікатор*— це послідовність латинських літер, цифр і знака підкреслення, яка розпочинається з латинської літери. У мові С# максимальна довжина ідентифікатора є необмеженою.

:*Стандартні ідентифікатори* використовуються як імена для стандартних (передбачених авторами мови програмування) типів даних, констант, підпро­грам (зокрема, стандартних математичних функцій). Приклад: int, long,double, char, bool та ін.

*Ідентифікатори корис­тувача* є іменами тих програмних об'єктів (констант, змінних, функцій тощо), які створює сам користувач. Службові слова та ідентифікатори користувача не повинні збігатися.

*Коментар –* невиконувана частина тексту програми, що ігнорується компілятором і служить для вставки деяких поміток у програмі тільки для програміста. Коментарі бувають однорядковими та багаторядковими. Однорядковий коментар починається з символів «//» і закінчується у кінці рядка. Тобто всі символи до кінця рядка вважаються коментарем.

Приклад.

// Це однорядковий коментар

Багаторядковий коментар починається з символів «**/\***» і закінчується символами «**\*/**».

Приrлад.

**/\*** Це

багаторядковий

коментар **\*/**

*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 |

*Розділові знаки* служать для відокремлення однієї лексеми від іншої. Ними є пробіл, табуляція, символ нового рядка, символ «;» та коментарі.

**ВЕЛИЧИНА. ТИП ВЕЛИЧИНИ**

У своїй роботі програміст має справу з таким поняттям, як величина. З точки зору програмування ***величини*** – це дані, що обробляються програмами. *Дані* — це інформація, введена у пам'ять комп'ютера або підготовлена до введення.

Носіями даних у програмах є *константи, змінні* (значення яких зберігається в оперативній пам'яті) та *файли* (на зовнішніх носіях інформації).

*Константи* — це величини, значення яких у процесі виконання програми не змінюється. *Змінні* — це величини, значення яких у процесі виконання програми можуть змінюватися. Імена констант і змінних, як і інших програм­них об'єктів, записують у формі ідентифікаторів. Кожна змінна і константа належать до визначеного типу.

**Тип даних –**  це сукупність властивостей певного набору даних, від яких залежать: діапазон значень, якого можуть набувати ці дані, а також сукупність операцій, які можна виконувати над цими даними.

З іншого боку **тип даних**  – це описання того, яку структуру, розмір мають комірки оперативної пам’яті при зберіганні відповідного елемента даних.

**Елемент даних певного типу –**  це комірка або комірки оперативної пам’яті, що мають фіксовану адресу, розряди яких розшифровуються згідно описання даного типу даних.

З кожним типом даних зв’язано своє унікальне ім’я (ідентифікатор), яке є синонімом певного описання елементу даних відповідного типу. Наприклад, ідентифікатор **byte** є синонімом опису : 8 послідовних розрядів містить ціле значення без знаку в діапазоні від 0 до 255 (у двійковому вигляді займає 1 байт).

Тип констант визначається компілятором автоматично, але позначивши змінну ідентифікатором, ще треба вказати її тип.

Слід зазначити, що тип даних – це абстрактний описання і тому прямо використовуватися він не може. Його використання здійснюється через елементи даних відповідного типу. Наприклад, елемент даних типу **byte** – це комірка оперативної пам’яті з певною адресою, розряди якої декодуються або розуміються згідно описання типу b**yte**. Оскільки безпосередньо, через вказівку адрес, працювати з елементами даних незручно, то кожному елементу даних ставиться у відповідність ідентифікатор (ім’я змінної чи константи), користуючись яким можна здійснювати доступ до елемента даних.

У зв’язку з цим можна дати інше означення константи та змінної.

Якщо елемент даних не може змінювати свого значення, тобто завжди містить одне і те ж саме значення, то відповідний ідентифікатор називається *константою даного типу.* Якщо елемент даних певного типу може змінювати своє значення під час виконання програми, то ідентифікатор, що зв’язаний з цим елементом даних називається змінною відповідного типу. *Значення змінної* – це елемент даних, з якими ця змінна пов’язана.

Отже, у програмах змінна характе­ризується такими ознаками: *іменем, типом* і *значенням.*

**Типи змінних у** С#

В мові підтримується стандартний набір типів даних. Однак кожен тип даних є об’єктом, що не передбачається в стандартному наборі. Для вирішення цього протиріччя в мові розрізнять: ***типи значень*** і ***типи посилань*** . Типи значень включать в себе прості типи змінних, а також перелікові типи даних і структур, тобто ті типи, які зазвичай не є об’єктами. До простих типів відносяться, наприклад, такі типи як char, int, float. Щоб перетворити їх в об’єкти і одночасно забезпечити їх використання в тих мовах, де вони не є об’єктами, кожному простому типу ставиться у відповідність об’єкт CLS.

До типів посилань відносяться типи класів, інтерфейсів, масивів та делегатів.

**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;** |

Зазначимо, що кожен з наведених типів є скорочення деякого системного типу. Наприклад, тип **Int** є альтернативною назвою структури **System.Int32**. В більшості випадків ці імена повністю рівноправні.

**Опис змінних**

Загальний опис:

без початкової ініцініалізації

<тип змінної> <ідентифікатор змінної>;

з початковою ініціалізацією

<тип змінної> <ідентифікатор змінної> [ = <значення>] ;

Приклад.

int a; // Без початкової ініціалізації.

int b=40; // Змінна b ініціалізовується значенням 40.

double s=3.67; // Змінна s ініціалізовується значенням 3.67.

**Опис констант**

Загальний опис

const <тип константи> <ідентифікатор констати> = <значення>;

Наприклад.

const int a = 10;

const float b = 2.5;

**Переліковний тип. Іменовані константи**

*Переліковний тип* це тип, який задається повним переліком свої значень. Значення переліконог типу є іменованими константами типів byte, sbyte, short, ushort, int,uint, long, ulong. Якщо при описані переліковного типу в явному вигляді не вказано тип його констант, вважається, що константи типу int. Всі константи повинні бути одного типу, який називається базовим типом. Тип char не може використовуватися в якості базового типу.

Переліковні типи є потомками від класу System.Enum. Загальна форма описання

<атрибути><модифікатори> enum <ім’я типу> : <базовий тип>

{

<атрибути> <іменована конст.1> [= <знач.1>],

<атрибути> <іменована конст.2> [= <знач.2>],

……………

<атрибути> <іменована конст.N> [= <знач.N>]

}

Приклад.

enum Week : Byte

{

Monday,

Tuesday,

Wednesday,

Thursday,

Friday,

Saturday,

Sunday

}

Іменовані константи можуть мати атрибути та можуть бути в явному вигляді проініціалізовані. довільними допустимими значеннями. В переліковному типі може існувати довільне число іменованих констант. При цьому діапазон значень цих констант не повинен перевищувати діапазон допустимих значень базового типу.

Наприклад

enum Week : Byte

{

Monday =2,

Tuesday =3,

Wednesday =4,

Thursday =5,

Friday =6,

Saturday =7,

Sunday =1,

Subbath =Saturday

}

При неявній ініціалізації іменованих констант, тобто, коли вони не мають ініціалізатора, їм присвоюється значення, встановлене за замовчуванням. При цьому діють наступні правила:

* якщо дана іменована константа є першою в списку іменованих констант, то вона ініціалізується нульовим значенням;
* в іншому випадку іменована константа ініціалізується значенням попередньої константи списку, збільшеним на одиницю. При цьому значення константи повинно знаходитись в діапазоні допустимих значень базового типу.

**Перетворення типів**

В мові розрізняють два види перетворень змінних – *явне*  та *неявне.*

**Неявне перетворення** типів використовується в тому випадку, коли дане перетворення є «природним». Тобто, наприклад, перетворюється величина типу float в величину типу double.

Приклад.

float f = 1.23;

double d = f; //Неявне перетворення

Таке перетворення є природним, оскільки обидва типи використовуються для представлення дійсних типів, причому цільовий тип має більший діапазон представлення і більшу точність. При такому перетворення не відбувається втрата інформації. При проведенні неявного переторення немає необхідності вказувати цільовий тип (тип до якого здійснюється перетворення).

**Явне перетворення** типів вимагає явного задання цільового типу, до якого здійснюється перетворення. Цільовий тип вказується в дужках перед значенням тип якого перетворюється.

<змінна> = (<цільовий тип>) <вираз>;

Приклад.

double d = 2.9;

float f = (float) d; // Явне перетворення

Для цього типу повинно існувати неявне перетворення до типу змінної, в якій буде збережено результат.

В C# всі типи, включаючи типи значень, можуть розглядатися як об’єкти, всі вони є потомками типу object. Тому методи об’єкта можуть викликатися для змінних будь-якого типу (або літералів відповідного типу).

Наприклад

using System;

class Demo

{

static void Main()

{

Console.WriteLine(17.ToString());

}

}

В тих випадках, коли потрібно використати вказівник на змінну, яка має тип значення, цю змінну копіюють в об’єкт, тобто присвоюють значення змінній типу object. Така операція називається *упакуванням значення (boxing)*, а зворотна їй операція – *розпакування значення (unboxing).* Наприклад

class Demo

{

static void Main()

{

byte i =17;

object obj =I; // Копіювання в об’єкт

byte j = (byte) obj // Вилучення з об’єкта

}

}

**Структура консольної програми**

Консоллю називають вікно операційної системи, в якій користувачі взаємодіють з операційною системою або консольною програмою шляхом набору текстового вводу з клавіатур і читання текстового виводу з екрана комп’ютера. У Windows консоль є вікном командного рядка і приймає команди операційної системи MS-DOS.

Точкою входу у будь-яку програму на мові С# є метод Main, що знаходиться у одному із класів.

|  |  |
| --- | --- |
| *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**

Введення та виведення даних у консольних програмах може бути здійснено з використанням класу Console.

*Виведення даних*

Для виведення даних можуть бути використані методи Write() та WriteLine(). Метод WriteLine()відрізняється від метода Write() тим, що після виведення інформації курсор переводиться на наступний рядок.

Метод Console.WriteLine(), як і Console.Write() мають декілька варіантів виклику, що відрізняються списком аргументів. Розглянемо деякі з варіантів виклику.

Виведення рядка тексту

Загальний вигляд

Console.WriteLine(<рядок тексту>);

|  |  |
| --- | --- |
| Приклад | На екрані |
| Console.WriteLine(“Hello!”); | Hello! |

Виведення числового значення

Загальний вигляд

Console.WriteLine(<значення>);

|  |  |
| --- | --- |
| Приклад | На екрані |
| int i=25;  Console.WriteLine(i); | 25 |

Виведення значення з текстовим повідомленням

Загальний вигляд

Console.WriteLine(<текст>+<значення>);

|  |  |
| --- | --- |
| Приклад | На екрані |
| 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 |

Виведення із використанням підстановочних знаків. При цьому способі виведення спочатку вказується рядок символів, у якому у фігурних дужках фіксується місце розташування аргументів для виведення (вказуємо номер аргументів, які нумеруються починаючи від 0), а потім, послідовно через кому перераховуємо аргументи для виведення.

|  |  |
| --- | --- |
| Приклад | На екрані |
| 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 |

Форматований вивід значень. Цей спосіб виведення можливий з використанням підстановочних знаків з параметрами. Розглянемо деякі з них:

d – десятковий формат. Дозволяє задати загальну кількість знаків (якщо кількість символів у числі менша, то зліва число доповнюється нулями).

|  |  |
| --- | --- |
| Приклад | На екрані |
| int i=25;  Console.WriteLine(“i={0:d5}”,i); | i=00025 |

f – формат з фіксованою точністю. Дозволяє задати кількість знаків після коми.

|  |  |
| --- | --- |
| Приклад | На екрані |
| double d = 2.853;  Console.WriteLine("d={0:f2} ", d); | d=2.85 |
| double d = 2.857;  Console.WriteLine("d={0:f2} ", d); | d=2.86 |

x – шістнядцятковий формат. Дозволяє виводити значення аргументу як шістнадцяткове число. Якщо параметр “x”, то цифри виводяться у нижньому регістрі, якщо ж параметр “X”, то цифри виводяться у верхньому регістрі.

|  |  |
| --- | --- |
| Приклад | На екрані |
| int i = 255;  Console.WriteLine("i={0:x} ", i); | i=ff |
| int i = 255;  Console.WriteLine("i={0:X} ", i); | i=FF |

c – грошовий формат (додає знак грошової одиниці, у залежності від налаштувань операційної системи).

|  |  |
| --- | --- |
| Приклад | На екрані |
| int i = 255;  Console.WriteLine("i={0:с} ", i); | i=255.00р. |

e – вивід числа в експоненціальній формі.

|  |  |
| --- | --- |
| Приклад | На екрані |
| double d = 362.853;  Console.WriteLine("d={0:e} ", d); | d=3.628530e+002 |

**Введення символьних даних**

Введення даних, як і виведення, можна здійснити використовуючи методи ReadKey,Read та ReadLine класу Console.

**Console.ReadLine()** дозволяє зчитати рядок символів із стандартного потоку вводу.

Приклад.

string s;

s=Console.ReadLine();

**Console.Read()** дозволяє зчитати код символу із стандартного потоку вводу.

|  |  |
| --- | --- |
| 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**

Для введення числових даних можна використати метод ReadLine для введення рядка символів та подальшого переведення рядкового представлення числа у числове значення відповідного типу. Переведення рядкового представлення числа у число можна використовуючи метод Parse відповідного типу

<Цільовий тип>.Parse(<Вираз>);

або методи класу Convert

Convert.To<Цільовий тип>(<Вираз>);

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**

***Вирази***

Виразом називають послідовність операцій, операндів і розділових знаків, що задають деякі обчислення. В залежності від значення, яке одержується в результаті цих обчислень, вираз поділяють на арифметичні та логічні вирази.

***Арифметичні вирази***

Арифметичним виразом називають вираз, в результаті обчислення якого одержуємо числове значення. При цьому можуть використовуватися бінарні та унарні операції арифметичні операції.

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 |

Окрім бінарних операцій в арифметичному виразі можуть бути присутні також унарні операції «+», «–» та операції інкременту «++» і декременту «– –». Унарний мінус використовується для зміни знаку. Операції інкременту і декрменту використовують для збільшення та зменшення значення змінної на одиницю. Ці операції можуть вживатися у префіксній та постфіксній формі.

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

Якщо унарну операцію вжито у префіксній формі у виразі, то вона виконується до використання значення змінної у виразі. Якщо ж унарну операцію вжито у постфіксній формі, то операція виконується після використання значення змінної у виразі.

|  |  |
| --- | --- |
| 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**

У мові C# є можливість здійснювати побітові операції над розрядами аргументів

|  |  |  |
| --- | --- | --- |
| 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**

Якщо при виконанні бінарної операції результат зберігається у змінній, що є першим аргументом, то можна використати так звані операції присвоювання.

|  |  |  |
| --- | --- | --- |
| 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***

**Логічним виразом** називається такий вираз, внаслідок обчислення якого одержується логічне значення типу bool ( або ).

Прикладом логічного виразу є вираз, що містить операції порівняння

|  |  |  |
| --- | --- | --- |
| 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 |

**Пріоритет операції**

Пріоритет Операції

1 () [] . (постфикс)++ (постфикс)-- new sizeof typeof

unchecked

2 ! ~ (имя типа) +(унарный) -(унарный)

++(префикс) --(префикс)

3 \* / %

4 + -

5 << >>

6 < > <= => is

7 == !=

8 &

9 ^

10 |

11 &&

12 ||

13 ?:

14 = += -= \*= /= %= &= |= ^= <<= >>=

**Складений оператор**

*Складений оператор* – це складний оператор, який об’єднує декілька операторів в одну групу.

Форма його запису наступна:

{

оператор 1;

оператор 2;

…………..

оператор n;

}

В даній конструкції “{“ – відкриваюча операторна дужка; “}” – закриваюча операторна дужка. Складений оператор визначається як єдиний оператор. Його можна вставляти в довільне місце програми, де дозволено використання одного простого оператора.

**Умовний оператор**

Дозволяє вибрати оператор, який буде виконуватися в залежності виконання чи невиконання деякої умови. Існують дві форми для даного оператора повна та скорочена.

**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;  } |

Приклад. Знайти максимальне та мінімальне із двох дійсних чисел.

using System.Collections.Generic;

using System.Linq;

using System.Text;

namespace ConsoleApplication1

{

class Program

{

static void Main(string[] args)

{ double b,d;

Console.Write("b=");

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

Console.Write("d=");

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

double max, min;

if (b > d)

{ max = b;

min = d;

}

else

{ max = d;

min = b;

}

Console.WriteLine("max={0} min={1}",max,min);

Console.ReadLine();

}

}

}

Результати роботи програми

b=3

d=8

max=8 min=3

**Умовний оператор (?:)**

Якщо значення виразу може дорівнювати одному із двох значень в залежності від виконання чи невиконання деякої умови, то можна скористатися умовним оператором **(?:)** .

|  |  |
| --- | --- |
| 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

Оператор switch дозволяє передавати керування одному з декількох операторів в залежності від значення виразу, який називають селектором вибору. У якості селектора вибору може бути вираз цілого типу, типу char, переліковного типу або типу string.

|  |  |
| --- | --- |
| **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();  }  }  } |

Якщо для декількох варіантів необхідно виконати одні і ті ж оператори, то ці оператори вказують тільки для одного з варіантів, а для всіх інших не вказуємо ні необхідних операторів, ні операторів break.

**Приклад**. З клавіатури вводиться оцінка у національній шкалі, необхідно вивести повідомлення про те, чи зараховано студенту залік.

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

namespace ConsoleApplication1

{

class Program

{

static void Main(string[] args)

{

int mark;

Console.Write("Mark = ");

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

switch (mark)

{

**case 1:**

**case 2:**Console.WriteLine("Незараховано.");

break;

**case 3:**

**case 4:**

**case 5:**Console.WriteLine("Зараховано.");

break;

default: Console.WriteLine("Неправильна оцінка.");

break;

}

Console.ReadKey();

}

}

}

**Приклад**. З клавіатури вводиться колір помідора, на екран необхідно вивести у якому він стані (росте, дозріває чи можна зірвати). При розв’язанні цього завдання використаємо оператор switch, у якому селектор і константи вибору є величинами типу string.

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

namespace ConsoleApplication3

{

class Program

{

static void Main(string[] args)

{

**string color;**

**Console.Write("Color = ");**

**color = Console.ReadLine();**

**switch (color)**

**{**

**case "Green" : Console.WriteLine("Ще росте");**

**break;**

**case "Yellow": Console.WriteLine("Дозріває.");**

**break;**

**case "Red": Console.WriteLine("Можна зірвати");**

**break;**

**default: Console.WriteLine("Помідор-мутант.");**

**break;**

**}**

Console.ReadKey();

}

}

}

Приклад. З клавіатури вводиться буква у нижньому регістрі, з’ясувати, чи є буква голосною. При розв’язанні цього завдання використаємо оператор switch, у якому селектор вибору та константи вибору типу char.

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

namespace ConsoleApplication3

{

class Program

{

static void Main(string[] args)

{

char c;

Console.Write("Input letter= ");

c = (char)Console.Read();

**switch (c)**

**{**

**case 'а' :**

**case 'о' :**

**case 'у' :**

**case 'и' :**

**case 'і' :**

**case 'е' : Console.WriteLine("Голосна");**

**break;**

**default: Console.WriteLine("Приголосна");**

**break;**

**}**

Console.ReadKey();

}

}

}

**Оператори циклу**

**Оператор циклу while**

Оператор while циклічно виконує свою інструкцію до тих пір, поки умова виконується (логічний вираз приймає значення true).

|  |  |  |
| --- | --- | --- |
| **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); |

**Оператор циклу for**

Загальна форма оператора наступна

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

<оператор>;

Оператор for виконує наступні дії:

1. Обчислює вираз ініціалізації. У цій частині допустима ініціалізація декількох лічильників циклу.
2. Перевіряється логічна умова. Якщо умова невірна то робота циклу завершується і передається управління наступному оператору.
3. Якщо умова істинна, виконується інструкція даного оператора.
4. Виконується приріст одного або декількох лічильників цикл (або виконується довільна інша операція.
5. Здійснюється перехід до кроку 2.

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**

Приклад. Знайти суму перших n натуральних чисел.

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

namespace ConsoleApplication3

{

class Program

{

static void Main(string[] args)

{

int n,sum=0;

Console.WriteLine("n=");

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

**for (int i = 1; i <= n; i++)**

**sum = sum + i;**

Console.ReadKey();

}

}

}

Проілюструємо випадок, коли частина ініціалізації та ітераційна частина мість декілька виразів, розділених комою.

**Приклад**. Обчислити значення виразу



using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

namespace ConsoleApplication3

{

class Program

{

static void Main(string[] args)

{

int n;

Console.WriteLine("n=");

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

double d;

int c;

double sum=0;

for ( **c = 1, d=0.3** ; c <= n ; **c++, d+=0.1**)

sum +=c/d;

Console.WriteLine("sum={0:f3}",sum);

Console.ReadKey();

}

}

}

Кожна з частин оператора циклу може бути відсутньою, але розділові знаки “ ; ” є обов’язковими. Так нескінчений цикл може бути задано так

for ( ; ; )

<оператор>;

**Оператор foreach**

Оператор foreach призначений для перебору значень колекції, виконуючи інструкцію для кожного її елемента.

Загальна форма цього оператора

foreach (<змінна типу елемента колекції> in <колекція>)

<оператор>;

**Оператори break та continue**

Оператор continue може бути використаний у будь-якому із циклів у випадку, коли немає потреби виконувати всі оператори у тілі циклу у поточній ітерації, а необхідно одразу перейти до наступної ітерації.

Приклад. Знайти добуток непарних натуральних чисел, що менші за K.

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

namespace ConsoleApplication3

{

class Program

{

static void Main(string[] args)

{

int K;

Console.Write("K=");

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

int mult=1;

**for** (int i = 1;i<=K ; i++)

{

if ((i % 2) == 0) continue;

mult \*= i;

}

Console.ReadKey();

}

}

}

Оператор break у циклах використовують для негайного завершення самого внутрішнього циклу, у тілі якого він знаходиться.

Приклад. Знайти найменше значення факторіалу натурального числа, що перевищує число K.

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

namespace ConsoleApplication3

{

class Program

{

static void Main(string[] args)

{

int K;

Console.Write("K=");

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

int fakt=1;

for (int i = 1; ; i++)

{

fakt \*= i;

if (fakt > K) **break;**

}

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

Console.ReadKey();

}

}

}

**Приклад**. Вивести на екран усі натуральні двоцифрові числа, у яких друга цифра не перевищує першу.

При розв’язанні цього завдання використаємо оператор break , який здійснює вихід тільки із вкладеного циклу (робота зовнішнього циклу продовжується).

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

namespace ConsoleApplication3

{

class Program

{

static void Main(string[] args)

{

for (int i = 1; i <= 9; i++)

{

for (int j = 1; j <= 9; j++)

{

if (j > i) **break;**

Console.WriteLine("{0}{1}",i,j);

}

}

Console.ReadKey();

}

}

}

Результат роботи програми

11

21

22

31

32

33

..

99

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

При розв’язанні складних задач застосовується структурний підхід до розробки алгоритмів за методикою «зверху-вниз». Цей підхід полягає у розбитті складної задачі на більш прості підзадачі та передбачає розв’язання цих підзадач з використанням функцій.

При розробці консольних програм у С# метод Main є статичним, тому будь-які функції, що можуть бути використані у консольних программах, також повинні бути статичними.

|  |  |
| --- | --- |
| **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();  }  }  } |

Якщо метод не повертає якого-небудь значення то його типом є void.

**Формальні параметри методів**

Формальні параметри в списку формальних параметрів розділяються комою. В якості модифікаторів аргументів можуть використовуватися наступні ключові слова

* ref
* out
* params

Описання метода створює окрему область дії імен для його аргументів і локальних змінних, які визначені в блоці, який утворює тіло методу.

При виклику методу створюється окрема копія аргументів і локальних змінних даного методу, після чого відбувається заповнення копій аргументів значеннями, що передаються в списку аргументів. В межах блоку зверненні до аргументів методу здійснюється по іменам, визначеним в списку аргументів.

Є чотири типи формальних параметрів:

* що передаються за значенням (немає ніякого модифікатора)
* передаються за вказівником (вказується модифікатор ref)
* формальні параметри, які використовуються для передачі вихідних значень (вказується модифікатор out)
* формальні параметри змінної довжини (вказується модифікатор params)

Оскільки модифікатор формального параметра є частиною списку формальних параметрів, то в класі можуть бути методи, які відрізняються модифікаторами формальних параметрів.

При передачі аргументу *за значенням* в копію даного аргументу, що створена при виклику методу, просто записується значення, яке передається в даному аргументі. Копія аргументу розглядається як локальна змінна, яка має тип відповідного формального параметра. Зміна значення цієї змінної ніяк не вплине на значення фактичного параметра.

**Приклад**. Створити функцію, що повертає середнє значення трьох дійсних чисел.

class Program

{

**static double Average(double c1, double c2, double c3)**

**{**

**return (c1 + c2 + c3) / 3;**

**}**

static void Main(string[] args)

{ double n, m, k;

Console.WriteLine("n=");

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

Console.WriteLine("m=");

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

Console.WriteLine("k=");

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

double ser= **Average(n, m, k);**

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

Console.ReadKey();

}

}

*При передачі аргументу за вказівником* в копію даного аргуенту, який має ім’я відповідного формального параметра записується адреса на даний аргумент. Тому, всі зміни стосовно даного формального параметра стосуються також і відповідного фактичного параметра. При описанні відповідного формального параметра та при записі відповідного фактичного параметра використовується модифікатор ref.

Такий спосіб пеередачі параметрів використовується, напиклад, тоді, коли в функції необхідно змінити фактичні параметри.

**Приклад.** Створити функцію, яка більше значення зменшує на 2, а менше збільшує на 3.

class Program

{

**static void Fun1(ref int a, ref int b)**

**{**

**if (a > b)**

**{**

**a -= 2;**

**b += 3;**

**}**

**else**

**{**

**b -= 2;**

**a += 3;**

**}**

**}**

static void Main(string[] args)

{ int n,m;

Console.WriteLine("n=");

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

Console.WriteLine("m=");

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

**Fun1(ref n,ref m);**

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

Console.ReadKey();

}

}

*Аргумент, в який буде записане вихідне значення методу* в основному аналогічний аргументу, який передається за вказівником. Основною відмінністю між ними є те, що для даного аргументу не є обов’язковою ініціалізація фактичного параметра до передачі цього аргументу методу. Але в самому методі фактичний параметр вважається неініціалізованим і має обов’язково бути проініціалізованим до завершення методу. При описанні формального параметра та відповідного фактичного викорстовується модифікатор out.

Такий метод передачі параметрів використовується, наприклад, тоді, коли аргумент є результатом роботи функції.

**Приклад.** Створити функцію, яка повертає максимальне та мінімальне значення з двох дійсних чисел.

class Program

{

static void MinMax(int c1, int c2, out int max, out int min)

{

if (c1 > c2)

{

max = c1;

min = c2;

}

else

{

max = c2;

min = c1;

}

}

static void Main(string[] args)

{ int n,m;

Console.WriteLine("n=");

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

Console.WriteLine("m=");

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

int max, min;

**MinMax(n, m, out max, out min);**

Console.WriteLine("max= {0} min={1}",max,min);

Console.ReadKey();

}

}

Отже, зміна значення фактичного параметра в результаті зміни значення формального праметра відбувається тільки у випадку коли передазча здійснюється за вказівником, або як вихідного параметра.

|  |  |  |
| --- | --- | --- |
| **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:** |

Якщо одним із аргументів метода є *одновимірний масив довільного розміру*, то цей аргумент передається останнім і має модифікатор params. Оскільки тип масив не визначено, то це може бути і масив вказівників.

При виклику методів, які мають аргумент з модифікатором params, діють наступні правила:

* при передачі в даному аргументі масивів, елементи якого мають тип, який має явне перетворення до типу елементів вказаного масиву, елементи вхідного масиву передаються по значенням.
* Якщо замість даного аргументу при виклику методу стоїть вираз, тип якого має явне перетворення до типу елемента масиву, то цей аргумент і всі наступні аргументи, тип яких має явне перетворення до типу елементу масива, вважаються елементами вхідного масиву (масиву, що передається). Ці елементи також передаються по значенню. При неможливості перетворення типів генерується помилка. При відсутності аргументу масив має нульову довжину. Для визначення довжини масиву використовується length

Такий спосіб передачі параметрів використовується тоді, коли кількість вхідних аргументів наперед невідома.

class Program

{

static void Max(out double max, params double[] a)

{

if (a.Length > 0)

{

max = a[0];

for (int i = 1; i < a.Length; i++)

{

if (max < a[i])

max = a[i];

}

}

else

{

max = 0;

}

}

static void Main(string[] args)

{

double max;

Max(out max, 0, 1, 2, 3, 4, 5);

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

Max(out max, 10, 3, 41);

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

Console.ReadKey();

}

}

32ультат роботи програмиродовжуєтьсяаємо оператор яких друга цифра не перевищує першу. дразу перейти до наступної ітерації.

**Масиви**

# *Поняття масиву*

Як відомо, будь-яку задачу людина, як і машина, розв’язує виконуючи кроки деякого алгоритму. Алгоритмом називають наперед задану скінчену послідов­ність чітких правил або команд (кроків), виконання яких призводить до одержання розв'язку задачі. Програма ж є одним із способів запису алгоритму.

У своїй роботі програміст виконує маніпуляції з величинами. З точки зору програмування, величини – це дані, що обробляються програмами. Носіями даних у програмах є константи, змінні(значення яких зберігається в оперативній пам'яті) та файли(які зберігаються на носіях інформації).

Константи– це величини, значення яких у процесі виконання програми не змінюється. Змінні– це величини, значення яких у процесі виконання програми можуть змінюватися. Імена констант та змінних, як і інших програм­них об'єктів, записують у формі ідентифікаторів (послідовності символів латинського алфавіти, цифр та символу нижнього підкреслювання “\_”, яка не може починатися з цифри). Кожна змінна і константа належать до визначеного типу. Тип даних **–** це сукупність властивостей певного набору даних, від яких залежать діапазон значень, якого можуть набувати ці дані, а також сукупність операцій, які можна виконувати над цими даними.

Усі типи даних розділяють на дві групи: скалярні (прості) та структуровані (складені). До скалярних відносять типи даних (цілі, дійсні, символьний, рядковий, логічний), які дозволяють опрацьовувати окремі числа, символи, рядки і т.д. Але у більшості задач доводиться розглядати сукупності величин одного типу (вектори, матриці, таблиці, числові послідовності і т.д.). Звичайно, можна було би для кожної величини ввести змінну. Але уявіть собі, що буде у випадку, коли кількість величин рівна 100. Тому для збереження сукупності елементів одного типу у мовах програмування визначається так званий структурований тип даних – *масив*.

Можна дати кілька означень масиву.

*Масив –*  це структура даних, що являє собою однорідну (за типом), фіксовану (за розміром і конфігурацією) сукупність нумерованих елементів.

*Масивом* називається скінчена послідовність елементів одного типу, які різняться порядковим номером.

*Масив* – формальне об’єднання декількох однотипних об’єктів (чисел, символів, рядків і т.д.), які розглядаються як єдине ціле.

Масиви відносяться до структур з прямим або довільнимдоступом. Щоб визначити окремий елемент масиву потрібно вказати його *індекс*. В одновимірному масиві (коли для ідентифікації елемента необхідно вказати значення одного індекса) індекс – це порядковий номер елементу масиву (нумерація починається з нуля). Якщо для ідентифікації елемента необхідно вказати декілька індексів, то такий масив називають багатовимірним. Кількість індексів при цьому називають *розмірністю,* кількість дозволених значень кожного індексу –  *діапазоном*, а сукупність розмірності та діапазону – *формою масиву.*

Класифікація масивів може здійснюватися за багатьма ознаками:

1. можливістю зміни значень елементів масиву (масиви-константи та масиви-змінні);
2. кількістю індексів, які необхідно вказати для того, щоб ідентифікувати елемент у масиві (масиви одновимірні, двовимірні, тривимірні і т.д.);

Базовим типом для всіх масивів є тип System.Array.

**Одновимірні масиви**

*Опис одновимірного масиву*

Одновимірний масив часто асоціюють із вектором. Загальний вигляд опису типу 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*

Для ідентифікації елемента масиву необхідно вказати ім’я масиву та індекс елемента, який записується у квадратних дужках після імені масиву

<ім’я масиву> [ <індекс> ]

**Приклад.**

b [ 0 ] = 2;

d [ 4 ] = 2.4;

r [ 1 ] = ‘М’ ;

Наведемо декілька важливих зауважень:

1. елементи масиву можуть бути довільного, але тільки одного типу;
2. значення індексу не повинно виходити за межі описаного діапазону;
3. оскільки процедури введення/виведення розраховані на введення та виведення значень простих типів, то масив потрібно вводити/виводити поелементно, тобто кожен елемент окремо.

*Початкова ініціалізація. Ініціалізатори масивів*

При описанні масивів можуть використовуватися інціалізатори масивів (конструкції, що дозволяють здійснювати початкову ініціалізацію значень елементів масиву). При цьому розмірність масиву та кількість вказуваних констнт повинні співпадати.

**Приклад.**

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

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

При ініціалізації перед ініціалізатором може бути опущений тип масиву, оскільки він повністю ввизначається типом поля або змінної. Так, наприклад, ініціалізатор

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

є скороченою формою запису ініціалізатора

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

і є еквівалентним наступному фрагменту програми

byte [ ] b = new byte [ 4 ];

b[0]=11;

b[1]=25;

b[2]=23

b[3]=14;

*Представлення одновимірного масиву в пам’яті ЕОМ*

Для збереження масивів в пам’яті ЕОМ використовується послідовне представлення. Тобто для збереження масиву наперед виділяється ділянка оперативної пам’яті величиною

<**кількість елементів**> \* <**кількість байтів для збереження одного елемента**>

і елементи розміщуються послідовно один за одним.

**Приклад.**

byte [ ] b = new byte [ 5 ];

Змінна b в пам’яті ЕОМ буде розміщена наступним чином

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  | Комірки |
| … | b[0] | b[1] | b[2] | b[3] | b[4] | … | **Імена комірок** |
|  | Область масиву b | | | | |  |  |

*Введення елементів масиву*

Масив відноситься до структурованих типів даних, тому введення і виведення масивів повинно здійснюватися поелементно. Як правило, введення елементів масиву здійснюється у циклі. Загальний вигляд циклу введення елементів масиву у консольних програмах може бути таким:

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*

Як і введення, виведення елементів масиву також здійснюється поелементно. Загальний вигляд циклу виведення елементів масиву може виглядати наступним чином:

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]);**

**}**

*Приклади задач*

Наведемо декілька прикладів задач, розв’язаних з використанням масивів.

**Приклад.** Дано , знайти .

***Розв’язання***

Для збереження векторів використаємо однойменні одновимірні масиви , і . Кількість елементів  будемо вводити з клавіатури.

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

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

{

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

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

}

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

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

{

c[i] = a[i] + b[i];

}

for (int i = 0; i < n; i++) //Виведення елементів масиву с

{

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

}

Console.ReadLine();

}

Результати роботи програми.

n=3

a[0]=1

a[1]=2

a[2]=3

b[0]=4

b[1]=5

b[2]=6

c[0]=5

c[1]=7

c[2]=9

**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();

}

*Функції для роботи з одновимірними масивами*

Як було зазначено раніше, базовим типом для всіх масивів є тип System.Array, який містить ряд функцій, які дозволяють здійснювати деякі маніпуляції над масивами.

Наприклад,

<ім’я масиву>.Sum() – дозволяє знайти суму елементв масиву,

<ім’я масиву>.Max() – дозволяє знайти максимальний елмент,

<ім’я масиву>.Min() – дозволяє знайти мінімальний елемент,

<ім’я масиву>.Average() – дозволяє знайти середнє значення.

*Копіювання масивів*

Оскільки масив є типом посилання, то для коректного копіювання масивів використовують спеціальний метод CopyTo

<масив 1>.CopyTo (<масив 2>, <поч. індекс>) – дозволяє скопіювати елементи <масиву 1> у <масив 2> починаючи з вказаного початкового індекса.

Приклад.

a.CopyTo(b,1);

В результаті елементи масиву a буде скопійовано у масив b починаючи з 1-го елемента (тобто буде виконано послідовність операцій присвоєння b[1]=a[0], b[2]=a[1], … ).

# Багатовимірні масиви

Багатовимірними масивами є масиви, в яких для ідентифікації елемента необхідно вказати більше одного індексу. Найбільш використовуваними є двовимірні масиви, тому їх буде детально розглянуто. З багатовимірними масивами маніпуляції виконуються аналогічно.

## Прямокутні масиви

Розглянемо прямокутні двовимірні масиви (у кожному рядку однакова кількість елементів). Оскільки двовимірні масиви часто асоціюються з двовимірними таблицями і матрицями, то прийнято вважати, що перший індекс позначає номер рядка, а другий – номер стовпця.

*Опис двовимірного масиву*

<тип елементів> [ , ] <ім’я масиву>;

**Приклад.**

int [ , ] A;

double [ , ] d;

*Виділення пам’яті*

Як і при роботі з одновимірними масивами, виділення пам’яті здійснюється з використанням оператора new

<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];

Змінну A можна відобразити у вигляді матриці



Як і при роботі з одновимірними масивами, кількість рядків і кількість стовпців можуть бути змінними величинами

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*

Як і при роботі з одновимірними масивами при використанні двовимірних маисів початкова ініціалізація може бути виконана з використанням ініціалізаторів (у фігурних дужках поступово вказують елементи кожного з рядків, як одновимірних масивів).

Прилад.

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

*Введення*

Введення двовимірних масивів здійснюється поелементно, як правило, з використанням двох циклів (вкладених). Наведемо приклад такого 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*

Виведення елементів також здійснюється поелементно. Наведемо приклад такого виведення

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]);

}

}

*Приклади задач*

**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***

Раніше було зазначено, що елементи масиву можуть буди довільного типу, в тому числі і масивами. У випадку, якщо маємо одновимірний масив, елементами якого є також одновимірні масиви то маємо двовимірний масив. При цьому кількість елементів у кожному з масивів може бути різною.

*Опис*

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

**Приклад.**

**double [ ] [ ] d;**

*Виділення пам’яті*

Виділення пам’яті для таких масивів відбувається у два етапи: на першому етапі виділяється пам’ять для масиву посилань на рядки, на другому – виділяється пам’ять під безпосередньо елементи рядків)

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;

Зауважимо, що використання ініціалізаторів можливе тільки для прямокутних масивів.

**Приклад.**  Знайти найстаршого учня у кожному з класів (кількість учнів у кожному із класів може бути різною.

Розв’язання

Для збереження віку учнів використаємо двовимірний масив  з різною кількістю елементів у рядку.

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

namespace ConsoleApplication1

{

class Program

{

static void Main(string[] args)

{

int[][] a;

int n;

Console.Write("Number of classes = ");

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

a = new int[n][];

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

{

int m;

Console.Write("Number of pupils in class {0} = ", i);

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

a[i] = new int[m];

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

{

Console.Write("Age: class {0} pupil {1} =", i,j);

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

}

}

Console.WriteLine("Age of seniors: ");

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

{

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

{

Console.Write("-- "+a[i][j]);

}

Console.WriteLine("--== max A[{0}]={1} ",i,a[i].Max());

}

Console.ReadLine();

}

}

}

**Простори імен**

Базовим елементом програми на мові C# є простір імен, який може містити описання типу або деякий вкладений (внутрішній) простір імен.

Описання визначає ім’я в тому просторі імен, в якому воно було здійснено. В одному й тому ж просторі імен не може бути описано два однаковим імені (виключення є перегрузка методів, індексаторів, операцій).

При роботі з просторами імен визначені наступні правила:

* *Глобальний* простір має включати в себе всі простори імен, визначені у всіх файлах програми і не включені в жоден інший простір.
* У всіх файлах програми описання, які здійснені в просторах імен, що мають одне й те ж саме ім’я відносяться до одного й того ж простору імен.
* Кожен опис нового класу, структури чи інтерфейсу представляє собою новий простір імен.
* Описання кожного нового переліковного типу створює новий простір імен.
* Кожен блок, включаючи блок оператора вибору, створює окремий простір імен для описання в них локальних змінних. Якщо цей блок є тілом методу, в цей простір імен включаються імена змінних, які є формальними параметрами.
* Кожен блок, включаючи блок оператора вибору створює окремий простір імен для міток. Оскільки простір імен вкладеного (внутрішнього) блоку включає в себе простір імен зовнішнього блоку, то в вкладеному блоці не може бути визначена мітка з тим же іменем, що й мітка з зовнішнього блоку.

Зазвичай порядок описання в просторі імен не має значення. Це відноситься перш за все до просторів імен типів, констант, методів, властивостей, індексаторів, операцій, конструкторів та деструкторів.

Порядок описання важливий тільки наступних випадках:

* порядок описання полів і локальних змінних визначає порядок їх ініціалізації;
* локальні змінні мають бути визначені до свого першого використання;
* порядок констант в описанні переліковного типу визначає відповідність їх значенням, якщо ці значення не задані явним чином.

Для вказування простору імен використовують ключове слово namespace.

namespace <простір імен>

{

….

}

На відміну від С++ в C# описання змінної в вкладеному (внутрішньому) блоці не перевизначає змінну, яка описана в зовнішньому блоці. Це правило діє навіть тоді, коли змінна в зовнішньому блоці описана після описання внутрішнього блоку. Наприклад в наступному фрагменті помилка

bool b = true;

if (b)

{

int i = 1;

}

int i = 2; //помилка

**Члени просторів імен та типів**

Простори імен та типи містять члени. Для доступу до них використовуються прості та повні імена. Прості імена використовуються в тому випадку, коли об’єкт та тип, якому належить той чи інший член, може бути визначений із контексту. Якщо ж контекст не дозволяє однозначно ідентифікувати член, використовується повне ім’я, яке складається з ідентифікатора об’єкта або класу, якому належить цей член, та простого його імені, розділені крапкою.

Члени типу можуть бути визначені в ньому або можуть бути наслідувані від свого базового класу. З базового класу наслідуються всі члени класу, крім його конструкторів та деструкторів, незалежно від унаслідуваних для них режимів доступу.

**Встановлення режимів доступу**

Для членів може бути встановлено один з наступних режимі доступу:

* public (відкритий) доступ до даного члену необмежений нічим;
* protected (захищений) доступ до члена може бути здійснено тільки з класу в якому він описаний та з типів, які є потомками для даного класу;
* internal (локальний) доступ до члену може бути здійснено тільки в проекті в якому він описаний;
* protected internal (локальний захищений) доступ до члену може бути здійснено тільки в проекті в якому він описаний, і тільки з класу в якому він описаний та з типів, які є потомками для даного класу.
* protected (закритий) доступ до членів може бути здійснено тільки з класу в якому він описаний.

**Область видимості**

Під областю видимості розуміють фрагмент програми, в якій можна звертатися до імені по його простому імені. Області видимості можуть бути вкладеними, і в внутрішній області видимості ім’я, яке використовується в зовнішній області видимості, може бути перевизначеним. При цьому ім’я, яке належить зовнішній області видимості, перевизначається (скрывается) у внутрішній області видимості. В цьому випадку доступ до перевизначеного імені можливий з використанням його повного імені.

При визначенні області видимості діють наступні правила:

* областю видимості члена одного простору імен, не вкладеного в інше, є програмний текст одиниці компіляції;
* область видимості члена одного простору імен, владного інший є тіла всіх просторів імен, повне ім’я яких співпадає з повним іменем зовнішнього простору імен або починаються з цього повного імені;
* область видимості імені, введеного або імпортованого директивою using , є описання члені простору імен в одинці компіляції та тілі просторі імен, які містять директиву using (директива using надає доступ до просторів імен або імен типів тільки в тих одиницях компіляції або в тілах просторів імен, в яких вона використовується, і дія цієї директиви не поширюється на зовнішні одиниці компіляції та простори імен);
* областю видимості члена класу є тіло класу, в якому він описаний. Крім того, область видимості члена класу поширюється на тіла тих класів, які є потомками цих класів (якщо не перевизначені);
* областю видимості члена структури є тіло структури, в якій він визначений;
* областю видимості члена переліковного типу є тіло переліковного типу в якому він визначений (описаний);
* областю видимості аргументу конструктора є інціалізатор та блок даного конструктора;
* областю видимості аргументу метода є блок даного метода;
* областю видимості аргументу індексатора є процедури доступу даного індексатора;
* областю видимості операнду перегруженої операції є блок даної операції;
* областю видимості локальної змінної, визначеної в блоці, є блок в якому ця змінна описана;
* областю видимості локальної змінної, визначеної в ініціалі заторі циклу for, є всі елементи даного оператора;
* областю видимості мітки є блок, в якому вона визначена.

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

**1.** **Класи та об’єкти**

**1.1. Поняття класу та об’єкта**

У більшості задач програміст має справу із реальними об’єктами деякої предметної області. Прикладами реальних об’єктів є автомобіль, літак, завод, людина, матриця, вектор тощо. Як правило, реальний об’єкт характеризується як певними даними так і набором функціональних можливостей. Іншими словами, об’єкти можуть характеризуватися даними та діями, що можуть виконувати самі об’єкти або які можна виконувати над ними.

**Приклад**. Прямокутник характеризується наступними даними:

1. довжина сторони *a*;
2. довжина сторони *b*.

Також для прямокутника можна визначити наступні дії:

1. знаходження периметру;
2. знаходження площі;
3. визначення рівності з іншим прямокутником, який задається за допомогою своїх сторін;
4. масштабування (збільшення або зменшення всіх сторін прямокутника у певну кількість разів).

Отже, необхідно описати наступну логічну структуру

Прямокутник

Сторона *a*

*Дані*

Сторона *b*

Дія *знаходження периметру*

Дія *знаходження площі*

*(з іншим прямокутником)*

*Функціональні можливості*

Дія *порівняння*

Дія *масштабування*

Для повноцінного опису об’єктів реальної дійсності використовується тип даних, який має назву *клас* і який дозволяє описувати як властивості об’єкту (за допомогою змінних), так і його поведінку або ж функціональні можливості (за допомогою функцій). Слід зазначити, що при визначенні класу описуються властивості й методи, які характерні для всіх об’єктів відповідного класу (групи) реальних об’єктів. Як довільний тип даних, клас визначає певну множину елементів або екземплярів цього типу. Окремо взятий екземпляр деякого класу називають *об’єктом*. Прийнято вважати, що *клас* – це шаблон, на основі якого може бути створено конкретний програмний *об’єкт*, що моделює реальний об’єкт певної предметної області. З точки зору мови програмування, *клас* – це тип даних, а *об’єкт* – це змінна типу клас. *Клас* і *об’єкт* є фундаментальними поняттями технології об’єктно-орієнтованого програмування. Суть цієї технології ґрунтується на моделюванні досліджуваних об’єктів предметної області з використанням класів.

**1.2. Опис класів**

У найпростішому випадку загальна схема опису класу наступна:

**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>

}

**}**

При цьому дані, що описано в класі, називають *полями,* а функції – *методами*. Разом поля і методи називають *членами класу*. Для створення та знищення об’єктів у описі класу можуть бути спеціальні методи, які називаються, відповідно, *конструктором* та *деструктором* (імена цих методів співпадають з іменем класу, у якому вони описані).

При описі членів класу можна вказувати *специфікатори доступу*, які визначають де вказані члени є доступними (де до них можна звернутись):

* **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.

За замовчуванням, якщо не вказано специфікатор доступу, вважається що член класу є закритим.

**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 ()  {  }  } |

Поля у класі можуть бути як константами так і змінними. При цьому опис полів у класі здійснюється як і опис звичайних змінних або констант. Реалізація методів у мові C# здійснюється всередині класу. Опис методів здійснюється так само як і опис функцій.

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

Як було зазначено раніше, для створення об’єктів використовуються спеціальні методи, які називають *конструкторами.* Ім’я конструктора співпадає з іменем класу. Тип для методу-конструктора не вказують. Конструктори можуть мати параметри, які дозволяють здійснити *ініціалізацію об’єкта*, тобто надати його полям початкових значень. У класі може бути визначено декілька конструкторів, які повинні відрізнятися списком формальних параметрів. В залежності від кількості та типу параметрів, чи їх відсутності, серед конструкторів можна виділити декілька типів.

**1.3.1. Конструктор без параметрів**. Якщо в класі явно не визначено жодного конструктора (і тільки в цьому випадку!), то він надається за замовчуванням (*конструктор за замовчуванням*). За визначенням такий конструктор не має формальних параметрів. Такий конструктор тільки виділить пам’ять для полів об’єкта, але поля не буде проініціалізовано. Тому бажано розробляти власний 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**. Як було зазначено раніше, опис конструктора може містити параметри, які, як правило, використовують для ініціалізації полів об’єктів.

**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;

}

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

}

Якщо в класі існує декілька конструкторів, то можна один конструктор викликати з іншого. Це робиться за допомогою ключового слова this та вказанням відповідних параметрів конструктора, що викликається.

**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**. У деяких випадках зручно мати у класі конструктор, який би давав можливість ініціалізувати поля об’єкта значеннями полів іншого об’єкта. Тоді у класі визначають конструктор з одним параметром, тип якого співпадає із типом класу. Такий конструктор називають *конструктором 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**. Наявність конструкторів у класі дозволяє створювати та ініціалізувати об’єкти у програмі. Об’єкт створюється за допомогою оператора new.

Загальне правило використання оператора new:

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

Приклади:

Rectangle r1 = new Rectangle(); // a=1, b=1

Rectangle r2 = new Rectangle(3); // a=3, b=1

Rectangle r3 = new Rectangle(4,5); // a=4, b=5

Rectangle r4 = new Rectangle(r2); // a=3, b=1

Оскільки клас відноситься до типів посилань, то у випадку, коли у програмі проводиться присвоювання

|  |
| --- |
| Rectangle r1 = new Rectangle();  Rectangle r2 = r1; |

то змінні r1 і r2 будуть посилатися на один і той самий об’єкт.

Щоб полегшити процес створення об’єктів та ініціалізації, у мові С# пропонується *синтаксис ініціалізії об’єктів*. За допомогою цього механізму можна створити новий об’єкт та присвоїти значення відкритим полям і/або відкритим властивостям у декількох рядках коду.

Приклад:

|  |
| --- |
| Rectangle r = new Rectangle { a=1, b=3 }; |

У наведеному прикладі спочатку *неявно* викликається конструктор без параметрів (або ж конструктор за замовчуванням, якщо в класі не визначено жодного конструктора), після чого здійснюється ініціалізація значень зазначених полів чи властивостей. Синтаксис ініціалізації можна застосовувати у поєднанні із будь-яким конструктором, тобто при бажані конструктор може (або ж має) бути викликаний і *явно*.

Приклади:

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

**1.4. Доступ до членів класу**

Після того як об’єкт створено є можливість здійснювати доступ до полів і методів об’єкта за допомогою *оператора "крапка"*.

| **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()**); |

Всередині методів класу можна звертатись до інших членів цього ж класу за їх іменами. Якщо ж імена формальних параметрів методу співпадають з іменами полів, описаних у класі, то доступ до цих полів можна здійснити з використанням покажчика на об’єкт класу this.

**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**

Як було відмічено вище перед тим як звертатися до поля чи методу об’єкта цей об’єкт має бути створеним. Але у С# є можливість описувати поля і методи, які можна використати у програмі без створення об’єктів класу, тобто як поля і методи класу. Такі члени класу називають *статичними* і описують з використанням ключового слова static.

*Статичні поля* завжди існують тільки в одному екземплярі (є спільними для усіх об’єктів). Всередині методів класу, де описано статичне поле, до нього можна звертатися як до звичайного (нестатичного) поля. За межами класу до відкритих статичних полів звертаються через ім’я класу із використанням оператора "крапка":

<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 |

Для ініціалізації статичних полів класу у мові C# використовують *статичний конструктор*. Статичний конструктор не може мати параметрів. Для статичного конструктора не дозволяється вказувати специфікатор доступу, тобто він є неявно закритим. Статичний конструктор викликається один раз при створенні першого об’єкта класу в програмі.

**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* відрізняються від нестатичних методів тим, що їх можна викликати без використання об’єкта. Оскільки у статичні методи покажчик this не передається, то всередині статичних методів можна звертатись тільки до статичних членів класу. Всередині методів класу, де описано статичний метод, до нього можна звертатися як до звичайного (нестатичного) метода. За межами класу до статичних методів звертаються через ім’я класу із використанням оператора "крапка":

<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**

Як було зазначено раніше, при створенні об’єктів деякого класу явно чи неявно (автоматично) викликається конструктор. У конструкторі, як правило, здійснюють ініціалізацію полів об’єкта, динамічно виділять необхідний об’єм пам’яті, підключають деякі додаткові ресурси та інші дії. Аналогічно, при знищенні об’єктів автоматично викликається метод, який називають *деструктором*. Деструктор викликається безпосередньо перед знищенням об’єкта, тому усередині деструктора виконують звільнення усіх виділених для цього об’єкта ресурсів: звільнення динамічно виділеної пам’яті, закритті файлів та ін. Якщо додаткових ресурсів при створенні об’єкта не виділяється, то власний деструктор описувати не обов’язково. Зазначимо, що деструктори викликають у порядку, зворотному до порядку виклику конструкторів.

Загальний вигляд деструктора:

~<Ім’я класу>()

{

<тіло деструктора>

}

Мова С# володіє механізмом звільнення ресурсів пам’яті, що називається «*збирач сміття*». Якщо до об’єкта не відбуваються звернення (у програмі не залишилося посилань на об’єкт), цей механізм звільняє пам’ять автоматично. Збирач сміття викликається періодично в процесі роботи програми. Деструктор викликається збирачем сміття перед знищенням об’єкта. Деструктор у класі може бути тільки один. Його ім’я співпадає з іменем класу та починається із символу ‘~’. У деструктора немає значення, що повертається, і немає параметрів. Він не має специфікаторів.

**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--;**

}

. . . . . . .

}

**2.** **Перевантаження операторів**

При описі класів, що характеризують поведінку математичних об’єктів, зручно було б використовувати традиційні математичні знаки операцій для виконання відповідних дій. Наприклад, при додаванні двох комплексних чисел природніше було б використовувати оператор «+», а не викликати, припустимо, метод Sum(). Для таких ситуацій у мові С# є зручний засіб, що називається *перевантаженням операторів*.

**2.1. Унарні оператори**

У мові C# у класах можна перевантажувати наступні унарні оператори:

+ - ! ~ ++ -- true false

Загальне правило перевантаження унарного оператора:

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

{

<тіло оператора>

}

Аргумент, який передається в оператор повинен мати тип класу, для якого цей оператор перевантажується.

Унарний оператор повинен повертати:

* для операторів +,-,!,~ величину будь-якого типу;
* для операторів ++, -- величину типу класу, для якого цей оператор перевантажується;
* для операторів true, false величину типу bool.

Унарний оператор не повинен змінювати значення аргументу. Унарний оператор, який повертає величину типу класу, для якого цей оператор перевантажується, повинен створювати новий об’єкт цього класу, виконати з ним необхідні дії і повернути його в якості результату.

Унарні оператори true, false повинні перевантажуватись разом.

| **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**

У мові C# у класах можна перевантажувати наступні бінарні оператори:

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

Загальне правило перевантаження бінарного оператора:

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

{

<тіло оператора>

}

Принаймні один аргумент у бінарному операторі повинен мати тип класу, для якого цей оператор перевантажується. Бінарний оператор може повертати величину будь-якого типу. Операції == та !=, > та <, >= та <= повинні перевантажуватись парами та повертати логічне значення. Бінарний оператор, який повертає величину типу класу, для якого цей оператор перевантажується, повинен створювати новий об’єкт цього класу, виконати з ним необхідні дії і повернути його в якості результату. Складні операції присвоювання (+=, -= і т.д.) перевантажувати не можна. Але якщо у класі перевантажена відповідна бінарна операція, то при виконанні складної операції присвоювання викликаються спочатку відповідна бінарна операція, а потім операція присвоювання.

| **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**

У мові C# є можливість перевантажувати операції явного та неявного 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>

}

Обидві операції виконують перетворення із типу аргумента в цільовий тип. Одним із цих типів має бути тип класу, у якому перевантажується операція.

| **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**

*Інкапсуляція* – один із основних принципів об’єктно-орієнтованого програмування, згідно з яким поля є внутрішніми даними об’єкта, а тому безпосереднє звертання до полів поза межами класу є недопустимим. Таке безпосереднє звертання несе у собі небезпеку їх неправильного використання. Наприклад, у класі Rectangle поля a і b використовують для збереження довжини сторін прямокутника. Але якщо ці поля описати зі специфікатором доступу public, то у програмі можна створити об’єкт цього класу і надати полям неприпустимі від’ємні значення.

Rectangle r1 = new Rectangle();

r1.a=-2;

r1.b=-9;

Для того, щоб уникнути таких помилок поля класу описують як закриті (зі специфікатором доступу private), тим самим забороняючи доступ до полів поза межами класу. Для коректного звертання до закритого поля об’єкта у класі додатково можна розробити методи для зчитування та встановлення його значення, у яких можна здійснити перевірку коректності значень, що необхідно надати закритому полю. Іноді, при описі закритого поля до його імені додають знак нижнього підкреслювання. Функція, яка повертає значення закритого поля, як правило, має префікс «Get\_»,а функція, яка використовується для встановлення значення закритого поля – «Set\_».

**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);**

}

}

Зараз звертання до закритих полів об’єкта у програмі можливе тільки з використанням описаних функцій зчитування та встановлення значень.

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**

У мові C# проблему коректного використання полів класу можна розв’язати із використанням *властивостей.* *Властивість –* церізновид метода, який слугує для організації коректного доступу до полів класу.

Загальний синтаксис опису властивості:

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

{

get

{

*// тіло функції зчитування значення*

return <результат>;

}

set

{

*// тіло функції встановлення значення*

}

}

Як правило, властивість оголошують відкритою (зі специфікатором доступу public). Якщо властивість використовується для доступу до деякого закритого поля, то тип властивості повинен співпадати із типом цього поля. Тип властивості не може бути void. При описі властивості може бути відсутньою або частина get або частина set, але не обидві одночасно. Якщо у описі властивості присутня тільки частина get, то така властивість називається *властивістю тільки для читання*. Якщо у описі властивості присутня тільки частина set, то така властивість називається *властивістю тільки для запису*. Метод get повинен містити оператор return. В методі set для доступу до значення, яке встановлюється, використовується неявний параметр value.

**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;

}

}

Характерною рисою властивостей є те, що із властивостями можна працювати як із звичайними полями об’єкта. В той же час, при наданні властивості деякого значення насправді викликається функція встановлення значення (блок set), а при звертанні до значення властивості викликається функція зчитування значення (блок get).

**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**

*Індексатори* є різновидом властивостей і дозволяють звертатися до полів класу за деяким індексом.

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*

}

}

Як і у випадку властивостей, як правило, індексатор оголошують відкритим (зі специфікатором доступу public). Тип індексатора не може бути void. При описі індексатора може бути відсутньою або частина get або частина set, але не обидві одночасно. Якщо у описі індексатора присутня тільки частина get, то такий індексатор називається *індексатором* *тільки для читання*. Якщо у описі індексатора присутня тільки частина set, то такий індексатор називається *індексатором* *тільки для запису*. Метод get повинен містити оператор return. В методі set для доступу до значення, яке встановлюється, використовується неявний параметр value.

У списку індексів вказують через кому опис індексів (тип індексу та його ім’я). Тип індексу та кількість індексів у списку може бути довільною, але найчастіше використовують один індекс цілого типу.

Індексатори в основному використовуються у класах для доступу до закритого поля-масиву (одновимірний або багатовимірний).

**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; |

**4.** **Успадкування**

**4.1. Поняття успадкування**

Однією із проблем у програмуванні є повторне використання створеного коду та його модифікація. У об’єктно-орієнтованому програмуванні цю проблему можна розв’язати із використанням успадкування. *Успадкування* – один із принципів об’єктно-орієнтованого програмування, згідно якого новий клас (*клас-нащадок*) описується на основі вже існуючого класу (*класу-предка*). При цьому клас-нащадок автоматично успадковує дані (поля) та функціональні можливості (методи) класу-предка.

Загальний синтаксис опису класу-нащадка:

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

{

...

}

У мові C# клас-предок при успадкуванні може бути тільки один.

**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;

}

}

Як зазначалось вище клас-нащадок отримує у спадок усі члени (окрім конструкторів) класу-предка − це його, так звана, *успадкована частина*. Також клас-нащадок може містити і нові члени − це його *власна частина*. У класі-нащадку доступ можливий тільки до тих членів успадкованої частини, які описані у класі-предку зі специфікаторами доступу public або protected. Доступ до захищених (protected) членів успадкованої частини можливий тільки в межах опису класу-нащадка. Закриті (private) члени класу-предка у класі-нащадку є недоступними.

**4.2. Успадкування та конструктори**

При успадкуванні за конструювання (виділення пам’яті та ініціалізацію) успадкованої частини класу-нащадка відповідає конструктор класу-предка, а за власну частину − конструктор класу-нащадка. Тому будь-який конструктор класу-нащадка повинен забезпечити коректне конструювання як базової частини, так і власної. При цьому слід пам’ятати наступне.

Спочатку конструюється успадкована частина. Для цього за замовчуванням викликається або конструктор за замовчуванням класу-предка (якщо у класі-предка не визначено жодного конструктора) або конструктор без параметрів класу-предка (якщо такий описано у класі-предка). За бажанням у мові C# є можливість *явного* виклику конструктора класу-предка:

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

{

<тіло конструктора класу-нащадка>

}

Якщо у класі-предка описано конструктори з параметрами але при цьому не описано конструктор без параметрів, то явний виклик конструктора класу-предка є обов’язковим.

**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;

}

}

У наведеному прикладі при описі конструктора з параметрами у класі-нащадку RParallelepiped ми змушені здійснити явний виклик конструктора з параметрами класу-предка Rectangle, оскільки у класі предка Rectangle відсутній конструктор без параметрів.

**4.3. Перевизначення методів**

Часто при визначенні класу-нащадка доводиться описувати однойменні дії, які також виконуються у класі предка, але мають інший зміст. Наприклад, розглянемо методи знаходження площі прямокутника (клас-предок Rectangle) і площі повної поверхні прямокутного паралелепіпеда (клас-нащадок RParallelepiped). Як у класі предка, так і у класі нащадка їх доцільно назвати Square. Опис у класі нащадка методу, ім’я якого використане у класі предка, називається *перевизначенням* (*простий поліморфізм*). При цьому застосовується *механізм приховування*, згідно якого метод, описаний у класі нащадка, “приховує” однойменний метод класу-предка. Та це не означає, що приховані методи стають недоступними. Завжди є можливість звернутися до відкритого чи захищеного прихованого методу класу-предка за допомогою ключового слова base:

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

При перевизначенні методів використовують специфікатор new. Його використання не є обов’язковим: це тільки рекомендація дотримання хорошого стилю, що полегшує читання коду. У випадку відсутності цього специфікатора помилки не буде. Компілятор тільки видасть відповідне рекомендаційне попередження.

**Приклад**. Наведемо можливу реалізацію методу Square у наших класах Rectangle та RParallelepiped. Знаходження площі поверхні будемо шукати за формулою

*площа поверхні паралелепіпеда = Периметр основи\*висоту +2\* площа основи.*

Оскільки клас RParallelepiped є нащадком класу Rectangle (прямокутник, що виступає в ролі основи), то площу поверхні буде обчислено за формулою

*площа поверхні паралелепіпеда = Периметр прямокутника\*висоту +2\* площа прямокутника.*

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**

У мові С# існує можливість заборони успадкування від класу. Це здійснюється за допомогою специфікатора sealed у списку специфікаторів класу.

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

public **sealed** class RParallelepiped: Rectangle

{

...

}

**5.** **Поліморфізм**

**5.1. Поняття поліморфізму**

Як вже було відмічено вище, при успадкуванні класів може виявитися, що деякі властивості об’єктів, зберігаючи назву, змінюються по суті (див. п. 4.3.). Для цього у мові програмування повинен бути передбачений механізм забезпечення *поліморфізму*. Дослівно значення цього терміну у перекладі з грецької означає «наявність багатьох форм». Іншими словами *поліморфізм* − це властивість деякого об’єкта приймати різні форми в залежності від ситуації. В об’єктно-орієнтованому програмуванні цей термін використовують по відношенню до методу. При цьому розрізняють три основні види 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)**

Передумовою складного поліморфізму є *правило неявного висхідного перетворення типів* при успадкуванні, згідно якого *змінній класу-предка, як покажчику, можна надавати адресу об’єкта, який є екземпляром довільного класу-нащадка.* Наприклад:

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

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

Із наведеного прикладу виникає питання: для об’єкта r буде викликатись метод Square класу-предка Rectangle чи метод Square класу-нащадка RParallelepiped? При успадкуванні класів до статичних поліморфних методів застосовується так званий *механізм раннього зв’язування − метод класу зв’язується із об’єктом під час компіляції програми у відповідності до типу об’єкта*. Так для нашого прикладу, оскільки об’єкт r має тип Rectangle та метод Square є статичним поліморфним, то у відповідності до механізму раннього зв’язування на етапі компіляції із об’єктом r буде пов’язано метод Square класу Rectangle.

*Механізм пізнього зв’язування* забезпечує *зв’язування методу із об’єктом під час виконання програми у відповідності до типу того об’єкта, на який він посилається.* Механізм пізнього зв’язування застосовується до *віртуальних методів*. Методи, оголошені віртуальними, на етапі компіляції зв’язуватися із об’єктом не будуть. Для кожного класу, що містить віртуальні методи автоматично будується спеціальна внутрішня *таблиця віртуальних методів* (ТВМ), у яку записуються адреси віртуальних методів. Цією таблицею користуються всі об’єкти даного класу для визначення адрес віртуальних методів на етапі виконання програми. Адреса ТВМ зберігається в об’єкті в спеціальному внутрішньому, невидимому для програміста полі. Запис адреси ТВМ в це поле відбувається неявно при виконанні конструктора, тому спроби виклику віртуальних методів до виклику конструктора призводять до помилок адресації.

У мові C# для оголошення у класі-предку віртуального методу використовується специфікатор virtual. Цей специфікатор не можна використовувати разом із специфікаторами static, abstract, private та override. Не можуть бути віртуальними конструктори класу. Для класу, який містить віртуальні методи не можна заборонити успадкування.

При перевизначенні віртуального методу у класі-нащадку використовують специфікатор override. Використання специфікатора override не є обов’язковим. За його відсутності метод буде перевизначено як статичний поліморфний.

**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;

Зараз при виклику методу Square об’єкта r буде викликатися метод Square, описаний у класі RParallelepiped.

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

**5.3. Перетворення типів при успадкуванні**

Мова C# підтримує також *явне низхідне перетворення типів* при успадкуванні. Наприклад:

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;

Особливістю явного низхідного перетворення є те, що воно здійснюється на етапі компіляції і тому слід бути уважним, щоб не створити помилку часу виконання. Наприклад:

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>

Оператор as діє подібним чином як і явне приведення. Однак, якщо приведення є неможливим, то повертає null − порожню адресу, а не породжує помилку часу виконання. Оператор is повертає значення true, якщо перший об’єкт не містить порожню адресу та приведення є можливим, інакше − false. Використовуючи ці оператори, наведений приклад можна модифікувати так:

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**

Властивості успадкування та поліморфізму дають можливість визначати клас, який є загальним описанням деякої групи подібних об’єктів реальної дійсності. Але дуже часто, описуючи такий клас, недоцільно чи навіть неможливо вказати реалізацію деяких спільних для всієї групи методів. У цьому випадку методи описують як *абстрактні* і реалізації для них не вказують.

У мові C# опис абстрактного методу здійснюється із використанням специфікатора abstract, не містить тіла та закінчується «;». Абстрактний метод є неявним віртуальним методом. Абстрактними можуть бути також властивості та індексатори. Неможливо оголосити конструктор абстрактним.

Клас, що містить принаймні один абстрактний член (метод, властивість або індексатор) називають *абстрактним класом*. При описі абстрактного класу також використовується специфікатор abstract. Абстрактний клас може використовуватися тільки як клас-предок. Не абстрактний клас, який є нащадком абстрактного повинен містити перевизначення усіх успадкованих абстрактних членів.

**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;

}

}

**Питання для самоконтролю**

**6.** **Інтерфейси**

**6.1. Поняття інтерфейсу. Опис інтерфейсів**

Абстрактний клас надає *поліморфний інтерфейс* (відкриті абстрактні методи, властивості або індексатори) для всіх своїх класів-нащадків. Однак цей поліморфний інтерфейс має ряд суттєвих обмежень.

По-перше, у кожному класі-нащадку обов’язково має підтримуватися цей поліморфний інтерфейс (потрібно реалізовувати відповідний набір абстрактних методів). Однак, може виникнути ситуація, у якій функціональність, яку задає поліморфний інтерфейс, має сенс не для всіх класів-нащадків. Наприклад у випадку абстрактного класу-предка Figure не має сенсу визначати метод площі для класу-нащадка Point (представляє точку), та є сенс для − Rect (представляє квадрат), Circle (представляє круг), тощо.

По-друге, досить часто при описі класів об’єктів реальної дійсності можна виділити окремі групи споріднених (схожих за функціональністю чи логічно-зв’язаних) методів, та разом з цим ці описувані класи не мають спільного класу-предка. Наприклад метод-властивість Volume (об’єм) характерна як для об’єктів класу RParallelepiped (прямокутний паралелепіпед, об’єм паралелепіпеда) так і для об’єктів класу Car (автомобіль, об’єм двигуна). Однак це абсолютно різні класи об’єктів і вони апріорі не можуть мати спільного класу-предка.

У таких випадках зручним є використання типу інтерфейс − дуже схожого типу до абстрактних класів. Абстрактні класи, у яких оголошено тільки абстрактні члени (методи, властивості, індексатори) називаються *інтерфейсами*. Для інтерфейсів, як і для класів, визначено поняття успадкування. Наведемо загальну схему описання інтерфейсу.

[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.>]);

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

}

Оскільки всі члени інтерфейсу є абстрактними (реалізації для них не наводиться), то, на відміну від абстрактних класів, при описанні членів у інтерфейсі специфікатор аbstrаct не використовується. Ще однією характерною особливістю інтерфейсів є те, що всі члени інтерфейсу є неявно відкритими (public) і не можуть супроводжуватися ніякими специфікаторами доступу.

**Приклад.** Для окремих класів геометричних фігур є сенс у визначенні таких числових характеристик як периметр та площа. Також для окремих класів геометричних фігур доводиться обчислювати радіус вписаного та описаного кола. Таку функціональність можна виділити у дві окремі групи методів та описати наступними інтерфейсами:

public interface IMeasurable

{

double Perimeter();

double Square();

}

public interface ICircumcircleIncircle

{

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

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

}

Імена інтерфейсів прийнято починати з великої літери “І” (ITriangle, IUnknown, тощо). Як і для абстрактного класу, не можна створити екземпляр типу інтерфейсу.

**6.2. Реалізація інтерфейсів**

Кажуть, що клас *реалізує* (*підтримує*) інтерфейс, якщо він містить реалізацію усіх абстрактних членів, які описані у інтерфейсі. У мові С# клас може реалізовувати декілька інтерфейсів одночасно. У цьому випадку вони вказуються через кому. Якщо клас є класом-нащадком та реалізовує деякі інтерфейси, то першим після двокрапки має бути вказане ім’я класу-предка, а вже потім список інтерфейсів. З урахуванням сказано загальне правило опису класу, що реалізує деякі інтерфейси таке:

[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>

}

Реалізація інтерфейсу у класі передбачає обов’язкову реалізацію всіх членів інтерфейсу. Кожен член інтерфейсу може бути реалізований одним із двох способів: *неявним* або *явним*.

**Неявна** **реалізація** окремого члену інтерфейсу у класі нічим не відрізняється від його реалізації у випадку якби він був власним членом цього класу. Неявна реалізація вимагає обов’язкову відкриту (public) реалізацію відповідного члену. У загальному випадку неявна реалізація методу інтерфейсу здійснюється у такий спосіб:

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

{

<реалізація методу>

}

**Явна реалізація**. Як було зазначено вище, окремий клас може реалізовувати довільну кількість інтерфейсів. Через це завжди існує ймовірність реалізації інтерфейсів зі членами, що мають однакові імена, і, отже, виникає необхідність в усуненні конфліктів на рівні імен. Для таких випадків у мові C# передбачений синтаксис явної реалізація інтерфейсів. У загальному випадку явна реалізація методу інтерфейсу здійснюється у такий спосіб:

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

{

<реалізація методу>

}

При явній реалізації члена інтерфейсу вказувати специфікатор доступу не потрібно, оскільки члени, інтерфейсу реалізовані явно, є неявно закритими (private).

**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**

Для об’єктів класу, який реалізує деякий інтерфейс доступ до функціональних можливостей цього інтерфейсу здійснюється через посилання на цей інтерфейс. Оскільки, інтерфейси не мають власних конструкторів, то єдиним засобом для створення посилання на той чи інший інтерфейс є операція явного перетворення типів для об’єктів класів, які підтримують даний інтерфейс. 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());

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

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

## DataGridView

Елемент управління DataGridView надає персоналізовану таблицю для відображення даних. DataGridView підтримує три режими роботи з даними:

1. ***Незв’язаний режим*** − режим, у якому дані зберігаються безпосередньо в самому елементі управління.
2. ***Зв’язаний режим*** − режим, у якому відображаються дані із зовнішніх джерел (наприклад, із колекції ArrayList, із таблиць баз даних).
3. ***Віртуальний режим*** − режим, у якому елемент управління приєднується до спеціального керованого кешу даних.

Обмежимось детальним оглядом тільки перших двох режимів.

***Незв’язаний режим***. Цей режим використовується в тому випадку, якщо необхідно вивести невеликий обсяг даних у вигляді таблиці. Доступ до даних у незв’язаному режимі здійснюється через наступну властивість:

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

Ця властивість повертає колекцію всіх рядків для даного елемента управління DataGridView. Через цю властивість є можливість звернутися до значення конкретної комірки персоналізованої таблиці за індексами цієї комірки:

|  |
| --- |
| 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; } |

При цьому у обох випадках слід пам’ятати, що індексація рядків та стовпців починається з нуля, а значення комірки є екземпляром типу object.

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 |

У класі DataGridView реалізована підтримка величезної кількості подій, що дає можливість відстежувати та обробляти для екземплярів цього класу широкий спектр обробки подій миші, зміни даних, тощо. Ми ж відмітимо тільки наступну:

|  |  |
| --- | --- |
| 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 |

Іноді для об'єкта DataGridView виникає потреба у налаштуванні зовнішнього вигляду окремої комірки. Ця можливість реалізується через наступну властивість класу DataGridViewCell:

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

Незв’язаний режим може виявитися зручним при програмуванні таких математичних об’єктів як матриці. Проілюструємо роботу із елементом управління DataGridView у незв’язному режимі на прикладі програмної реалізації наступного завдання.

|  |
| --- |
| *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;  }  }  } |

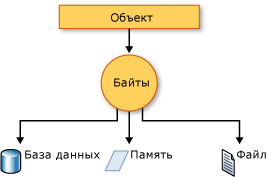
***Зв’язаний режим.*** Зв'язаний режим підходить для управління даними за допомогою автоматичної взаємодії із джерелом даних. Коли елемент управління прив'язаний до даних, рядки з даними передаються і приймаються без необхідності явного управління з боку користувача. Клас DataGridView підтримує модель прив'язки даних Windows Forms. Це означає зокрема, що у якості джерела даних може бути будь-який тип, який реалізує інтерфейс IList або IList<T>. Тобто у якості джерела даних можуть виступати списки та масиви.

Для інкапсуляції джерела даних використовується клас BindingSource. Цей клас спрощує прив'язку елементів управління на формі до даних, надаючи засоби управління грошовими одиницями, передаючи повідомлення про зміни і виконуючи інші функції зв'язку між елементами управління Windows Forms та джерелами даних. Це досягається шляхом приєднання компонента BindingSource до джерела даних за допомогою властивості DataSource елемента управління DataGridView. Усі наступні взаємодії з даними здійснюються за допомогою викликів компонента BindingSource. Нижче наведені основні властивості та методи класу BindingSource, які надають доступ до даних:

|  |  |
| --- | --- |
| 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 |

Проілюструємо роботу із елементом управління DataGridView у зв’язному режимі на прикладі програмної реалізації наступного завдання.

Серіалізація являє собою процес перетворення об'єкта в потік байтів з метою збереження його в пам'яті, в базі даних або у файлі. Її основне призначення - зберегти стан об'єкта для того, щоб мати можливість відтворити його при необхідності. Зворотний процес називається десеріалізації.



Об'єкт серіалізуєтся в потік, який переносить не тільки дані, але й відомості про тип об'єкта, його версію, мову і регіональні параметри, а також ім'я збірки. З цього потоку об'єкт можна зберегти в базі даних, файлі або пам'яті.

Для серіалізації об'єкта потрібно об'єкт, який буде серіалізуватись, потік, який буде містити серіалізований об'єкт, і об'єкт Formatter. Простір імен System.Runtime.Serialization містить класи, необхідні для серіалізації та десеріалізації об'єктів.

|  |
| --- |
| *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;  }  }  } |